HETEROGENEOUS COMPUTING WORKSHOP

Proceedings

Sixth Heterogeneous Computing Workshop

(HCW '97)

Proceedings

Sixth Heterogeneous Computing Workshop

(HCW '97)

April 1, 1997 Geneva, Switzerland

Edited by

Debra Hensgen, Naval Postgraduate School

Sponsored by

The IEEE Computer Society

Office of Naval Research

19971006 082

DTIC QUALITY INSPECTED 4

IEEE Computer Society Press
Los Alamitos, California

Washington • Brussels • Tokyo

IEEE Computer Society Press
10662 Los Vaqueros Circle
P.O.Box 3014
Los Alamitos, CA 90720-1264

Copyright © 1997 by The Institute of Electrical and Electronics Engineers, Inc.
All rights reserved.

Copyright and Reprint Permissions: Abstracting is permitted with credit to the source. Libraries may photocopy beyond the limits of US copyright law, for private use of patrons, those articles in this volume that carry a code at the bottom of the first page, provided that the per-copy fee indicated in the code is paid through the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923.

Other copying, reprint, or republication requests should be addressed to: IEEE Copyrights Manager, IEEE Service Center, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331.

The papers in this book comprise the proceedings of the meeting mentioned on the cover and title page. They reflect the authors' opinions and, in the interests of timely dissemination, are published as presented and without change. Their inclusion in this publication does not necessarily constitute endorsement by the editors, the IEEE Computer Society Press, or the Institute of Electrical and Electronics Engineers, Inc.

IEEE Computer Society Press Order Number PR07879
IEEE Order Plan Catalog Number 97TB100103
Library of Congress number 96-80211
ISBN 0-8186-7879-8
Microfiche ISBN 0-8186-7881-X

Additional copies may be ordered from:

IEEE Computer Society Press Customer Service Center 10662 Los Vaqueros Circle P.O. Box 3014 Los Alamitos, CA 90720-1314 Tel: +1-714-821-8380 Fax: +1-714-821-4641 Email: cs.books@computer.org	IEEE Service Center 445 Hoes Lane P.O. Box 1331 Piscataway, NJ 08855-1331 Tel: +1-908-981-1393 Fax: +1-908-981-9667 misc.custserv@computer.org	IEEE Computer Society 13, Avenue de l'Aquilon B-1200 Brussels BELGIUM Tel: +32-2-770-2198 Fax: +32-2-770-8505 euro.ofc@computer.org	IEEE Computer Society Ooshima Building 2-19-1 Minami-Aoyama Minato-ku, Tokyo 107 JAPAN Tel: +81-3-3408-3118 Fax: +81-3-3408-3553 tokyo.ofc@computer.org

Editorial production by Penny Storms
Cover by Alex Torres
Printed in the United States of America by Technical Communication Services

The Institute of Electrical and Electronics Engineers, Inc.

Table of Contents

Message from the General Chair

This is the 6th Heterogeneous Computing Workshop, also known as HCW '97. Heterogeneous computing is a very important research area with great practical impact. The topic of heterogeneous computing covers many types of systems. A heterogeneous system may be a set of machines interconnected by a wide-area network and used to support the execution of jobs submitted by a variety of users. A heterogeneous system may be a suite of high-performance machines tightly interconnected by a fast dedicated local-area network and used to process a set of production tasks, where the subtasks of each task may execute on different machines in the suite. A heterogeneous system may also be a special-purpose embedded system, such as a set of different types of processors used for automatic target recognition. In the extreme, a heterogeneous system may consist of a single machine that can reconfigure itself to operate in different ways (e.g., in different modes of parallelism). All of these types of heterogeneous systems (as well as others) are appropriate topics for this workshop series. I hope you find the contents of these proceedings informative and interesting, and encourage you to look also at the proceedings of past and future HCWs.

Many people have worked very hard to make this workshop happen. I thank Richard F. Freund, NRaD, for founding this workshop series, and guiding its continued success. Debbie Hensgen, Naval Postgraduate School, was this year's Program Committee Chair, and she assembled the excellent program and collection of papers in these proceedings. Debbie did this with the assistance of her Program Committee, whose members are listed in these proceedings. The Vice-General Chair was Dan Watson, Utah State University, who helped me in many ways, including handling workshop publicity.

This workshop is held each year in conjunction with the International Parallel Processing Symposium (IPPS). Viktor Prasanna, University of Southern California, is the Symposium Co-Chair of IPPS '97. The HCW series is very appreciative of his constant cooperation and assistance.

This year the workshop is cosponsored by the IEEE Computer Society and the Office of Naval Research. I thank Dr. Andre M. van Tilborg, Director of the Math, Computer, & Information Sciences Division of the Office of Naval Research, for arranging ONR support to fund publication of the workshop proceedings (under grant number N00014-97-1-0121).

Penny Storms, IEEE Computer Society Press, was responsible for publishing these proceedings. I have worked with Penny before, and as always she was very efficient, effective, professional, and pleasant.

Richard F. Freund, Chair of the HCW Steering Committee, nominated me to be General Chair of HCW '97, and my nomination was approved by the Steering Committee. I appreciate the trust they had in me to accomplish this task.

I thank my secretary Carol Edmundson for her assistance with my duties for this workshop. I thank my wife, Janet, for her advice on many workshop related matters, and for continuing to teach me, by example, to be a kinder, gentler person. Finally, I apologize to my one-year-old daughter Sky for letting my workshop responsibilities cut into our playtime.

H.J. Siegel
School of Electrical and Computer Engineering
Purdue University

Message from the Program Committee Chair

I am very pleased to have had the privilege of chairing this year's workshop and hope that you will gain significantly from reading these proceedings, listening to our talks, and participating in our closing discussion. The program committee and I are proud to have four outstanding case studies along with 14 excellent regular papers. I wish to thank our large, hard-working program committee for their enthusiasm and cooperative spirit as we each, along with external reviewers Mike Zyda, Jon Weissman, Cynthia Irvine, and Geoffrey Xie, reviewed the numerous submissions. Special thanks to Dan Watson for all of his work on publicity; to our Case Studies co-chairs John Antonio and Taylor Kidd for finding the excellent projects we will hear about today; to Viktor Prasanna for organizing the excellent conference to which this workshop belongs; and to Penny Storms for her hard work in bringing these proceedings together. And mostly, thanks to H. J. Siegel for keeping everything running on schedule and for his nearly endless voice mail reminders.

Debra Hensgen
Naval Postgraduate School

Committees

General Chair	H. J. Siegel, *Purdue University*
Vice General Chair	Dan Watson, *Utah State University*
Program Committee Chair	Debra Hensgen, *Naval Postgraduate School*
Case Studies Co-Chairs	John K. Antonio, *Texas Tech University* Taylor Kidd, *Naval Postgraduate School*
Steering Committee	Richard F. Freund, Chair, *NRaD* Francine Berman, *UCSD* Jack Dongarra, *University of Tennessee* Debra Hensgen, *Naval Postgraduate School* Paul Messina, *Caltech* Jerry Potter, *Kent State University* Viktor Prasanna, *USC* H. J. Siegel, *Purdue University* Vaidy Sunderam, *Emory University*
Program Committee Members	Ishfaq Ahmad, *Hong Kong U. of Science & Tech.* Giovanni Aloisio, *Università degli Studi di Lecce* Cosimo Anglano, *Università di Torino* Francine Berman, *UCSD* Steve Chapin, *University of Virginia* Partha Dasgupta, *Arizona State University* Hank Dietz, *Purdue University* Mary Eshaghian, *New Jersey Institute of Technology* Andrew Grimshaw, *University of Virginia* Nayeem Islam, *IBM T. J. Watson Research Center* Gary Johnson, *George Mason University* Domenico Laforenza, *CNUCE - Institute of the Italian NRC* David Lilja, *University of Minnesota* Miron Livny, *University of Wisconsin - Madison* Bob Lucas, *DARPA* Richard C. Metzger, *Rome Laboratory* Lantz Moore, *University of Cincinnati* Jim Patterson, *Boeing Info. and Support Services* Ranga S. Ramanujan, *Architecture Technology Corp.* Rick Stevens, *Argonne National Laboratory* Vaidy Sunderam, *Emory University* Maria Uspenski, *EcoSoftware, Inc.* Dan Watson, *Utah State University* Chip Weems, *University of Massachusetts - Amherst* Elizabeth Williams, *Center for Computing Sciences* Albert Y. Zomaya, *University of Western Australia*

Session 1

System Support for Heterogeneous Computing

Session Chair

Richard F. Freund
NRaD, San Diego, CA, USA

Dynamic Load Balancing of Distributed SPMD Computations with Explicit Message-Passing

M. Cermele, M. Colajanni, G. Necci
Dipartimento di Informatica, Sistemi e Produzione
Facoltà di Ingegneria
Università di Roma "Tor Vergata"
Roma, Italy 00133
{cermele, colajanni}@utovrm.it

Abstract

Distributed systems have the potentiality of becoming an alternative platform for parallel computations. However, there are still many obstacles to overcome, one of the most serious is that distributed systems typically consist of shared heterogeneous components with highly variable computational power. In this paper we present a load balancing support that checks the load status and, if necessary, adapts the workload to dynamic platform conditions through data migrations from overloaded to underloaded nodes. Unlike task migration supports for task parallelism and other data migration frameworks for master/slave-based parallel applications, our support works for the entire class of SPMD regular applications with explicit communications such as linear algebra problems, partial differential equation solvers, image processing algorithms. Although we considered several variants (three activation mechanisms, three load monitoring techniques and four decision policies), we implemented only the protocols that guarantee program consistency. The efficiency of the strategies is tested in the instance of two SPMD algorithms that are based on the PVM library enriched by special-purpose primitives for data management. As additional contribution, our research keeps the entire support for dynamic load balancing transparent to the programmer. Even if the technical details are out of the scope of this paper, we point out that the only visible interface of our support is the activation phase.

1 Introduction

The widespread diffusion of distributed systems motivates the attempts to exploit the potential parallelism intrinsic in these computing environments. Their architecture and the autonomy of the cluster components resemble the asynchronous activity of MIMD distributed-memory machines. In addition, frameworks such as PVM and MPI greatly help to fill the gap between parallel and distributed platforms by hiding the heterogeneity of cluster components to the programmer. However, these libraries do not overcome the inefficiencies caused by the unpredictable variability of usually shared resources. Since any dynamic modification of the underlying platform deteriorates the performance of *distributed parallel computations*, a satisfactory efficiency can be achieved only by keeping the workload proportional to the computational power of each processor.

In this paper, we address heterogeneity through the notion that each node may have a different computational capacity. We propose a transparent support which aims at dynamically balancing the workload of Single Program Multiple Data (SPMD) regular computations with near-neighbor and/or multicast communications. A large number of parallel programs belongs to this class: linear algebra problems, partial differential equation solvers, image processing algorithms. For these algorithms, the same code runs on several nodes while the data space is partitioned among the nodes. Since the workload on each processing unit is a function of the number of elements contained in its subdomain, we can keep the workload balanced by means of data migrations from overloaded to underloaded nodes. When data migration schemes are feasible, they are preferable to task migration approaches for two main reasons: higher efficiency because the information to transmit is more limited and a new process startup is not required, and more precise balancing thanks to the finer granularity of the load that can be re-distributed.

In particular, we are interested in balancing strategies which are efficient but, at the same time, guarantee full consistency of the parallel program's execution. Our strategy consists of periodically evaluating the status of the platform and providing data reconfigurations if the differences among old and new load values pass a given threshold. We discuss three activation mechanisms, three load monitoring techniques, four decision policies and implement all those that give

0-8186-7879-8/97 $10.00 © 1997 IEEE

adequate guarantees of correctness. In particular, our support implements an explicitly activated protocol, a distributed load status evaluation, a centralized decision policy and a concurrent data reconfiguration. Even if the proposed load balancing schemes are best suited to SPMD parallelism, some of the methods can be applied elsewhere. We compare under different scenarios four decision policies that decide to reconfigure on the basis of parameters such as system average imbalance, system maximum imbalance, current load, current load combined with past information.

Both the SPMD applications and the dynamic support are written in C. They use PVM for the *communication* and *system inquiry* primitives [9], and a special-purpose library for the *data inquiry* primitives. The dynamic balancing support described in this paper is, in fact, part of a wider project named DAME that aims at combining the simplicity of the SPMD paradigm with the efficiency when this programming style is adopted on distributed systems. For this reason, the entire support for dynamic load balancing is kept hidden from the programmer: the only visible interface is the `check_load()` function which is provided for the explicit activation of the framework. Since the details about this project are out of the scope of this paper, the reader can refer to [5, 6] for the technical issues about heterogeneous data distribution, and data inquiry primitives that keep this heterogeneity hidden from the programmer. The goals of the entire project and other experimental results are reported in [8].

This paper is organized as follows. Section 2 contains a summary of the related work and the main motivations for a new approach. Section 3 focuses on the load balancing model that we adopt for SPMD applications. Section 4 describes the load monitoring and decision phases, and presents four decision policies that we compare in the experiments. Section 5 analyzes the activation and the data migration phases. Moreover, it outlines the effects of the load balancer on initial data distribution and application's processes. Section 6 presents the experimental results that are obtained for different load balancing schemes and scenarios. Section 7 gives some final remarks and outlines future work.

2 Related work and motivation

The increasing interest in distributed parallel computing has opened new problems in dynamic load balancing that is a well studied subject in other areas. While much attention has been devoted to develop efficient reconfiguration schemes for task parallel programs running on parallel or distributed platforms [1, 7, 17, 3, 4, 18], the results in the area of data parallelism are more limited. These computations allow a reconfiguration protocol based on data migration that is more efficient than task migration. On the other hand, there are higher risks of incorrect execution because data migration affects the initial domain partition on which the SPMD implementation has been based. The necessity of dynamic data redistribution can be caused by *internal factors*, such as for irregular data parallel computations, or *external factors*, such as in distributed systems that provide inherently dynamic platforms. Data migration techniques have been adopted for irregular algorithms [11, 14, 15] and for regular applications running on distributed platforms [13, 3, 16, 10]. The load balancing strategies proposed in this paper were designed to face the problems related to these latter environments. To the best of our knowledge, none of the results achieved in this area yet refers to SPMD algorithms with explicit communications that are the most common parallel computations. Therefore, our contribution represents an important step to facilitate the portability of these programs on distributed platforms.

Related works that consider programming models very distant from SPMD programs with communications are *Dataparallel C* [13] and *Piranha* [2]. *Dataparallel C* is a run-time support which allows SIMD shared memory applications to run efficiently on clusters of workstations. Its programming environment and the migration support are based on the virtual processor concept. *Piranha* dynamically adapts Linda computations to the available workstations. However, these programs are implemented on the basis of a virtual shared memory paradigm.

On the other hand, there are two recent results that concern a paradigm closer to ours that is, SPMD master/slave computations without communications [3, 10]. The *Application Data Movement* described in [3] is a run-time support that furnishes a set of functions that help the programmer to implement adaptive workload distribution of master/slave programs through data migration. Hamdi and Lee [10] propose a data redistribution method for parallel image processing. The main novelty of this support with respect to other load balancing strategies, which are activated only at the end of fixed intervals or phases [14, 7, 16], is that data migration can occur even within an iteration. This method is indispensable especially if we consider computations, such as image processing, that are characterized by few very long iterations.

As main contributions, we provide a support that works for regular SPMD applications containing near-neighbor and/or multicast communications, allows the programmer to activate the dynamic balancer at any computation point (as well as in [10]), guarantees the transparency of the load balancing phases.

3 Load balancing model

As computing model, we consider a regular SPMD application with explicit communications. Moreover, we assume a simple machine model consisting of p nodes. Each node executes one *internal* process of the SPMD application, and may execute other *external* processes. By analogy, we call *external workload* the load represented by other (sequential or parallel) jobs that run on the same workstation used by our parallel application that causes *internal workload.* The nodes are fully connected as in an Ethernet-based or tokenring network. Each internal process has its own address space, and the need for any data entry placed in the memory of other nodes requires explicit message passing. The dynamic load balancing model we are proposing for SPMD computations is similar to that given in [17], and it is based on the following phases, that can be implemented in different ways:

1. **Activation mechanism**: This phase represents one of the possible points of interaction between the load balancing support and the internal processes. The load balancer can be activated *explicitly* (by some function called by the application) or *implicitly* (without any application's intervention). In addition, this activation can determine a barrier for the internal processes (*synchronous*) or not (*asynchronous*).

2. **Load monitoring**: Once the support has been activated, each process evaluates the status of the external workload on the respective node. This phase is usually executed in a distributed way. The alternatives regard the ways in which the load parameter can be evaluated: by adopting some external functions (*active*) or by using the evolution of the parallel application itself (*passive*).

3. **Decision**: This phase determines, on the basis of the load parameters, whether the workload should be reconfigured or not, and how to redistribute it. Several centralized and distributed policies have been proposed in literature. Moreover, the decision can be taken on the basis of the last evaluated parameters or using a combination of present and past information.

4. **Reconfiguration (Data migration)**: This phase represents another main point of interaction between the load balancer and the application. Therefore, we may have the same alternatives described for the phase 1: synchronous/asynchronous, implicit/explicit. Moreover, the migration strategy can be centralized or distributed.

At present, our framework follows a centralized policy and is activated explicitly by a call to the `check_load()` function which we provide together with the run-time support. The following alternatives are available: the *activation mechanism* is explicit with two alternatives for the internal process (synchronous or asynchronous); the *load monitoring* is active with two available schemes (micro-benchmarks or Unix functions); the *decision* is centralized with four available policies; the *reconfiguration* is implicit, distributed and synchronous.

Without the aim of considering all of the feasible alternatives, we illustrate the adopted choices and give some motivations for each of them. Let us first focus on the two phases which are independent of the parallel application (*load monitoring* and *decision*) and postpone the discussion about *activation* and *reconfiguration* phases to Section 5.

4 Application independent phases

4.1 Load monitoring

In a heterogeneous system with different workstation speeds, each node has a nominal power, and a duty cycle η_i that is, the fraction of node processing capacity which is consumed by local tasks. We call *available capacity* c_i the computing power that remains for executing the distributed parallel program. When we normalize each c_i to the cumulative capacity of all the nodes, we obtain a *relative available capacity* χ_i, where $\sum_{i=1}^{p} \chi_i = 1$. Since in a distributed system the duty cycle may change during the execution of the parallel application, we denote $\eta_i(t)$, $c_i(t)$ and $\chi_i(t)$ as functions of time. The goal of the load monitoring phase is to determine a measure for $c_i(t)$. There are two main alternatives for estimating this value: by means of external functions (*active* methods) or by using the application itself (*passive* methods). This latter is an ideal solution that aims at avoiding extra-overheads caused by the active methods. See for example [13, 10].

Our first idea for implementing a load monitoring was to use a passive method in which the load parameter was extrapolated by the time difference for executing a significative portion of code without communications or synchronizations among the processes. However, in the present version of the support, we discarded the hypothesis of adopting a passive method for three main reasons: 1) the programmer would have been required to select this portion of 'test' code, thus contradicting the aim of transparency; 2) we could have no guarantee about the quantity of involved operations to give a precise estimate about $c_i(t)$; 3) although active methods are more time wasting, they guarantee transparency, application independency, and a more accurate estimate of the actual load.

Therefore, we adopt two active methods for load monitoring. The first uses the Unix system call that gives different kinds of information about current computational power (number of tasks in the run queue, 1-minute load average, rate of CPU context switches, etc.). Following the results reported in [12], we adopt the number of tasks in the run queue of each node as basis for measuring the external workload $\eta_i(t)$. The second possibility is to estimate the current load through a synthetic micro-benchmark that is, a purposely implemented function which gives an immediate estimate about the available capacity $c_i(t)$ for executing scientific-based programs. Due to space limitations, we could not include the code of the adopted micro-benchmark, however it is to be noted that a different class of parallel applications would require a different micro-benchmark. The Unix call is faster in the evaluation of the load (between 80 and 100 milliseconds) but requires some additional computations to estimate the available capacity $c_i(t)$. Moreover, this estimate causes some approximation in the load information. On the other hand, the micro-benchmark is a slower technique (between 0.5 and 2 seconds, depending on the machine power) but gives an immediate estimate about $c_i(t)$.

Both methods are available in our support. For example, in our experiments, we adopt the Unix function for the LU factorization algorithm, and the micro-benchmark for the heat equation solver. In both instances, the load information $c_i(t)$ is sent to the *reconfiguration master* that executes the decision policy. Typically, the most powerful node of the platform is chosen as the master.

4.2 Decision policies

This phase takes two important decisions: *whether to redistribute* and *how to redistribute*. We discuss four policies that are based on a centralized approach. The master process is responsible for collecting the load parameters, executing one decision policy, and broadcasting the decisions to the internal processes. This message consists of three parts: *operation* (to reconfigure or not), *node information* (map of sender and receiver nodes), *data information* (map of data to transmit). Even if centralized approaches tend to be more time consuming and less feasible than distributed strategies as the number of processors in the system becomes large, we preferred them because they better guarantee the consistency of a generic SPMD application, and allow the master to keep track of global load situation.

Once the master has received the available capacities $c_i(\bar{t})$, at check-load time $\bar{t}$, from all the nodes $i = 1, \ldots, p$, the first step of our decision policies is to evaluate the relative values $\chi_i(\bar{t})$. Then, each node estimates $\Delta_i(\bar{t}) = \mid \chi_i(\bar{t}) - \chi_i(t_l) \mid$ that is, the difference between the current capacity $\chi_i(\bar{t})$ and the relative power $\chi_i(t_l)$ which was available at the time of the last data redistribution t_l. Thereafter, each decision policy implements a different strategy to determine the overall *imbalance factor* $\Delta(\bar{t})$. Our intention is to compare decision policies that are focused on the average imbalance *vs.* strategies oriented to check the maximum node imbalance; policies using instantaneous information *vs.* policies using also historical information. In general, the reconfiguration is carried out when the imbalance factor passes a threshold ϑ. The following criteria were implemented:

a) *System Average Imbalance* (SAI)

This policy decides to reconfigure when the overall system is considered imbalanced that is, when

$$\Delta(\bar{t}) = 1/p \sum_{i=1}^{p} \Delta_i(\bar{t}) > \vartheta_a$$

b) *System Maximum Imbalance* (SMI)

This policy assumes that, once the available capacity of a node changes more than the threshold ϑ_b, any reconfiguration will improve program efficiency. Therefore, it decides to reconfigure when at least one node i has

$$\Delta_i(\bar{t}) = \mid \chi_i(\bar{t}) - \chi_i(t_l) \mid > \vartheta_b$$

c) *Number of Imbalanced Nodes on the basis of Current load* (NINC)

It reconfigures when $\Delta(\bar{t}) = 1/p \sum_{i=1}^{p} \Delta_i(\bar{t}) > \vartheta_c$

where $\Delta_i(\bar{t}) = \begin{cases} 1 & \text{if } \mid \chi_i(\bar{t}) - \chi_i(t_l) \mid > \lambda \\ 0 & \text{otherwise} \end{cases}$

This policy is similar to SMI. However, NINC tends to reduce the reconfigurations to the instances in which more than one node is imbalanced. In our experiments, a node is considered imbalanced when its relative capacity has changed more than $\lambda = 0.4$, which is the default threshold value. Therefore, the behavior of this policy is more sensitive to ϑ_c that is, the fraction of imbalanced nodes in the system.

d) *Number of Imbalanced Nodes on the basis of current and Past load* (NINP)

It reconfigures when $\Delta(\bar{t}) = 1/p \sum_{i=1}^{p} \Delta_i(\bar{t}) > \vartheta_d$

where

$$\Delta_i(\bar{t}) = \begin{cases} 1 & \text{if } \sum_{j=1}^{h} f(j) \mid \chi_i(\bar{t}) - \chi_i(\bar{t}_j) \mid > \lambda \\ 0 & \text{otherwise} \end{cases}$$

and $\chi_i(\bar{t}_j)$, for $j = 1, \ldots, h$, denote the relative powers that were available at the last h check-load points.

This method measuring the load imbalance in all check-load points from the last reconfiguration is similar to the NINC criterion. However, the new policy measures the load imbalance of a node over a period of time rather than using only the most recent information. The goal of this criterion is to reduce the effects of temporary power variations. In our experiments we set $h = 4$ and $\lambda = 0.4$, while the function $f(j) = e^{-j/2}/G$ is a *decay factor* that assigns a smaller weight to the older load parameters, and G is the normalizing factor.

It is to be noted that the four policies have the same linear complexity $O(p)$. Since this phase is much faster than any other phase of the balancing process, the decision does not represent a big bottleneck even if it is centralized. The choice of the most suitable threshold for any kind of SPMD application and platform condition is one of the theoretical issues (the other is the optimal frequency of check-load points) that deserves further study. The experiments are carried out by empirically choosing policy thresholds that work fine for the kind of considered applications.

The second basic issue in the decision policy is how to redistribute data. An optimal solution to this problem is very hard to find if we consider task parallel programs, because it involves integer programming solution. Conversely, in the instance of data parallel computations, the possibility of partitioning the data space into different size portions allows us to assign to each node a computational workload which is proportional to its speed. This simple optimal technique can be achieved because we redistribute a workload (data) that has a much finer granularity than a task load. Since our framework works under the hypothesis that the internal workload is a function of the quantity of owned data, we can suppose that the complexity of the algorithm executed by each node is approximated by a law such as $O(n^C)$ where n is the one dimensional size of the local domain. For example, if the data domain is $M \times N$, the local domain of a node i at time $\bar{t}$ should be kept as close as possible to (*balancing equation*)

$$m_i \times n_i = (M \times N)[\chi_i(\bar{t})]^{2/C}.$$

To allow the run-time support to automatically determine the right data size to assign to each node i (that is, computing m_i and n_i), the programmer has to specify the average *computation complexity* C and the *basic grain*. This latter denotes the minimum portion of data domain that can be moved during dynamic reconfigurations. Depending on the chosen domain decomposition, the *basic grain* can be a single element, or a multiple of a row or column.

For a 1D decomposition, the *basic grain* can only be a multiple of row or column. As example, if it is set to a single row, we have $n_i = N$, and we assign to each node i a number of rows equal to

$$m_i = M[\chi_i(\bar{t})]^{2/C}.$$

For a 2D decomposition, the most common basic grain is a single element because this choice allows us to achieve best load balancing. Our data distribution support obtains a 2D decomposition by virtually partitioning the nodes in horizontal subnets, and assigning the same set of rows to the nodes of the same subnet (see nodes 1–3 and 4–6 in Figure 1.b). Since the nodes of a subnet $\mathcal{S}$ have the same number of rows, each m_i, for $i \in \mathcal{S}$, has the following identical value

$$m_i = M[\sum_{j \in \mathcal{S}} \chi_j(\bar{t})]^{2/C}.$$

Once evaluated the variables m_i, we can use the balancing equation to obtain each n_i value, that is,

$$n_i = (N/[\sum_{j \in \mathcal{S}} \chi_j(\bar{t})]^{2/C})[\chi_i(\bar{t})]^{2/C}.$$

5 Application dependent phases

5.1 Activation and reconfiguration

The activation and reconfiguration phases are the interface between the load balancing support and the application. Several protocols could be chosen on the basis of the degree of interference that one allows between the support and internal processes. We propose a partial transparent framework that does not put any responsibility for data reconfiguration on the programmer, however it requires the programmer to specify the points of the application where the support has to be activated and the frequency of the activation. To this purpose, we furnish a `check_load()` primitive that the programmer can insert into one or more point of the SPMD code. Since no control is done by the compiler, the entire responsibility for the `check_load()` insertion is left to the programmer. For this reason, instead of using 'unsafe' transparent protocols, we adopt solutions that have partial or total interference with the application. Let us briefly discuss three protocols with explicit activation.

- *Synchronous activation - Synchronous reconfiguration* (SASR)
  In this case, each call to the `check_load()` primitive interrupts the execution of the parallel application and activates the load balancing support. The information about current computational power is then evaluated and sent to the reconfiguration master. All the internal processes wait until the decision phase is completed. This choice has many advantages: the protocol is easy to implement, there is maximum safety because the entire load balancing process is carried out while the internal processes are blocked, and the information about the platform load is the most updated possible. However, the drawback of this protocol is that every time the reconfiguration is not necessary, the `check_load()` call represents an unuseful barrier. For this reason, it can be

adopted for highly-coupled applications and/or when the probability of reconfiguration is high.

- *Asynchronous activation - Synchronous reconfiguration* (AASR)
  A way for mitigating the overhead of SASR is to synchronize the internal processes only when necessary. To this purpose, we adopt a different protocol that is based on a couple of `check_load(1)` and `check_load(2)` calls. The former call activates on each node i an independent evaluation of the capacity $c_i(t)$ which is then sent to the reconfiguration master. While the internal processes continue their operations, the master decides about the convenience of reconfiguration. The best point for calling `check_load(2)` depends on the following trade-off: it should not occur too late in order to avoid obsolescence of load parameters, and not too soon to guarantee that almost all of the nodes have sent their load parameters and the decision is completed with high probability. At the occurrence of the `check_load(2)` call, each node waits for the message about the reconfiguration. However, this call causes a global barrier only when the master decides that the reconfiguration is necessary.
  This protocol limits the overheads to the necessary points and guarantees the same safety of SASR because the reconfiguration is carried out while the internal processes are blocked. On the other hand, AASR does not work on the same precise information as the SASR because the load parameters do not exactly refer to the same instant. We mitigate the effects due to this problem by using higher thresholds that take into account the probability of some error about the node capacity estimation.

- *Asynchronous activation - Asynchronous reconfiguration* (AAAR)
  The AASR protocol aims at avoiding any interference between the support and internal processes, thereby allowing asynchronous activations and data reconfigurations. However, we discarded this protocol because of the serious risks of inconsistency that may affect an SPMD algorithm with explicit communications running on our support. The main reason is that most parallel instructions of these programs work on the basis of a data partition. Since any data reconfiguration modifies this representation, it is very risky and, most of the time, impossible to allow asynchronous data migrations. Additional details about the reconfiguration phase will clarify this issue in Section 5.2.2.

Therefore, even if the synchronous reconfiguration protocols do not seem optimal from the efficiency point of view, they augment the simplicity and correctness of the reconfiguration strategy. Since all processes are blocked in the same execution point, data migrations can be easily carried out in a distributed and transparent way.

5.2 Implementation issues

The explicit management of the load balancing phases makes distributed parallel programming very difficult. Hence, it was our main intention to keep this framework as hidden as possible from the programmer. In this section, we outline the guidelines adopted for designing a transparent data migration support which is explicitly activated by the `check_load()` primitive.

5.2.1 The `check_load()` function

This function represents the only explicit interface between the load balancing support and the application. It can be called at any point of the parallel program which is considered *safe* by the programmer. This task is not as complicated as it seems, because we are in the context of SPMD algorithms. The basic rule is to avoid `check_load()` calls between `send()` and `receive()` primitives, and in branches of the algorithm that could not be executed by some node. However, since no control is carried out by the compiler, the entire responsibility for the `check_load()` insertion is left to the programmer. This function has seven parameters: *activation, interval, load, decision, threshold, grain, complexity.*

Let us first focus on the activation and interval parameters that concern the interface between the application and load balancing support. There are three basic ways of using the `check_load()` primitive that depend on the activation protocol, and the number of activations that the programmer wants in a main iteration. We distinguish the following instances: a `check_load(0,0)` denotes one *synchronous* activation for each call; a `check_load(0,`i`)` denotes one *synchronous* activation every i main iteration steps; a couple of `check_load(1,`i`)` and `check_load(2,`i`)` denotes one *asynchronous* activation every i main iteration steps. It is to be noted that the AASR protocol does not admit more than one activation during a main iteration step. If for certain algorithms it is preferable to execute more than one iterations before waiting for the decision, the programmer has to use values greater than two in the first parameter of the second `check_load()` call.

The parameters of the `check_load()` function are as follows,

Activation: For the SASR protocol, use 0. For the AASR protocol, use 1 in the activation call, 2 for waiting the decision from the reconfiguration master in the same iteration step, and $k > 2$ for waiting the decision after $k - 2$ iteration steps.

Interval: This parameter denotes the frequency of activation of the load balancing support. Any value greater than 0 is expressed as number of main iterations, while the 0 value means that each `check_load()` call causes the activation of the framework. If the programmer wants to activate the support more times during the same iteration, he has to use more `check_load(0,0)` calls in the same program. These multiple calls are allowed for the SASR protocol only.

Load: The two available active methods are *micro-benchmark* and *Unix system call.*

Decision: The available decision policies are SAI, SMI, NINC, NINP (see Section 4.2).

Threshold: This parameter depends on the adopted decision policy. We use a real number, in which the value before the point denotes the chosen λ, and the value after the point corresponds to ϑ. For example, a 35.60 parameter denotes $\lambda = 0.35$ and $\vartheta = 0.6$. A null value in either camp is adopted when a policy requires a single threshold or to specify the default parameter.

Grain: This parameter represents the basic grain that can be redistributed. The programmer can choose among *point, row, column,* taking into account the adopted data distribution. As example, for 1-dimensional decompositions, the basic grain has to be a multiple of one row/column, while a 2-dimensional decomposition allows even a single point as basic grain.

Complexity: This value describes the average computational complexity that the application has on its data domain for each iteration step, and is crucial to determine the right amount of data that has to be transmitted in case of reconfiguration (see Section 4.2).

5.2.2 Data management

The data distribution choice is the first step in the implementation of any SPMD program. It determines the mapping of data entries onto nodes, and strongly influences load balancing because in SPMD computations the internal workload is usually proportional to the data assigned to each node. In the instance of regular problems solved onto homogeneous platforms, the choice of the best data distribution is not a particularly difficult task because all the nodes are assumed to have identical and static power. On the other hand, the quality of data distribution on heterogeneous and variable platforms, even for regular algorithms, depends on dynamic factors that cannot be anticipated during implementation. Since the program should potentially run for several kinds of data distributions, it should be *decomposition-independent.* For this reason our load balancing support is part of a wider framework, called DAME, that provides the programmer with the tools for adopting the SPMD paradigm on a distributed system as on a parallel platform. Even if a detailed description of DAME is out of the scope of this paper, we give some details about the *adaptive data distribution* support (ADD) because it is related to the dynamic load balancer. For more details, you can refer to [6].

- At *implementation-time*, ADD allows the programmer to choose the regular data distribution (*data partition image*) that is most appropriate for the application without caring about the platform irregularities. As common rule, many studies have evidenced that, for distributed parallel computing, 1-dimensional decompositions work better than 2-dimensional decompositions. Moreover, a block distribution is usually suitable to regular computations, while a fine-grain scatter distribution is preferable for irregular computations.
  In addition, ADD provides the programmer with a set of *data inquiry primitives* (such as data owner identifiers, local data extractors, index conversions, local loop ranges) that have to be used in any operation which is related to the *data partition image.*

- At *load-time*, ADD achieves initial load balancing by automatically determining the heterogeneous data distribution (*actual data partition*) which best adapts itself to the computational status of the platform. Figure 1.a and 1.b show a 1-dimensional and 2-dimensional block decomposition of a 2-dimensional data structure on a computing platform consisting of six heterogeneous nodes having relative capacities equal to $\chi(t_0) = (0.19, 0.09, 0.17, 0.15, 0.29, 0.11)$.

- At *run-time*, ADD translates the data inquiry primitives referring to the *data partition image* into accesses to the *actual data partition.* This latter, in fact, can be dynamically modified by the load balancer support.

When the master decides that the initial data partition has to be reconfigured, both the load balancer and ADD are involved. On the basis of the results of the decision policy, the load balancer carries out data migrations that modifies the *actual data partition* only. The application is indirectly informed about the new

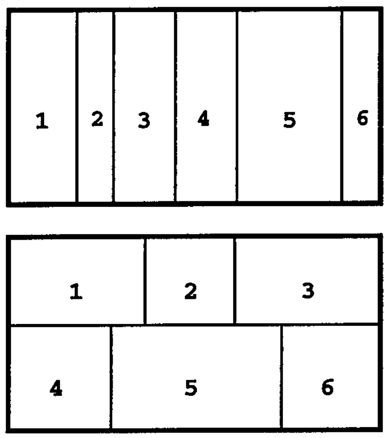


Figure 1: *Actual data partition* for 1D and 2D block decomposition of a 2D data domain.

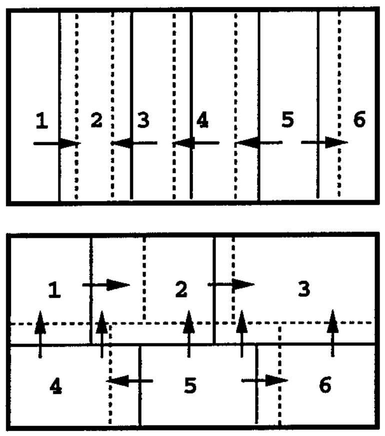


Figure 2: *Actual data partition* after a data reconfiguration. Dotted lines denote subdomain boundaries before reconfiguration.

data distribution through the results of the data inquiry primitives. In fact, they are used at implementation time with respect to the *data partition image* which is not subject to dynamic modifications, but refer to the *actual data partition* when they are called at run-time.

All the data structures that describe the *actual data partition* have to be updated before allowing ADD to access them. This inter-dependence prevents us from using asynchronous reconfiguration protocols such as AAAR. When the master decides about the reconfiguration, each internal process has to be blocked. In such a way, the master process can inform each node about the data entries that have to be transmitted and received. Data migrations occur in a distributed way among the nodes that own parts of neighbor data domain. If a 1-dimensional data distribution is adopted, each node may communicate with the two neighbor nodes. Figure 2.a shows the data migration occurring in the same platform of Figures 1 under the hypothesis that the relative capacities become $\chi(\bar{t}) = (0.11, 0.18, 0.21, 0.18, 0.14, 0.18)$. If a 2-dimensional data distribution is adopted and the basic grain is a point, data exchange occurs first horizontally and then, if necessary, vertically (Figure 2.b). If one excludes the worst cases, that are very rare, the total number of `send()` for a node usually does not overcome six and is much less in average.

6 Experimental results

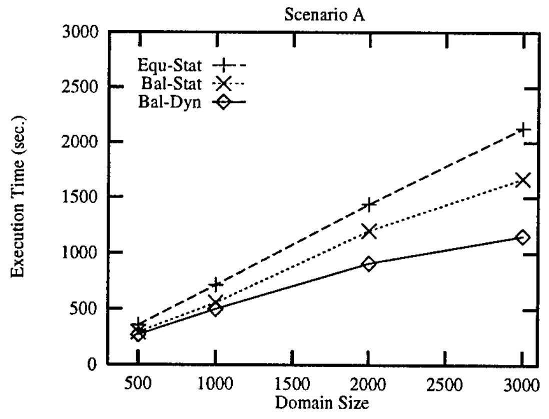
We carried out a set of experiments to measure the performance of our load balancing framework when applied to SPMD applications with explicit communications. The focus is mainly on the decision policies. The platform consists of six heterogeneous workstations which are connected through an Ethernet-based network. The initial relative capacities are given by the following vector $\chi(t_0) = (0.19, 0.09, 0.17, 0.15, 0.29, 0.11)$.

Experiments were performed during night hours when we had 'dedicated' workstations and we could assume almost exclusive access to the network. In order to emulate external workload, we loaded some node with special processes. Since even during night hours a distributed system cannot be considered a stable platform, we reduced the variability of the execution times by carrying out six repeated runs for each experiment. The results shown in this section are the average measurements of these six runs. We implemented two popular SPMD algorithms in C using PVM 3.3 primitives and the library provided by DAME:

1) *LU factorization.* This algorithm contains multicast communications and is partly irregular in the sense that, at each iteration step, it works on a smaller data domain. We consider a domain consisting of a square dense matrix that is row partitioned in a cyclic way. The basic redistribution grain is fixed to a row, while the complexity coefficient C is equal to 2.

2) *Heat equation solver.* This algorithm evaluates the temperature of each grid point by means of a suitable linear combination of the temperature of its adjacent points at the previous time step according to the finite difference method. It is a regular algorithm that contains only near-neighbor communications. We consider a 2-dimensional domain that is partitioned in blocks. Even for this algorithm, the complexity coefficient C is equal to 2.

The system variability is reflected through a set of four scenarios. In *scenario A*, two nodes are loaded by external processes that have an interval of activation and disactivation equal to 300 seconds (*average variability*). In *scenario B*, the external workload is on five nodes with same average interval of activation and disactivation as *scenario A*. In *scenario C*, the external workload is on half nodes with activation/disactivation interval equal to 400 seconds (*low variability*). In *scenario D*, the external workload is on half nodes with interval equal to 120 seconds (*high variability*).



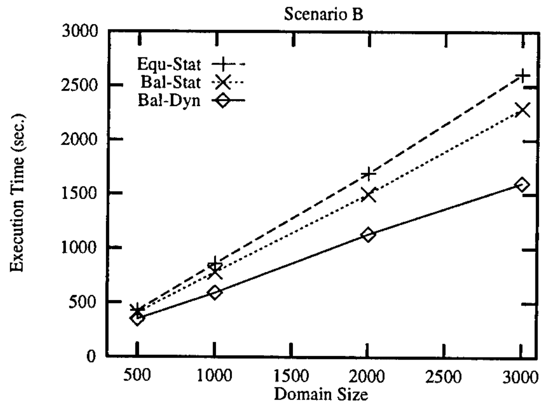
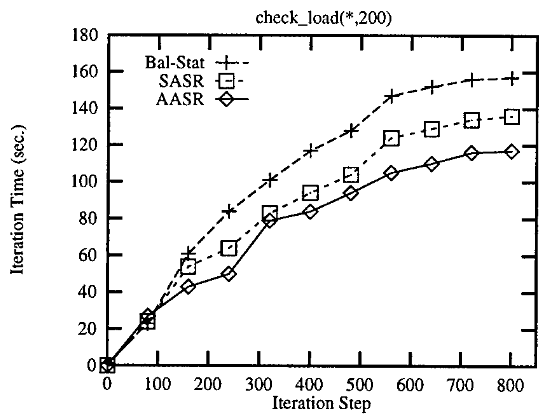


Figure 3: Execution times of the parallel solution of the 2-dimensional heat equation after 100 iteration steps with variable domain size. The platform consists of 6 machines that are subject to workload variations according to scenario A and scenario B.

The first set of experiments aims at evaluating the advantages of our load balancing support. To this purpose, we compared the performance of the heat equation algorithm with *actual data partition* equal to *data partition image* and no `check_load()` call (*equ-stat*), the same algorithm with *actual data partition* proportional to initial node capacities and no `check_load()` call (*bal-stat*), and the algorithm using dynamic *actual data partition* and `check_load(0,7,bench,SMI,.45)` calls (*bal-dyn*). Figures 3 show the execution times (in seconds) of one hundred iteration steps of the heat equation solver for a rectangular data domain in which one dimension is fixed to 300 and the other is variable (see values on the ascissa). In particular, Figure



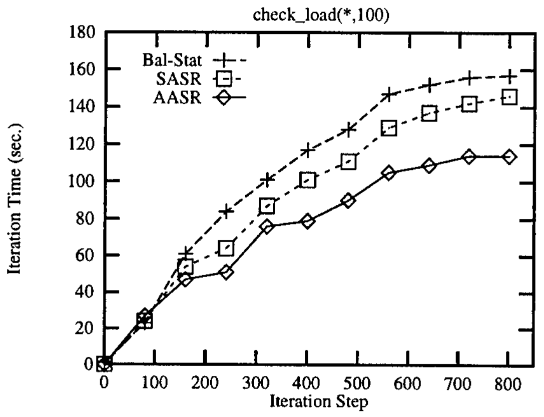
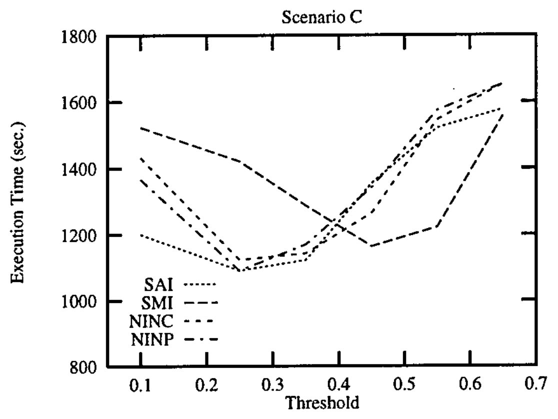


Figure 4: Execution times (cumulative) of LU factorization of a 800 × 800 dense matrix solved on four workstations without dynamic balancing (no balan), with synchronously (SASR) and asynchronously activated load balancing (AASR). Each node of the platform is subject to very frequent workload variations (every 40 secs). The first set of runs adopts `check_load(*,200)` while the second uses `check_load(*,100)`.

3.a refers to the scenario A, while Figure 3.b to the scenario B. We can see that our support guarantees a better efficiency for any dimension of the data domain, even when it adopts a simple policy such as the SMI, and the protocol SASR which is affected by many overheads due to synchronous activations and micro-benchmarks.

The second set of experiments compares the synchronous with the asynchronous activation protocol. Figures 4 show the performance of these policies running on a platform subject to frequent workload variations for different activation intervals. As expected, AASR behaves better than SASR in both cases. The difference in favor of AASR is even larger if we consider parallel applications that have less intrinsic synchronizations than LU factorization algorithm.

In the third set of experiments, we test the sensitiv-



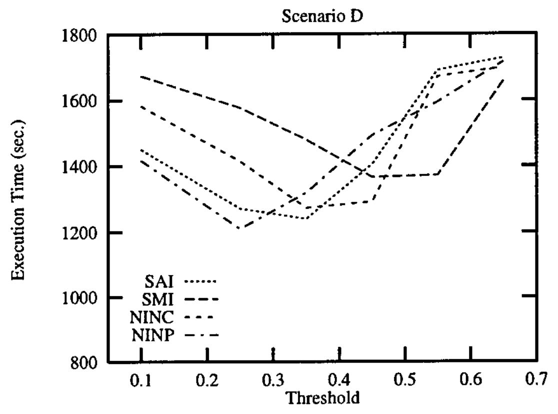


Figure 5: Execution times of 100 iterations of the heat equation solver on a 300 × 3000 domain by using dynamic load balancing every 7 iteration steps with different decision policies and variable thresholds.

ity of the decision policies to the threshold ϑ. This is a key parameter for the performance of the load balancer because we have to solve the trade-off between low thresholds that allow a more balanced workload at the price of higher overheads, and high thresholds that reduce overheads, but have major risks of long imbalanced executions. Since the choice of the best threshold depends on many variables, among which the application characteristics and platform conditions are the most important, a global solution is out of the scope of this paper. For this reason, we carried out only an empirical analysis within our test algorithms and scenarios. Figures 5 show the execution times of 100 iterations of the heat equation solver on a 300 × 3000 domain running under scenarios C and D as a function of the threshold. It is to be noted that each decision policy performs best in correspondence of a rather large interval of threshold values. Hence, any value in this interval can be considered acceptable.

The last set of experiments aims at comparing the decision policies applied to the same heat equation solver used in the third set of experiments under the four scenarios A-D. The `check_load()` adopts the AASR protocol with interval parameter set to 7, while the thresholds for each decision strategy are set to the best values found in the previous experiments. In particular, the Figures 6-9 show the execution times required to complete each iteration step by the slowest node of the platform, without considering the costs due to the `check_load()` calls. Each graph reports the execution times as sum of application time and overheads due to the load balancing support. The peaks in each curve denote an imbalanced situation created by the activation of synthetic workloads. All the experiments are subject to the same external workloads, however they can occur at different iteration steps depending on the speed of the processes. In faster (that is, more balanced) executions, the peaks that denote external workloads appear in correspondence of higher iteration steps.

Figure 6 compares the strategies SAI, SMI and NINC under the scenario A. SAI and SMI reduce the global execution time of more than 20% with respect to the algorithm without `check_load()`, while NINC (and NUNC that for these experiments behaves very similarly to NINC) provides more limited improvements. In particular, SMI recognizes any consistent power variation and reacts immediately (all the peaks but one are smaller than in other strategies). SAI is less reactive than SMI because it considers the global imbalance: it does not react to single variations, while it soon reconfigures when the entire system is imbalanced. On the other hand, NINC, that in our platform requires at least two nodes to be imbalanced, is insensitive to variations occurring in one node only.

Figure 7 compares the same decision strategies under the scenario B that provides a more imbalanced platform than A. In this instance, the strategies SAI and NINC perform best, while SMI is very poor. The problems of this strategy are due to the fact that it always reconfigures, thus causing more overheads (344 seconds compared to 206 and 256 of the other two strategies) without reducing the execution time of the algorithm (compare 1543 seconds to 1470 and 1408). This demonstrates that, in highly variable platforms, it is not convenient to always reconfigure, because the new *actual data partition* may soon be invalidated. The curves SAI and NINC have less skews and perform generally better than SMI. However, the long peaks around iterations 45-65 denote their inability to react to a serious imbalance which is due, with high probability, to only one node.

Figures 8 and 9 evaluate the trade-off between a strategy using only the last information and a policy using even past information about the node capacities. The comparison between NINC and NINP is carried out on a low and highly variable platform (that is, scenario C and scenario D, respectively). We can anticipate that in both instances, the policy using past information does not achieve better results than the strategy without memory. In particular, in the scenario C, NINC suddenly reacts to each variation, while NINP usually reacts at the successive `check_load()` call. Therefore, NINP causes more iterations to be executed in an imbalanced condition. Figure 8 shows that the NINP peaks are always longer than the NINC peaks, while in a low variable scenario there is not a great difference between NINC and NINP overheads. The results in case of highly imbalanced platform (Figure 9) have a more difficult interpretation. NINC always reconfigures but, being a myopic policy, it often has to reconfigure again at the successive `check_load()` call. This causes a greater overhead for NINC than for NINP (266 *vs.* 246 seconds) which is due to the higher number of data migration for the former policy. NINP, being sensitive to long fluctuations only, performs well when it avoids following any brief variation of the platform, and performs poorly when it reacts too late to an imbalanced situation.

Even if it is difficult to deduct any global conclusion, because there are many other system and application parameters that could be considered, our experiments indicate the following.

- AASR is usually more advantageous than SASR.
- SMI and NINC with low thresholds are preferable when most of the machines are stable and just one-two workstations are subject to high load variations (for example, some machines of the pool are used occasionally for external jobs). In this case, NINP performs poorly, while SAI is acceptable even if it is sensible to overall variations only and we are considering load variations on a small subset of nodes.
- The global policies such as SAI and NINP are preferable when the platform is highly unstable. Moreover, SAI seems usually more stable than NINP, while NINC and SMI do not improve much the algorithm performance.
- Using past information is rarely convenient. Typically, NINC does not perform worse than NINP and it is often better. We observed the NINP policy to give better results than NINC only in a (not reported) scenario characterized by long periods of load variations and some critical situations of short duration, to which it was usually more convenient not to react.

7 Conclusions

The problem of load balancing distributed parallel computations touches on many theoretical and practical issues. In this paper, we have presented some methods that work for regular SPMD algorithms containing near-neighbor and/or multicast communications. Even if our framework does not implement all the possible strategies, it achieves two important results: it shows the feasibility of dynamically adapting data distribution for this wide class of parallel applications, it hides from the programmer the entire reconfiguration process but the activation phase. In addition, the experiments demonstrate this framework to be an efficacious support for balancing SPMD computations and to maintain the efficiency even when the platform is subject to highly dynamic variations. In particular, our results show that the asynchronous activation protocol is usually preferable to the synchronous strategy, while an asynchronous reconfiguration is unfeasible for our framework. No one decision policy proved best for all the applications and scenarios used in our experiments, however the SAI, that looks at the global system imbalance, seems to be the most stable.

This paper leaves open two interesting problems, such as the choice of the best threshold value and the optimal checkpoint frequency, that are currently under study. We intend to provide the programmer with a support that, on the basis of the chosen decision policy, class of applications (independent tasks, moderate synchronous tasks, highly synchronous tasks) and status of the platform (quiet, moderately variable, highly variable), sets autonomously the *threshold* and *interval* parameters of the `check_load()` function.

Acknowledgements

The authors thank the anonymous reviewers for their comments that helped improve the quality of this presentation.

References

[1] F. Bonomi, and A. Kumar, "Adaptive optimal load balancing in a nonhomogeneous multiserver system with a central job scheduler", *IEEE Trans. on Computers*, v. 39, n. 10, Oct. 1990, pp. 1232-1250.

[2] N. Carriero, and D. Kaminsky, "Adaptive parallelism and Piranha", *IEEE Computers*, v. 28, n. 1, Jan. 1995, pp. 40-49.

[3] J. Casas, R. Konuru, S.W. Otto, R. Prouty, and J. Walpole, "Adaptive load migration systems for PVM", *Proc. of Supercomputing '94*, Washington, DC, Nov. 1994, pp. 390-399.

[4] J. Casas, D.L. Clark, R. Konuru, S.W. Otto, R.M. Prouty, and J. Walpole, "MPVM: A migration transparent version of PVM", *Usenix Computing Systems*, v. 8, n. 2, Spring 1995, pp. 171-216.

[5] M. Cermele, and M. Colajanni, "Supporting irregular data distribution for heterogeneous clusters", *Proc. of 9th Int. Conf. on Par. and Distr. Comp. Sys.*, Dijon, France, Sept. 1996.

[6] M. Cermele, and M. Colajanni, "Nonuniform and dynamic domain decomposition for hypercomputing", to appear in *Parallel Computing,* 1997; (available as Tech. Rep. RI.95.19 via anonymous ftp://ftp.info.utovrm.it/pub/projects/Ri95-15.ps).

[7] A.N. Choudhary, B. Narahari, and R. Krishnamurti, "An efficient heuristic scheme for dynamic remapping of parallel computations", *Parallel Computing,* v. 19, 1993, pp. 621-632.

[8] M. Colajanni, and M. Cermele, "DAME: An environment for preserving efficiency of data parallel computations on distributed systems", *IEEE Concurrency,* 1997.

[9] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM: Parallel Virtual Machine – A Users's Guide and Tutorial for Networked Parallel Computing.* Cambridge: MIT Press, 1994.

[10] M. Hamdi, and C.-K. Lee, "Dynamic load balancing of data parallel applications on a distributed network", *Proc. of 1995 Int. Conf. on Supercomputing,* Barcelona, July 1995, pp. 170-179.

[11] R. von Hanxleden, and L.R. Scott, "Load balancing on message passing architectures", *Journal of Parallel and Distributed Computing,* v. 13, 1991, pp. 312-324.

[12] T. Kunz, "The influence of different workload descriptions on a heuristic load balancing scheme", *IEEE Trans. on Software Engineering,* v. 17, n. 7, July 1991, pp. 725-730.

[13] N. Nedeijkovic, and M.J. Quinn, "Data-parallel programming on a network of heterogeneous workstations", *Concurrency: Practice and Experience,* v. 5, n. 4, June 1993, pp. 257-268.

[14] D.M. Nicol, and J.H. Saltz, "Dynamic remapping of parallel computations with varying resource demands", *IEEE Trans. on Computers,* v. 37, n. 9, Sept. 1988, pp. 1077-1087.

[15] R. Ponnusamy, J. Saltz, A. Choudary, Y.-S. Hwang, and G. Fox, "Runtime support and compilation methods for user-specified data distributions", *IEEE Trans. on Parallel and Distributed Systems,* v. 6, n. 8, Aug. 1995, pp. 815-831.

[16] Schnekenburger, and M. Huber, "Heterogeneous partitioning in a workstation network", *Proc. of 1994 Heterogeneous Computing Workshop,* 1994, pp. 72-77.

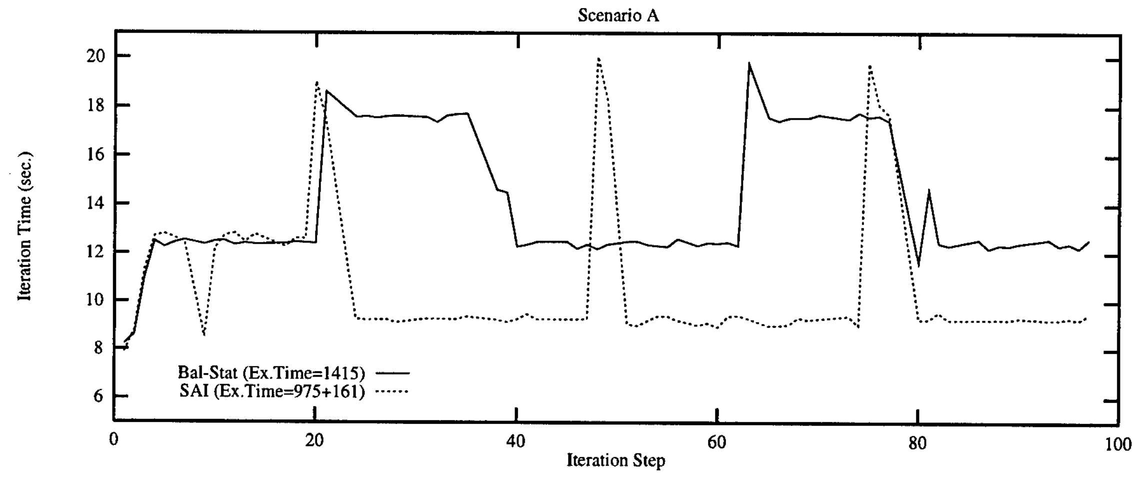
[17] M.H. Willebeek-LeMair, and A.P. Reeves, "Strategies for dynamic load balancing on highly parallel computers", *IEEE Trans. on Parallel and Distributed Systems,* v. 4, n. 9, Sept. 1993, pp. 979-993.

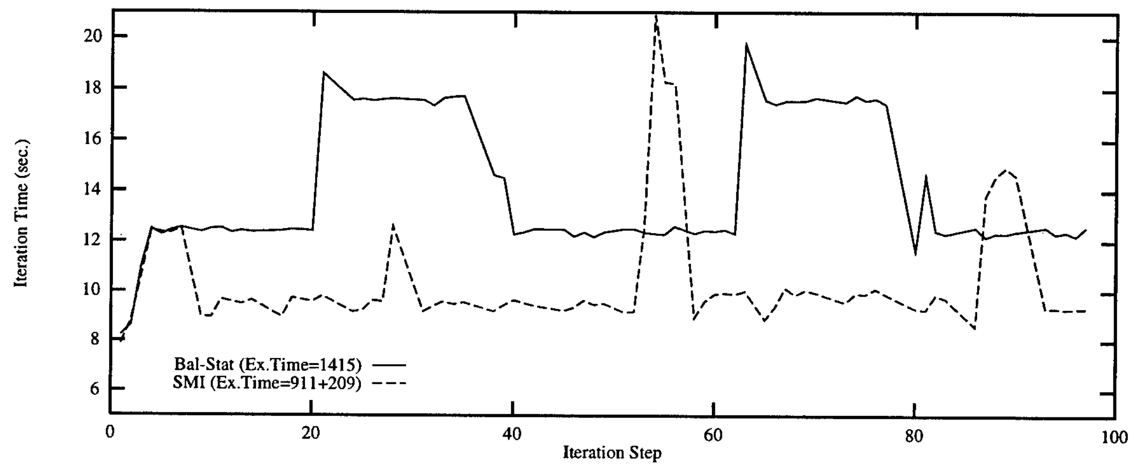
[18] S. Zhou, X. Zheng, J. Wang, and P. Delisle, "Utopia: a load sharing facility for large, heterogeneous distributed computer systems", *Software – Practice and Experience,* v. 23, n. 12, Dec. 1993, pp. 1305-1336.

Michele Cermele is pursuing its PhD in computer science at the University of Roma "Tor Vergata", Italy. His research interests are in the area of parallel processing, network-based computing, and performance evaluation methods for computer systems and networks. He received the Laurea degree in computer engineering *summa cum laude* from the University of Roma "Tor Vergata", in 1994. From January 1997, he is a *visiting scholar* at the Department of Electrical and Computer Engineering of Duke University, NC.

Michele Colajanni is a Researcher at the Department of Computer Science, Systems and Management of the University of Roma "Tor Vergata", Italy. His research interests include adaptive distributed computing, parallel algorithms, and performance modeling and analysis. He received the Laurea degree in computer science from the University of Pisa, in 1987, and the PhD in computer engineering from the University of Roma "Tor Vergata", in 1991. In 1992 and 1993, he received two grants from the Italian National Research Council (CNR) for conducting researches on high performance computing. In 1996, he was a *visiting scientist* at the IBM T.J. Watson Research Center in Yorktown Heights, NY. Dr Colajanni is a member of the Association for Computing Machinery, and IEEE Computer Society.

Giovanni Necci received the Laurea degree in Computer Engineering from the University of Roma "Tor Vergata", in 1996. He is currently a candidate to the PhD course in Computer Science at the University of Roma "Tor Vergata". His research interests are in load balancing for parallel and distributed systems, and stochastic methods for performance analysis.





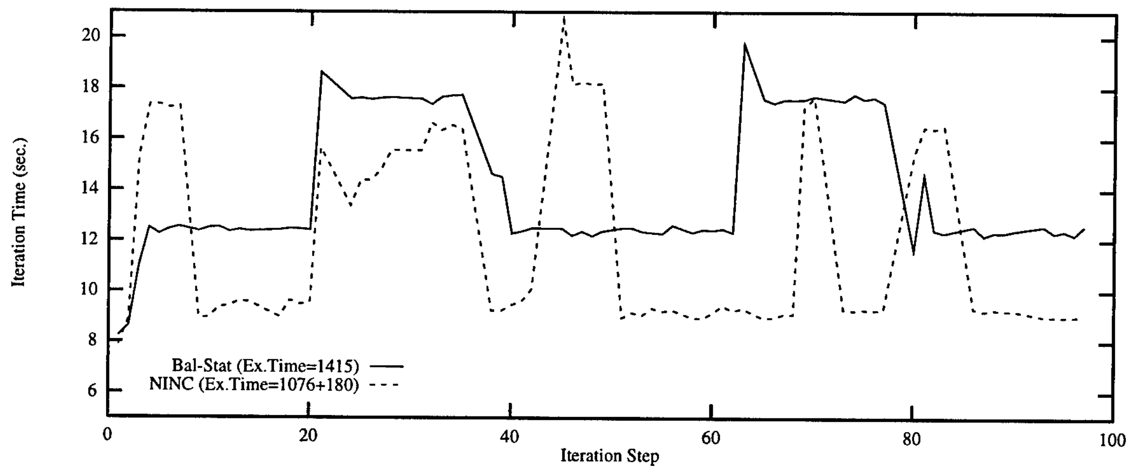


Figure 6: Iteration times of the heat equation solver on a 300 × 3000 domain without dynamic balancing and with `check_load(*,7)` calls using three decision policies which are based on current load information. The platform is subject to workload variations according to scenario A.

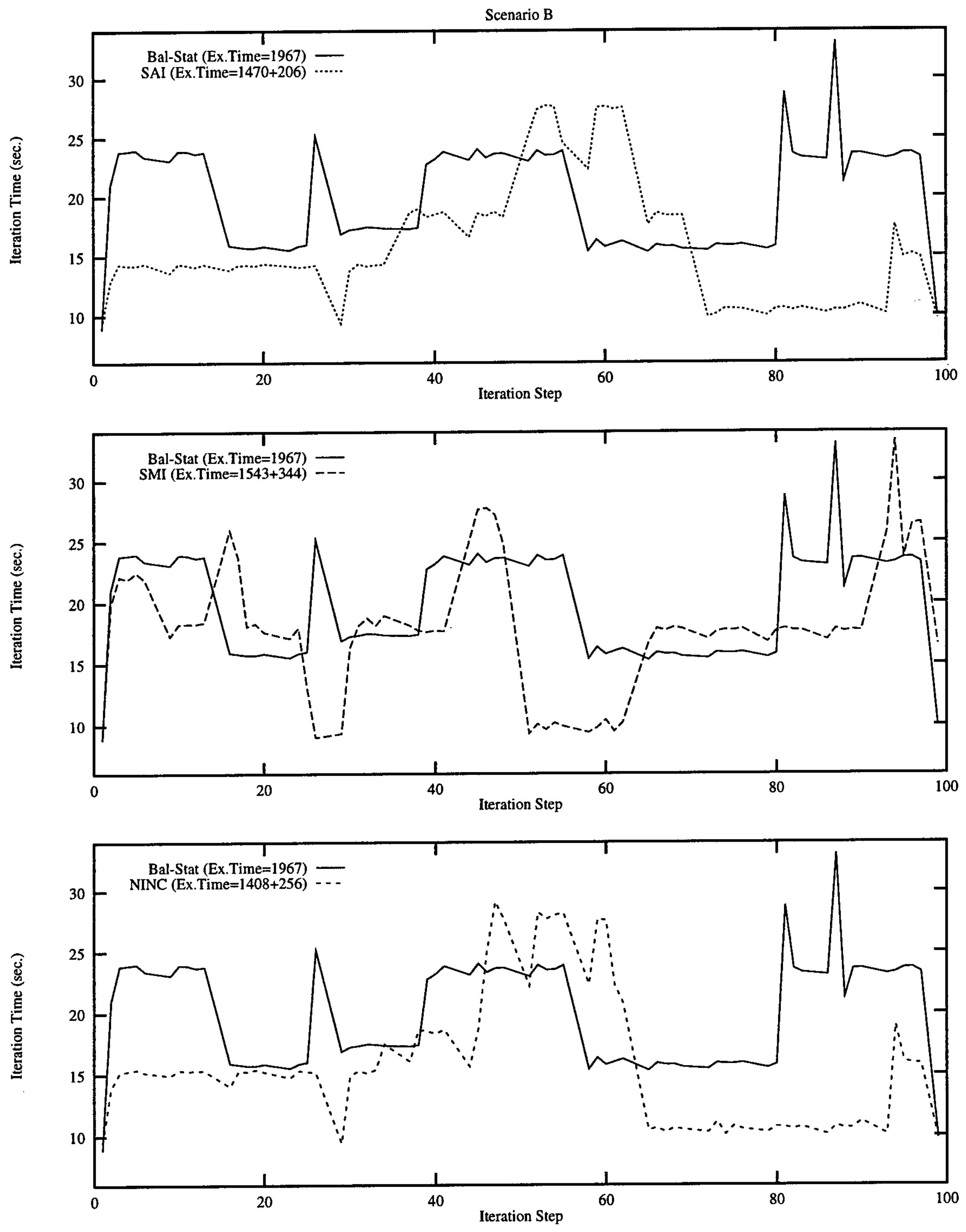


Figure 7: Iteration times of the heat equation solver without dynamic balancing and with `check_load(*,7)` calls using three decision policies which are based on current load. The platform is subject to workload variations according to scenario B.

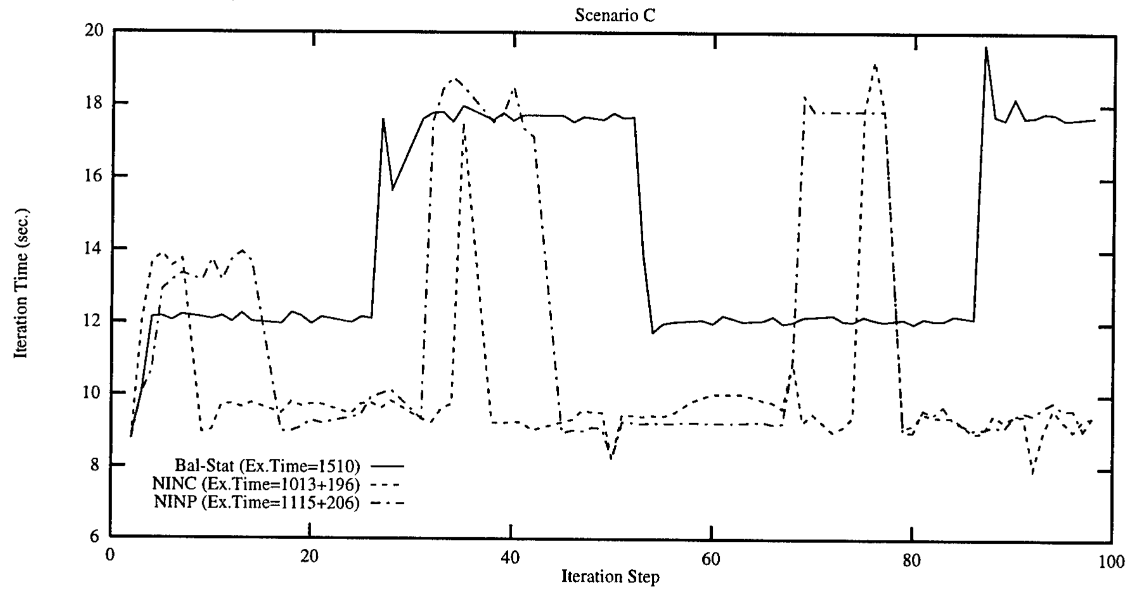


Figure 8: Iteration times of the heat equation solver without dynamic balancing (no bal), with a decision policy which is based on current load information (NINC), and a decision policy which adopts past and current information (NINP). The platform is subject to workload variations according to scenario C.

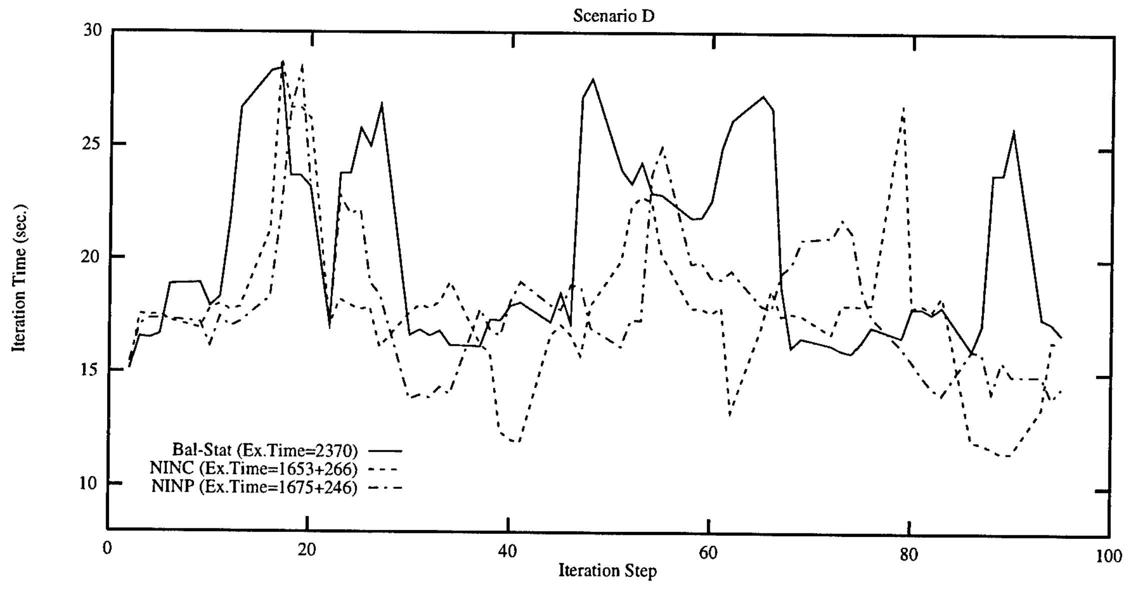


Figure 9: Iteration times of the heat equation solver without dynamic balancing (no bal), with a decision policy which is based on current load information (NINC), and a decision policy which adopts past and current information (NINP). The platform is subject to workload variations according to scenario D.

The MOL Project: An Open, Extensible Metacomputer*

A. Reinefeld, R. Baraglia†, T. Decker, J. Gehring,
D. Laforenza†, F. Ramme, T. Römke, J. Simon

PC^2 – Paderborn Center for Parallel Computing, Germany
† CNUCE – Institute of the Italian National Research Council

Abstract

Distributed high-performance computing—so-called metacomputing—refers to the coordinated use of a pool of geographically distributed high-performance computers. The user's view of an ideal metacomputer is that of a powerful monolithic virtual machine. The implementor's view, on the other hand, is that of a variety of interacting services implemented in a scalable and extensible manner.

In this paper, we present MOL, the Metacomputer Online environment. In contrast to other metcomputing environments, MOL is not based on specific programming models or tools. It has rather been designed as an open, extensible software system comprising a variety of software modules, each of them specialized in serving one specific task such as resource scheduling, job control, task communication, task migration, user interface, and much more. All of these modules exist and are working. The main challenge in the design of MOL lies in the specification of suitable, generic interfaces for the effective interaction between the modules.

1 Metacomputing

"Eventually, users will be unaware they are using any computer but the one on their desk, because it will have the capabilities to reach out across the national network and obtain whatever computational resources are necessary" [41]. This vision, published by Larry Smarr and Charles Catlett in their seminal CACM article on metacomputing, sets high expectations: *"The metacomputer is a network of heterogeneous, computational resources linked by software in such a way that they can be used as easily as a personal computer."*

The advantages of metacomputing are obvious: Metacomputers provide true supercomputing power at little extra cost, they allow better utilization of the available high-performance computers, and they can be flexibly upgraded to include the latest technology. It seems, however, that up to now no system has been built that rightfully deserves the name 'metacomputer' in the above sense. From the user's point of view, the main obstacles are seen at the system software level, where non-standardized resource access environments and incompatible programming models make it difficult for non-experts to exploit the available heterogeneous systems. Many obstacles in the cooperative use of distributed computing systems can be overcome by providing a homogeneous, user-friendly access to a reliable and secure virtual metacomputing environment that is used in a similar way as a conventional monolithic computer today.

Some of these issues are addressed by "Metacomputer Online (MOL)", an initiative that has been founded with the goal to design and implement the nucleus of a practical distributed metacomputer. MOL is part of the Northrhine-Westphalian Metacomputer Inititiative that has been established in 1995, and it is embedded in several other European initiatives [31].

The MOL group has implemented a first, incomplete 'condensation point' of a practical metacomputer, which is now being extended and improved. Clearly, we could not tackle all relevant obstacles at the same time. We initially concentrated on the following issues that are deemed most important in the design of a first prototype:

- provision of a generic, user-friendly resource access interface,
- support of interoperability of existing codes using different message passing standards,

*This work is partly supported by the EU ESPRIT Long Term Research Project 20244 (ALCOM-IT) and by the Northrhine Westphalian Initiative "Metacomputing: Distributed Supercomputing"

0-8186-7879-8/97 $10.00 © 1997 IEEE

- effective global scheduling of the subsystems for high throughput and reduced waiting times,
- support of concurrent use in interactive and batch mode,
- mechanisms for automatic remote source code compilation and transparent data distribution,
- automatic selection of adequate compute nodes from a pool of resources to be assigned to the tasks of a parallel application,
- dynamic re-placement of user tasks by means of performance prediction of the source code on the heterogeneous nodes,
- provision of data management libraries and programming frames to help non-expert users in program design and optimal system utilization.

Clearly, this list is not complete. Further items can and should be added for a full metacomputing environment. Depending on their attitude and current task, users might have different expectations on the services a metacomputer should provide. As a consequence, a metacomputer cannot be regarded as a closed entity, but it is rather a highly dynamical software system that needs continuous adaptation to the current user demands.

The MOL project aims at integrating existing software modules in an open, extensible environment. Our current implementation supports PVM, MPI and PARIX applications running on LAN- or WAN-connected high-performance computers, such as Parsytec GC, Intel Paragon, IBM SP2, and UNIX workstation clusters.

This paper presents the current status of MOL. It gives an overview about the system architecture and it illustrates how the separate modules interact with each other. Section 2 reviews related work which influenced the design of MOL or which can be integrated into MOL at a later time. Section 3 gives a conceptual view and discusses the general MOL architecture. In Section 4, we elaborate on the interface design and show how the MOL components interact with each other. Section 5 to 7 describe the MOL components in more detail, and Section 8 gives a brief summary and outlook.

2 Previous Work

In the past few years, several research projects have been initiated with the goal to design a metacomputer environment. Some of them follow the top-down approach, starting with a concrete metacomputing concept in mind. While this approach seems compelling, it usually results in a "closed world metacomputer" that provides a fixed set of services for a well-defined user community.

The second category of projects follow the bottom-up approach, initially focusing on some selected aspects which are subsequently extended towards a full metacomputing environment. In the following, we review some projects falling into this category.

Parallel programming models have been a popular starting point. Many research projects targeted at extending existing programming models towards a full metacomputing environment. One such example is the Local Area Multicomputer *LAM* developed at the Ohio Supercomputer Center [13]. LAM provides a system development and execution environment for heterogeneous networked computers based on the MPI standard. This has the advantage that LAM applications are source code portable to other MPI systems. The wide-area metacomputer manager *WAMM* developed at CNUCE, Italy, is a similar approach based on PVM [5, 6]. Here, the PVM programming environment has been extended by mechanisms for parallel task control, remote compilation and a graphical user interface. Later, WAMM has been integrated into MOL and extended to the support of other programming environments, as shown below.

Object oriented languages have also been proposed as a means to alleviate the difficulties of developing architecture independent parallel applications. *Charm* [26], *Trapper* [38], *Legion* and *Mentat* [25] for example, support the development of portable applications by object oriented parallel programming environments. In a similar approach, the *Dome* project [2] at CMU uses a C++ object library to facilitate parallel programming in a heterogeneous multi-user environment. Note, however, that such systems can hardly be used in an industrial setting, where large existing codes (usually written in Fortran) are to be executed on parallel environments.

Projects originating in the *management of workstation clusters* usually emphasize on topics such as resource management, task mapping, checkpointing and migration. Existing workstation cluster management systems like *Condor*, *Codine*, or *LSF* are adapted for managing large, geographically distributed 'flocks' of clusters. This approach is taken by the Iowa State University project *Batrun* [42], the Yale University *Piranha* project [14], and the Dutch *Polder* initiative [34], both emphasizing on the utilization of idle workstations for large-scale computing and high-throughput computing. Complex simulation applications, such as air pollution and laser atom simulations have been run in the *Nimrod* project [1] that supports multiple executions of the same sequential task with different parameter sets.

These projects could benefit by the use of special metacomputing schedulers, such as the Application-Level Scheduler *AppLeS* developed at the University of California in San Diego [9]. Here, each application has its own AppLeS scheduler to determine a performance-efficient schedule and to implement that schedule in coordination with the underlying local resource scheduler.

Networking is also an important issue, giving impetus to still another class of research projects. *I-WAY* [22], as an example, is a large-scale wide area computing testbed connecting several U.S. supercomputing sites with more than a hundred users. Aspects of security, usability and network protocol are among the primary research issues. Distributed resource brokerage is currently investigated in the follow-up *I-Soft* project. In a bilateral project, Sandia's massively parallel Intel Paragon is linked with the Paragons located at Oak Ridge National Laboratories using GigaNet ATM technology. While also termed a metacomputing environment, the consortium initially targets at running specific multi-site applications (e.g., climate model) in distributed mode.

The well-known Berkeley *NOW* project [33] emphasizes on using networks of workstations mainly for improving virtual memory and file system performance by using the aggregate main memory of the network as a giant cache. Moreover, highly available and scalable file storage is provided by using the redundant arrays of workstation disks, and—of course—the multiple CPUs are used for parallel computing.

3 MOL Architecture

We regard a metacomputer as a dynamic entity of interacting modules. Consequently, interface specifications play a central role in the MOL architecture, as illustrated in Figure 1. There exist three general module classes:

1. programming environments,
2. resource management & access systems,
3. supporting tools.

(1) Programming environments provide mechanisms for communicating between threads and they allow the use of specific system resources. MPI and

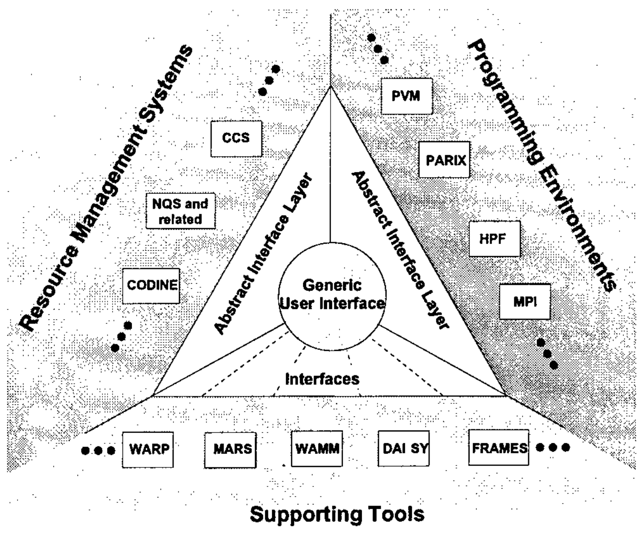


Figure 1: MOL architecture

PVM, for example, are popular programming environments supported by MOL. More important, MOL also supports the communication and process management of *heterogeneous* applications comprising software using different programming environments. The MOL library "PLUS" makes it possible, for example, to establish communication links between MPI- and PVM-code that are part of one large application. PLUS is almost transparent to the application, requiring only a few modifications in the source code (see Sec. 5).

(2) Resource management & access systems provide services for starting an application and for controlling its execution until completion. Currently there exists a large number of different resource management systems, each with its specific advantage [3]. Codine, for example, is well suited for managing (parallel) jobs on workstation clusters, while NQS is specialized on serving HPC systems in batch mode. Because the specific advantages of these systems supplement each other, we have developed an *abstract interface layer* with a high-level resource description language that provides a unified access to several (possibly overlapping) resource management systems.

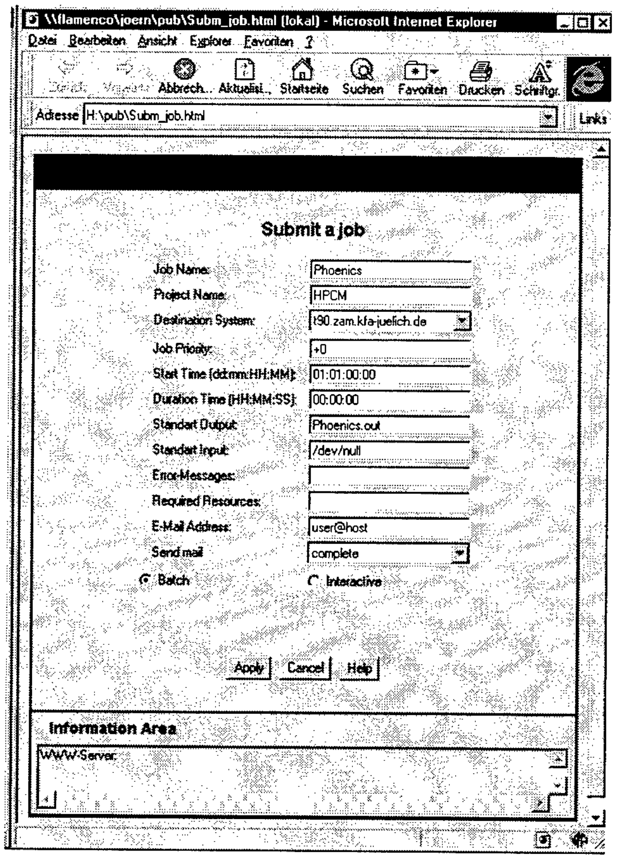
(3) Supporting tools provide optional services to metacomputer applications. Currently, the MOL toolset includes software modules for dynamic task migration (MARS), for easy program development by using programming templates (FRAMES), and also data libraries for virtual shared memory and load balancing (DAISY). The wide-area metacomputer manager WAMM plays a special role in MOL: On the one hand, it may be used just as a resource management system for assigning tasks to resources, and on the other hand,

it also provides automatic source code compilation and cleanup after task execution.

4 Interface Layers

As described, MOL facilitates the interaction between different resource management systems and different programming environments. However, this is not enough: A metacomputer application may also require some interaction between these two groups. As an example, consider a PVM application running on a workstation cluster that is about to spawn a process on a parallel computer. For this purpose, the application makes a call to the parallel system's management software in order to allocate the necessary resources. Likewise, a metacomputer application may need a mechanism to inform the communication environment about the temporary location of the participating processes.

In addition, tools are needed for efficient task mapping, load balancing, performance prediction, program development, etc. There is a wealth of supporting tools available to cover most aspects of metacomputing. However, as these tools are typically very complex, it is hard to adapt their interfaces for our specific needs. In the MOL framework shown in Figure 1, the supporting tools only need to interact with two abstract interface layers rather than with all systems below. Thus, new software must be integrated only once, and updates to new releases affect only the tool in question.


\\flamenco\joern\pub\Subm_job.html (lokal) - Microsoft Internet Explorer
Datei Bearbeiten Ansicht Explorer Favoriten ?
Abbrech... Aktualisi... Startseite Suchen Favoriten Drucken Schriftgr.
Adresse H:\pub\Subm_job.html
Links
Apply Cancel Help

Submit a job

Job Name:	Phoenics
Project Name:	HPCM
Destination System:	t90.zam.kfa-juelich.de
Job Priority:	+0
Start Time (dd:mm:HH:MM):	01:01:00:00
Duration Time (HH:MM:SS):	00:00:00
Standart Output:	Phoenics.out
Standart Input:	/dev/null
Error-Messages:	
Required Resources:	
E-Mail Address:	user@host
Send mail	complete

[✓ Batch] Interactive

Information Area

WWW-Server:

Figure 2: Prototype GUI of Paragon/Parsytec metacomputer

4.1 MOL User Interface

Metacomputing services will be typically accessed via Internet or Intranet world-wide-web browsers, such as the Netscape Navigator or the Microsoft Explorer. MOL provides a generic graphical user interface (GUI) based on HTML and Java for interactive use and for submitting batch jobs. At the top level, the GUI displays general information on status and availability of the system services. Users may select from a number of alternatives by push-down buttons. In doing so, context sensitive windows are opened to ask for context specific parameters required by the requested services. Figure 2 shows the MOL window used for submitting interactive or batch jobs.

This concept is called 'interface hosting': The generic interface acts as a host for the sub-interfaces of the underlying modules. Interface hosting gives the user control over the complete system without the need to change the environment when the user wants to deal with another feature or service of the metacomputer.

The different skills of users are met by a Java-based graphical interface and a shell-based command oriented interface. In addition, an API library allows applications to directly interact with the metacomputer services. All three interfaces,

- the graphical user interface (Java)
- the shell-based command line interface
- the API library

provide the same functions.

Figure 2 illustrates the graphical user interface used to submit a job to a Parsytec/GC and a Paragon located in Paderborn and Jülich. The two machines are installed 300 kilometers apart. They are interconnected via the 34Mbps German Research WAN. Note that the same interface is used for submitting batch jobs as well as for starting interactive sessions.

A later version of the interface will include a graph editor, allowing the user to describe more complex interconnection topologies and hierarchical structures.

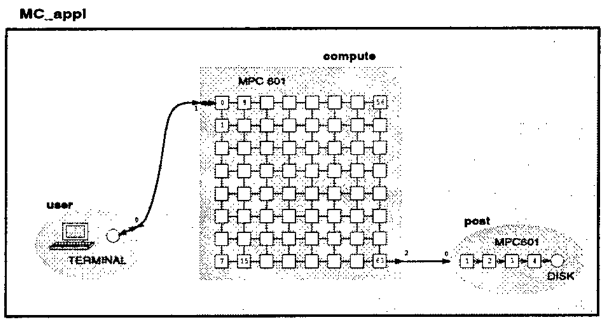


Figure 3: Simple heterogeneous computing example

When the actual system configuration has been determined by the resource scheduler, the resulting topology will be presented to the user by a graph-drawing tool [32].

4.2 A MOL User Session

In this section, we describe the specification and execution of a example user application that is executed in the MOL environment. Suppose that a large grid structured application shall be run on a massively parallel system with 64 PowerPC 601 nodes, each of them having at least 16 MB of main memory. As shown in Figure 3, the result of this computation shall be post-processed by another application with a pipeline structure. Depending on the available resources, the resource management system may decide to collapse the pipeline structure to be run on a single processor, or to run the grid and the pipeline on the same machine. For interactive control, the user terminal is linked to the first node of the grid part of the application. For the concrete resource description used to specify this metacomputer scenario see Figure 6 below.

Within MOL, such resource specifications are generated by tools that are integrated into the GUI. When submitting a resource request, the request data is translated into the internal format used by the abstract resource management (RM) interface layer. The result is then handed over to the scheduler and configurator. The RM interface layer schedules this request and chooses a time slot when all resources are available. It contacts the target management systems and takes care that all three programs will be started at the same time. Furthermore, it generates a description file informing the abstract programming environment interface where the programs will be run and which addresses to be used for establishing the first communication links. At program run time, the programming environment interface layer establishes the interconnection between the application domains and takes care of necessary data conversion.

Though this example is not yet completely realized, it gives an idea on how the modules would interact with each other by means of abstract layers and how the results are presented to the user by the generic interface.

5 Programming Environments Layer

In contrast to many other heterogeneous computing environments, MOL is not restricted to a specific programming environment. Applications may use any programming model supported by the underlying hardware platforms. This could be either a vendor supplied library or a standard programming model such as PVM and MPI.

However, many programming models are homogeneous, that is, applications can only be executed on the system architecture they have been compiled (linked) for. While there also exist some heterogeneous programming libaries, they usually incur significant performance losses due to the high level of abstraction. Portable environments are typically implemented on top of the vendor's libraries, making them an order of magnitude slower than the vendor supplied environments. Clearly, users are not willing to accept any slow-downs in their applications, just because the application might need some infrequent communication with some remote system. This problem is addressed by PLUS.

5.1 PLUS – A Linkage Between Programming Models

PLUS stands for *Programming Environment Linkage by Universal Software Interfaces* [12]. It provides a lightweight interface between different programming environments which are only used for communicating with remote hosts. The efficiency of the native environment remains untouched.

The interfaces are embedded into the various environments. A PVM application, for example, uses ordinary PVM-IDs for addressing non-PVM processes. Consequently, a PARIX process reachable via PLUS is represented within a PVM application by an ordinary PVM-ID. Of course, PLUS takes care that these 'pseudo PVM-IDs' do not conflict with the real PVM-IDs.

Furthermore, PLUS allows to create processes by overlaying existing routines (PVM) or by extending the available function set with its own procedure (in the case of MPI). Thereby, PLUS adds a dynamic process model to programming environments like MPI1,

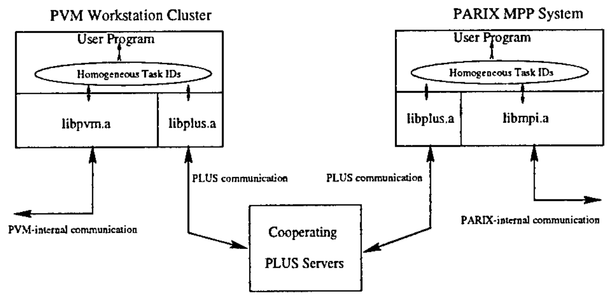


Figure 4: PLUS architecture

which do not provide such facilities. PLUS has no internal resource management functionality, but it forwards the requests to the abstract resource management layer, which offers more powerful methods than could be integrated into PLUS.

With this concept, PLUS allows program developers to integrate existing codes into the metacomputer environment, thereby lowering barriers in the use of metacomputers.

PLUS Architecture. Fig. 4 depicts the architecture of the abstract interface layer for programming environments developed in the PLUS project. Note that the regular PVM communication is not affected by PLUS. Only when accessing a PVM-ID that actually represents an external process (a PARIX process in this example), the corresponding PLUS routine is invoked. This routine performs the requested communication via the fastest available protocol between the PVM cluster and the PARIX system. Usually, communication will be done via UDP.

PLUS libraries are linked to each other via one or more PLUS server. Since most of the PLUS code is contained in these servers, the link library could be kept rather small. The servers manage the translation of different data representations, the message routing along the fastest network links, the dynamic creation of new processes, and much more. The number of active PLUS servers and their configuration may change dynamically at run time according to the needs of the user's application.

PLUS Performance. On a 34 Mb/s WAN interconnection, the PLUS communication has been shown to be even faster than TCP [12]. This is because the UDP based communication protocol of PLUS builds a virtual channel on which the messages are multiplexed. A sliding window technique allows to assemble and re-order the acknowledgements in the correct sequence.

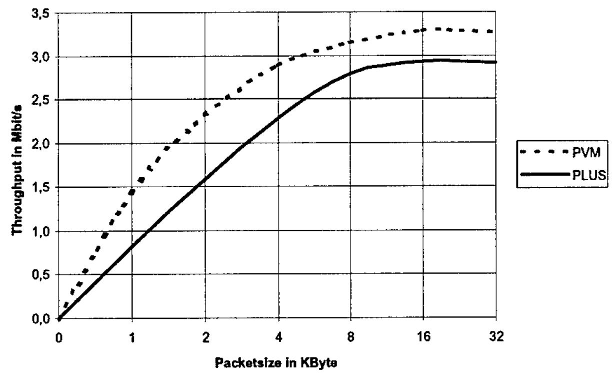


Figure 5: PLUS versus PVM on a 10Mb/s LAN

Figure 5 shows a less favorable case where the PLUS communication is outperformed by PVM. A parallel Parsytec CC system under AIX has been interconnected to a SUN-20 via 10Mb/s Ethernet. In both runs, the same PVM code has been used: once communicating via the internal PVM protocol, and the other time via the PLUS protocol. Here, PLUS is about 10 to 20% slower, because the PLUS communication needs two Internet hops, one hop from the CC to the PLUS daemon, and another from the daemon to the target SUN. The PVM tasks, in contrast, need only one Internet hop for communicating between the two corresponding PVM daemons. Moreover, since the PLUS communication libraries are designed in an open and extensible way, they do not contain routing information.

Note, that the example shows a worst case, because PLUS would only be used to communicate between *different* programming environments. Any internal communication, e.g. within PVM, is not affected by PLUS.

Extending PLUS. The current version of PLUS supports PVM, MPI and PARIX, where the latter is available on Parsytec systems only.

The PLUS library consists of two parts: a generic, protocol-independent module, and a translation module. The translation module contains a small set of abstract communication routines for the specific protocols. These routines are quite small, typically comprising less then one hundred lines of code.

New programming environments can be integrated to PLUS by implementing a new translation module. This allows applications to communicate with any other programming model for which translation modules are available in PLUS.

6 Resource Management Layer

Resource management is a central task in metacomputing environments, permitting oversubscribed resources to be fairly and efficiently shared. Historically, supercomputer centers have been using batch queuing systems such as NQS [28] for managing their machines and for scheduling the available computing time. Distributed memory systems employ more complex resource scheduling mechanisms, because a large number of constraints must be considered in the execution of parallel applications. As an example, interactive applications may compete with batch jobs, and requests for the use of non-timeshared components or for special I/O facilities may delay other jobs.

On a metacomputer, the scheduler is responsible for assigning appropriate resources to parallel applications. Scheduling is done by the resource management system that allocates a particular CPU at a particular time instance for a given application, and by the operating system running on a certain compute node. In parallel environments, we distinguish two levels of scheduling:

- At the *task level*, the scheduling of tasks and threads (possibly of different jobs) is performed by the operating system on the single MPP nodes. The scheduling strategy may be uncoordinated, as in the traditional time-sharing systems, or it can be synchronized, as in gang-scheduling. The latter, however, is only available on a few parallel systems.

- At the *job level*, the scheduling is usually architecture independent. A request may specify the CPU type, memory requirements, I/O facilities, software requirements, special interconnection structures, required occupation time and some attributes for distinguishing batch from interactive jobs. Here, the request scheduler is responsible for re-ordering and assigning the submitted requests to the most appropriate machines.

With few exceptions [9], the existing schedulers were originally designed for scheduling serial jobs [27, 3]. As a consequence, many of them do not obey the 'concept' of parallel programs [37]. Typically, a stater (master) process is launched that is responsible for starting the rest of the parallel program. Thus, the management has limited knowledge about the distributed structure of the application. When the master process dies unexpectedly (or has been exempted because the granted time is expired) the rest of the parallel program is still existent. Such orphaned processes can cause serious server problems, and in case of MPP systems (like the SP2) they can even lock parts of the machine.

With its open architecture, MOL is not restricted to the usage of specific management systems. Rather, we distinguish three types of management systems, each of them specialized to a certain administration strategy and usage profile:

The first group of applications comprise *sequential and client-server programs*. The management software packages of this group have emerged from the traditional vector-computing domain. Examples are Condor, Codine, DQS and LSF.

The second group contains resource management systems tailored to the needs of *parallel applications*. Some of them have their roots in the NQS development (like PBS), while others were developed to overcome problems with existing vendor solutions (like EASY) or do focus on the transparent access to partitionable MPP systems (like CCS described below).

The last group consists of resource management systems for *multi-site applications*, exploiting the whole power of a metacomputing environment. Currently, only few multi-site applications exists, and the development of corresponding resource management systems is still in its infancy. But with increasing network performance such applications and their need for a uniform and optimizing management layer is gaining importance.

6.1 Computing Center Software CCS

CCS [15] provides transparent access to a pool of massively parallel computers with different architectures [35]. Today, parallel machines ranging from 4 to 1024 nodes are managed by CCS. CCS provides user authorization, accounting, and for scheduling arbitrary mixtures of interactive and batch jobs [23]. Furthermore, it features an automatic reservation system and allows to re-connect to a parallel application in the case of a breakdown in the WAN connection—an important feature for remote access to a metacomputer.

Specification Language for Resource Requests. Current management systems use a variety of command-line (or batch script) options at the user interface, and hundreds of environment variables (or configuration files) at the operator interface. Extending such concepts to a metacomputing environment can result in a nightmare for users and operators.

Clearly, we need a general resource description language equipped with a powerful generation and anima-

```
INCLUDE  <default_defs.rdl>                         -- default definitions and constant declarations

DECLARATION
BEGIN UNIT MC_appl;

  DECLARATION
      BEGIN SECTION main;                           -- main computation on grid
          EXCLUSIVE;                                -- use separate processors for each item
          DECLARATION
              FOR  i=0  TO  (main_x * main_y - 1)  DO
                   { PROC i; compute = pbl_size; CPU = MPC601; MEMORY = 16; }; OD
          CONNECTION
              FOR  i=0  TO  (main_x - 1)  DO
                   FOR  j = i * main_y  TO  (i * main_y + main_y-2) DO
                        PROC j LINK 0   <=>   PROC j+1 LINK 2; OD OD
              FOR  i=0  TO  (main_y - 1)  DO
                   FOR  j=0  TO  (main_x - 2) DO
                        PROC  (j * main_y + i)  LINK 1 <=> PROC (main_y * (j + 1) + i)  LINK 3; OD OD

            ASSIGN  LINK 1  <=>  PROC 0 LINK 3;        -- from user terminal
            ASSIGN  LINK 2  <==  PROC (main_x * main_y-1) LINK 1; -- to postprocessing
      END SECTION

      BEGIN SECTION post;                           -- pipelined post-processing
         SHARED;                                    -- the items of this section may be run on a single node
         DECLARATION
             FOR  i=1  TO  post_len  DO
                  { PROC i; filter = post; CPU = MPC601; }; OD
             { PORT Ausgabe; DISK; };
         CONNECTION
             FOR  i=1  TO  (post_len - 1)  DO
                  i  LINK 0   ==>   i + 1  LINK 1; OD
             PROC  post_len  LINK 0  ==>   PORT Ausgabe LINK 0;
           ASSIGN  PROC 1    LINK 1  <==   LINK 0;    -- link to higher level
      END SECTION

      BEGIN SECTION user;                           -- user control section
         DECLARATION
             { PORT IO; TERMINAL; };
         CONNECTION
            ASSIGN   IO LINK 0    <=>    LINK 0;
      END SECTION

      CONNECTION                                    -- connecting the modules
          user LINK 0     <=>     main LINK 1;
          main LINK 2     ==>     post LINK 0;
END UNIT -- MC_appl
```

Figure 6: Specification of the system shown in Figure 3 using the Resource Description Language RDL

tion tool. Metacomputer users and metacomputer administrators should not have to deal directly with this language but only with a high-level interface. Analoguously to the popular *postscript* language used for the device-independent representation of documents, a metacomputer resource description language is a means to specify the resources needed for a metacomputer application. Administrators should be able to use the same language for specifying the available resources in the virtual machine room.

In a broad sense, resource representations must be specified on three different abstraction levels: The *vendor-level* for describing the internal structure of a machine. The *operator-level*, which is the metacomputer itself, for describing the interconnections of the machines within the network and their general properties. And the *user-level*, for specifying a logical topology to be configured for a given application.

Within MOL, we use the *Resource Description Language RDL* [4], that has been developed as part of the CCS resource management software [35]. RDL is used

- at the administrator's level for describing type and topology of the participating metacomputer components, and
- at the user's level for specifying the required system configuration for a given application.

Figure 6 shows an RDL specification for the example discussed in Figure 3. It is the task of the resource scheduler to determine an optimal mapping between the two specifications: The specification of the application structure on the one hand, and the specification of the system components on the other hand. Better mapping results are obtained when the user requests are less specific, i.e., when classes of resources are specified instead of specific computers.

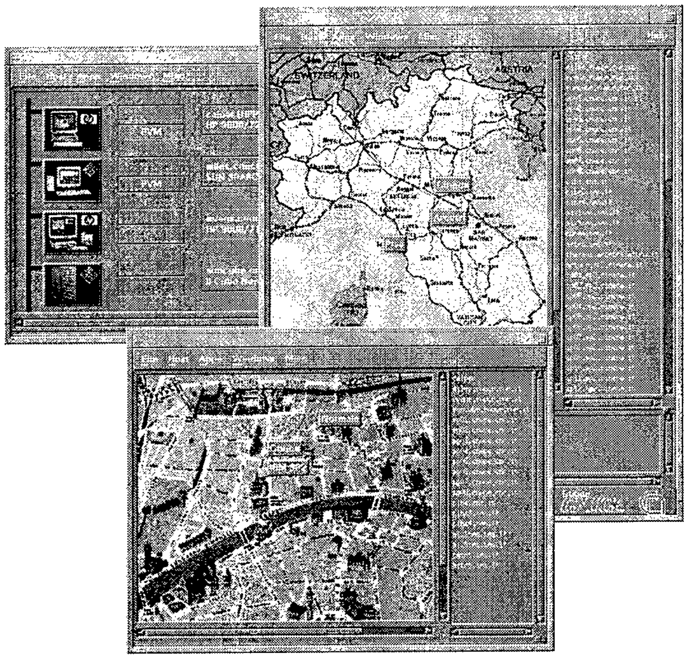


Figure 7: Snapshot of WAMM user interface

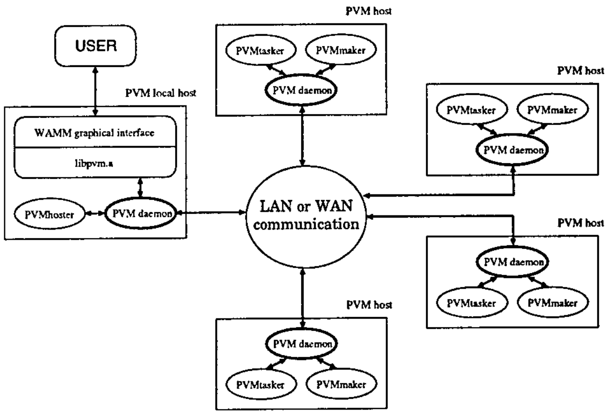


Figure 8: WAMM architecture

7 Supporting Tools

The MOL toolset contains a number of useful tools for launching and executing distributed applications on the metacomputer. We first describe the "wide area metacomputer manager" WAMM, which provides resource management functionality as well as automatic source code compilation and cleanup after task execution.

Another class of tools allows for dynamic task migration at execution time (MARS), which makes use of the performance prediction tool WARP.

Data libraries for virtual shared memory and load balancing (DAISY) provide a more abstract programming level and program templates (FRAMES) facilitate the design of efficient parallel applications, even for non-experts.

7.1 Graphical Interface WAMM

The *Wide Area Metacomputer Manager WAMM* [43, 5, 6] supports the user in the management of the computing nodes that take part in a parallel computation. It controls the virtual machine configuration, issues remote commands and remote source code compilations, and it provides task management. While ealier releases of WAMM [5, 6] were limited to systems running PVM only, with the PLUS library it is now possible to use WAMM on arbitrary systems [7].

WAMM can be seen as a mediator between the top user access level and the local resource management. It simplifies the use of a metacomputer by adopting a "geographical" view, where the hosts are grouped in tree structured sub-networks (LANs, MANs or WANs). Each item in the tree is shown in an OSF/Motif window, using geographical maps for networks and icons for hosts as shown in Figure 7. The user can move through the tree and explore the resources by selecting push buttons on the maps. It is possible to zoom from a wide-area map to a single host of a particular site. A traditional list with Internet host addresses is also available.

Metacomputer Configuration. The metacomputer can be configured by writing a configuration file which contains the description of the nodes that make up the metacomputer, i.e. all the machines that users can access. This file will be read by WAMM at startup time.

Application Development. The development and execution of an application requires some preparatory operations such as source code editing, remote compilation and execution. In WAMM, programmers develop their code on a local machine. For remote compilation, they only have to select hosts where they want to do the compilation and to issue a single `Make` command from the popup menu. In a dialog box the local directories containing the source files and corresponding `Makefiles` can be specified along with the necessary parameters.

WAMM supports remote compilation by grouping all source files into a single, compressed `tar` file. A

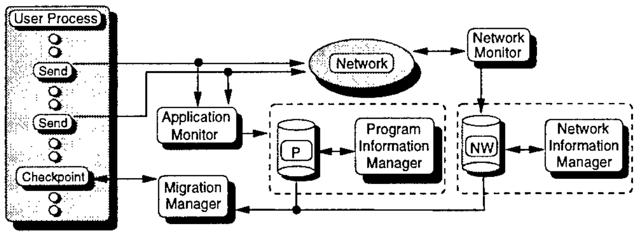


Figure 9: MARS System Architecture

PVMMaker task, which deals with the remote compilation, is spawned on each participating node and the compressed tar file is sent to all these tasks. The rest of the work is carried out in parallel by the PVMMakers. Each PVMMaker receives the compressed file, extracts the sources in a temporary working directory, and executes the make command. WAMM is notified about the operations that have been executed, and the result is displayed in a control window to show the user the status of the compilation.

Tasks Execution and Control. Application are started by selecting the Spawn pushbutton from the Apps popup menu. A dialog box is opened for the user to insert parameters, such as number of copies, command line arguments, etc. The output of the processes is displayed in separate windows and/or saved in files. When the output windows are open, new messages from the tasks are shown immediately (Fig. 7). A Tasks control window can be opened to control some status information on all the PVM tasks being executed in the virtual machine.

7.2 MARS Task Migrator

The *Metacomputer Adaptive Runtime System MARS* [24] is a software module for the transparent migration of tasks during runtime. Task migrations may become necessary when single compute nodes or sub-networks exhibit changing load due to concurrent use by other applications. MARS uses previously acquired knowledge about a program's runtime behavior to improve its task migration strategy. The knowledge is collected during execution time without increasing the overall execution time significantly. The core idea is to keep all information gathered in a previous execution of the same program and to use it as a basis for the next run. By combining information from several runs, it is possible to find regularities in the characteristic program behavior as well as in the network profile. Future migration decisions are likely to benefit by the acquired information.

The MARS runtime system comprises two types of instances (Fig. 9): *Monitors* for gathering statistical data on the CPU work-load, the network performance and the applications' communication behavior, and *Managers* for exploiting the data for computing an improved task-to-processor mapping and task migration strategy.

As MARS is designed for heterogeneous metacomputers, we cannot simply migrate machine-dependent core images. Instead, the application code must be modified by a preprocessor to include calls to the runtime system at certain points where task migration is allowed. The user's object code is linked to the MARS runtime library which notifies an *Application Monitor* and a *Network Monitor* each time a **send** or **receive** operation is executed.

The Network Monitor collects long-term statistics on the network load. The Application Monitor monitors the communication patterns of the single applications and builds a task dependency graph for each execution. Dependency graphs from successive execution runs are consolidated by a *Program Information Manager*. The resulting information is used by the *Migration Manager* to decide about task migrations whenever a checkpoint is reached in the application.

In summary, two kinds of data are maintained: application specific information (in dependency graphs) and system specific information (in system tables) for predicting the long-term performance of the network and CPU work-load. Further information on MARS can be found in [24].

7.3 Runtime Predictor WARP

WARP is a Workload Analyzer and Runtime Predictor [39, 40]. Performance prediction tools provide performance data to compilers, programmers and system architects to assist in the design of more efficient implementations. While being an important aspect of high-performance computing, there exist only few projects that address performance prediction for clusters of workstations and no project targets the metacomputing scenery.

WARP is used within MOL to advise the resource management systems for better hardware exploitation by maximizing the through-put or by minimizing the response time. More specifically, the performance prediction figures of WARP provide valuable input for the initial mapping and dynamical task migration performed by MARS. The WARP system [39, 40] includes modules for

- compiling resource descriptions into a suitable internal representation,

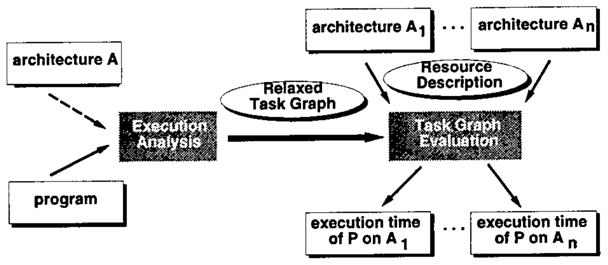


Figure 10: WARP architecture

- analyzing resource requests of parallel programs, and
- predicting the execution time of parallel programs.

Architectural Model. In WARP, a parallel machine is described by a hierarchical graph. The nodes in the graph denote processors or machines with local memory. The edges denote interconnections between the machines. The nodes are weighted with the relative execution speed of the corresponding machines, which have been measured by small benchmark programs or routines for testing the functional units of the CPU (floating-point pipelines, load & store operations in the memory hierarchy, etc.). The edges are weighted with the latency and bandwidth of the corresponding communication performance between the two nodes, which can be either a local mechanism (e.g., shared memory) or a network communication. A monitoring tool in WARP detects the basic load of time-sharing systems, that is used to model the statistical load reducing the potential performance of the machine or interconnection.

Task Graph Construction. WARP models the execution of a parallel program by a task graph consisting of a set of sequential blocks with corresponding interdependencies. The nodes and edges of a task graph are weighted with their computation and communication loads. Task graphs are constructed either by static program analysis or by executing the code on a reference system. In the latter case, the load factors are reconstructed from the execution time of the sequential blocks and communications.

Clearly, the task graphs – representing execution traces – must not be unique. In cases where the control flow depends on the execution sequence of the sequential blocks or on the communication pattern, stochastic graphs are be used. Moreover, detailed event tracing can result in extremely large task graphs, depending on the size of the system. Fortunately, most parallel applications are written in SPMD or data parallel mode, executing the same code on all processors with local, data-dependent branches. Only a few equivalence classes of processor behavior are modeled by WARP, with statistical clustering used as a standard technique for identifying data equivalence classes. Also regular structures derived by loops and subroutines are used for a task graph relaxation.

Task Graph Evaluation. Task graph are evaluated for predicting the execution time under modified task allocation schemes and/or changed resources. The task graph evaluation is done by a discrete event simulator which constructs the behavior of the parallel program running on a hardware with a given resource description. Resource contention is modeled by queuing models. The execution times and communication times are then adjusted to the relative speed of the participating resource and the expected contention.

7.4 Data Management Library DAISY

The data management library *DAISY* (Fig. 11) comprises tools for the simulation of shared memory (DIVA) and for load balancing (VDS) in a single comprehensive library. A beta release of DAISY is available for Parsytec's PARIX and PowerMPI programming models. With the improved thread support of MPI-2, DAISY will also become available on workstation clusters.

Load Balancing with VDS. The virtual data space tool *VDS* simulates a global data space for structured objects stored in distributed heaps, stacks, and other abstract data types. The work packets are spread over the distributed processors as evenly as possible with respect to the incurred balancing overhead [19]. Objects are differentiated by their corresponding class, depicted in Figure 11 by their different shape. Depending on the object type, one of three distribution strategies is used:

- *Normal* objects are weighted by a load measure given by the expense of processing the object. VDS attempts to distribute the object such that all processors have about the same load. A processor's load is defined as the sum of the load-weights of all placed objects.

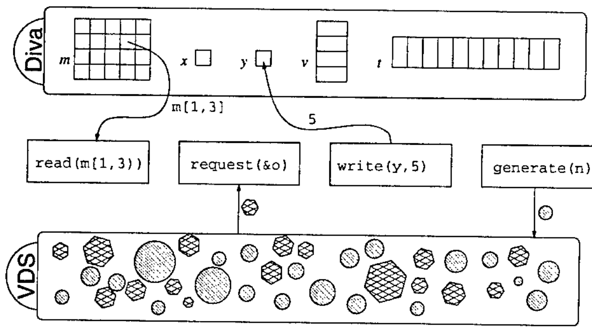


Figure 11: : The two *DAISY* tools: Distributed shared memory (Diva) and the load-balancing layer VDS.

- In some applications, such as best-first branch-and-bound, the execution time is not only affected by the number of objects, but also by the order in which the objects are processed. In addition to the above described quantitative load balancing, some form of qualitative load balancing is performed to ensure that all processors are working on promising objects. VDS provides *qualitative load balancing* by means of the weighted objects. Besides their load-weight, these objects possess a quality tag used to select the next suitable object from several alternatives.
- The third kind of object, *thread objects*, provide an easy and intuitive way to model multi-threaded computations as done in CILK [10]. In this computational model, threads may send messages (results) to their parents. With later versions it will be possible to send data across more than one generation (e.g. to grandparents) [21].

In the current VDS release, we implemented a work-stealing method [11] for thread objects as well as diffusive load balancing for normal and weighted objects.

Shared Memory Simulation with DIVA. The distributed variables library *DIVA* provides functions for simulating shared memory on distributed systems. The core idea is to provide an access mechanism to distributed variables rather than to memory pages or single memory cells. The variables can be created and released at runtime. Once a global variable is created, each participating processor in the system has access to it.

For latency hiding, reads and writes can be performed in two separate function calls. The first call initiates the variable access, and the second call waits

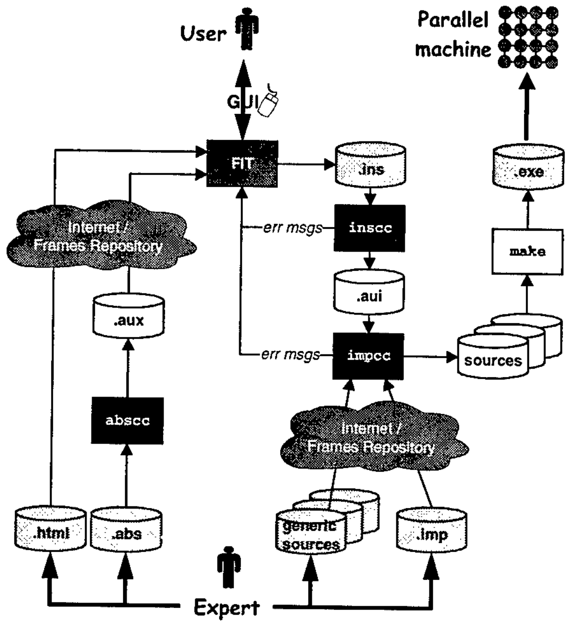


Figure 12: The frame model

for its completion. The time between initiation and completion of a variable access can be hidden by other local instructions or variable accesses.

7.5 Programming Frames

Programming frames facilitate the development of efficient parallel code for distributed memory systems. Programming frames are intended to be used by non-experts, who are either unfamiliar with parallel systems or unwilling to cope with new machines, environments and languages. Several projects [16, 8, 17, 18] have been initiated to develop new and more sophisticated ways for supporting the programming of distributed memory systems via libraries of basic algorithms, data structures and programming frameworks (templates). Like LEDA for the sequential case [30], each of these approaches provides non-experts with tools to program and exploit parallel machines efficiently.

Our frames are like black boxes with problem dependent holes. The basic idea is to comprise expert knowledge about the problem and its parallelization into the black box and to let the users specify the holes, i.e. the parts that are different at each instantiation. The black boxes are either constructed using efficient basic primitives, standard parallel data types, communication schemes, load balancing and mapping facilities, or they are derived from well optimized, complete applications. In any case, frames contain efficient state-of-the-art techniques focusing on reusability and portability – both important aspects in metacomputing. This will save software development costs and improve the reliability of the target code.

Figure 12 depicts our frame model. Each generated target executable is built from three different specifications. The *abstract specification* defines the parameters of the problem (the holes in the black box). It remains the same for all instances of that frame, and is generally given by an expert. Furthermore, the abstract specification is used to generate a graphical user interface (GUI) and for the type consistency check in the instance level.

At the *instance level*, values are assigned to the parameters specified in the abstract level. New problem instances are generated by the user by giving new instance specifications via the GUI.

The implementation level contains a number of implemented sources, e.g. for MPI or PVM, with respect to a given abstract frame. An *implementation specification* consists of a list of source files and rules for modifying those sources to get new ones that comply with the values given in the instance level. The rules are best described as replacements and generations. New frames can also be composed by modifying and/or composing existing basic frames.

Three major tools are used to process the frame specifications. The first checks the abstract specification. The second one checks the instance specification against the abstract specification. The build tool, finally, generates the target executables taking the specifications for both an instance and an implementation. Our preliminary versions of these tools and the GUI are portable across systems with POSIX compliant C compilers. As GUIs are considered vital for the acceptance of the programming frames, our implementation is based on the graphical TCL/TK toolkit. The interface reads the abstract specification and prompts the user for the frame parameters. The output is an instance specification according to our model. A Java interface to the frames will be available in the near future.

8 Summary

We have presented an open metacomputer environment that has been designed and implemented in a collaborative effort in the Metacomputer Online (MOL) initiative. Due to the diversity in the participating hard- and software on the one hand, and due to the heterogeneous spectrum of potential users on the other hand, we believe that a metacomputer cannot be regarded as a closed entity. It should rather be designed in an open, extensible manner that allows for continuous adjustment to meet the user's demands by a dynamically changing HW/SW environment.

With the MOL framework, we have linked existing software packages by generic, extensible interface layers, allowing future updates and inclusion of new soft- and hardware. There are three general classes of metacomputer modules:

- programming environments (e.g., PVM, MPI, PARIX, and the PLUS linkage module),
- resource management & access systems (Codine, NQS, PBS, CCS),
- supporting tools (GUIs, task migrator MARS, programming frames, data library DAISY, WAMM, WARP performance predictor,...).

All of these modules exist. Linked by appropriate generic interfaces, the modules became an integral part of the MOL environment. From the hardware perspective, MOL currently supports geographically distributed high-performance systems like Parsytec GC, Intel Paragon, and UNIX workstation clusters that are run in a dedicated compute cluster mode.

As a positive side-effect, the collaboration within the MOL group has resulted in a considerable amount of (originally unexpected) synergy. Separate projects, that have been started as disjoint research work, now fit together in a new framework. As an example, the task migration manager MARS derives better migration decisions when being combined with the performance predictor WARP. An even more striking example is the linkage of WAMM with PLUS [7]. Previously, the WAMM metacomputer manager was limited to PVM only. The PLUS library now extends the application of WAMM to a much larger base of platforms.

Acknowledgements

MOL is the colaborative effort of many individuals. We are indebted to the members of the Metacomputer Online Group, the members of the Northrhine-Westphalian Initiative Metacomputing, and to the members of ALCOM-IT, especially Jordi Petit i Silvestre (UPC).

References

[1] D. Abramson, R. Sosic, J. Giddy, B. Hall. *Nimrod: A tool for performing parameterized simulations using distributed workstations.* 4th IEEE Symp. High-Perf. Distr. Comp. (August 1995).

[2] J.N.C. Árabe, A.B.B. Lowekamp, E. Seligman, M. Starkey, P. Stephan. *Dome: Parallel programming environment in a heterogeneous multi-user environment.* Supercomputing 1995.

[3] M.A Baker, G.C. Fox, H.W. Yau. *Cluster computing review.* Techn. Report, Syracuse Univ., Nov. 1995.

[4] B. Bauer, F. Ramme. *A general purpose Resource Description Language.* Reihe Informatik aktuell, Hrsg R. Grebe, M. Baumann, Parallel Datenverarbeitung mit dem Transputer, Springer-Verlag, (Berlin), 1991, 68-75.

[5] R. Baraglia, G. Faieta, M. Formica, D. Laforenza. *WAMM: A Visual Interface for Managing Metacomputers.* EuroPVM'95, Ecole Normale Supèrieure de Lyon, Lyon, France, September 14-15, 1995, 137–142.

[6] R. Baraglia, G. Faieta, M. Formica, D. Laforenza. *Experiences with a Wide Area Network Metacomputing Management Tool using IBM SP-2 Parallel Systems.* Concurrency: Practice and Experience, John Wiley & Sons, Ltd., Vol.8, 1996, in press.

[7] R. Baraglia, D. Laforenza. *WAMM integration into the MOL Project.* CNUCE Inst. of the Italian Nat. Research Council, Pisa, Internal Rep., 1996.

[8] R. Barret, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, H. van der Vorst. *TEMPLATES for the Solution of Linear Systems: Building Blocks for Iterative Methods.* Tech. Rep., CS-Dept., Univ. of Tennessee, 1993.

[9] F. Berman, R. Wolski, S. Figueira, J. Schopf, G. Shao. *Application-level scheduling on distributed heterogeneous networks.* Tech. Rep., Univ. California, San Diego. `http://www-cse.ucsd.edu/-groups/hpcl/apples`.

[10] R.D. Blumhofe, C.F. Joerg, B.C. Kuszmaul, C.E. Leiserson, K.H. Randall, Y. Zhou. *Cilk: An Efficient Multithreaded Runtime System.* Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, July 19-21, 1995, Santa Barbara, California, 207-216.

[11] R.D. Blumhofe, C.E. Leiserson. *Scheduling Multithreaded Computations by Work Stealing.* Proc. of the 36th Ann. Symposium on Foundations of Computer Science (FOCS '95), 356-368, 1995.

[12] M. Brune, J. Gehring, A. Reinefeld. *A Lightweight Communication Interface Between Parallel Programming Environments.* HPCN'97, Springer LNCS.

[13] G. D. Burns, R. B. Daoud, J. .R. Vaigl. *LAM: An Open Cluster Environment for MPI.* Supercomputing Symposium '94, Toronto, Canada, June 1994

[14] N. Carriero, E. Freeman, D. Gelernter, D. Kaminsky. *Adaptive Parallelism and Piranha.* IEEE Computer 28,1 (1995), 40–49.

[15] *Computing Center Software CCS.* Paderborn Center for Parallel Computing. `http://www.-uni-paderborn.de/pc2/projects/ccs`

[16] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* PhD, Research Monographs in Par. and Distr. Computing, MIT Press.

[17] M. Danelutto, R. Di Meglio, S. Orlando, S. Pelagatti, M. Vanneschi. *A Methodology for the Development and the Support of Massively Parallel Programs.* J. on Future Generation Computer Systems (FCGS), Vol. 8, 1992.

[18] J. Darlington, A.J. Field, P.G. Harrison, P.H.J. Kelly, D.W.N. Sharp, Q. Wu. *Parallel Programming Using Skeleton Functions.* Proc. of Par. Arch. and Lang. Europe (PARLE '93), Lecture Notes in Computer Science No. 694, Springer-Verlag, 1993.

[19] T. Decker, R. Diekmann, R. Lüling, B. Monien. *Towards Developing Universal Dynamic Mapping Algorithms.* Proc. of the 7th IEEE Symposium on Parallel and Distributed Processing, SPDP'95, 1995, 456-459.

[20] J.J. Dongarra, S.W. Otto, M. Snir. D. Walker. *A message passing standard for MPP and Workstations.* CACM 39,7(July 1996), 84–90.

[21] P. Fatourou, P. Spirakis. *Scheduling Algorithms for Strict Multithreaded Computations.* Proc. of the 7th Annual International Symposium on Algorithms and Computation (ISAAC '96), 1996, to appear.

[22] I. Foster. *High-performance distributed computing: The I-Way experiment and beyond.* Procs. of the EURO-PAR'96, Lyon, 1996, Springer LNCS 1124, 3–10.

[23] J. Gehring, F. Ramme. *Architecture-Independent Request-Scheduling with Tight Waiting-Time Estimations,* IPPS Workshop on Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science 1162, 1996.

[24] J. Gehring, A. Reinefeld. *MARS – A Framework for Minimizing the Job Execution Time in a Metacomputing Environment.* Future Generation Computer Systems (FGCS), Elsevier Science B. V., Spring 1996.

[25] A. Grimshaw, J.B. Weissman, E.A. West, E.C. Loyot. *Metasystems: An approach combining parallel processing and heterogeneous distributed computing systems.* J. Par. Distr. Comp. 21 (1994), 257–270.

[26] L. V. Kale, S. Krishnan. *CHARM++: A Portable Concurrent Object Oriented System Based On C++.* Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA), September 1993

[27] J.A. Kaplan, M.L. Nelson. *A Comparison of Queuing, Cluster and Distributed Computing Systems.* NASA Technical Memo, June 1994.

[28] B.A. Kinsbury. *The Network Queuing System.* Cosmic Software, NASA Ames Research Center, 1986.

[29] M.J. Litzkow, M. Livny. *Condor–A hunter of idle workstations.* Procs. 8th IEEE Int. Conf. Distr. Computing Systems, June 1988, 104–111.

[30] K. Mehlhorn, S. Näher. *LEDA, a Library of Efficient Data Types and Algorithms.* MFCS 89, LNCS Vol. 379, 88-106

[31] Metacomputer Online (MOL). `http://www.-uni-paderborn.de/pc2/projects/mol/`

[32] B. Monien, F. Ramme, H. Salmen : *A Parallel Simulated Annealing Algorithm for Generating 3D Layouts of Undirected Graphs.* Proc. of Graph Drawing '95, Springer LNCS, Vol. 1027, pp. 396-408.

[33] David Patterson et al. *A Case for Networks of Workstations: NOW.* IEEE Micro. http://now.CS.Berkeley.EDU/Case/case.html

[34] *Polder.* `http://www.wins.uva.nl/projects/-polder/`

[35] F. Ramme, T. Römke, K. Kremer. *A Distributed Computing Center Software for the Efficient Use of Parallel Computer Systems.* HPCN Europe, Springer LNCS 797, Vol. II, 129-136 (1994).

[36] F. Ramme, K. Kremer. *Scheduling a Metacomputer by an Implicit Voting System.* 3rd IEEE Int. Symposium on High-Performance Distributed Computing, San Francisco, 1994, 106-113.

[37] W. Saphir, L.A. Tanner, B. Traversat. *Job Management Requirements for NAS Parallel Systems and Clusters.* IPPS Workshop on Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science 949, 319-336, 1995.

[38] L. Schäfers, C. Scheidler, O. Krämer-Fuhrmann. *Software Engineering for Parallel Systems: The TRAPPER Approach.* 28th Hawaiian International Conference on System Sciences, January 1995, Hawaii, USA

[39] J. Simon, J.-M. Wierum. *Performance Prediction of Benchmark Programs for Massively Parallel Architecture.* 10th Annual Intl. Conf. on High-Performance Computer HPCS'96, June 1996.

[40] J. Simon, J.-M. Wierum. *Accurate Performance Prediction for Massively Parallel Systems and its Applications.* Euro-Par'96 Parallel Processing, August 1996, LNCS 1124, 675–688.

[41] L. Smarr, C.E. Catlett. *Metacomputing.* Communications of the ACM 35,6(1992), 45–52.

[42] F. Tandiary, S.C. Kothari, A. Dixit, E.W. Anderson. *Batrun: Utilizing ilde workstations for large-scale computing.* IEEE Parallel and Distr. Techn., Summer 1996, 41–48.

[43] *WAMM – Wide Area Metacomputer Manager.* Available at `http://miles.cnuce.cnr.it` or via ftp at `ftp.cnr.it` in the directory `/pub/wamm`.

A Programming Environment for Heterogenous Distributed Memory Machines

Dmitry Arapov, Alexey Kalinov, Alexey Lastovetsky, Ilya Ledovskih
Institute for System Programming, Russian Academy of Sciences
25, Bolshaya Kommunisticheskaya str., Moscow 109004, Russia
lastov@ispras.ru

Ted Lewis
Naval Postgraduate School, Code CS, Monterey, CA 93943-5118
lewis@cs.nps.navy.mil

Abstract

mpC is a programming language of medium level for distributed memory machines (DMM). The language is an ANSI C superset based on the notion of network comprising virtual processors of different types and performances connected with links of different bandwidths. It allows the user to describe a network topology, create and discard networks, distribute data and computations over the networks. In other words, the user can specify (dynamically) the topology of his application, and the mpC programming environment will use this (topological) information in run time to ensure the efficient execution of the application on any particular DMM. The paper outlines the most principal features of mpC and its programming environment making them suitable tools to write efficient and portable parallel programs for heterogenous DMMs.

1. Introduction

The mpC language and its programming environment was initially developed to support programming for massively parallel computers, first of all for high-performance distributed memory machines (DMMs). In brief, our motivation of mpC was as follows.

Programming for DMMs is based mostly on message-passing function extensions of C or Fortran, such as PVM [1] and MPI [2]. But it is tedious and error-prone to program in a message-passing language, because of its low level. Therefore, high-level languages that facilitate parallel programming have been developed for DMMs. They can be divided into two classes depending on the parallel programming paradigm - task parallelism or data parallelism - underlying them. Task parallel [3-4] and data parallel [5-11] programming languages allow the user to implement different classes of parallel algorithms. But efficient implementation of many problems needs parallel algorithms that can not be implemented in pure data parallel or task parallel styles. We have developed the mpC language (as an ANSI C superset) which supports both task and data parallelism, allows both static and dynamic process and communication structures, enables optimizations aimed at both communication and computation, and supports modular parallel programming and the development of a library of parallel programs.

The mpC language is based on the notion of network consisting of virtual processors of different types and performances connected with links of different bandwidths. The user can describe network topology, create and discard networks, and distribute data and computations over the networks. That is, the user can specify (dynamically!) in details virtual parallel machine which performs his application.

In other words, the user can specify the topology of his application, and the programming environment will use this (topological) information **in run time** to ensure the efficient execution of the application on any particular DMM.

Currently, the mpC programming environment includes a compiler, a run-time support system, a library, and a command-line user interface.

The compiler translates a source mpC program into ANSI C code with calls to functions of the run-time support system.

Run-time support system manages the computing space which consists of a number of processes running over target DMM as well as provides communications. It has a precisely specified interface and encapsulates a particular communication package (currently, a small subset of MPI). It ensures platform-independence of the rest of system components.

The library consists of a number of functions which sup-

0-8186-7879-8/97 $10.00 © 1997 IEEE

port debugging mpC programs as well as provide some low-level efficient facilities.

The command-line user interface consists of a number of shell commands supporting the creation of a virtual DMM and the execution of mpC programs on the machine. While creating the machine, its topology is detected by a topology detector running a special benchmark and saved in a file used by the run-time support system.

When developing the mpC programming environment, we used a network of workstations running MPI as a target parallel machine and found, that the principles, on which mpC is based, make this programming language and its programming environment be very convenient tools for development of efficient and portable parallel programs for heterogenous networks of workstations.

The point is that all programming environments for DMMs which we know of have one common property. Namely, when developing a parallel program, either the user has no facilities to describe the virtual parallel system executing the program, or such facilities are too poor to specify an efficient distribution of computations and communications over the target DMM. Even topological facilities of MPI (as well as MPI-2) have turned out insufficient to solve the problem. So, to ensure the efficient execution of the program on a particular DMM, the user must use facilities which are external to the program, such as boot schemes and application schemes [12]. If the user is familiar with both the topology of target DMM and the topology of the application, then, by using such configurational files, he can map the processes which constitute the program onto processors which make up DMM, to provide the most efficient execution of the program. But if the application topology is defined in run time (that is, if it depends on input data), it won't be successful.

The mpC language allows the user to specify an application topology, and its programming environment uses the information in run time to map processes onto processors of target DMM resulting in efficient execution of the application.

Section 2 of the paper outlines the mpC language. Section 3 sketches the mpC programming environment. Section 4 demonstrates how mpC may be used to develop efficient and portable irregular applications for DMMs. Section 5 demonstrates how mpC may be used to develop efficient and portable regular applications for heterogeneous DMMs. In addition, sections 4 and 5 tell more about the mpC language.

More about the language and its programming environment may be found in [13-17] as well as at http://www.ispras.ru/~mpc. In addition, the corresponding free software is available at http://www.ispras.ru/~mpc.

2. Outline of the mpC language

In mpC, the notion of *computing space* is defined as a set of typed virtual processors of different performance connected with links of different bandwidth accessible to the user for management. There are several processor types, but most common virtual processors are of the `scalar` type. A virtual processor has an attribute characterizing its relative performance. A directed *link* connecting two virtual processors is a one-way channel for transferring data from source processor to the processor of destination.

The basic notion of the mpC language is *network object* or simply *network*. Network comprises virtual processors of different types and performances connected with links of different bandwidths. Network is a region of the computing space which can be used to compute expressions and execute statements.

Allocating network objects in the computing space and discarding them is performed in similar fashion to allocating data objects in the storage and discarding them. Conceptually, creation of new network is initiated by a virtual processor of some network already created. This virtual processor is called a *parent* of the created network. The parent belongs to the created network. The only virtual processor defined from the beginning of program execution till program termination is the pre-defined virtual *host-processor* of the `scalar` type.

Every network declared in an mpC program has a type. The type specifies the number and types and performances of virtual processors, links between these processors and their lengths characterizing bandwidths, as well as separates the parent. For example, the type declaration

```
/*1*/      nettype Rectangle {
/*2*/        coord I=4;
/*3*/        node {
/*4*/          I<2 : fast scalar;
/*5*/          I>=2: slow scalar;
/*6*/        };
/*7*/        link {
/*8*/          I>0:  [I]<->[I-1];
/*9*/          I==0: [I]<->[3];
/*10*/       };
/*11*/       parent [0];
/*12*/     };
```

introduces network type `Rectangle` that corresponds to networks consisting of 4 virtual processors of the `scalar` type and different performances interconnected with undirected links of the normal length in a rectangular structure.

In this example, line 1 is a *header* of the network-type declaration. It introduces the name of the network type.

Line 2 is a *coordinate declaration* declaring the coordinate system to which virtual processors are related. It introduces integer coordinate variable `I` ranging from 0 to 3.

Lines 3-6 are a *node declaration*. It relates virtual processors to the coordinate system declared and declares their types and performances. Line 4 stands for the predicate ***for all*** *`I<4`* ***if*** *`I<2`* ***then*** *fast virtual processor of the `scalar` type is related to the point with coordinate `[I]`*. Line 5 stands for the predicate ***for all*** *`I<4`* ***if*** *`I>=2`* ***then*** *slow virtual processor of the `scalar` type is related to the point with coordinate `[I]`*. Performance specifiers `fast` and `slow` specify relative performances of virtual processors of the same type. For any network of this type, this information allows the compiler to associate a weight with each virtual processor of the network normalizing it in respect to the weight of the parent. Note, that the virtual host-processor is always of the `scalar` type and normal performance.

Lines 7-10 are a *link declaration*. It specifies links between virtual processors. Line 8 stands for the predicate ***for all*** *`I<4`* ***if*** *`I>0`* ***then*** *there exists undirected link of normal length connecting virtual processors with coordinates `[I]` and `[I-1]`*, and line 9 stands for the predicate ***for all*** *`I<4`* ***if*** *`I==0`* ***then*** *there exists undirected link of normal length connecting virtual processors with coordinates `[I]` and `[3]`*. Note, that if a link between two virtual processors is not specified explicitly, it is meant not absence of a link but existence of a very long link.

Line 11 is a *parent declaration*. It specifies that the parent has coordinate [0].

With the network type declaration, the user can declare a network identifier of this type. For example, the declaration

```
  net Rectangle r1;
```

introduces identifier `r1` of network.

The notion of *distributed data object* is introduced in the spirit of C* [9] and Dataparallel C [10]. Namely, a data object distributed over a region of the computing space comprises a set of components of any one type so that each virtual processor of the region holds one component. For example, the declarations

```
  net Rectangle r2;
  int [*]Derror, [r2]Da[10];
  float [host]f, [r2:I<2]Df;
  repl [*]Di;
```

declare:

- integer variable `Derror` distributed over the entire computing space;
- integer 10-member array `Da` distributed over the network `r2`;
- undistributed floating variable `f` belonging to the virtual host-processor;
- floating variable `Df` distributed over a subnetwork of network `r2`;
- integer variable `Di` replicated over the entire computing space.

By definition, a distributed object is *replicated* if all its components is equal to each other.

The notion of *distributed value* is introduced similarly.

In addition to a network type, the user can declare a parametrized family of network types called *topology* or *generic network type*. For example, the declaration

```
/*1*/     nettype Ring(n, p[n]) {
/*2*/       coord I=n;
/*3*/       node {
/*4*/         I>=0: fast*p[I] scalar;
/*5*/       };
/*6*/       link {
/*7*/         I>0:  [I]<->[I-1];
/*8*/         I==0: [I]<->[n-1];
/*9*/       };
/*10*/      parent [0];
/*11*/    };
```

introduces topology `Ring` that corresponds to networks consisting of `n` virtual processors of the `scalar` type interconnected with undirected links of normal length in a ring structure.

The header (line 1) introduces parameters of topology `Ring`, namely, integer parameter `n` and vector parameter `p` consisting of `n` integers.

Correspondingly, coordinate variable `I` ranges from `0` to `n-1`, line 4 stands for the predicate ***for all*** *`I<n`* ***if*** *`I>=0`* ***then*** *fast virtual processor of the `scalar` type, whose relative performance is specified by the value of `p[I]`, is related to the point with coordinate `[I]`*, and so on.

Here, performance specifier `fast*p[I]` includes so-called power specifier `*p[I]`. In general, the value of the expression in a power specifier shall be positive integer. Any operand in the expression should consist only of coordinate variables, constants and generic parameters. If the value of the expression is equal to 1, the power specifier may be omitted.

It is meant that in the framework of the same network-type declaration any performance specifier with the `fast` keyword specifies more powerful virtual processor than a performance specifier with the `slow` keyword. It is meant also that the greater value of the expression in a power specifier the more performance is specified.

With the topology declaration, the user can declare a network identifier of a proper type. For example, the fragment

```
  repl m, n[100];
  /* Computing m, n[0],...,n[m-1] */
  {
    net Ring(m,n) rr;
    ...
  }
```

introduces identifier `rr` of the network, the type of which is defined completely only in run time. Network `rr` consists of `m` virtual processors the relative performance of `i`-th virtual processor being characterized by the value of n[i].

A network has a computing space duration that determines its lifetime. There are 2 computing space durations: static, and automatic. A network declared with *static* computing space duration is created only once and exists till termination of the entire program. A new instance of a network declared with *automatic* computing space duration is created on each entry into the block in which it is declared. The network is discarded when execution of the block ends.

Now, let us consider a simple mpC program computing the dot product of two vectors. The program is correct but not good in the sense of efficiency.

```
/*1*/  nettype Star(n) {
/*2*/    coord I=n;
/*3*/    node { default: scalar;};
/*4*/    link { I>0: [0]<->[i];};
/*5*/    parent [0];
/*6*/  };
/*7*/  #define N 100
/*8*/  void [*]main()
/*9*/  {
/*10*/    double [host]x[N];
/*11*/    double [host]y[N];
/*12*/    double [host]z;
/*13*/    double sqrt();
/*14*/    .../*Input of x and y */
/*15*/    {
/*16*/      net Star(N) s;
/*17*/      double [s]dx, [s]dy, [s]dz;
/*18*/      dx=x[];
/*19*/      dy=y[];
/*20*/      dz=dx*dy;
/*21*/      z=[host]dz[+];
/*22*/      z=([host]sqrt)(z);
/*23*/    }
/*24*/    .../* Output of z */
/*25*/  }
```

The program includes 2 functions - main defined here and library function sqrt. Lines 8-25 contain a definition of main. Lines 10-12 contain definitions of arrays x, y and variable z all belonging to the virtual host-processor. Line 13 contains a declaration of function identifier sqrt.

In general, mpC allows 3 kinds of functions. Here, functions of two kinds are used: main is a *basic* function, and sqrt is a *nodal* function.

A call to *basic function* is executed on the entire computing space. Its arguments should either belong to the virtual host-processor or be distributed over the entire computing space, and its value should be distributed over the entire computing space. In contrast to other kinds of function, a basic function can define networks. In line 8, construct [*], placed just before the function identifier, specifies that main is an identifier of basic function.

Nodal function can be executed completely by any one virtual processor. Only local data objects of the executing virtual processor may be defined in such a function. In addition, the corresponding component of an externally-defined distributed data object can be used in the function. A declaration of nodal function (e.g., in line 13) does not need any additional specifiers.

Line 16 defines the automatic network s with the virtual host-processor as a parent.

Line 17 defines 3 automatic variables dx, dy, and dz all distributed over s.

Line 18 contains unusual unary postfix operator []. In general, its operand should either designate an array or be a pointer. In this case, expression x[] designates array x as a whole, and the statement in line 18 scatters elements of array x to components of distributed variable dx.

Similarly, the statement in line 19 scatters elements of array y to components of distributed variable dy.

The statement in line 20 is also executed on network s. But unlike 2 previous statements, its execution does not need any communications between virtual processors constituting network s. In fact, this statement is divided into a set of independent undistributed statements each of which is executed by the corresponding virtual processor using the corresponding data components. Such statement are called *asynchronous* statements. In particular, this statement multiplies (in parallel) components of dx and dy and assigns the result to components of dz.

In line 21, the result of postfix unary operator [+] is distributed over s. All its components are equal to the sum of all components of operand dz. Here, the result of prefix unary operator [host] is the component of its operand belonging to the virtual host-processor. So, the statement in line 21 assigns the sum of all components of dz to z.

Finally, line 22 calls to nodal function sqrt on the virtual host-processor and assigns the value returned to z.

To support modular parallel programming as well as the writing of libraries of parallel programs, so-called network functions are introduced in addition to basic and nodal functions.

3. The mpC programming environment

Currently, the mpC programming environment includes a compiler, a run-time support system (RTSS), a library, and a command-line user interface.

The compiler translates a source mpC program into the ANSI C program with calls to functions of RTSS.

RTSS manages the computing space which consists of a number of processes running over target DMM as well as provides communications. It has a precisely specified interface and encapsulates a particular communication package (currently, a small subset of MPI). It ensures platform-independence of the rest of system components.

The library consists of a number of functions that support debugging mpC programs as well as provide some low-level efficient facilities.

The command-line user interface consists of a number of shell commands supporting the creation of a virtual parallel machine and the execution of mpC programs on the machine. While creating the machine, its topology is detected by a topology detector running a special benchmark and saved in a file used by RTSS.

Our compiler uses optionally either the SPMD model of target code, when all processes constituting a target message-passing program run identical code, or a quasi-SPMD model, when it translates a source mpC file into 2 separate target files - the first for the virtual host-processor and the second for the rest of virtual processors.

All processes constituting the target program are divided into 2 groups - the special process called *dispatcher* playing the role of the computing space manager, and general processes called *nodes* playing the role of virtual processors of the computing space. The special node called *host* is separated. The dispatcher works as a server accepting requests from nodes. The dispatcher does not belong to the computing space.

In the target program, every network or subnetwork of the source mpC program is represented by a set of nodes called *region*. At any time of the target program running, any node is either free or hired in one or several regions. Hiring nodes in created regions and dismissing them are responsibility of the dispatcher. The only exception is the pre-hired host-node representing the mpC pre-defined virtual host-processor. Thus, just after initialization, the computing space is represented by the host and a set of temporarily free (unemployed) nodes.

Creation of the network region involves the parent node, the dispatcher and all free nodes. The parent node sends a creation request containing the necessary information about the network topology to the dispatcher. Based on this information and the information about the topology of the virtual parallel machine, the dispatcher selects the most appropriate set of free nodes. After that, it sends to every free node a message saying whether the node is hired in the created region or not. Deallocation of network region involves all its members as well as the dispatcher.

The dispatcher keeps a queue of creation requests that cannot be satisfied immediately but can be served in the future. It implements some strategy of serving the requests aimed at minimization of the probability of occurring a deadlock. The dispatcher detects such a situation when the sum of the number of free nodes and the number of such hired nodes that could be released is less than the minimum number of free nodes required by a request in the queue. In this case, it terminates the program abnormally specifying a deadlock.

4. Irregular applications

4.1 Programming in mpC

Let us consider an irregular application simulating the evolution of a system of stars in a galaxy (or set of galaxies) under the influence of Newtonian gravitational attraction.

Let our system consist of a number of large groups of bodies. It is known, that since the magnitude of interaction between bodies falls off rapidly with distance, the effect of a large group of bodies may be approximated by a single equivalent body, if the group of bodies is far enough away from the point at which the effect is being evaluated. Let it be true in our case. So, we can parallelize the problem, and our application will use a few virtual processors, each of which updates data characterizing a single group of bodies. Each virtual processor holds attributes of all the bodies constituting the corresponding group as well as masses and centers of gravity of other groups. The attributes characterizing a body include its position, velocity and mass.

Finally, let our application allow both the number of groups and the number of bodies in each group to be defined in run time.

The application implements the following scheme:

```
Initializing the galaxy
          on the virtual host-processor
Creation of the network
Scattering groups over
          virtual processors
Parallel computing masses of groups
Interchanging the masses among
          virtual processors
while(1) {
  Visualization of the galaxy
          on the virtual host-processor
  Parallel computation of centers of
          gravity of groups
  Interchanging the centers among
          virtual processors
  Parallel updating groups
  Gathering groups
          on the virtual host-processor
}
```

The corresponding mpC program looks as follows:

```
#define DELTA 3600.0
#define INTERVAL 3

/*The maximum number of groups*/
#define MaxGs 30
```

```
/*The maximum number of bodies in a group*/
#define MaxBs 600

typedef double Triplet[3];
typedef
  struct {Triplet pos; Triplet v; double m;}
  Body;

/*The number of groups*/
int [host]M;

/*The numbers of bodies in groups*/
int [host]N[MaxGs];

repl dM, dN[MaxGs];

/*The galaxy timer*/
double [host]t;

/*Bodies of a galaxy*/
Body (*[host]Galaxy[MaxGs])[MaxBs];

nettype GalaxyNet(m, n[m]) {
  coord I=m;
  node { I>=0: fast*n[I] scalar;};
  link (J=m){
    J>0: length*(-1) [J]->[0];
    J>0: length*1    [I]->[J];
  };
};

void [host]Input(), UpdateGroup();
void [host]VisualizeGalaxy();

void [*]Nbody(char *[host]infile)
{
  /*Initializing Galaxy, M and N*/
  Input(infile);

  /*Broadcasting the number of groups*/
  dM=M;

  /*Broadcasting the numbers of bodies*/
  /*in groups*/
  dN[]=N[];
  {
    net GalaxyNet(dM,dN) g;
    int [g]myN, [g]mycoord;
    Body [g]Group[MaxBs];
    Triplet [g]Centers[MaxGs];
    double [g]Masses[MaxGs];
    repl [g]i;
    void [net GalaxyNet(m, n[m])]Mintegrity
      (double (*)[MaxGs]);
    void [net GalaxyNet(m, n[m])]Cintegrity
      (Triplet (*)[MaxGs]);

   mycoord = I coordof body_count;
    myN = dN[mycoord];

    /*Scattering groups*/
    for(i=0; i<[g]dM; i++)
      [g:I==i]Group[] = (*Galaxy[i])[];

    for(i=0; i<myN; i++)
      Masses[mycoord]+=Group[i].m;
    ([([g]dM,[g]dN)g])Mintegrity(Masses);
    while(1) {
      if(((int)(t/DELTA))%INTERVAL==0)
        VisualizeGalaxy();
      Centers[mycoord][]=0.0;
      for(i=0; i<myN; i++)
        Centers[mycoord][] +=
          (Group[i].m/Masses[mycoord])*
          (Group[i].pos)[];
      ([([g]dM,[g]dN)g])Cintegrity(Centers);
      ([g]UpdateGroup)(Centers, Masses,
                       Group, [g]dM);
      t+=DELTA;
      if(((int)(t/DELTA))%INTERVAL==0)
        /*Gathering groups*/
        for(i=0; i<[g]dM; i++)
          (*Galaxy[i])[]=[g:I==i]Group[];
    }
  }
}

void [net GalaxyNet(m,n[m]) p] Mintegrity
  (double (*Masses)[MaxGs])
{
  double MassOfMyGroup;
  repl i, j;
  MassOfMyGroup=(*Masses)[I coordof i];
  for(i=0; i<m; i++)
    for(j=0; j<m; j++)
      [p:I==i](*Masses)[j] =
        [p:I==j]MassOfMyGroup;
}

void [net GalaxyNet(m,n[m]) p] Cintegrity
  (Triplet (*Centers)[MaxGs])
{
  Triplet MyCenter;
  repl i, j;
  MyCenter[] = (*Centers)[I coordof i][];
  for(i=0; i<m; i++)
    for(j=0; j<m; j++)
      [p:I==i](*Centers)[j][] =
        [p:I==j]MyCenter[];
}
```

This mpC source file contains the following external definitions:

- definitions of variables M, t and arrays N, Galaxy all belonging to the virtual host-processor;

- a definition of variable dM and array dN both replicated over the entire computing space;
- a definition of network type GalaxyNet;
- a definition of basic function Nbody with one formal parameter infile belonging to the virtual host-processor;
- definitions of network functions Mintegrity and Cintegrity.

In general, a *network function* is called and executed on some network or subnetwork, and its value is also distributed over this region of the computing space. The header of the definition of the network function either specifies an identifier of a global static network or subnetwork, or declares an identifier of the network being a special formal parameter of the function. In the first case, the function can be called only on the specified region of the computing space. In the second case, it can be called on any network or subnetwork of a suitable type. In any case, only the network specified in the header of the function definition may be used in the function body. No network can be declared in the body. Only data objects belonging to the network specified in the header may be defined in the body. In addition, corresponding components of an externally-defined distributed data object may be used. Unlike basic functions, network functions (as well as nodal functions) can be called in parallel.

Network functions Input and VisualizeGalaxy, both associated with the virtual host-processor, as well as the nodal function UpdateGroup are declared and called here.

Automatic network g, executing most of computations and communications, is defined in such a way, that it consists of M virtual processors, and the relative performance of each processor is characterized by the number of bodies in the group which it computes.

So, the more powerful is the virtual processor, the larger group of bodies it computes, and the more intensive is the data transfer between two virtual processors, the shorter link connects them (length specifier length*(-1) specifies a shorter link than length*1 does).

The mpC programming environment bases on this information to map the virtual processors constituting network g into the processes constituting the entire computing space in the most appropriate way. Since it does it in run time, the user does not need to recompile this mpC program, to port it to another DMM.

The result of the binary operator coordof (in the first statement of the inner block of function Nbody) is an integer value distributed over g, each component of which is equal to the value of coordinate I of the virtual processor to which the component belongs. The right operand of operator coordof is not evaluated and used only to specify a region of the computing space. Note, that coordinate variable I is treated as an integer variable distributed over the region.

Call expression ([g]UpdateGroup)(....) causes parallel execution of nodal function UpdateGroup on each of virtual processors of network g. It is meant, that function name UpdateGroup is converted to a pointer-to-function distributed over the entire computing space, and operator [g] cuts from this pointer a pointer distributed over g. So, the value of expression [g]UpdateGroup is a pointer-to-function distributed over g. Therefore, expression ([g]UpdateGroup)(....) denotes a distributed call to a set of undistributed functions.

Network functions Mintegrity and Cintegrity have 3 special formal parameters. Network parameter p denotes the network executing the function. Parameter m is treated as a replicated over p integer variable, and parameter n is treated as a pointer to the initial member of an integer unmodifiable m-member array replicated over p. The syntactic construct ([(dM,dN)g]), placed on the left of the name of the function called in the call expressions in function Nbody, just specifies the actual arguments corresponding to the special formal parameters.

4.2 Experimental results

We compared the running time of our mpC program to its carefully written MPI counterpart. We use 3 workstations - SPARCstation 5 (hostname gamma), SPARCclassic (omega), and SPARCstation 20 (alpha), connected via 10Mbits Ethernet. There were 23 other computers in the same segment of the local network. We used LAM MPI version 5.2 [12] as a particular communication platform.

The computing space of the mpC programming environment consists of 15 processes, 5 processes running on each workstation. The dispatcher runs on gamma and uses the following relative performances of the workstations obtained automatically upon the creation of the virtual parallel machine: 1150 (gamma), 331 (omega), 1662 (alpha).

The MPI program is written in such a way to minimize communication overheads. All our experiments deal with 9 groups of bodies. We map 3 MPI processes to gamma, 1 process to omega, and 5 processes to alpha, providing the optimal mapping if the numbers of bodies in these groups are equal to each other.

The first experiment compares the mpC and MPI programs for homogeneous input data when all groups consist of the same number of bodies. Figure1 shows the running time of both programs simulating 15 hours of the galaxy evolution depending on the number of bodies in groups.

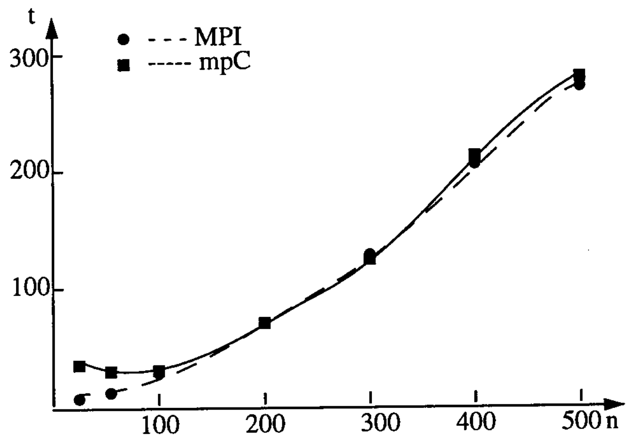


Figure 1. Running time of the MPI and mpC programs for homogenous input data.

In fact, it shows how much we pay for the usage of mpC instead of pure MPI. One can see that the running time of the MPI program consists about 95-97% of the running time of the mpC program. That is, in this case we loose 3-5% of performance.

The second experiment compares these programs for heterogeneous input data. Let our groups consist of 10, 10, 10, 100, 100, 100, 600, 600, and 600 bodies correspondingly.

The running time of the mpC program does not depend on the order of the numbers. In any case, the dispatcher selects:

- 4 processes on gamma for virtual processors of network g computing two 10-body groups, one 100-body group, and one 600-body group;

- 3 processes on omega for virtual processors computing one 10-body group and two 100-body groups;

- 2 processes on alpha for virtual processors computing two 600-body groups.

The mpC program takes 94 seconds to simulate 15 hours of the galaxy evolution.

The running time of the MPI program essentially depends on the order of these numbers. It takes from 88 to 391 seconds to simulate 15 hours of the galaxy evolution depending on the particular order. Figure 2 shows the relative running time of the MPI and mpC programs for different permutations of these numbers. All possible permutations can be broken down into 24 disjoint subsets of the same power in such a way that if two permutations belong to the same subset, the corresponding running time is equal to each other. Let these subsets be numerated so that the greater number the subset has, the longer time the MPI program takes. In figure 2, each such a subset is represented by a bar, the height of which is equal to the corresponding value of t_{MPI}/t_{mpC}.

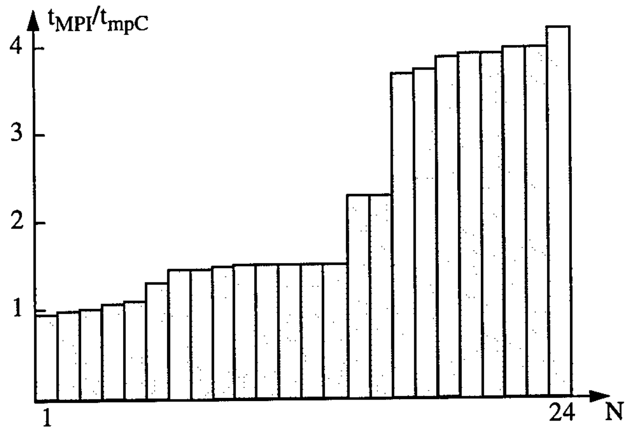


Figure 2. The relative running time for different permutations of the numbers of bodies in groups.

One can see that almost for all input data the running time of the MPI program exceeds (and often, essentially) the running time of the mpC program.

5. Regular applications

5.1 Programming in mpC

Let us consider a regular application multiplying 2 dense square nxn matrices X and Y.

Our mpC program will use a number of virtual processors, each of which computes a number of rows of the resulting matrix Z. Both dimension n of matrices and the number of virtual processors involved in computations are defined in run time. So, our application implements the following scheme:

```
Initializing X and Y
    on the virtual host-processor
Creating a network
Scattering rows of X over
    virtual processors of the network
Broadcasting Y over
    virtual processors of the network
Parallel computing submatrices of Z
Gathering the resulting matrix Z
    on the virtual host-processor
```

The corresponding mpC program looks as follows:

```
/*1*/    nettype SimpleNet(n) {
/*2*/     coord I=n;
/*3*/    };

/*4*/    nettype Star(m, n[m]) {
/*5*/     coord I=m;
/*6*/     node {I>=0: fast*n[I] scalar;};
/*7*/    link {I>0:  [I]->[0], [0]->[I];};
/*8*/    parent [0];
/*9*/   };

/*10*/  void [*]MxM(float *x, float *y,
/*11*/              float *z, repl n) {
/*12*/   repl double *powers;
/*13*/   repl nprocs, nrows[MAXNPROCS], n;
/*14*/
/*15*/   MPC_Processors_static_info
/*16*/                  (&nprocs,&powers);
/*17*/   Partition(nprocs,powers,nrows,n);
/*18*/   {
/*19*/    net Star(nprocs, nrows) w;
/*20*/    ([([w]nprocs)w])ParMult(
/*21*/      [w]x,[w]y,[w]z,[w]nrows,[w]n);
/*22*/   }
/*23*/  }

/*24*/  void [net SimpleNet(p)v] ParMult(
/*25*/   float *dx, float *dy, float *dz,
/*26*/   repl *r, repl n)
/*27*/  {
/*28*/   repl s=0;
/*29*/   int myn, i;
/*30*/   int *d, *l, c;
/*31*/
/*32*/   myn=r[I coordof r];
/*33*/   ([(p)v])MPC_Bcast(&s, dy, 1,
/*34*/                     n*n, dy, 1);
/*35*/   d=calloc(p, sizeof(int));
/*36*/   l=calloc(p, sizeof(int));
/*37*/   for(i=0, d[0]=0; i<p; i++) {
/*38*/     l[i]=r[i]*n;
/*39*/    if(i+1<p) d[i+1]=l[i]+d[i];
/*40*/   }
/*41*/   c=l[I coordof c];
/*42*/   ([(p)v])MPC_Scatter(&s, dx ,d,
/*43*/                        l, c, dx);
/*44*/   SeqMult(dx, dy, dz, myn, n);
/*45*/  ([(p)v])MPC_Gather(&s,dz,d,l,c,dz);
/*46*/  }

/*47*/ void SeqMult(float *a, float *b,
/*48*/              float *c, int m, int n)
/*49*/ {
/*50*/  int i, j, k, ixn;
/*51*/  double s;
/*52*/
/*53*/  for(i=0; i<m; i++)
/*54*/   for(j=0, ixn=i*n; j<n; j++) {
/*55*/    for(k=0, s=0.0; k<n; k++)
/*56*/     s+=a[ixn+k]*(double)(b[k*n+j]);
/*57*/    c[ixn+j]=s;
/*58*/   }
/*59*/ }

/*60*/  void Partition(int p, double *v,
/*61*/                 int *r, int n)
/*62*/  {
/*63*/   int sr, i;
/*64*/   double sv;
/*65*/
/*66*/   for(i=0, sv=0.0; i<p; i++)
/*67*/    sv+=v[i];
/*68*/   for(i=0, sr=0; i<p; i++) {
/*69*/    r[i]=(int)(v[i]/sv*n);
/*70*/    sr+=r[i];
/*71*/   }
/*72*/   if(sr!=n) r[0]+=n-sr;
/*73*/  }
```

Formal parameters x, y, and z of basic function MxM are distributed over the entire computing space, and parameter n is replicated over the entire computing space. It is meant that n holds the dimension of matrices. It is also meant that x points to nxn-member array, and the component of this distributed array belonging to the virtual host-processor holds matrix X. Similarly, [host]y points to an array holding matrix Y, and [host]z points to an array holding resulting matrix Z.

Lines 15-16 calls to library nodal function MPC_Processors_static_info on the entire computing space returning the number of actual processors and their relative performances. So, after this call replicated variable nprocs will hold the number of actual processors, and replicated array powers will hold their relative performances.

Line 17 calls to nodal function Partition on the entire computing space. Based on relative performances of actual processors, this function computes how many rows of the resulting matrix will be computed by every actual processor. So, after this call nrows[i] will hold the number of rows computed by i-th actual processor.

Line 19 defines automatic network w. Its type is defined completely only in run time. Network w, which executes the rest of computations and communications, is defined in such a way, that the more powerful the virtual processor, the greater number of rows it computes. The mpC environment will ensure the optimal mapping of the virtual processors constituting w into a set of processes constituting the entire computing space. So, just one process from processes running on each of actual processors will be involved in multiplication of matrices, and the more powerful the actual processor, the greater number of rows its process will compute.

Lines 20-21 call to network function ParMult on network w. In this call, topological argument [w]nprocs specifies a network type as an instance of parametrized network type SimpleNet, and network argument w specifies a region of the computing space treated by function `ParMult` as a network of this type.

In lines 24-26, the header of the definition of function `ParMult` declares identifier `v` of a network being a special network formal parameter of the function. Since network `v` has a parametrized type, topological parameter `p` is also declared in this header. In the function body, special formal parameter `p` is treated as an unmodifiable variable of type `int` replicated over network formal parameter `v`. The rest of formal parameters (regular formal parameters) of the function are also distributed over `v`.

Actually, `p` holds the number of virtual processors in network `v`, `n` holds the dimension of matrices, `r` points to `p`-member array, `i`-th element of which holds the number of rows of the resulting matrix that `i`-th virtual processor of network `v` computes. Each component of `dy` points to an array to contain `nxn` matrix `Y`. Each component of dz points to an array to contain the rows of `Z` computed on the corresponding virtual processor of v. Each component of dx points to an array to contain the rows of `X` used in computations on the corresponding virtual processor. In addition, throughout the function execution the components of `dx`, `dy`, `dz` belonging to the parent of network `v` are reputed to point to arrays holding matrices `X`, `Y` and `Z` correspondingly.

Line 28 defines variable `s` replicated over v. Lines 29-30 define variables `myn`, `i`, `d`, `l` and `c` all distributed over v.

After execution of the asynchronous statement in line 32, each component of `myn` will contain the number of rows of the resulting matrix that computes the corresponding virtual processor.

Lines 33-34 call to so-called *embedded* network function `MPC_Bcast` which is declared in a standard mpC header as follows:

```
int [net SimpleNet(n)] MPC_Bcast(
     repl const *coordinates_of_source,
     void *source_buffer,
     const source_step,
     repl const count,
     void *destination_buffer,
     const destination_step);
```

This call broadcasts matrix `Y` from the parent of v to all virtual processors of v. As a result, each component of the distributed array pointed by `dy` will contain this matrix.

An embedded network function looks like a library network function, but a compiler knows its semantics. In particular, it will generate different code for different types of arguments corresponding to source and destination buffers.

Statements in lines 35-40 are asynchronous. They form two `p`-member arrays `d` and `l` distributed over `v`. After their execution, `l[i]` will hold the number of elements in the portion of the resulting matrix which is computed by the `i`-th virtual processor of v, and `d[i]` will hold the displacement which corresponds to this portion in the resulting matrix. Equivalently, `l[i]` will hold the number of elements in the portion of matrix `X` which is used by `i`-th virtual processor of v, and `d[i]` will hold the displacement which corresponds to this portion in matrix `X`.

The statement in line 41 is also asynchronous. After its execution, each component of `c` will hold the number of elements in the portion of the resulting matrix which is computed by the corresponding virtual processor (equivalently, the number of elements in the portion of matrix `X` which is used by this virtual processor).

Lines 42-43 call to embedded network function `MPC_Scatter` which is declared as follows:

```
int [net SimpleNet(n) w] MPC_Scatter(
     repl const *coordinates_of_source,
     void *source_buffer,
     const *displacements,
     const *sendcounts,
     const receivecount,
     void *destination_buffer);
```

This call scatters matrix `X` from the parent of v to all virtual processors of v. As a result, each component of dx will point to an array containing the corresponding portion of matrix X.

Line 44 calls to nodal function `SeqMult` on v, computing the corresponding portions of the resulting matrix on each of its virtual processors in parallel (`SeqMult` implements traditional sequential algorithm of matrix multiplication).

Finally, line 45 calls to embedded network function `MPC_Gather` which is declared as follows:

```
int [net SimpleNet(n) w] MPC_Gather(
   repl const *coordinates_of_destination,
   void *destination_buffer,
   const *displacements,
   const *receivecounts,
   const sendcount,
   void *source_buffer);
```

This call gathers resulting matrix `Z` each virtual processor of v sending its portion of the result to the parent of v.

5.2 Experimental results

We measured the running time of our mpC program multiplying two dense square matrices. We used three Sun SPARCstations 5 (hostnames `gamma`, `beta`, and `delta`), SPARCclassic (`omega`), and HP 9000-712 (`zeta`) connected via 10Mbits Ethernet. There were more than 20 other computers in the same segment of the local network.

We used LAM MPI Version 6.0 as a particular communication platform as well as a new improved benchmark for detecting relative performances of workstations. In addition, all executables, which took part in the experiment, were generated by GNU C compiler with optimization option -O2.

Eight virtual parallel machines were created:

- g consisting of gamma (its relative performance detected during the creation of this virtual parallel machine was equal to 324);
- gd consisting of gamma (323), and delta (330);
- gbd consisting of gamma (324), beta (331), and delta (331);
- gbdz consisting of gamma (324), beta (327), delta (330), and zeta (510);
- zg consisting of zeta (510), and gamma (323);
- zgb consisting of zeta (509), gamma (321), and beta (325);
- zgbd consisting of zeta (466), gamma (328), beta (327), and delta (329);
- zo consisting of zeta (506), and omega (147).

The computing space of each of these virtual parallel machines was constituted by 5 processes running on each of workstations (that is, for example, the computing space of gbdz was constituted by 20 processes). As a base of the comparison we used the running time of a sequential C program implementing the same algorithm which was used in function SeqMult.

Table 1 gives the time of running the mpC program on four virtual parallel machines (g, gd, gbd, and gbdz) dependent on the dimension of multiplied matrices, and compares it to the time of running the sequential C program on workstation gamma. Machines g, gd, and gbd are homogeneous ones, meantime machine gbdz is heterogeneous.

Figure 3 illustrates how the mpC program allows to speed up the multiplication of two dense square matrices, if the user starts from single workstation gamma and enhances his computing facilities step by step by means of adding workstations delta, beta and zeta.

Table 1: Time to multiply two nxn matrices (sec)

n	g	g	gd	gbd	gbdz
	C	mpC	mpC	mpC	mpC
100	0.32	0.40	0.53	0.61	0.70
200	2.55	2.61	2.00	1.91	2.05
300	9.33	9.66	6.11	5.25	4.96
400	31.2	32.2	17.9	13.9	11.6
500	54.7	55.6	31.0	23.4	19.0
600	125.	125.	68.0	49.0	37.0
700	196.	196.	106.	75.0	58.0
800	320.	323.	172.	123.0	88.

Note, that the running time of the mpC program substantially depends on the network load. We monitored the network activity during our experiments. We have observed up to 32 collisions per second. The collisions occurred more often during broadcasting large data portions. The collisions resulted in visible degradation of the network bandwidth.

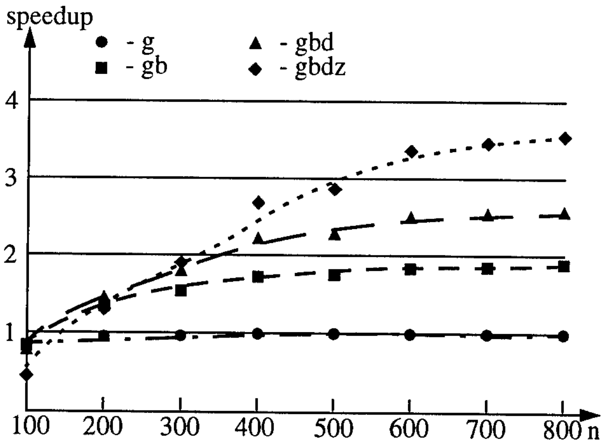


Figure 3. Speedups computed relative to sequential code running on workstation gamma.

Table 2 compares contribution of communications and computations in the total running time of the mpC program (results for gbdz are presented). The first column shows matrix dimensions, an the second one shows percentage of communications in the total running time.

Table 2:Contribution of communications in the total running time (gbdz)

n	Communications (%)
100	40
200	55
300	48
400	38
500	35
600	26
700	24
800	21

Communications in our mpC program consist of three parts: scattering matrix X, broadcasting matrix Y, and gathering the resulting matrix. Table 3 compares contribution of each of these parts in the total communication time (for the gbdz virtual parallel machine).

While analyzing the presented results, it is necessary to take into account some peculiarities of both the implementation of MPI, which we used, and our local network.

Our local network does not support fast communications. It is based on 10Mbits Ethernet and uses old-fashioned network equipment. In addition, there are 26 computers in our segment of the network connected via cascade of 4 hubs. To characterize our network, it is enough to say that ftp transfers data from gamma to alpha at the rate of 300-400Kbytes/s. It means that real bandwidth of our network is about 25-30% of its maximum bandwidth.

Table 3: Contribution of broadcast, scatter, and gather in the total communication time (gbdz)

n	bcast	scatter	gather
100	70%	18%	12%
200	78%	11%	11%
300	78%	10%	12%
400	79%	10%	11%
500	79%	10%	11%
600	79%	10%	11%
700	79%	10%	11%
800	76%	13%	11%

On the other hand, LAM MPI Version 6.0 ensures sending large floating arrays at the rate of 50-60Kbytes/s. In addition, it doesn't use multicasting facilities of our network when broadcasting.

Nevertheless, even under these conditions, our mpC program has demonstrated good speedup comparing with the sequential C program.

If the implementation of MPI ensured the communication rate comparable with the real bandwidth of the local network and used its multicasting facilities, contribution of communications in the total running time of our mpC program would not exceed 5-7%. If, in addition, we used 100Mbits Ethernet and up-to-date network technologies (for example, replaced hubs with switching devices), contribution of communications in the total running time of the mpC program would not exceed 1-2%. That is, the mpC programming environment can ensure practically ideal speedup of the presented mpC program for up-to-date heterogeneous networks of workstations.

Table 4 gives the time of running the mpC program on four heterogeneous virtual parallel machines (zg, zgb, zgbd, and zo) dependent on the dimension of multiplied matrices, and compares it to the time of running the sequential C program on workstation zeta.

Table 4: Time to multiply two nxn matrices (sec)

n	z	zg	zgb	zgbd	zo	zo
	C	mpC	mpC	mpC	mpC	MPI
100	0.18	0.43	0.52	0.57	0.67	0.91
200	1.52	1.67	1.70	1.79	2.36	4.29
300	6.80	5.66	5.08	4.90	7.09	14.2
400	17.3	14.2	11.7	11.1	16.4	33.0
500	36.2	26.0	21.0	19.0	32.8	68.0
600	66.8	53.0	41.0	37.0	58.5	120.
700	113.	83.0	64.0	56.0	97.0	200.
800	180.	134.	102.	88.0	152.	306.

In addition, the table compares the mpC program and its manually written MPI counterpart on machine zo.

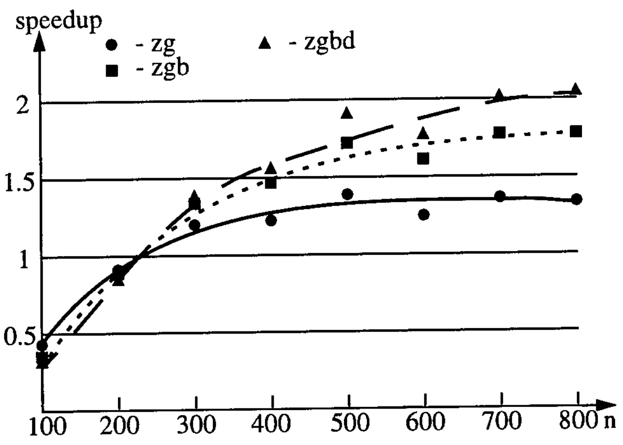


Figure 4. Speedups computed relative to sequential code running on workstation zeta.

Figure 4 illustrates how the mpC program allows to speed up the multiplication of two dense square matrices, if the user starts from single powerful workstation zeta and enhances his computing facilities step by step by means of adding less powerful workstations gamma, beta, and delta. One can see that the mpC programming environment ensures good speedup in this case also.

Another interesting result can be extracted from tables 1 and 4. One can see that the slow network consisting of workstations gamma and delta (virtual parallel machine gd), the performance each of which is about 60% of the performance of workstation zeta, demonstrates a little bit higher performance (when multiplying two dense square matrices) than single workstation zeta.

Finally, figure 5 shows clearly, that even for very heterogeneous distributed memory machine consisting of high-performance HP workstation zeta and low-performance Sun workstation omega, the mpC program allows to utilize its parallel potential, speeding up the multiplication of two dense square matrices (comparing to the sequential C program running on zeta). At the same time, the use of its MPI counterpart, which distributes the workload equally, does not allow to do it slowing down the matrix multiplication essentially.

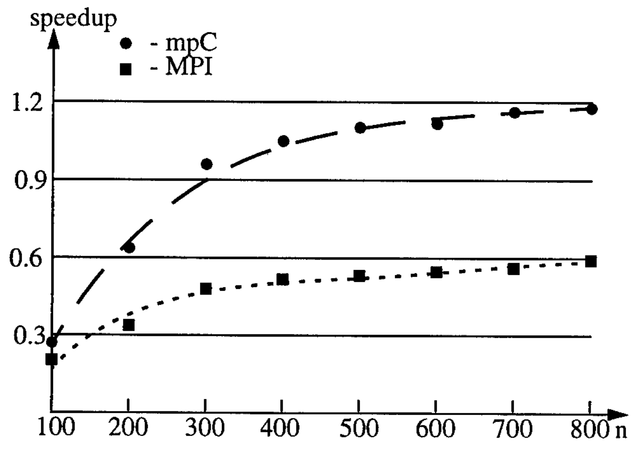


Figure 5. Speedups for MPI and mpC programs both running on machine zo.

6. Summary

The key peculiarity of mpC is its advanced facilities for managing such resources of DMMs as processors and links between them. They allow to develop parallel programs for DMMs that once compiled will run efficiently on any particular DMM, because the mpC programming environment ensures optimal distribution of computations and communications over DMM in run time.

The mpC language is a medium-level one. It demands from the user more than high-level parallel languages (say, Fortran D), but much less than MPI or PVM.

Like MPI and PVM, mpC supports efficient programming a particular DMM. Like MPI, the user does not need to rewrite (and, moreover, to recompile) an mpC program to port it to other DMMs.

At the same time, MPI (as well as MPI-2) does not ensure efficient porting to other DMMs, that is, it does not ensure, that a program, running efficiently on a particular DMM, will run efficiently after porting to other DMM. The mpC language and its programming environment do it.

Advantages of mpC are especially clear when programming heterogeneous (irregular) applications or/and programming for heterogeneous DMM.

It makes mpC and its programming environment suitable tools for development of libraries of parallel programs, especially for heterogeneous DMMs.

The paradigm of parallel programming, supported by mpC, foresees explicit specification of a virtual parallel machine executing computations and communications. At the same time, mpC also supports implicit parallel programming, when parallelism is reduced to calls to library basic functions (like function `Nbody` from section 4.1) that just encapsulate parallelism.

Acknowledgments

The work was supported by Office of Naval Research (USA) and partially by Russian Basic Research Foundation.

References

[1] V. Sunderam, "PVM: A framework for parallel distributed computing", *Concurrency: Practice and Experience,* 2(4), 1990, pp.315-339.

[2] Message Passing Interface Forum, "MPI: A Message-passing Interface Standard", *International Journal of Supercomputer Applications*, 8(3/4), 1994.

[3] I. Foster, and K. M. Chandy, *Fortran M: a language for modular parallel programming. Preprint MCS-P327-0992,* Argonne National Lab, 1992.

[4] K. M. Chandy, and C. Kesselman, *CC++: A Declarative Concurrent Object Oriented Programming Language. Technical Report CS-TR-92-01,* California Institute of Technology, Pasadena, California, 1992.

[5] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng, and M.-Y. Wu, *Fortran D Language Specification. Center for Research on Parallel Computation,* Rice University, Houston, TX, October 1993.

[6] B. Chapman, P. Mehrotra, and H. Zima, "Programming in Vienna Fortran", *Scientific Programming,* 1(1), 1992, pp.31-50.

[7] High Performance Fortran Forum., *High Performance Fortran language specification, version 1.0.* Rice University, Houston, TX, May 1993.

[8] *CM-5 Technical Summary. The CM Fortran Programming Language,* Thinking Machines Corp., November. 1992.

[9] *CM-5 Technical Summary. The C* Programming Language,* Thinking Machines Corporation, November 1992.

[10] P. J. Hatcher, and M. J. Quinn, *Data-Parallel Programming on MIMD Computers.* The MIT Press, Cambridge, MA, 1991.

[11] M. Philippsen, and W. Tichy, "Modula-2* and its compilation", *First International Conference of the Austrian Center for Parallel Computation,* Salzburg, Austria, 1991.

[12] *Trollius LAM MPI Version 5.2.* Ohio State University, 1994.

[13] A. Lastovetsky, *The mpC Programming Language Specification. Technical Report,* Institute for System Programming, Russian Academy of Sciences, Moscow, December 1994.

[14] A. Lastovetsky, "mpC - a Multi-Paradigm Programming Language for Massively Parallel Computers", *ACM SIGPLAN Notices, 31(2),* February 1996, pp.13-20.

[15] D. Arapov, A. Kalinov, and A. Lastovetsky, "Managing the Computing Space in the mpC Compiler", *Proceedings of the 1996 Parallel Architectures and Compilation Techniques (PACT'96) conference,* Boston, October 1996.

[16] D. Arapov, A. Kalinov, and A. Lastovetsky, "Resource Man-

agement in the mpC Programming Environment", *Proceedings of the 30th Hawaii International Conference on System. Sciences (HICSS'30)*, IEEE Computer Society Press, Maui, HI, January 1997.

[17] D. Arapov, A. Kalinov, A. Lastovetsky, I. Ledovskih, and T. Lewis, "A Parallel Language for Modular Distributed Programming", *Proceedings of the 2nd Aizu International Symposium on Parallel Algorithms/Architectures Synthesis (pAs'97)*, IEEE Computer Society Press, Aizu-Wakamatsu, Japan, March 1997.

Dmitry M.Arapov works for the Institute for System Programming, Russian Academy of Sciences. His research interests include parallel and distributed programming, compilers, object-oriented programming. He received his MS in mathematics from the Moscow State University in 1984. He teaches at the Moscow State University. Previously, he worked for the Keldysh Institute, Russian Academy of Sciences, where he took part in the development of software for Russian aerospace project "Buran".

Alexey Ya. Kalinov is a senior researcher at the Institute for System Programming, Russian Academy of Sciences. His research interests are in parallel and distributed programming and computer modelling man-steering vehicles. He received his MS in mathematics and engineering from the Moscow Aviation Institute in 1980, and his PhD in engineering from the Heavy-Machinery Research Institute in 1990. Previously, he took part in the implementation of the SDL language under a contract with BNR (Canada).

Alexey L. Lastovetsky is a leading researcher at the Institute for System Programming, Russian Academy of Sciences. His research interests include parallel and distributed programming, programming languages, compilers, and theory of programming languages. He received his MS in mathematics and engineering and PhD in computer science from the Moscow Aviation Institute in 1980 and 1985, respectively. Earlier, he has developed an algebraic approach to semantics of programming languages and an ANSI C superset for vector and superscalar computers. He teaches at the Moscow State University and at the Moscow Institute of Physics and Technology. He is on the editorial board of *Programmirovanie* (a journal of Russian Academy Sciences on computer science translated as *Programming and Computer Software*). He is on the advisory committee of the software track of 30th and 31st Hawaii International Conferences on System Sciences.

Ilya N. Ledovskih is a researcher at the Institute for System Programming, Russian Academy of Sciences. His research interests include programming languages and compilers. He received his MS in mathematics and engineering from the Moscow Aviation Institute in 1990. Previously, he took part in implementation of Fortran 77 for Russian Cray-like supercomputer "Elektronika SSBIS" as well as was one of most principal contributors in the implementation of an ANSI C superset for vector and superscalar computers.

Theodor G. Lewis is professor and chair of computer science at the Naval Postgraduate School in Monterey, California. A past editor-in-chief of both *IEEE Computer* and *IEEE Software*, he has published extensively in the areas of parallel computing and real-time software engineering. He received a BS in mathematics from Oregon State University in 1966 and MS and PhD degrees from Washington State University in 1970 and 1971, respectively. He is a member of the IEEE Computer Society.

UbiWorld: An Environment Integrating Virtual Reality, Supercomputing, and Design

Terrence Disz, Michael E. Papka, and Rick Stevens
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439
{disz,papka,stevens}@mcs.anl.gov

Abstract

UbiWorld is a concept being developed by the Futures Laboratory group at Argonne National Laboratory that ties together the notion of ubiquitous computing *(Ubicomp) with that of using virtual reality for rapid prototyping. The goal is to develop an environment where one can explore Ubicomp-type concepts without having to build real Ubicomp hardware. The basic notion is to extend object models in a virtual world by using distributed wide area heterogeneous computing technology to provide complex networking and processing capabilities to virtual reality objects.*

1 Introduction

In the Futures Laboratory [1] in the Mathematics and Computer Science (MCS) Division at Argonne National Laboratory (ANL), our research agenda is driven partly by discussions of advanced computing scenarios. We find that by suspending disbelief momentarily and by engaging in serious discussion of such topics as off-planet infrastructure, green nomadic computing, and molecular nanotechnology, we are able to project beyond the current set of problems and to conceptualize innovative solutions.

UbiWorld is a result of this fertile ground where concepts converge and evolve. This convergence is evident in the off-planet infrastructure problem, or People to Mars scenario. It's a safe assumption that support for people on Mars will be primarily computing, that the computing will be ubiquitous and nearly invisible, and that "green" technology will be used to minimize power consumption will be important, as will the deployment of nanotechnology to manufacture devices. On Mars, computers will always outnumber people. Computers will undoubtedly be heterogeneous, it being hard to imagine a single architecture deployed in devices from gloves and boots to landers, flight control decks, and mining machines. Computers will need to be transparently interconnected; reliability and fault tolerance will be critical; and programming and code maintenance will be significant activities. Everything of value will be available on mars.net from anywhere on the planet. Immersive telepresence will be a critical capability to overcome the obstacle of distance.

These requirements push the boundaries of computing and networking as we know them and as we can imagine them in the near future. Today, we don't even have the tools to experiment with implementations of some of these ideas. We can, however, conduct experiments in a virtual world, creating and designing objects out of "pure thought-stuff," to borrow a phrase from Frederick Brooks. This is the concept behind UbiWorld.

2 Ubiquitous Computing

In the beginning, there were mainframe computers. Access to mainframes has historically been characterized by many people per computer, batch operations, text input, and paper output. Today, we are living in the era of the personal computer. Personal computer use is characterized by one person per computer, multithreaded interactive use, multimedia, windows, and mouse interface. The next wave of computing will be ubiquitous computing, characterized by many computers per person and a transparent interface, used to amplify one's powers, not replace them. Ubiquitous computing means computers will become as invisible to us today as text is [2]. There was a time when the written word was the sole province of the experts, guarded and used sparingly, much as computing has been. Text technology has undergone a transformation from being written on clay tablets, then coarse paper, up to today's refined paper and display technology. Believers in ubiquitous computing see a day when the same transformation will occur with respect to computing; users will not be any more aware of the computers in their lives than we are aware today of

U.S. Government Work Not Protected by U.S. Copyright

the text in which this document is written. We are already beginning to see this happen with the integration of computers in automobiles: the driver is really unaware of the computer and its function.

The ubiquitous computing philosophy originated at Xerox Parc in 1988 [3], pioneered by Mark Weiser. He conceived of Ubicomp as nonintrusive, mobile, flexible computing, highly integrated into the working and living environment. Ubicomp is not virtual reality (VR). VR techniques, which put people into artificial worlds, primarily pose a computing and graphics horsepower problem. Ubicomp forces the computer to live in the real world with people. It is the integration of human factors, computer science, engineering, and social science. The human factors issues go well beyond yet another human computer interface problem and will not be solved with another windowing system. The computer science issues span all areas—networking, operating systems, distribution of memory and processing, naming, resource management, etc. A general discussion of these issues can be found in [4]. Engineering will have to make advances in nanotechnology to manufacture the devices we can foresee. No one knows the social implications of everyone's having full-time computing and network access.

An example of an Ubicomp object is intelligent curtains that contain light and temperature sensors and control room lighting and temperature. As developers of UbiWorld, we talk about binoculars with image processing software to detect and highlight objects in a scene, to track eyes and vary the focus, or to perform other image transformations. We imagine scrap paper that will be aware of the identity of the user, will auto connect to the user's environment, and will automatically store and retrieve all notes made in a personal database accessed via contact with a table or desk. We discuss many other examples, limited only by our imagination.

Research in ubiquitous computing conducted at Xerox Parc followed standard experimental science protocols [5]. Devices were conceived and prototypes constructed and tried out on willing subjects. There were three prototypes of note: the Xerox ParcTab, a palm-sized device; the Pad, a notebook-sized device; and the Liveboard, a wall hanging device. Applications were constructed to perform e-mail, take notes, schedule meetings, check weather, etc. [6]. The primary lesson we take from the work, however, is that technology today is nowhere near what is required to design and perform experiments in truly ubiquitous computing. A discussion of the many compromises that Xerox had to make in the development of the ParcTab can be found in [6].

3 UbiWorld

UbiWorld is an experimental system combining virtual reality, advanced networking, and supercomputing to explore the implications of ubiquitous computing. We use a virtual reality system as a design and evaluation environment. Instead of actually building devices, we use VR techniques to model the representation of devices. We use advanced networking to link VR objects with computational servers to represent the behavior of the objects. Using these techniques, we can explore devices that are not yet possible to build.

Today's hardware capabilities fall short of what ubiquitous computing will need in terms of power consumption, miniaturization, network bandwidth, and computing power. Until these capabilities can be met, we feel that experimenting in virtual spaces is a productive method of exploring the concepts in ubiquitous computing. The UbiWorld project builds on existing software and projects in MCS. It serves to focus those efforts and leverages our long-standing expertise in software engineering and our strong development environment.

Starting with the CAVE^{TM} family of display devices we integrate tools for the construction of 3D objects into the existing library [7]. Using these objects as models, we can then imbed new information technology within them. These products might be hand-held computers, intelligent paper, image-processing binoculars, desks, clothing, jewelry, cups, eyeglasses, or carpeting. The plan is to couple the virtual objects with remote computers via fine-grained heterogeneous computing technology and to provide Ubicomp behavior and functionality to the models. The 3D objects will be placed in virtual rooms, thereby creating a shared virtual world (in a collection of CAVEs, ImmersaDesks, or whatever) where users can experiment with using the virtual devices.

Each object in the world has behavior controlled by a program running on the network. The behavior could be one that, in the real object, would be provided by a local computer or by a combination of local computer and network connection to remote processors or databases. These "behavior" processes are able to communicate with each other by using a shared protocol (UbiWorldcomm). The objects also react and are influenced directly by interactions with the virtual world and its users. If someone picks up an object in the UbiWorld, that object knows it has been picked up and then "does the right thing," which may mean communicating with other objects or computing something or displaying some network stream. For example, one's coffee cup could be displaying live video, or one could talk to an earring and make a phone call.

UbiWorld will let us debug Ubicomp years before we have the technology to build it. It will enable us to change specifications without changing hardware and to identify software requirements. Through UbiWorld we can explore the boundaries of embedded computing and network computing. It serves as a testbed for heterogeneous computing tools and systems such as fine-grained networking protocols, image composition mechanisms, and agent integration.

4 UbiWorld Design

A fundamental principle in the design of UbiWorld is the separation of an object's representation from its behavior. As shown in Figure 1, the user interacts with the representation of an object. The behavior of the object is specified independently of the representation. The actual computation of the behavior is performed on a simulation server, separate from the representation computing. Furthermore, as seen in Figure 2, the world in which the object interacts is viewed as an orthogonal issue from the object itself. The virtual world, its representation, and its properties (such as light and gravity) are computed, stored, and rendered separately from the objects we choose to place in the world. In Figure 2, we show a virtual world record-and-playback engine that builds on work in progress at MCS. This engine, which we call VR Voyager, can be thought of as a virtual world server, storing virtual worlds, serving them to clients, recording VR experiences, and playing them back on demand.

In addition, we introduce the concept of "distributed rendering." This is an attempt to overcome the bottlenecks introduced into today's VR systems by lack of graphics rendering power. By employing rendering engines in distributed machines, we can bring much greater rendering power to bear than would otherwise be possible. Also, it gives us the ability to bring the rendering closer to the source of the data generated in the simulations. Distributed rendering is a new research area introduced into the Futures Lab by the UbiWorld project. We are studying ways to composite the separately rendered images into a single image in the VR theater, by tapping into the OpenGL pipeline, for instance. Along with distributed rendering, we also introduce the concept of "aware networking." Aware networking means that networking resources are not precisely known at all times and implies an adaptive nature imbedded in the objects as they seek to join networks. Objects can join networks by using a variety of bandwidths and protocols.

UbiWorld is a classic example of using the power of virtual reality systems to prototype devices that are impossible or too expensive to build. To accomplish our goals, we must construct tools that simplify the connection of physical representations to computer simulations that are prototyping the hardware. This then gives us the ability to construct and test the usefulness of hardware that is impossible to build with today's technology.

5 UbiWorld Development Environment

The UbiWorld development environment is a combination of hardware and software environments. The hardware is a combination of supercomputers, networks, and display devices. The software covers all major areas of software development from the low-level network code all the way up to high-level scripting languages that allow users to configure the environment.

5.1 Hardware

The following two subsections will address the display environments in which the UbiWorld system will be tested, and the various hardware limitations.

5.1.1 Display Environments

The current display environment is comprised of three virtual reality devices.

- **CAVE**

  The CAVE^{TM} (CAVE Automatic Virtual Environment) is a 10 x 10 x 9 foot room that uses rear-projected high-resolution projectors to produce an immersive 3D environment (Figure 3). The CAVE environment, originally developed by the Electronic Visualization Laboratory (EVL) at the University of Illinois at Chicago, produces a 3D stereo effect by displaying in alternating succession the left and right eye views of the scene as rendered from the viewer's perspective [7]. These views are then seen by the user through a pair of LCD shutter glasses whose lenses open and close forty-eight times a second in synchronization with the left- and right-eye views. The correct viewer-centered projection is calculated based on the viewer's position and orientation as determined by a electromagnetic tracking system. The position and orientation of a 3D wand are also tracked; this wand allows for navigation of and input into the virtual world. Along with the visual

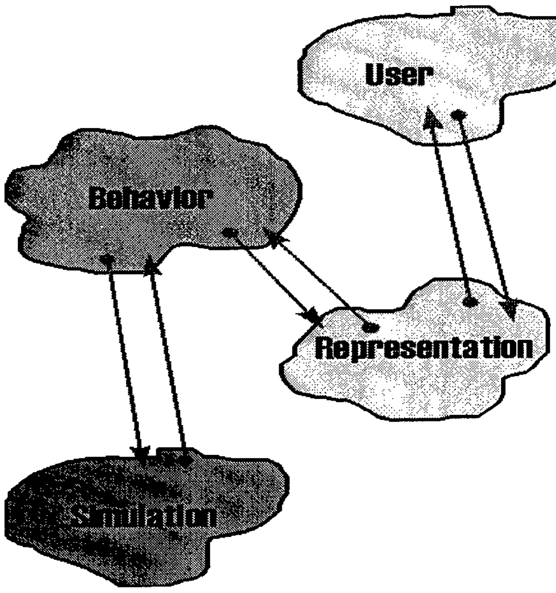


Figure 1: The Connections of Behavior, Representation, Simulation, and the User within the UbiWorld model.

feedback of the CAVE environment, a complete 3D audio environment is available to the user.

- **ImmersaDesk**

  The ImmersaDesk is based on the same rear-projection technology as the CAVE (Figure 4). It is a fully interactive, 3D immersive environment that is about the size of a large drafting table. The ImmersaDesk allows for one tracked viewer, along with two to three passive viewers.

- **InfinityWall**

  The InfinityWall is a large rear-projected system that is created from compositing four standard 1280 x 1024 screens together to create one large high-resolution screen. The InfinityWall can be used as a large ImmersaDesk, where the images are projected in stereo and the viewer is tracked, or can substitute for a large high-resolution workstation. The NII/Wall was developed by EVL, the National Center for Supercomputing Applications, and the University of Minnesota, with support from Silicon Graphics, Inc.

5.1.2 Hardware Limitations

Although this development environment is satisfactory for early experiments, we believe that several improvements will need to be made in virtual technologies to fully realize the benefits of the UbiWorld project.

- **Resolution**

  The current resolution of the CAVE is 1280 x 768 on each wall. If we were to attempt a resolution close to the capabilities of the human eye, we would need a resolution of 4,800 x 3,800 [8]. That resolution is not available at this time, but we believe we can achieve that resolution on selected areas of the screen by using a "high-resolution window." By using a separate projector and rendering engine and by driving the location of the projector based on gaze direction, we can provide an area of high resolution at the place where the user is looking. This high-resolution window is another of the research projects in the Futures Lab that is motivated by the UbiWorld project.

- **Tracking**

  The resolution of the tracking system is another weakness of the current environment. Today, our effective sampling rate is approximately 100 ms, with a spatial resolution of approximately 1 inch. In the future, for fine-grained manipulation of objects, we expect a sampling rate closer to 1 ms and a spatial resolution of 1 mm.

- **Haptics**

  Today, our CAVE environment has no haptic devices. The UbiWorld project requires that we bring these devices into the CAVE and learn to register their actions with virtual representations.

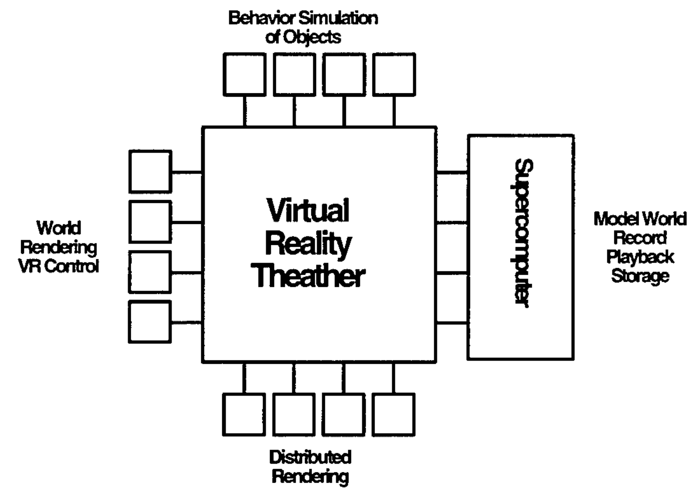

Behavior Simulation of Objects
World Rendering VR Control
Virtual Reality Theather
Supercomputer
Model World Record Playback Storage
Distributed Rendering

Figure 2: Separation of UbiWorld Spaces

Force and feedback will be required to fully prototype actions of devices in our virtual world.

- **Control Interfaces**

  Software in the CAVE today is programmed as an extension of current windows-based systems. We use menus, visual displays of pick lists, radio buttons, dials, etc. We believe these interfaces are wholly inadequate for use in UbiWorld. Devices and objects represented in UbiWorld may be activated by voice, by absolute position or proximity to other objects, by proximity to or by sensing features in the virtual world, or by any other means that we haven't thought of yet. We need a new paradigm to allow freedom and innovation in control interfaces. This is yet another active research project spawned by requirements of the UbiWorld project.

5.2 Software

Although more tools are required, we have a good group of software already available to us for use in the UbiWorld project.

5.2.1 CAVElib

The CAVE library [7], developed at EVL to work with the CAVE family of display devices, provides basic VR functionality and viewer-centered perspective transforms automatically. This frees the VR programmer to focus on the graphics of the problem at hand, not the viewer perspective problem. The CAVE library provides basic navigation functions, tracking of the user and wand, and interaction with the wand buttons and joystick.

5.2.2 CAVEcomm

The CAVEcomm library is a communications library that aids developers of virtual reality applications in the area of remote communications [9, 10]. The remote communications can be either virtual reality device to virtual reality device or virtual reality device to supercomputer. Using the CAVEcomm library, users register their virtual reality applications and/or supercomputing simulations with a broker. The broker process handles the connections of separate entities. The broker manages resources and connections but does not handle data traffic. Once the broker has set up the connection between the entities, they send the actual data traffic only between each other. CAVEcomm is specifically designed to work with the CAVE group of virtual reality devices, namely, the CAVE, the ImmersaDesk, the CAVE simulator, and the I-finityWall. The ideas of CAVEcomm can be extend however, to any virtual reality-based system.

CAVEcomm has been used to connect virtual reality applications to supercomputing simualtions that are running in real time. It has been used to connect two CAVEs that are geographically separated, allowing the users to collaborate on a joint task or to demonstrate something in one CAVE to users in another.

5.2.3 CAVEav

The CAVEav library brings multimedia capabilities to the CAVE. Using the library, programmers can connect to video sources on the network and texture map the resulting video stream onto objects in the CAVE. Live video streams from other CAVE, for instance, can

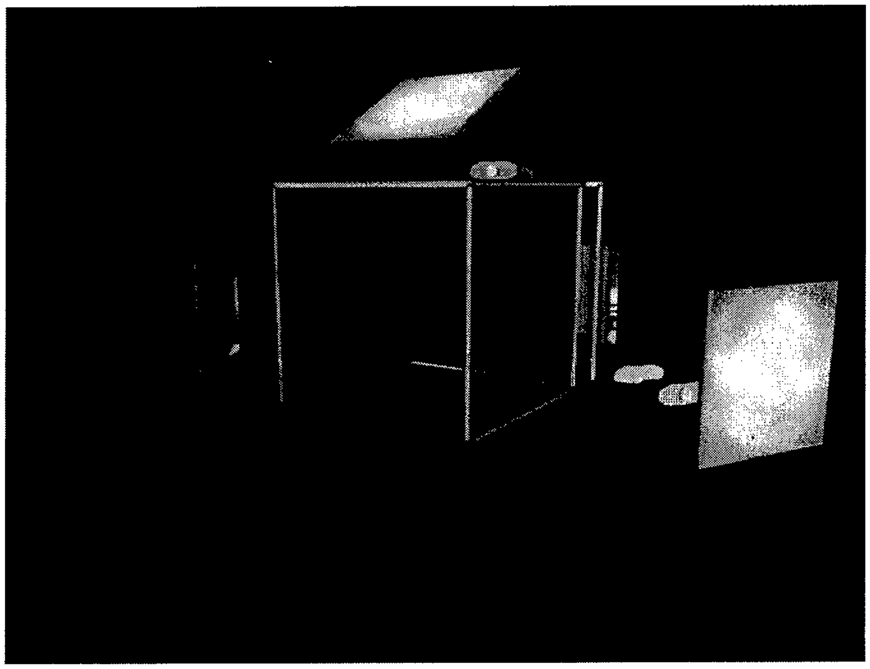


Figure 3: CAVE Virtual Environment (Milana Huang, EVL, 1994)

be texture mapped onto avatars showing remote users. Live streams from robots or instruments are used to provide an immersive telepresence capability. Prerecorded streams can be used to provide instruction or backgrounds.

A prototype system demonstrating these features has been developed at ANL. A small robot mounted with a variety of video and audio components is connected to the CAVE; from within the CAVE, a user can navigate the robot and interact with the environment within which the robot lives [11]. This system is being used to test the requirements and to expose the difficult problems within a toolkit of this nature.

5.2.4 CAVE-VRML Modeler

Three-dimensional virtual computer environments such as the CAVE should have the capacity to be a working development environment, not just an interactive display environment. One aspect of a development environment is 3D modeling. Currently, no 3D modelers work well in conjunction with the CAVE. Often, in going from the modeler to the CAVE, "what you see is what you get" is not always true. Frequently, objects modeled on a workstation look quite different in the CAVE, particulary with respect to the object's scale, color, and lighting. One of the new projects in the Futures Lab is the development of an interactive modeling system to be used in the CAVE. This system will allow users to create objects in the native environment and will import/export VRML-based objects that can be used in or taken from other VR environments. Users will be able to affect the transformations (rotate, scale, translate) of the object as well as its material properties (ambience, diffusivity, shininess, specularity,, etc.). The ability to edit materials and lighting is significant given the fact that many objects look very different in the CAVE due to the physical components of the CAVE, (i.e., projectors and screens). This CAVE modeling tool will also have the ability to edit the object's shape, not just its extraneous properties. Users will be able to edit the polygons of the object; adding, deleting, and moving vertices will give users the ability to redefine and combine existing shapes or create new objects from scratch. The created worlds and environments will be exportable to VRML.

5.2.5 VR Voyager

The Voyager multimedia recording and playback system has been under development in the Futures Lab for the past two years [12]. It uses an IBM SP2 for multistream, multimedia record and playback of network-based sources. We propose to use the Voyager system as the basis for a new virtual world server,

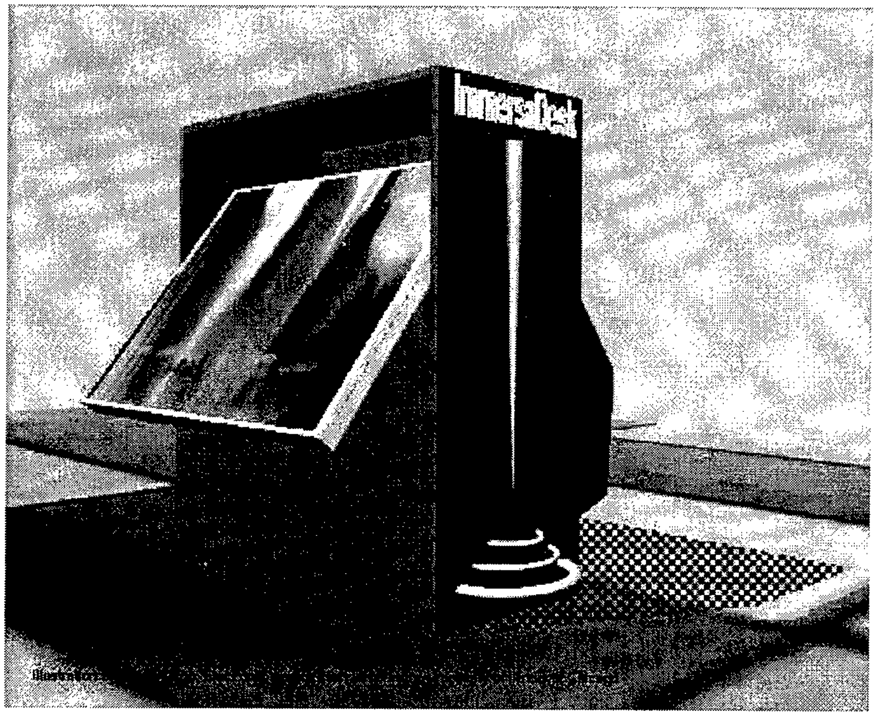


Figure 4: ImmersaDesk Virtual Environment (Jason Leigh, EVL, 1995)

providing record and playback of VR experiences.

6 UbiWorld Goals and Requirements

Our goal is to test UbiWorld objects and worlds in a task-based manner, under different scenarios. We will provide several different environments, for example, a home, office, airplane, hotel, car, field, restaurant, laboratory, and shop. Each of these environments must be rendered with high-resolution textures, complex spaces and lighting, and varying degress of scene complexity. Each of these models is different, but the requirements of ubiquitous computing span all of these environments. In each different environment, Ubicomp objects must behave appropriately. Scenario spaces will be used for experimenting with device functionality and interaction by requiring task-based explorations. For instance, users may be required to order a meal, prepare a talk, negotiate a contract, drive to an unknown destination, or buy groceries.

6.1 Virtual Device Requirements

To accomplish this goal, we believe that computing objects in the UbiWorld must possess or make use of at least the following features.

6.1.1 Innovative Representational Design

Current thinking, as evidenced by the ParcTab, is too restricted by technology limitations to create truly innovative design. We would like to be able to take the best from advanced industrial design and apply it to the design of future Ubicomp devices in UbiWorld. The type of advanced design philosophy we have in mind is embodied in publications such as *Arbitare* magazine and the book *The Art Factory, Design in Italy Towards the Third Millenium* [13]. In this latter text, the author analyzes the birth of virtual industry, links between the fashion system and the media system, the rediscovery of art and craft traditions, and renewed ecological awareness of materials in terms of contemporary and New Wave design.

6.1.2 Novel Information Technology Components

Using supercomputer and external multimedia servers and resources, designers in the UbiWorld will be able to specify and simulate behavior associated with advanced CPU capabilities, imaging technology, sensors, actuators, multimedia components, communications capabilities, etc.

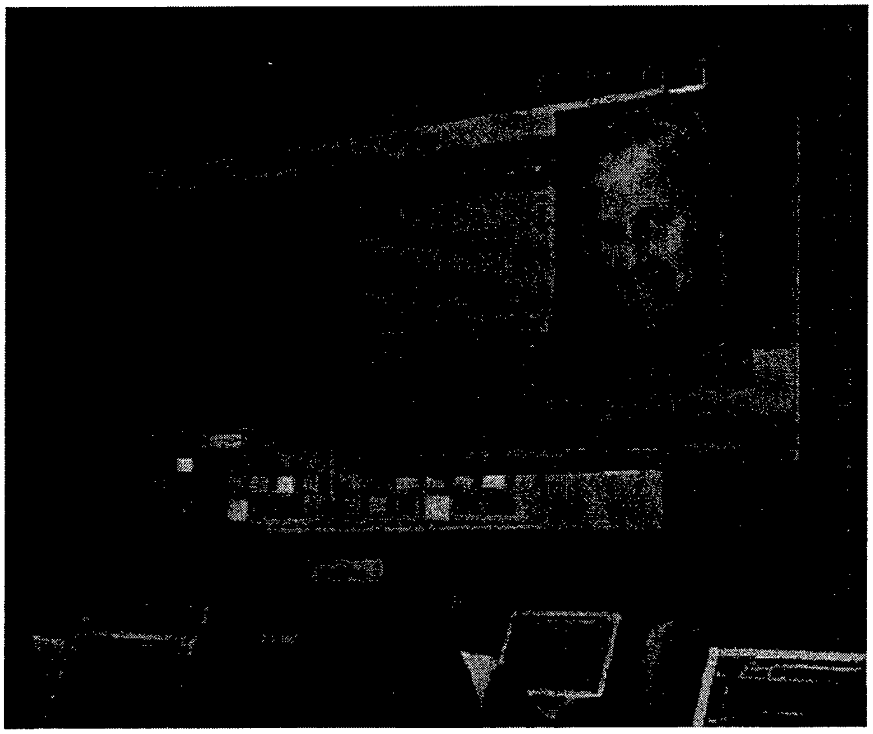


Figure 5: InfinityWall Virtual Environment (Jason Leigh, EVL, 1995)

6.1.3 Transparent Networking

Transparent, or "aware," networking is assumed to be a fundamental capability in the UbiWorld. The network in this case is transparent to the user. UbiWorld devices automatically connect to whatever other device is appropriate, using mutually acceptable bandwidths and protocols.

6.1.4 Device/Space Awareness

Devices in the UbiWorld must exhibit awareness of other devices and of the space in which they are operating. If a user carries a device to a different space, the device must automatically be aware of the new space context and take appropriate action.

6.1.5 Reactive/Proactive/Proxy Behavior

UbiWorld devices should, of course, react properly to users requests, but beyond that, they should be proactive as necessary. An example is the loading of a new context when the user enters a new space or the proactive downloading of news or information known to be of interest to the user. Based on current research within the artificial intelligence community on agent based systems, one can already see a trend emerging. Agents are being constructed that passively view the behaviors of users and learn about their interests. Using methods similiar to these techniques, we hope to construct within the UbiWorld a framework that allows the Ubicomp devices to be reactive and proactive.

6.1.6 Integration Functions

In the UbiWorld, we believe the network really is the computer. As devices near other devices or objects in the space, they should be able to interrogate the devices and perform acts of spontaneous integration if it is of benefit to do so. Benefit in this case could be defined as access to greater bandwidth, computing power or advanced capabilities. The intelligent scrap paper idea fully embodies this idea, since the scrap paper integrates with desktops for greater bandwidth or with imaging devices to capture multimedia information.

6.2 UbiWorld Design Problems

We see four critical design problems in the UbiWorld project:

- Object shape, form, and representation
- Computing and communications internals
- Functional behavior

- Integration with and awareness of environment

In the UbiWorld project, we believe it is important that these design problems be separated. We want to feel free to experiment with the form and shape of an object independent of its other attributes. Since the object is virtual, we can experiment with communications and computing internals without being encumbered by physical packaging or power problems. Functional behavior is simulated via the computational servers and can be varied at will, independent of the packaging or other factors. Finally, integration with and awareness of the environment are accomplished via simulated sensors and connection interfaces. For instance, when an object such as a piece of intelligent paper is placed on a table, we expect it to perform the above mentioned "spontaneous integration" with the table. Through its sensors, it must become aware of the table and its capabilities. By using an aware networking approach, the intelligent paper negotiates with the table to establish a connection, thereby integrating its functionality with that of the table.

For each of these design problems, appropriate tools are essential. Where existing tools are inadequate, it will be necessary to build or invent more robust tools.

6.3 Technical Challenges

The scope of the UbiWorld project pushes the bounds of current VR and networking technology and gives rise to a host of technical challenges. The following list is not meant to be comprehensive but serves to enumerate those problems we feel are most important to focus on now in the implementation of the UbiWorld.

- **Scalability**

  Adding objects into a UbiWorld environment stresses protocols, bandwidths, computing, and rendering power. Important research issues regarding scalability of these components and their interactions must be investigated.

- **Latency**

  We know that there are limits on the user-perceived latencies in an interactive system. The latency of the total system (including latency from the graphics system, the tracking systems, the networks and the computation engines) cannot exceed 100 ms - 1000 ms, depending on the user's experience level [14]. Taylor et al. studied this phenomenon in multiple-tracked, network-connected VR systems and offered data and research issues to be investigated to mitigate the latencies in the system [15].

- **Object Representation**

  The representation of objects within the UbiWorld model must be flexible, so that they can be changed easily, without affecting the underlying simulation of the objects. It is important to allow for a variety of different representations to be attached to the simulation, to allow users to experiment with different interfaces. This component is one of the true strengths of the connection of Ubicomp devices to VR, since expensive prototypes do not need to be built to try out a new device.

- **Behavior Specification**

  An open question is how best to specify the behavior of an object. For our purposes, there are three categories of behavior:

  - Representation Dynamics
  - Functional Mechanics
  - Computational and Communication

  It is not likely that the same tools will be appropriate for each of these tasks. A suite of tools will need to be developed to enable specification of each of these behaviors and their interactions.

- **Object Binding and Brokering**

  Resource brokering is the use of computationally enhanced entities to aid in the requesting, allocation, and management of remote capabilities. Much in the same way that data hiding is used in object-oriented programming to make the interfaces easier to use and understand, wide-area resource brokering can be used to simplify the user interaction with a large-complex set of computational and collaborative resources. One approach can be attacked by expanding on what we have learned from traditional system management software and current work in cluster management. Through the use of resource brokering, the level of complexity needed to configure and control a large virtual world is reduced to a manageable state for the user.

- **Process Mapping and Execution Control**

  The management and control of processes become important in the UbiWorld model. The ability to map a new process into an existing computational framework is an essential component of the UbiWorld model. As a new user enters the UbiWorld or a new Ubicomp device is introduced, the process controlling the simulation will have to be seamlessly integrated. At the computational level that will require mapping the new process onto a computational resource and then controling the execution from startup to termination.

- **Evaluation and Measurement**

  As in any scientific endeavor, we desire to measure, evaluate, and report on our work in a rigorous manner. Tools are required to enable instrumentation of all the computational and communications processes and their relationships. Evaluation and reporting tools are essential to reduce and analyze the data generated by instrumented codes.

- **Distribution**

  Distribution covers a whole set of problems related to using networked resources: the mapping of processes to processors; the distribution of databases over networked resources; the issues of redundancy, failure recovery, and the other problems usually associated with distributed computing — all are present in the UbiWorld project.

- **Naming and Identification**

  The current Internet provides mechanisms for naming computers and Web pages. Future environments will require the ability to name a much wider variety of objects with varying degrees of persistence and scope [16]. Mechanisms are also required for locating objects based on different criteria. Proposals for Universal Resource Names are a step in this direction but are designed for long-lived objects. We believe that a new class of naming mechanism will be needed that can refer to much shorter-lived objects that would be the topic of communication between user agents and simulation spaces. Brokering and name translations mechanisms are also needed. These will need to be high performance and scalable and to incorporate hooks for security and access rights.

- **Security**

  Fine-grained, scalable authentication, authorization, and accounting mechanisms will be required to control access to information and computational resources by both users and computational entities. Complex issues include secure communication of high-bandwidth multimedia data, access control of dynamically created entities, multi-entity interactions, security of archived data, object-level security in virtual environments, and delegation of authority between users and associated computational entities.

7 Tools

The underlying infrastructure for the construction of UbiWorld requires the combination of a wide area of computer science disciplines. Techniques and tools need to be borrowed from the fields of computer graphics, artificial intelligence, and systems to name but a few.

We have identified a set of existing tools for use in implementing the UbiWorld concept. These tools will not be sufficient in the long term, but most represent a suitable beginning for work on the UbiWorld requirements.

7.1 Representation

The physical representation of the objects within UbiWorld, such as intelligent paper or image-processing glasses, needs to modeled in such a way that the objects can be easily changed and modified. Currently one can use a wide variety of desktop modeling and CAD packages to physically design the objects. These packages are not sufficient to develop UbiWorld objects to final state. While it is important to separate the representation from the function, the modelers require the ability to provide hooks or connections to a behavior toolkit. We believe that the Open Inventor^{TM} and VMRL modeling formats lend themselves most to this goal.

7.2 Behavior Specification

For the representational dynamics of an object, the tools such as VRML or OpenInventor come to mind. For the functional (or mechanical) behavior, procedural systems such as Java or C++ are available today, but they are too low level to use in the long term. For the specification of the computational and communications behavior, we can also use Java or C++, or other still too low-level systems such as nperl or Nexus.

7.3 Object Binding/Brokering

Binding of objects to resources can currently be accomplished by hard-wiring the connection or soft-wiring the connection through the use of a brokering system such as CORBA or the LabSpace broker.

7.4 Process Mapping and Execution Control

The design of UbiWorld calls for object behavior to be computed on separate computational servers. The mapping of many object processes to computational servers and control of the execution is a problem we have been examining for some time. For SC'95, a team at ANL developed custom software for scheduling and mapping processes to network-based processors. The system deployed at SC'95 was the I-WAY Point of Presence (I-POP) machine. The I-POP machine was specifically set up to manage security issues and was configured specifically for the I-WAY [17]. It allowed use of process mapping and control software such as Nexus and MPI. Subsequent to the I-WAY project, a team at ANL has been integrating the Adaptive Communication Environment (ACE) [18] system into a parallel message-passing toolkit.

Each of these solutions has shortcomings, and we fully expect that UbiWorld requirements will force the design or evolution of a new, more sophisticated system for process mapping and control.

7.5 Evaluation and Measurement

It is important to be able to measure and quantify the progress that is being made. Since UbiWorld requires the combination of such a wide variety of different aspects of the field of computing, we are required to connect various areas that are only now beginning to be tested. The connection of supercomputers to immersive virtual reality display devices is just one area. For the first time, issues such as latency, for example, need to be looked at outside their normal meanings to supercomputer users and to graphics programmers. Therefore, as UbiWorld is constructed, measurements and evaluation need to be done. Currently, we are using PABLO from the University of Illinois for instrumenting VR and simulation programs. We use MPI logging and the Upshot display system for tracing and evaluating MPI programs.

8 Conclusion

UbiWorld is a project that pushes beyond the bounds of current technology and forces us to think of heterogeneous computing in new terms. Rather than making incremental changes in existing technology, UbiWorld gives us a chance to leapfrog into a new problem space where heterogeneous computing is the norm, not the exception. In this space, we can more clearly see the technical challenges that await us, and we can proceed to invent solutions well ahead of technological progress that can implement these solutions.

The scope of UbiWorld is very broad and invites research on many fronts. We have identified some of these issues, such as network latency, scalability, object representation and specification, process and data mapping, object brokering and binding, security, and measurement. Each of these issues is deserving of a focused research effort in the futures-oriented ubiquitous computing scenario, and we invite the heterogeneous computing community to engage in discussions and research in this rich problem space.

Acknowledgments

We thank Ian Foster, Ivan Judson, Bill Nickless, Bob Olson, and Matt Szymanski for all their help and input. This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

References

[1] Terrence L. Disz, Remy Evard, Mark W. Henderson, William Nickless, Robert Olson, Michael E. Papka, and Rick Stevens, "Designing the future of collaborative science: Argonne's futures laboratory," *IEEE Parallel and Distributed Technology Systems and Applications*, vol. 3, no. 2, pp. 14–21, Summer 1995.

[2] Mark Weiser, "The computer for the 21st century," *Scientific American*, vol. 265, no. 3, pp. 94–104, September 1991.

[3] Mark Weiser, "The world is not a desktop," *Interactions*, vol. 1, no. 1, pp. 7–8, January 1994.

[4] Ian Foster, Michael E. Papka, and Rick Stevens, "Tools for distributed collaborative environments: A research agenda," in *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Compututing*. IEEE, 1996, pp. 23–29, IEEE Computer Society Press.

[5] Mark Weiser, "Some computer science issues in ubiquitous computing," *Communications of the ACM*, vol. 36, no. 7, pp. 74–83, July 1993.

[6] Roy Want, Bill N. Schilit, Norman I. Adams, Rich Gold, Karin Petersen, David Goldberg, John R. Ellis, and Mark Weiser, "The parctab ubiquitous computing experiment," Tech. Rep. CSL-95-1,

Xerox PARC, Xerox Palo Alto Research Center, March 1995.

[7] C. Cruz-Neira, D. J. Sandin, and T. A. DeFanti, "Surround-screen projection-based virtual reality: The design and implementation of the CAVE," in *Computer Graphics (Proceedings of SIGGRAPH '93)*. ACM SIGGRAPH, August 1993, pp. 135–142, Addison Wesley.

[8] M. McKenna and David Zeltzer, "Three dimensional visual display systems for virtual environments," *Presence*, vol. 1, no. 4, pp. 421–458, 1992.

[9] T. L. Disz, M. E. Papka, M. Pellegrino, and R. Stevens, "Sharing visualization experiences among remote virtual environments," in *International Workshop on High Performance Computing for Computer Graphics and Visualization.* 1995, pp. 217–237, Springer-Verlag.

[10] Terrence L. Disz, Michael E. Papka, Michael Pellegrino, and Matthew Szymanski, *CAVEcomm Users Manual v1.0*, Mathematics and Computer Science Division, Argonne National Laboratory, 1.0 edition, August 1996, ANL/MCS-TM-218.

[11] Ivan Judson and Rick Stevens, "Architecture of an immersive virtual reality telepresence system," Private Communication, Argonne National Laboratory, April 1997.

[12] Rick Stevens, Terrence Disz, Robert Olson, Michael E. Papka, and Remy Evard, "Voyager: A next generation hypermedia server to support the construction of virtual organizastions," 1995, Grant Proposal, Submitted to DOE 1995.

[13] Stefano Casciani, *The Art Factory Disgn in Italy Towards the Third Millennium*, Springer-Verlag, 1996.

[14] A. Liu, G. Tharp, L. French, S. Lai, and L. Stark, "Some of what one needs to know about using head-monunted displays to imporve teleoperator performance," *IEEE Transactions on robotics and Automation*, vol. 9, no. 5, pp. 638–648, 1993.

[15] Valerie E. Taylor, Milana Huang, Thomas Canfield, Rick Stevens, Daniel Reed, and Stephen Lamm, "Performance modeling of interactive, immersive virtual envrionments for finite element simulations," *The International Journal of Supercomputing Applications and High Performance Computing*, vol. 10, no. 2/3, pp. 141–151, November 1996.

[16] John Kunze, "Functional recommendations for Internet resource locators," Febuary 1995, Network Working Group, RFC 1738.

[17] Ian Foster, Jonathan Geisler, William Nickless, Warren Smith, and Steve Tuecke, "Software infrastructure for the i-way high-performance distributed computing experiment," in *5th IEEE Symposium on High Performance Distributed Computing.* IEEE, 1996.

[18] Douglas C. Schmidt, "The adaptive communication environment an object-oriented network programming toolkit for developing communication software," in *Sun Users Group Conference*, December 1993.

Case Study

Mercury Computer Systems' Modular Heterogeneous RACE® Multicomputer

Thomas H. Einstein
Mercury Computer Systems, Chelmsford, MA, USA

Mercury Computer Systems' Modular Heterogeneous RACE® Multicomputer

Thomas H. Einstein
Mercury Computer Systems, Inc.
199 Riverneck Road
Chelmsford, MA 01824-2820
USA

Abstract

A heterogeneous multicomputer is a multicomputer composed of two or more different types of processors. This paper describes the rationale for heterogeneity in a multicomputer and gives a typical example of a heterogeneous system in the form of a RACE multicomputer composed of a mixture of Analog Devices' SHARC 21060 and the IBM/Motorola/Apple PowerPC 603p processors. These two processors have complementary attributes, and the advantages and limitations of each are described.

Multicomputers generally implement a sequence of different processing algorithms. The "optimal" processor that maximizes throughput at each step in the processing flow is generally a function of the algorithm to be executed at that step. Other factors that also influence the optimal mix of processors in a heterogeneous multicomputer include physical processing density, hardware cost, and ease of programmability.

1.0 Introduction

A multicomputer is defined as a computing system composed of multiple, independent but cooperating processors. The different processors in a multicomputer communicate with each other over a common data communication fabric. In its simplest form, this communication fabric may consist of a single common bus, such as VME or PCI, that is shared by all of the multicomputer's processors. Another high-performance example of such a fabric is a crossbar switching network that consists of multiple independent data paths over which several independent data transactions can be in process concurrently. The principal purpose of a multicomputer is to satisfy the processing requirements of applications that require the processing throughput exceeding that of a single processor.

A modular multicomputer employs component processors designed to be modular, much like the building blocks in a Lego™ set. This enables expandability by literally "plugging" additional processors into the common interprocessor communication fabric.

There are both physical and logical implications to this configuration. The physical implication is merely that the processor modules be constructed on individual circuit boards or cards that plug into a standard connector on the communication fabric. VME circuit boards and connectors are an example of such a standard connector interface. The logical implication of this modularity is in the multicomputer's operating system software. Each processor must use a common interprocessor communication system that can support data communication with any other processor or group of processors in the multicomputer. Multicomputer software will be discussed in more detail in a subsequent section.

Finally, a heterogeneous multicomputer is defined as a multicomputer composed of two or more different types of processors. One example of such a system is Mercury Computer Systems' RACE multicomputer. In addition to being heterogeneous, the Mercury system is also modular and may be configured to contain from four to several hundred processors, consisting of any combination of the following processors: Intel i860, IBM/Motorola/Apple PowerPC 603p, Analog Devices SHARC 21060, and Texas Instruments C80.

Why heterogeneous multicomputing? Even when several different processors have comparable MFLOP throughput ratings, each processor type may have particular attributes that make it ideally matched to certain applications. For example, the SHARC 21060 is ideally matched to vector signal processing applications such as FFTs, as well as for embedded applications that require high processing densities in terms of both MFLOPS/volume and/or MFLOPS/watt. In

0-8186-7879-8/97 $10.00 © 1997 IEEE

contrast, the PowerPC is ideal for executing scalar (non-DSP) applications characterized by arbitrary C code.

In general, algorithms that have a high ratio of computation-to-data accesses will execute more efficiently on the PowerPC than on the SHARC, whereas the converse is true for algorithms having a low ratio of computation-to-data accesses. A more detailed comparison of the SHARC and PowerPC processors, together with the relative advantages and disadvantages of each, will be given in a subsequent section. It will be shown that the SHARC and PowerPC are truly complementary processors that justify the concept of heterogeneous multicomputing.

The remainder of this paper will discuss a Mercury modular heterogeneous RACE multicomputer composed of the following two processor types: SHARC and PowerPC 603p.

This paper is organized as follows: Section 2 briefly introduces the following basic logical components of Mercury's RACE multicomputer: Compute Elements (CEs), Compute Nodes (CNs), CN RACEway interface ASICs, RACE crossbars, and the RACEway crossbar switching network. Also described in Section 2 is how these modules can be physically interconnected to form a RACE multicomputer.

The only components in this multicomputer that are unique to a given processor type are the CEs themselves and the CN RACEway interface ASICs. The CN RACEway Interface ASIC for a given processor type provides a common interface to RACEway communication from that processor. The interface commonality provided by these ASICs is the hardware key that enables a RACE multicomputer to be heterogeneously configured from a number of different processor types.

The RACEway crossbar switching network is an essential element of the RACE multicomputer's scalability. This crossbar network provides multiple concurrent paths between different CEs of the multicomputer. Much like in a modern digital telephone switching network, a number of independent messages can be transferred concurrently over different paths between CEs, and/or CEs and I/O interfaces, at data rates of up to 160 MB/s per path. Each path is dynamically switchable in less than two microseconds.
As additional processors are added to a multicomputer, the interprocessor communication bandwidth requirements naturally increase. Unlike a bus, whose bandwidth capacity is fixed, as more processor nodes are added to a RACE multicomputer, the crossbar network automatically expands to provide the additional bandwidth required. This expansion is realized by adding crossbar sub-network modules to an existing crossbar network to accommodate the added nodes.

Section 3 presents a very-high level overview of the software in a Mercury RACE multicomputer. A separate copy of Mercury's real-time runtime environment, MC/OS, executes in every CE of the multicomputer.

The MC/OS runtime environment consists primarily of the following three components: MCexec™, Interprocessor Communication System (ICS) and Hardware Abstraction Layer (HAL). MCexec is a standard, single-processor real-time operating system that handles task scheduling, context switching, timer services, etc. for the processor on which it resides. ICS is a set of Application Programming Interfaces (APIs) that handles all interprocessor communication. ICS allocates and manages shared memory buffers, performs address mapping of remote nodes, and manages all interprocess data transfers, both DMA and programmed I/O.

HAL is a set of processor-specific software that provides the interface between both MCexec and ICS and each processor's unique hardware (i.e. DMA controllers, interrupt registers and vectors, address mapping registers, etc.). The purpose of HAL is to make the MCexec and ICS components of MC/OS as processor-independent as possible. Thus, most of the processor-specific elements of MC/OS have been encapsulated in the HAL.

Section 4 introduces the concept of heterogeneous processing, using a system composed of SHARC and PowerPC processors as a specific example. The unique attributes of the SHARC and PowerPC processor chips are described as well as which chip is best suited to which kind of application. In a nutshell, the PowerPC is best suited for the execution of algorithms that are characterized by a high ratio of computation-to-data accesses. Typical algorithms in this category include those that involve the evaluation of transcendental functions such as sine, cosine, sqrt, atan, log, etc., as well as long FIR filters, matrix inversion and multi-point data interpolation. In contrast, the SHARC is best suited for vector operations characterized by a relatively low ratio of computation-to-data accesses, as well as for computation of large-size (i.e. > 512-point) FFTs.

Other important considerations that effect processor choice include physical processing density, ease of coding, and code portability. The SHARC is one of the densest (in terms of MFLOPS/volume and MFLOPS/watt) processing chips available today. This factor is an especially important issue in embedded systems for military applications. On the other hand, coding SHARC applications generally requires an extra level of learning if the chip's full processing potential is to be realized. In particular, the SHARC user should make use of Mercury's hand-coded Scientific Algorithm Library (SAL) and multibuffering routines wherever possible, using C code only as 'glue.'

In contrast, compiled C code runs extremely efficiently on the PowerPC, and in some cases even more so than on corresponding SAL functions. Thus, if the user has already written a large application in C that the user wants to port to a Mercury RACE multicomputer, porting that application to the PowerPC will usually be significantly easier than to the SHARC. Section 4 enumerates the various pros and cons of each chip. It is up to the user to make the trade-offs between conflicting goals in deciding which chip to use for which processing step in the application.

fabric may be a bus, a series of separate buses connected by bridges, a set of fixed point-point links, or a configurable multi-transaction interconnect network such as Mercury's RACEway. The individual paths that comprise this fabric may be either serial or parallel links.

A corresponding high-level diagram of a typical Mercury RACE multicomputer is illustrated in Fig. 2. The system consists of a variety of processor, I/O, and bridge nodes connected to the terminal ports of the RACEway interconnect. The salient features of this system are as follows:

- Use of Mercury's RACEway high-bandwidth crossbar switching network as the multiprocessor interconnect fabric.

- Self-contained CNs with local DRAM and DMA controllers.

- Ability to include different processor types in the multicomputer (heterogeneous CNs).

- Use of bridges between RACEway and standard buses such as VME, VSB, and PCI.

- Complete modularity and scalability. The system is modular in that any type of node (i.e., CN, I/O, etc.) can be connected to any terminal port on RACEway. All nodes present the same interface to RACEway.

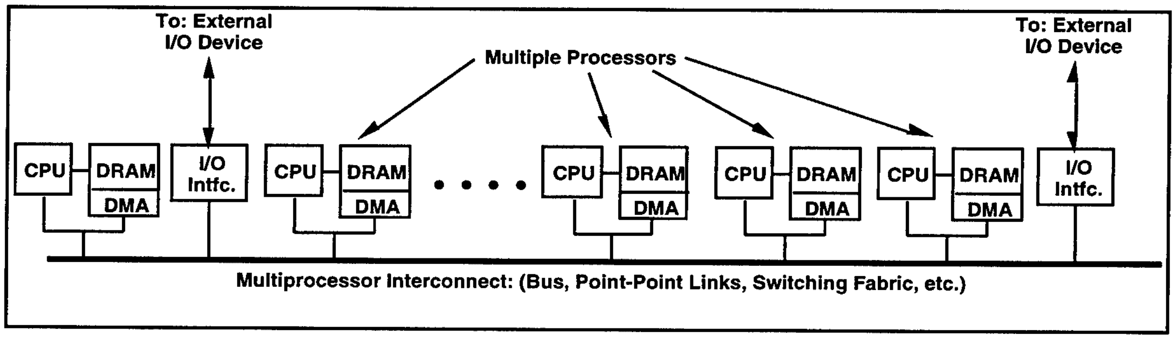


Fig. 1. Typical Multicomputer System

2.0 Mercury's RACE Multicomputer

Fig. 1 illustrates a high-level block diagram of a typical multicomputer. The system portrayed in Fig. 1 is composed of three generic-node types: processor nodes, I/O nodes, and bridge nodes, all of which are interconnected by a common data communication fabric. This interconnection

The system is scalable in that it can be expanded merely by expanding the RACEway interconnect to provide more terminal ports and then populating them with additional CN, I/O, or bridge nodes.

All nodes in the multicomputer have a common interface to the RACEway crossbar switching network consisting of 32 parallel data lines and 8 control lines. Data is transferred between each node and the crossbar network synchronously at a

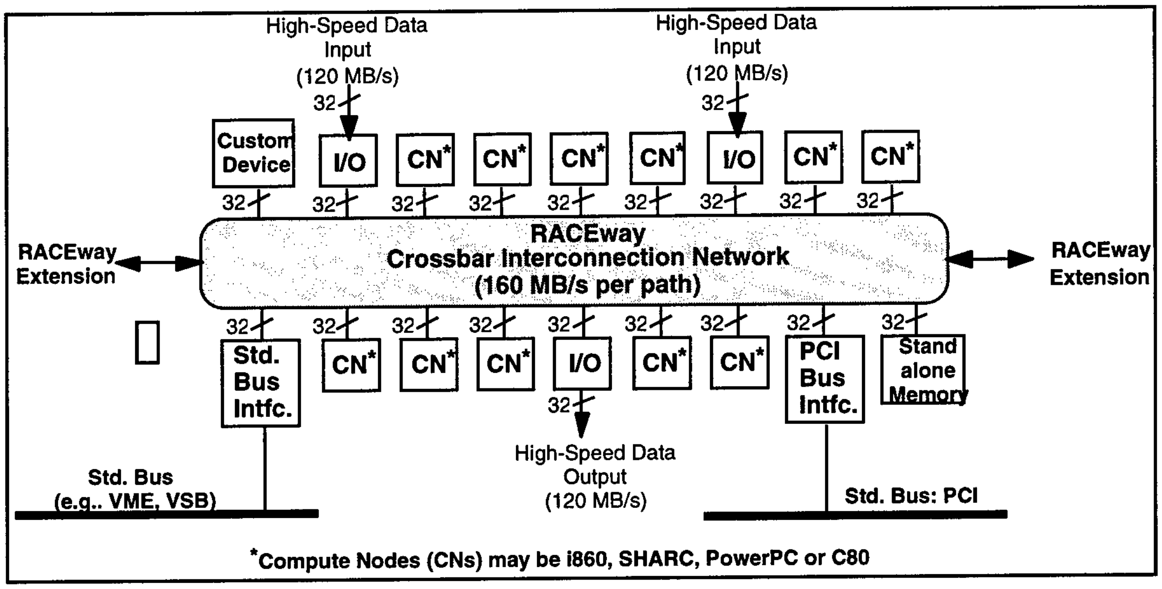

High-Speed Data Input (120 MB/s)
High-Speed Data Input (120 MB/s)
32
Custom Device
I/O
CN*
RACEway Crossbar Interconnection Network (160 MB/s per path)
RACEway Extension
RACEway Extension
Std. Bus Intfc.
PCI Bus Intfc.
Stand alone Memory
High-Speed Data Output (120 MB/s)
Std. Bus (e.g.. VME, VSB)
Std. Bus: PCI
*Compute Nodes (CNs) may be i860, SHARC, PowerPC or C80

Fig. 2. RACE Multicomputer

clock rate of 40 MHz, providing a data bandwidth of 160 MB/s per path.

A RACE crossbar switching network is composed of six-port crossbar switches. Each switch can either interconnect any three port pairs from among the six crossbar ports to each other, providing an aggregate data transfer bandwidth of 480 MB/s, or can cause data entering one of the six ports to be broadcast to some subset of the remaining five ports on that crossbar. A large number of different network topologies may be configured from these switches. The aggregate data bandwidth of a given network is equal to 160 MB/s, times the number of independent parallel paths provided by that network. For large networks, this aggregate network bandwidth can reach several GB/s. Furthermore, the resulting crossbar networks are easily expandable to accommodate the addition of more CNs to a given multicomputer.

Expansion of a given crossbar network not only provides more node ports to which additional CNs may be connected, but also increases the aggregate network bandwidth by adding more transaction paths that can be used concurrently. Further details on the operation and capabilities of Mercury's RACEway crossbar switching network may be found in Refs [1], [2].

Fig. 3 shows a "thinned" Fat -Tree architecture of the generic RACEway interconnect that was illustrated in Fig. 2. The various compute, I/O and bridge nodes that comprise the multicomputer illustrated in Fig. 2 are shown connected to the bottom level of this inverted tree architecture.

The "thinned" Fat -Tree architecture illustrated in Fig. 3. is not the only network interconnect topology that can be constructed with the currently available six-port RACE crossbar switches. Other network topologies that could be constructed with these switches include two-dimensional (2-D) and three-dimensional (3-D) meshes and rings.

Note that the scalability of the RACE multicomputer derives from the following two attributes of the RACE architecture.

- Scalability of the RACEway crossbar interconnection network. A given network may be expanded either by adding additional crossbars or by interconnecting sub-networks of crossbars.
- Modularity of the system nodes. The system nodes (CNs, I/O, and bridges) may be considered both physical and logical building blocks that can be plugged into any terminal port of the crossbar network. This is somewhat analogous to plugging a telephone or FAX machine into any port of a telephone network.

Of particular interest with regard to heterogeneous computing are the CNs illustrated in Figs. 2 and 3. Each CN consists of the following basic items: from one to three CPUs, all of the same type; DRAM memory of between

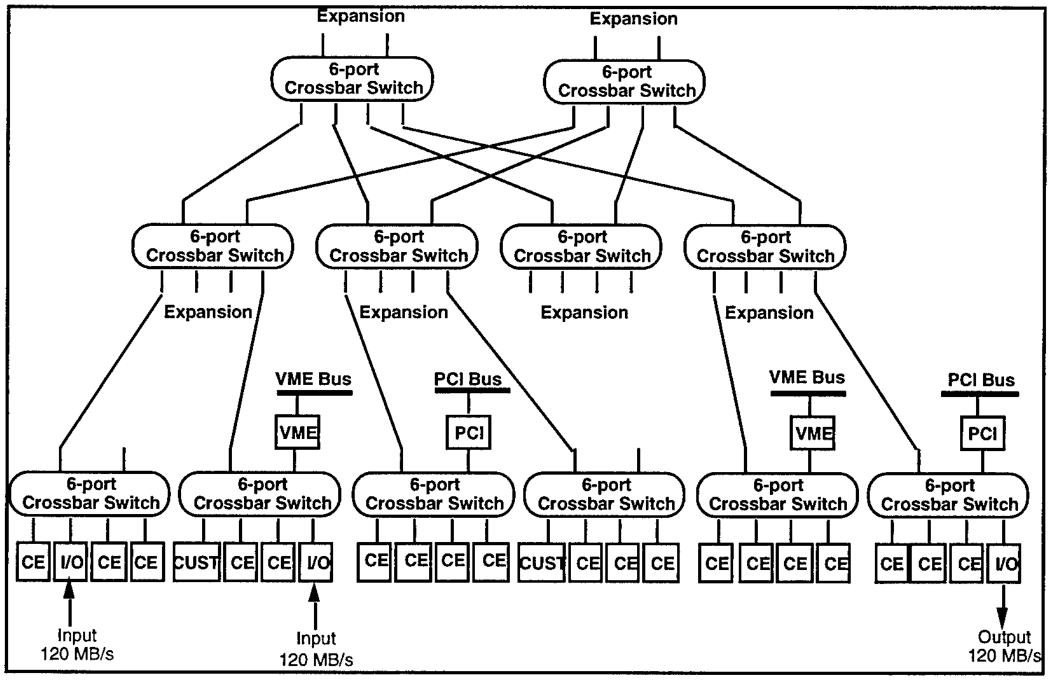


Fig. 3. Thinned Fat-Tree Crossbar Switching Network

8 and 64 MBs; and a RACEway CN interface ASIC.

The principal components of the CN ASIC are: address mapping logic that enables the local CN to access any DRAM location in any remotely located CN on RACEway; a DMA controller for performing rapid transfers between local DRAM and any other remote CN, I/O, or bridge node on RACEway; processor-support functions such as timers; and finally, interfacing logic for effecting RACEway transfers.

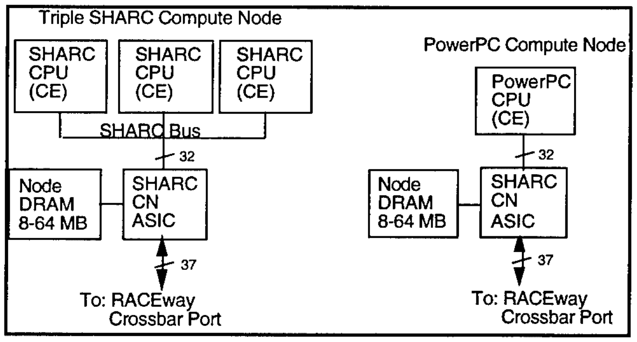


Fig. 4. SHARC and PowerPC Compute Notes

Clearly, there is a unique CN for each processor type. However, every CN ASIC, regardless of type, has the same interface to RACEway. Currently, RACE CNs are available for the i860, SHARC, and PowerPC.

Fig. 4 describes the architecture of the CNs for the SHARC and PowerPC. The SHARC CN consists of one to three SHARC CPUs sharing a common RACEway interface and DRAM. Multiple SHARC CPUs are connected together on a common internal bus. In contrast, the CN for the PowerPC consists of only a single CPU, together with a DRAM and CN RACEway Interface.

Each CPU in a multi-CPU CN functions as an autonomous CE that executes independently of any other CE. Therefore, each CE operates under control of its own copy of Mercury's MC/OS runtime environment. Thus, the SHARC CN consists of three independently executing CEs that happen to share a common DRAM and RACEway interface.

In contrast, the PowerPC CN consists of only a single CE, making the distinction between a CE and CN somewhat moot in that case. In summary, a CN is defined as a computational node that attaches to RACEway, whereas a CE is defined as an independent processing element. In the case of multiple CEs that are connected to a common node, as in the case of a multiple SHARC node, each CE operates under control of its own copy of the operating system and does not necessarily have to execute the same application code as the other CEs at that node.

3.0 Mercury's Multicomputing Software

The basic software components of the Mercury RACE multicomputer are:

- The MC/OS runtime environment
- The Scientific Algorithm Library (SAL)
- Cross-compilers and cross-assemblers for each supported processor type

A different version of each of these software components is required for each supported CE type; however, the APIs to these components are identical across all CE-specific versions. This makes the user's application code nearly independent of processor type. However, to get optimal performance on a given processor, the user should be aware of each processor's features and limitations, and tailor the code accordingly. In most cases, execution of the user's application code on any given processor will be optimized by using the SAL functions wherever possible. All functions in the SAL library have been assembly language coded individually for each processor, to provide optimal performance on that processor.

A separate copy of MC/OS executes in each CE of the multicomputer. These copies are functionally identical; however, as was mentioned earlier, a different version is used for each processor type.

MC/OS consists of the following three principal components.

- **MCexec**
  A standard, uniprocessor multitasking real-time OS that handles task scheduling, timers, and interrupt servicing.

- **ICS**
  A component that handles all interprocess communications. It allocates and manages shared-memory buffers in both local and remote CNs, does address mapping of remote nodes, and manages all data transfers between local and remote shared-memory buffers using either the local CE's DMA controller or programmed I/O, as appropriate.

- **HAL**
  This is the processor-specific code that forms the interface between MCexec, ICS and the given processor's DMA controller, timer, interrupt control and other internal registers.

MCexec and ICS both operate on top of the HAL. While the HAL code varies from processor to processor, the implementations of MCexec and ICS are largely processor-independent. Therefore, although a new version of HAL must be supplied for each new processor type implemented in the RACE multicomputer, existing versions of both MCexec and ICS are largely portable between different types of processors. Fig. 5 illustrates how the different components of MC/OS relate to each other and to the hardware of the host processor.

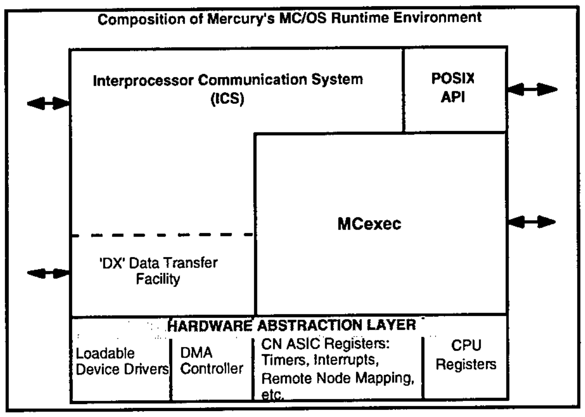

Composition of Mercury's MC/OS Runtime Environment
Interprocessor Communication System (ICS)
POSIX API
MCexec
'DX' Data Transfer Facility
HARDWARE ABSTRACTION LAYER
Loadable Device Drivers
DMA Controller
CN ASIC Registers: Timers, Interrupts, Remote Node Mapping, etc.
CPU Registers

Fig. 5. MC/OS Organization

The basic size of MC/OS varies between 550 and 700 KBs, depending on the CE type. Additional memory is also required at each CE for the Configuration Data Base (CDB) and any temporary buffers that may be dynamically allocated by MC/OS. The CDB is a table in each CE's DRAM that contains the crossbar routing information from that CE to every other CE in the system; clearly, the size of the CDB increases in proportion to the number of nodes in the multicomputer. The total memory consumed by each copy of MC/OS and its associated tables is approximatley 700 KBs to 1.2 MBs, depending on the CE type and size (i.e., number of CEs) in the multicomputer. The copy of MC/OS and its associated tables for a given CE is stored in that CE's local-node DRAM memory. Recall that in the case of CNs that consist of multiple CEs, a separate copy of MC/OS is stored in the common CN DRAM, for each CE.

4.0 Heterogeneous Computing with SHARC and PowerPC CEs

A currently popular multicomputer configuration is one based on a mixture of the SHARC 21060 and PowerPC 603p processing chips. Architecturally, these are two very different processors. The PowerPC is a RISC chip, and the SHARC is a DSP chip. The principal functional differences between these two chips are:

- type of fast, on-chip memory used
- processor clock speed

The 200 MHz PowerPC uses a 16 KB write-back cache having a bandwidth of 800 MB/s, while the SHARC uses a 512 KB on-chip SRAM having an effective bandwidth of up to 480 MB/s. With regard to clock speed, the PowerPC 603p operates at a clock speed of 200 MHz, while the SHARC operates at a clock speed of only 40 MHz. In addition, the SHARC's architecture has been optimized for the computation of FFTs, with the ability to execute the following three instructions, in parallel, in one clock cycle:

- (a+b)
- (a-b)
- c*d

However, the 5:1 clock-speed ratio between the PowerPC and the SHARC does not necessarily mean that the PowerPC has five times the throughput of the SHARC. For some DSP operations, such as large FFTs, the SHARC can execute up to twice as fast as the PowerPC, despite its considerably lower clock speed.

Furthermore, the choice of "best" processor for a given application depends not only on the specific algorithm to be implemented, but also on power and space contraints imposed by the application. The latter is an especially important issue in embedded systems. For these and other reasons, a heterogeneous multicomputer comprised of both SHARCs and PowerPCs, is often the optimal (in terms of size and cost) solution for many applications.

4.1 Processing Attributes of the PowerPC

Specifically, the PowerPC is best suited for executing algorithms that have a high ratio of computation-to-DRAM memory accesses, (>10 operations per floating point data access), such as transcendental functions, matrix inversion, and FFTs etc., with the exception of large-size FFTs. The PowerPC is also the processor of choice for scalar processing applications written in C and for performing double-precision arithmetic.

The reason is that the PowerPC can execute a floating-point instruction between 1.3 and 3.0 clock cycles, resulting in throughputs of between 67 and 167 MFLOPS, provided that all operands and results were previously cached. Even higher throughputs are possible for compound operations in which the intermediate results can be retained in the CPU's internal registers rather than being written to and then read back from cache between successive operations. Indeed, the C compiler for the PowerPC takes maximum advantage of the CPU's internal registers to produce highly efficient code. In general, however, the principal limitation on the PowerPC's floating-point execution speed is the single-precision cache bandwidth of 800 MB/s (200 M combined single-precision floating-point operand and result cache accesses per second).

Eventually, however, new data must be accessed from, and the corresponding results written back to, DRAM. When this occurs things can slow down dramatically. For sake of argument, assume a 160 MB/s DRAM. Because of the operation of the write-back cache, the effective DRAM access rate of the PowerPC is reduced from 160 MB/s to between 50 and 80 MB/s — more than an order of magnitude less than the cache bandwidth of 800 MB/s. As a result, operations characterized by a low ratio of computation-to-DRAM accesses ($\leq$ 10 operations/floating-point operand accessed) become I/O-bound by the above-cited DRAM-cache access limit. This causes the effective throughput to decrease dramatically. For example, in the case of three-access floating-point operations, (i.e. those that produce a single result from a pair of operands), the aforementioned DRAM/cache access bandwidth constraint can reduce the sustainable throughput to a value as low as 4 MFLOPS.

However, computationally intensive operations such as those cited earlier, tend to be compute-bound, and are thus less effected by the decrease in DRAM data access bandwidth associated with the operation of the write-back cache.

As mentioned earlier, an exception to the above occurs for complex FFTs (in-place) longer than 1024 points (512 points for out-of-place FFTs). Although the ratio of computation-to-data accesses for FFTs increases with FFT size, the limited size (i.e., 16 KB) of the PowerPC cache now places a new limit on performance. A complex input data stream longer than 1024 elements (8 KBs), together with the required FFT weights, will not fit into a 16 KB cache. Thus, complex FFTs of length 2048 or greater must be decomposed into a 2-D array whose maximum row length is less than or equal to 1024. Each row of this array is then accessed, its FFT computed, and the transformed rows written back to DRAM. The resulting *columns* of this 2-D array are then accessed from DRAM, multiplied by a complex "twiddle factor" vector, and the corresponding column FFTs are computed. Although this process is handled automatically by the SAL and is transparent to the user, the DRAM accesses associated with this 2-D FFT decomposition process slow down FFT throughput significantly.

In summary, the PowerPC is best suited for computation of the following types of algorithms:

- Algorithms (with the exception of large FFTs) that have a ratio of computation to data accessed in excess of 10 floating-point operations per data item accessed from/written to DRAM.

- C-coded scalar processing algorithms.

- Algorithms or procedures whose implementation requires more than 50 KBs of code.

- Computations requiring double-precision arithmetic.

In contrast, the PowerPC is poorly suited for processing long vectors (including large FFTs) whose length is such that the combination of all vector operands and results exceeds the size of the 16 KB cache.

4.2 Processing Attributes of the SHARC

The SHARC is ideally suited for processing of vectors of any length, including FFTs of all sizes. Indeed, the SHARC has been customized for efficient computation of FFTs, as it can perform the following two additions and a multiply (i.e., three operations) in one clock cycle:

- (a+b)
- (a-b)
- c*d

This gives the SHARC a throughput rate for FFTs of nearly 120 MFLOPS. In general, the throughput limits for the SHARC are as follows:

- 120 MFLOPS for FFTs
- 80 MFLOPS for multiply-accumulate operations (e.g. dot products)
- 40 MFLOPS for most other floating-point operations

To realize the above rates, all operands and results must be accessed from/written to the SHARC's 512 KB SRAM. This on-chip SRAM effectively plays the role of a cache. However, unlike the PowerPC, the operation of the SHARC rarely becomes I/O-limited.

The SHARC's 512 KB SRAM is divided into two independent banks of 256 KBs each. Furthermore, each of these two banks is dual-ported; each bank having both a processor and an I/O port. These dual ports enable either SRAM bank to be simultaneously accessed by both the processor and external DRAM memory at combined access rates of up to 320 MB/s (i.e. 160 MB/s through each of the two ports). In addition, the SHARC CPU also has two independent buses that are connected to both SRAM banks through a multiplexor, enabling the CPU to make separate data accesses to both memory banks simultaneously, at rates of 160 MB/s from each bank.

In summary, the SHARC has three internal data buses that can each simultaneously make accesses to this SRAM, two CPU buses and an external I/O bus to DRAM. Simultaneous accesses can be made to either of the two SRAM banks by the external I/O bus and one of the two CPU buses. Simultaneously, the other CPU bus can also access data from the other SRAM bank. Each of these three buses has a bandwidth of 160 MB/s. Therefore, data can be transferred between external DRAM and the on-chip SRAM at 160 MB/s, while the CPU is simultaneously accessing (read and/or write) data from/to SRAM over its two internal buses at an aggregate rate of 320 MB/s.

The bottom line is that on the SHARC, data can always be moved between DRAM and SRAM at rates of up to 160 MB/s without impacting CPU activity. As a result, the SHARC has a better balance of processing and I/O throughput for operations that are characterized by a low ratio of processing to data accesses (i.e., less than 10 floating-point operations per DRAM data access). On the PowerPC, the processor throughput for such operations becomes throttled to rates of between 4 and 10 MFLOPS by the effective cache-DRAM access bandwidth limit of 50 to 80 MB/s. However, on the SHARC, because of the greater DRAM access bandwidth available, such operations can potentially execute two to three times faster than on the PowerPC, despite the SHARC's slower clock speed.

In summary, the SHARC is best suited for the following types of applications.

- Computation of FFTs of any length. Effective throughput rates in excess of 100 MFLOPS can be sustained for all FFT sizes. Complex FFTs larger than 4K must be decomposed into a 2-D array and the intermediate results written back to DRAM, as described previously for the PowerPC. However, in the case of the SHARC, much of the requisite I/O between SRAM and DRAM can be overlapped with ongoing computation so that the associated computations never become I/O-bound.

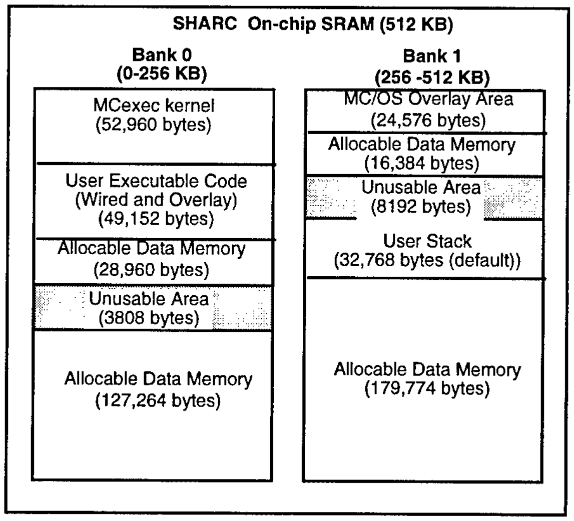
- Processing of long vectors. Because I/O and computation can be overlapped on the SHARC, long vectors can be strip-mined without incurring a performance penalty.

- Executing algorithms that are characterized by a low ratio of computation-to-memory accesses.

- Applications in which there is a premium on physical density (MFLOP/volume, MFLOP/watt).

The above attributes make the SHARC ideal for repetitive, FFT-intensive, embedded signal processing applications such radar, sonar, communications and medical imaging.

In contrast, the SHARC is poorly suited for execution of user application programs whose executable code occupies more than 50 KBs of memory, involve the use of double-precision arithmetic, and/or programs that make frequent use of MC/OS services. The reason is that the SHARC can execute only code located in SRAM and does not have double-precision hardware. However, only about 50 KBs of the 512 KBs in SRAM are allocated to the storage of user code. The remainder is allocated to segments of MC/OS and data. A map of SRAM memory allocations for a Mercury SHARC CN is illustrated in Fig. 6.



SHARC On-chip SRAM (512 KB)
Bank 0 (0-256 KB)
Bank 1 (256 -512 KB)
MCexec kernel (52,960 bytes)
User Executable Code (Wired and Overlay) (49,152 bytes)
Allocable Data Memory (28,960 bytes)
Unusable Area (3808 bytes)
Allocable Data Memory (127,264 bytes)
MC/OS Overlay Area (24,576 bytes)
Allocable Data Memory (16,384 bytes)
Unusable Area (8192 bytes)
User Stack (32,768 bytes (default))
Allocable Data Memory (179,774 bytes)

Fig. 6. SHARC SRAM Allocations

Note that not even all of MC/OS can fit into SRAM! Consequently, only the MCexec kernel is loaded into SRAM. All other MC/OS services, such as the ICS routines, are loaded into SRAM as overlays on an as-needed basis. Similarly, only 50 KBs of SRAM are allocated to storage of the user's application code. User programs consisting of more than 50 KB of code must be broken up into multiple overlay segments, each smaller than 50 KBs, that can be subsequently loaded, as needed, from DRAM into the allocated 50 KBs segment of SRAM.

Although this overlay mechanism does allow user code that extends over more than 50 KBs to be run on the SHARC, each overlay takes up to 333 usec to complete, during which time the CPU is idled. Although this amount of overhead may be tolerable in certain applications, it usually is unacceptable in most high-performance real-time signal processing applications. Consequently, execution of application programs that contain more than 50 KBs should generally be avoided unless it is determined that the overhead associated with bringing in each new overlay can be tolerated. Similarly, the execution of software such as I/O drivers, that may cause MC/OS overlays to be invoked, should also be avoided on the SHARC.

Another potential limitation of the SHARC relative to the PowerPC is that the SHARC is somewhat more difficult to program than the PowerPC. This is primarily so for two reasons.

First, in order to get maximum performance from the SHARC, the user should write the code in terms of calls to Mercury's SAL functions to the greatest degree possible. Use of SAL functions is much more efficient than executing compiled C code on the SHARC.

Second, unlike the case of cache, the user *must* explicitly manage all data transfers between DRAM and SRAM, as well as being aware of which SRAM memory banks his data is stored in. For optimum performance, the results of an operation (i.e., a SAL call) should generally be stored in the opposite SRAM memory bank from the operands. Furthermore, the user should also arrange all data transfers between SRAM and DRAM so they are performed concurrently with program execution, using either the on-chip or the external DMA controller. Generally, the on-chip DMA controller should be used to write data from SRAM to DRAM, while the external (on the CN ASIC) DMA controller is used to write data from DRAM to SRAM. Mercury supplies a set of data transfer routines for the SHARC, called the Multi-buffering Facility, that largely automates this process and makes use of these two DMA controllers transparent to the user. However, the user is still responsible for assigning the data buffers to the proper SRAM memory banks and for making sure that the SRAM-DRAM I/O operations are overlapped with program execution as much as possible. Failure to do so will result in sub-optimal performance.

5.0 A Sample Heterogeneous Processing Application

The processor topology of a multicomputer for a typical dataflow processing application is illustrated in Fig. 7. The processing operations to be performed are first partitioned into a number of sequential sub-process processing stages. At each stage, the data to be processed is subdivided into segments that are distributed over a corresponding number of processors, each of which executes the same sub-process independently and in parallel, on its associated data segment.

A processing stage is defined as a sequence of processing steps that can be performed in parallel on different segments of data, without requiring communication with any other processors at that stage. A new processing stage is defined whenever the nature of the data segments to be processed changes. Therefore, a redistribution of data always occurs between any two adjacent processing stages. The number of stages into which a given process is partitioned is a function of the application.

A 2-D FFT is a simple example of a simple two-stage processing application. The data to be processed by a 2-D FFT may be considered to be a 2-D array, or matrix. The first processing stage consists of performing the row FFTs, in which the rows of the matrix are distributed over the processors assigned to that stage. The transformed row output(s) of each processor in this first stage, are then redistributed among the processors in the second stage so that each second-stage processor receives an integer number of matrix *columns*. The second-stage processors then perform the requisite column FFTs to complete the computation of the 2-D FFT.

In the multicomputer dataflow model of Fig. 7,

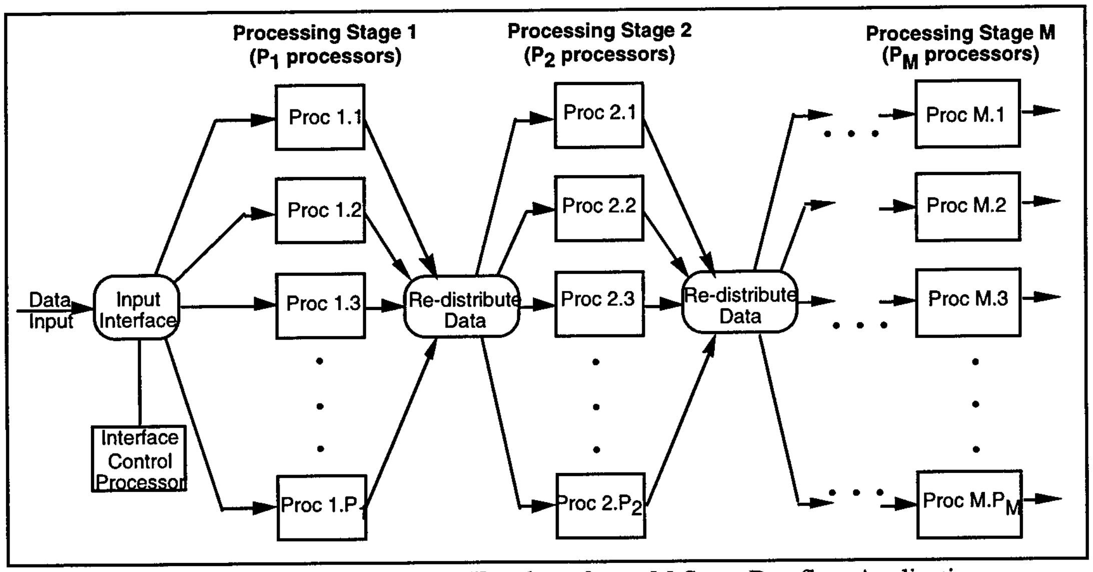


Fig. 7. Multicomputer Processor Topology for an M-Stage Dataflow Application

all processors used within a given stage would be the same type. This makes sense because all processors used at a given stage execute the same algorithm in parallel, albeit on different data. However, the processor type used at a given stage may vary from one stage to the next, depending upon the nature of the algorithm being implemented at each stage.

The general rules for choosing the optimum processor for a given processing stage, as a function of the algorithm being implemented to maximize throughput, were described in the preceding section. In general, FFT-intensive processing algorithms and those algorithms characterized by a low ratio of processing-to-data accesses are more efficiently implemented using the SHARC processor. In contrast, algorithms that are highly computationally intensive -i.e., those characterized by a high ratio of computation to data accesses — such as those involving transcendental functions, iterative procedures, or complex multi-point interpolation — are usually more efficiently implemented using the PowerPC. The PowerPC would also be the processor of choice for algorithms or processes whose implementation entails more than 50 KBs of code, or that require extensive interaction with the operating system. Typical applications include I/O drivers, system-name servers, system control programs, etc.

6.0 Conclusion

The rationale for a heterogeneous multicomputer has been described. A specific example of such a system, a Mercury RACE multicomputer composed of SHARC and PowerPC processors, has been presented. It was shown that each of these processors is ideally suited for different types of algorithms.

Briefly, the SHARC is optimized for execution of vector-oriented signal processing algorithms, especially those involving FFTs. One of the principal limitations of the SHARC is that in order to realize maximum processing throughput, the compiled code to be executed should not exceed 50 KBs in length. Observance of this constraint avoids the need for code overlays. In addition, to obtain peak performance from the SHARC, the user's code should make use of Mercury's SAL routines wherever possible.

In contrast, the PowerPC is generally superior for execution of algorithms or processes that are characterized by a high ratio of computation to data accesses, such as those involving transcendental functions, iterative procedures or complex multi-point interpolation. The PowerPC would also be the processor of choice for algorithms or processes whose executable code is more than 50 KBs in length, that involve double-precision arithmetic, or those that require extensive interaction with the operating system.

A typical multicomputing application generally consists of a sequence of different types of algorithms, the execution of each of which may be optimized by one or the other of the above two processor types. When each algorithm is implemented on its corresponding optimal processor type, the result is a heterogeneous multicomputer.

The above preferences are only general guidelines that assume maximum performance per processor is the only parameter to be optimized. However, in a practical, real-world application, other factors such as processor density (i.e. MFLOP/m^3 and/or MFLOP/watt) and ease of programming may be of equal or greater importance than just minimizing the total number of processors. In this regard, the SHARC usually offers the highest processing density, while the PowerPC is easier to program.

In summary, although processor heterogeneity provides a means of optimizing a multicomputer configuration for a given application, there are usually a number of different optimization criteria that should be considered such as:

- Minimization of processor count
- Minimization of total system hardware cost
- Maximization of processor density (especially in embedded systems)
- Minimization of software development costs (i.e. ease of programming)

In general, each of the above objectives, taken alone, results in a different optimal heterogeneous solution. Every situation is different, and it is up to the user to make trade-offs between the various optimization criteria listed above that are appropriate to a specific situation.

References

[1] "The RACE Multicomputer" Vol. 1, Hardware Theory of Operation - Processors, I/O Interfaces, and the RACEway Interconnect, Mercury Computer Systems, Inc., 1995.
[2] Einstein, Tom, "RACEway Interlink - A Real-Time Multicomputing Interconnect Fabric for High-Performance VMEbus Systems," VMEbus Systems magazine, February, 1996.

Dr. Thomas Einstein is a consulting systems engineer at Mercury Computer Systems, Inc., where he serves as a technical consultant on radar and other defense signal processing applications of Mercury's high-performance multicomputers.

Prior to joining Mercury in 1993, Dr. Einstein was a systems engineer at Atlantic Aerospace Electronics Corporation, where he served as the chief systems engineer on the company's milimeter wave radar program. Previously, he was a staff member at MIT Lincoln Laboratory for 24 years where he was involved in the development of a number of ground-based and airborne radar systems. He began his career in defense electronics while serving in the U.S. Navy as a shipboard electronics officer on the destroyer USS Borie, DD-704.

Dr. Einstein holds a ScD in mechanical engineering from MIT, a MS in instrumentation engineering from the Case Institute of Technology, and a BS in mechanical engineering from the University of New Hampshire.

Session 2

Mapping and Scheduling Systems

Session Chair

John Antonio
Texas Tech University, Lubbock, TX, USA

A Scheduling Expert Advisor for Heterogeneous Environments

Mihai G.Sirbu and Dan C. Marinescu
Computer Sciences Department
Purdue University, West Lafayette, IN 47906, USA
{sirbu, dcm}@cs.purdue.edu

Abstract

In this paper we discuss intelligent agent support for parallel and distributed computing in a heterogeneous environment. We provide an overview of the Bond environment and of services provided by a network on intelligent agents, then we discuss in depth the Scheduling Expert Advisor, SEA. The SEA processes a high level description of a computational task provided by a user and converts it into a set of facts and rules. An expert system starts the execution of one program or a group of programs based on the scheduling information compiled earlier.

Intelligent agent support for heterogeneous parallel and distributed computing

While intelligent agents [5, 6, 7] are used extensively for information retrieval and data mining, there are virtually no reports of their application in the area of parallel and distributed computing. In this paper we discuss an environment for parallel and distributed computing and present one of it major components, the Scheduling Expert Advisor. A feature distinguishing the Bond environment from other efforts in this area is the extensive use of knowledge processing. It seems natural that in a distributed environment based upon the client-server paradigm, at least some of the services be provided by intelligent agents.

The task of accommodating heterogeneity poses challenges difficult to carry out by less sophisticated means then knowledge processing. Take for example data and program migration, one of the activities needed in such an environment. Data migration can be accomplished by a script including commands to `tar, compress, ftp, rlogin`, etc. But each of the steps mentioned above may fail and a script able to handle such errors is likely to be very complex. When one adds the requirement to move data among systems with different operating systems e.g. Unix and NT, this solution becomes impractical. For example, the task of finding if enough space is available on the target system, one of the low level actions performed during data and program migration is considerably easier to implement as a set of facts and rules than as a script.

The intelligent agents in the Bond environment are specialized expert systems acting as servers able to perform tasks like program migration, data migration, scheduling, mapping, exporting objects, and so on. At the heart of the Bond systems are resource databases, which provide information about all the objects available to individual members of a group. Programs, data, and hardware objects are shared or used exclusively by the members of the group. The information about the services available in the system is provided by a name server, the *oracle,* running at a known port. All services including those provided by intelligent agents register themselves with the oracle.

The Scheduling Expert Advisor works in a network of expert advisors to accomplish its task. The Mapping Expert Advisor selects a target system, the Data Replication and the Program Migration Expert Advisors make the programs and data objects needed for the computation available at the target system.

To exploit the benefits of knowledge processing we had to provide effective mechanism for the intelligent agents to collaborate with one another, and to adapt their behavior according to the feedback provided by the environment as a result of their action. The major contributions of this paper are such mechanisms. In the Bond environment, the facts and the rules used by an inference engine are modified dynamically as a result of user interactions, actions of other intelligent agents as well as feedback from the environment.

The Scheduling Expert Advisor, SEA, is developed in Clips [1-3]. Additional functions for socket communication are written in C. SEA interacts with clients through TCP sockets using ASCII strings. We consider using a KQML [4, 11] interface for the expert advisors. Clients and test programs are written in Tk [12] and Expect [10].

0-8186-7879-8/97 $10.00 © 1997 IEEE

Rule-based expert systems

This section gives artificial intelligence background and provides some insight into the operation of expert systems. An expert system starts with information about an abstract universe model and then infers additional knowledge [9]. The new knowledge can be stored in the form of both facts and rules. The following discussion follows loosely the CLIPS language [1-3, 8], but the presented concepts are valid for any rule-based system. In an expert system, information is stored as *facts* (individual items) and *rules* (algorithmic knowledge). A fact stores knowledge about the problem universe, and is represented as an n-tuple ($n \geq 1$), in which the first element is a fact identifier and the other optional elements are fact arguments. A rule represents procedural information, and is a construct of the type:

```
IF (antecedent) THEN (consequent)
```

Alternative component names are *Left Hand Side,* LHS, for the antecedent, and *Right Hand Side*, RHS, for the consequent. If all the terms of the antecedent are true, the rule is activated. The system triggers one of the activated rules, and evaluates the expressions of the RHS in sequential order.

Figure 1 illustrates the block architecture of a rule-based expert system. The facts are stored in the *working memory* and the rules are stored in the *production memory.* The *Inference Engine* (IE) runs a three-step infinite cycle of matching, selecting and execution. The first step matches the available facts against all rules. This is done by special algorithms to improve efficiency and avoid combinatorial complexity. The activated rules are placed on a list called the system's *agenda.* Next, the IE sorts the agenda and selects the top rule for execution. The sorting criteria is central to the operation of the expert system. In the third step, the RHS actions of the triggered rule are executed. As side effects, changes in the working and production memory can occur. Usually only facts are changed, although the mechanism for dynamic rule changes is present. The original rule is then removed from the agenda to prevent repeated activation by the same facts. The IE cycle continues with a new matching step, and stops if no rule is activated.

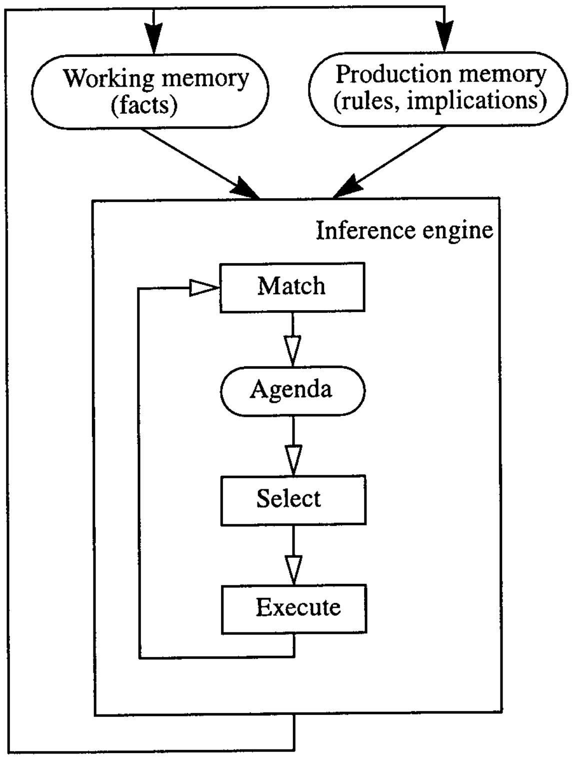


Figure 1: Architecture of a Rule-based Expert System

The Bond environment

The Bond environment [14] currently under development at Purdue University is designed to support concurrent execution of parallel and/or sequential programs on computing platforms with different architecture and system software, interconnected by a high speed network. We consider a model of parallel and distributed computing which allows an individual working in a group to provide a high level description of the problem to be solved and let an intelligent environment determine a sequence of actions optimal in some sense leading to the desired result. To accomplish this goal the environment has several inference engines and maintains a set of resource databases containing the description of the computing platforms and networks, information about the programs, the services, and the data available to the group, and to each individual within the group.

Bond is a groupware system which supports batch as well as interactive execution. It is designed to run on top of different operating systems, makes no assumptions concerning the communication libraries used by the parallel programs, and supports the management of hardware and software objects. It consists of a kernel, resource databases, remote services including Expert Advisors, and a user interface. The user interface provides access to a set of computing engines interconnected by a high speed network. The environment allows a user to provide a high level description of the problem to be solved, including execution and data dependencies. The Scheduling Expert Advisor converts this description into a set of complex tasks and returns a task schedule to the kernel. The Bond kernel uses other agents e.g. Program and Data Replication Advisors, the Mapping Expert Advisor, etc. to execute simple tasks. Each simple task implies running a program with a particular data set on a target system under the supervision of a Bond process. This supervisory process informs the environment about the outcome of the execution and allows the Scheduling Advisor to proceed with the scheduling of the next task or to attempt an error recovery procedure.

When activated, Bond creates a user environment, reflecting information from shared and private resource databases. The services and the Expert Advisors invoked in behalf of a user share the same view of the environment. The set of services and Expert Advisors are distributed and they can be accessed via an *oracle*. The system is open-ended, as new services are added they are registered with the oracle. Some of the services are replicated and the oracle directs a request for service to the server capable of providing the service in an optimal way.

Other Expert Advisors use facts stored in shared knowledge bases to determine if similar tasks have been carried out previously, and based upon the size of the current problem suggest alternative ways to carry out the computations, provide estimates of the execution time on different configurations. The Data Replication Advisor determines if the data needed for the computation is available at the execution site and performs a variety of operations related to data staging. For example it determines if enough storage space is available at the execution site, then establishes if data conversion is necessary, if so decides where it should take place, compresses and eventually encrypts the data and finally makes a copy of the data at the execution site. The Program Movement Advisor provides similar functionality for program staging. When the remote execution completes, the EA extracts the relevant facts and stores them into shared knowledge bases.

Overview of the scheduling expert advisor

This paper shows the use of an expert advisor for the scheduling of complex program execution sequences. The data and execution dependencies of the component programs are encoded in a set of facts and rules which control the expert system. Rule activation models the scheduling of programs which have all their dependencies satisfied. The process is simple and has significant advantages over the static approach using scripts.

The Scheduling Expert Advisor, SEA, is a layer positioned between the problem description provided by the user and the Bond execution environment, as shown in Figure 2. The SEA processes the High Level Description,

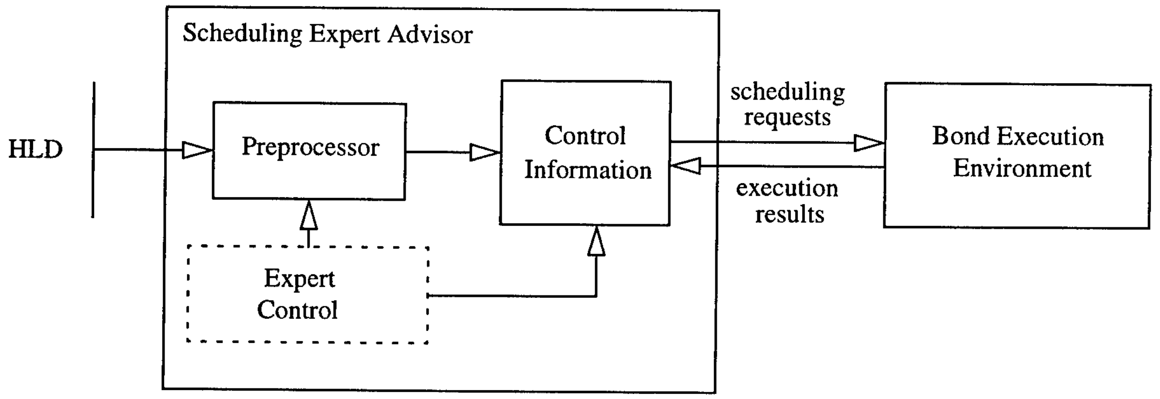


Figure 2: The Scheduling2 Expert Advisor processes a high level description of the problem and converts it into Scheduling Control Information (facts and rules). Then it sends scheduling requests to the environment.

HLD, and generates a knowledge-based representation of the problem. Next, the SEA submits program scheduling requests to the Bond execution environment. The exit status of the executed programs is returned to the SEA, which updates its internal state. Successful executions validate conditions for the other programs, which are then scheduled. The user can program actions to be taken in case of program failure, for example start an expert advisor specific for error recovery or take direct control of the execution.

The High Level Description, is processed by the SEA and converted into a set of rules and facts called Scheduler Control Information, SCI. The process is similar to compiling a source program into intermediate code. The SCI is in fact an independent expert system which automatically schedules programs in the Bond environment according to the input HLD. The design principle follows the inference engine algorithm described in Section . In a rule-based expert system, the antecedents of each rule are matched with all the available facts. If all the conditions of a rule are satisfied, the rule is activated. One active rule is executed based on the selection mechanism. The scheduling of a program in a complex processing follows the same principle. Some conditions have to be satisfied before the program can be started. The most common conditions are data and execution dependencies. Data dependencies appear when a previous program has to terminate successfully to provide input data for the next step. An example of execution dependency arises when the programs in a group have to be co-scheduled to exchange intermediate results. A group is scheduled only when all the component programs are ready for execution. The basic idea behind the Scheduling Expert Advisor is to associate rules with the scheduling of programs, and antecedent conditions with execution and data dependencies. When all the dependencies are satisfied, the rule is activated and the program is scheduled.

An important difference between the SEA and an usual expert system is the asynchronous nature of the SEA program scheduling, presented in Figure 3. The schedul-

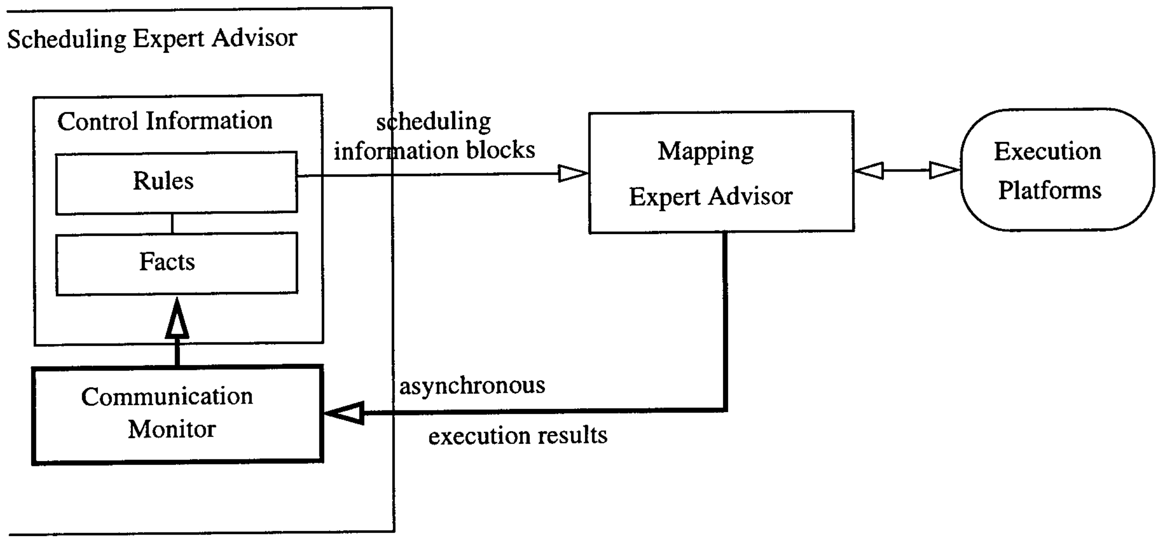


Figure 3: Asynchronous Communication with the SEA. The SEA sends requests to the MEA which in turn generates mappings. The Bond execution environment responds when a mapped activity completes. The MEA informs the SEA that the request was satisfied. The result is entered as a new fact of the SCI.

ing information is sent to the execution environment, but it is unknown how long the execution will take. Some programs might take hours or even days to complete. As such, scheduling of a program is a complete rule by itself. While an expert system normally runs only one active rule at a time, the SEA can schedule all ready programs at the same time, providing an added superconcurency bonus. The downside is that a connection has to be open to receive asynchronous return codes when an individual program terminates. The return codes are converted into facts which are placed in the working memory, activating in turn processing rules.

When a program is scheduled by the SEA, the necessary information is passed to the Mapping Expert Advisor, MEA, which selects an execution target, starts the program in the control environment, and monitors its execution. The completion code, `OK/Error`, is reported to the SEA. Only after a successful completion of a program its output files are available as input for other programs. The process continues until all available programs are executed. In case of a program failure additional information can be reported to the SEA for diagnostic and recovery purposes.

The SEA is loosely coupled with the user interface and with the Bond Execution Environment, BEE. Any interface able to generate the HLD of a problem workflow is acceptable. On the BEE side, SEA generates scheduling information blocks used by the Mapping Expert Advisor. Any program able to parse such a block can be used for scheduling execution, if it returns valid completion information.

Task execution specification

A graphical interface allows the user to specify the workflow of a given problem. The sequence of programs, with associated input and output files is described. Parallel execution of program groups are specified. Links are established between output and input files of various programs. The GUI translates this description into an intermediate representation which is parsed by the Scheduling Expert Advisor. We have named the graphical presentation of the workflow High Level Description, or HLD. This description is similar to different module interconnection languages [13].

The central HLD concept is the *Execution Block*, or *Block* for short. There are basic execution blocks and composite ones.The basic execution block is an executable program and its internal structure is presented in Figure 4.

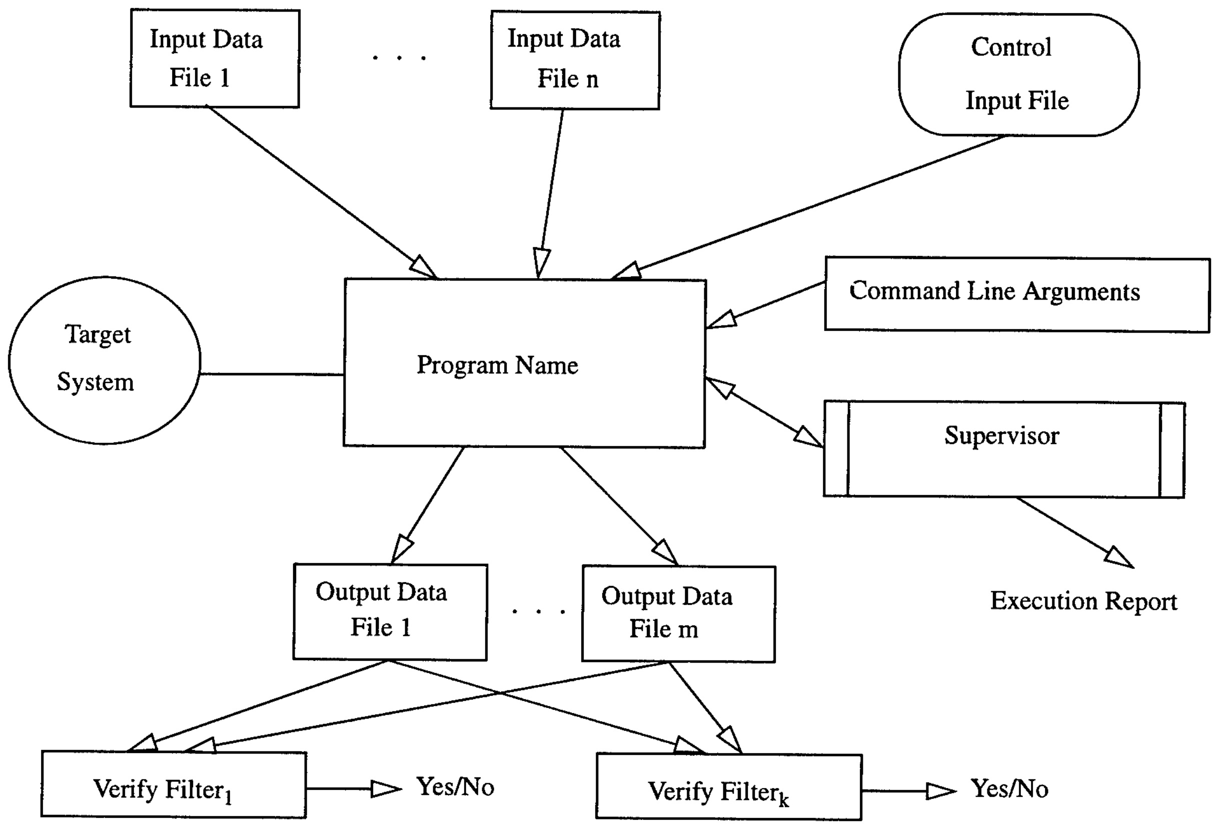
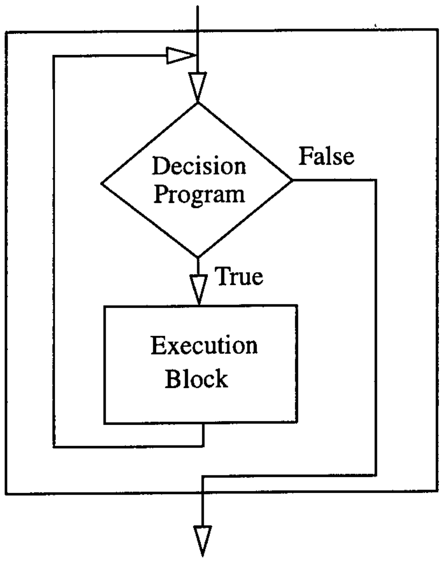


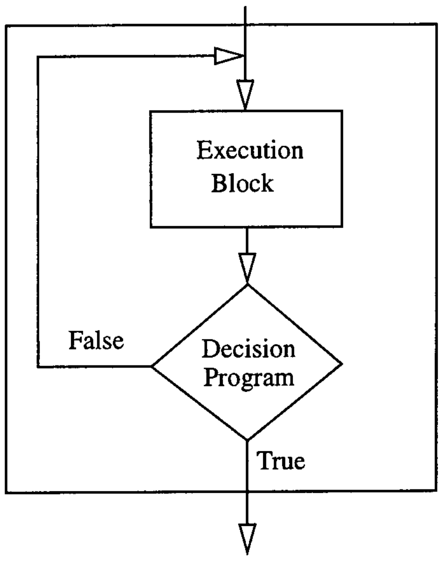
Figure 4: A basic execution block and its internal structure

Each basic block may have 0 to n *Input Files*. The *Control Input File* guides execution with specific options, so it is processed separately from the other input files. Command line arguments for the program are provided, although in most cases the information is in the control input file. The user can select a *Target System* for the program execution, or this decision can be left to the Mapping EA. A *Supervisor* program monitors the execution of the program, and determines if the execution was successful or not. The supervisor returns the *Execution Report* to the MEA. The program generates a number of *Output Data Files*, and the output results are validated by *Verify Filters* (VF_1-VF_k). The VFs are started by the supervisor in case of successful execution, and report if the output complies with their requirements or not. The verify filters are similar to assertions in various programming languages.

Composite execution blocks are created by recursive application of composition rules on basic blocks. The list of control composition rules contains: (1) block sequence, (2) loop, (3) forced alternative, (4) computed alternative, and (5) group with parallel execution of component blocks. In a *block sequence*, individual blocks are executed only after the previous block in the sequence has successfully terminated. It is similar to sequential execution of the statements in a programming language. The loops are composed of an execution block and a decision program, as shown in Figure 5. The structures are similar to traditional programming language. The role of a logical



a) Initial Test Loop: While-Do



b) Final Test Loop: Repeat-Until

Figure 5: HLD Loop Structures

expression is taken by a *Decision Program*, which controls the execution flow. A *forced alternative* arises when there are a number of equivalent programs that have the same processing effect, and the user manually selects one before execution. The *computed alternative* involves a Decision Program which selects one of the available paths in the description. A *parallel group* contains programs that must be scheduled at the same time due to communication dependencies.

By combining multiple blocks we can use combination techniques to group any number of subblocks into a higher level block. Usually each program has a number of additional elements associated with it, such as names of input and output files.

The scheduling expert advisor

The Scheduling Expert Advisor has the following functions:

- Parse a new HLD and convert it into rules and facts (pre-processing).
- Determine the programs available for execution at any time.
- Schedule the program that can be run in the current step. A scheduling request is passed to the Mapping Expert Advisor for each individual program or group of programs.
- Run in asynchronous or synchronous mode. In the first case each program is scheduled when all the conditions are fulfilled. In the second case all programs available for execution in one step are co-scheduled; whenever a program terminates, a new scheduling cycle is started.
- Report programs that have not been executed and might never be, due to a possible HLD programming error.
- Clear the working and production memory of the current HLD, and prepare for a new script.

Generation of the new rules and facts for a HLD and the scheduling of programs are the most important functions of the SEA.

The program scheduling results from the interaction of the new generated rules. The state transition diagram of a generic program is presented in Figure 6. Each arrow in the transition diagram represents a single rule or a set of rules in the Scheduler Control Information. The rules are named after the destination state, for example the rules leading to the Valid State are called Valid Rules. Most states are represented by SEA facts, named as in the transition diagram. Initially, the programs are in the *Start State*, when partial dependencies are satisfied. There is no specific fact associated with a program in the Start State. When all data dependencies are satisfied, the *Ready Rule* executes, and the program enters the *Ready State,* which is marked by a corresponding fact. The Ready Rule triggers asynchronously, whenever all the data dependencies of a program are satisfied. Each target program has its unique Ready Rule in the SCI rule base.

Valid Rules control the HLD execution flow. In *Run Mode*, a program can be scheduled at any time, so the transition from the Ready state to the Valid state is immediate and asynchronous. In *Step Mode*, the programs can be

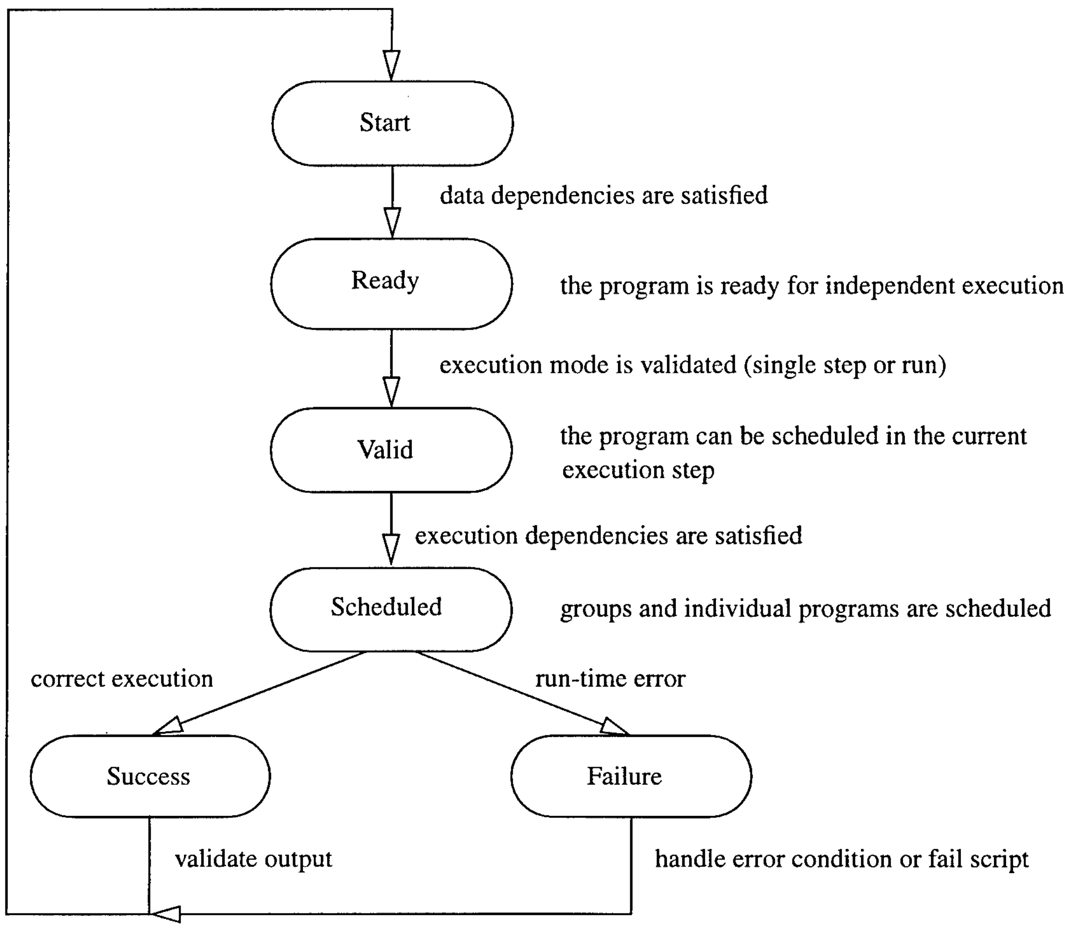

Start
data dependencies are satisfied
Ready
the program is ready for independent execution
execution mode is validated (single step or run)
Valid
the program can be scheduled in the current execution step
execution dependencies are satisfied
Scheduled
groups and individual programs are scheduled
correct execution
run-time error
Success
Failure
validate output
handle error condition or fail script

Figure 6: State Transition Diagram of a Program in the Scheduling Expert Advisor

scheduled only at predetermined events (*steps*). All the programs holding in the Ready State during the previous cycle are validated synchronously. Programs entering Ready State after the validation transition have to wait there until the next validation cycle. There is a Valid Rule for the Run Mode, and a Valid Rule for the Step Mode. These rules are common for all programs. If only the Run Mode is desired, the Valid Rules and the Valid state are eliminated from the transition diagram, and the Scheduling Rules respond to the Ready facts and not to the Valid facts. The execution mode is selected by the initial execution request and can be changed at any time. The same SCI rule base is used in all execution modes.

The *Scheduling Rules* differ for groups of programs that must be co-scheduled and for individual programs which are executed alone. An independent program is scheduled as soon as it enters the Valid state. Groups of programs are scheduled when all their execution dependencies are satisfied, in most cases when all the programs in the group are in the Valid state. Each Scheduling Rule creates a *Scheduling Block* of information which is passed to the Mapping Expert Advisor and an internal hook for the execution result. The Scheduling Rules are triggered asynchronously. Once a program has executed, the results are entered as specific *Success* or *Failure* facts in the knowledge base. In case of success, the output files are marked as valid, and can satisfy data dependencies of other programs. In case of failure, the SEA tries to recover from the error or lets the user handle the error.

Example

The preprocessing step of the Scheduling Expert Advisor converts the text or graphic HLD description into a set of facts and rules used to control the SEA. In this section we present a possible structure of the SCI facts and rules, the structure actually used by the Bond SEA.

The facts represent the status of various input files or control conditions. The preprocessor determines the set of *Absolute Input Files,* AIF, not generated by HLD programs and used as input for one of the HLD blocks. Bond checks if AIFs exist in the system. For the available AIF, the preprocessor adds a "`file file-name valid`" fact to the SCI. A fact "`file file-name invalid`" marks the missing AIF files.

The following rule types are generated by the preprocessor, as seen in Figure 6: (1) ready rule, (2) program or group scheduling rule, (3) successful execution rule, and (4) failed execution rule. In the following examples we ignore the Valid state which is controlled by shared rules.

The ready rules detect when a program has all the execution conditions satisfied. For example, `program_N` depends only on its input files:

```
(defrule ready_program_N
   (file input_data_file_1 valid)
   . . .
   (file input_data_file_k valid)
=>
   (assert (ready program_N
              input_data_file_1 ...
              input_data_file_k)
   )
)
```

The step control changes the program state from Ready to Valid. If the program has no other execution constrains, the generated scheduling rule is:

```
(defrule schedule_program_N
   factx <- (valid program_N args)
=>
   (retract factx)
   (schedulef program_N args)
)
```

A group of programs (`program_A` to `program_J`) which execute concurrently has a single scheduling rule:

```
(defrule schedule_group_M
   fact_a <- (Valid program_A args_a)
   . . .
   fact_j <- (valid program_J args_j)
=>
   (retract fact_a)
   . . .
   (retract fact_j)
   (schedulef program_A args_a)
   . . .
   (schedulef program_J args_j)
)
```

The `schedulef` function passes the execution request to the Mapping Expert Advisor, and prepares a hook to insert the execution results in the working memory. The generated facts are:

```
(result program_N OK)
(result program_N error)
```

The rule for a successful execution validates the output files of the current program:

```
(defrule success_program_N
   factq <- (result program_N OK)
=>
   (retract factq)
   (validate output_data_file_1)
   . . .
   (validate output_data_file_m)
)
```

The validation procedure removes a possible invalid fact for the file name and generates a valid fact for the file (`file file_name valid`). The scheduling process continues with the new facts (valid files), which can trigger execution of the next programs.

The execution failure rule triggers an error recovery procedure or fails the entire process:

```
(defrule error_program_N
   factq <- (result program_N error)
=>
   (... report and process error)
   (... revalidate program_N | ignore
                              | STOP)
)
```

Conclusions

There are major differences between an execution controlled by a program or a script and one controlled by an expert advisor. (a) A program or a script has limited adaptability properties, it needs to invoke error recovery routines to handle error conditions. An expert advisor can provide recovery rules invoked automatically when an error condition occurs. (b) In an expert system environment individual operations can be enhanced without changing the entire system. Rules in an expert system are loosely coupled. Changes to one rule generator will not propagate to other generators or control rules. (c) In an expert advisor, the dependencies can be specified in any order. A number of conditions must be valid before a program can be started. There is no order in which these conditions have to be entered or fulfilled. Programs are scheduled for execution only when all conditions are satisfied. Several unrelated programs that can be executed concurrently. If dependencies have to be satisfied in a specific order, we generate a chain of ready rules. The first rule checks the first dependency and then activates the second ready rule. The second rule checks the next dependency and so on, until all conditions have been satisfied. (d) Scheduler Control Information can be saved to a file and loaded in a separate expert system. This expert system can work independently to create a specialized Scheduling Expert Advisor to control execution of a particular workflow. (e) A simple deadlock detection mechanism is available. If programs are placed into a circular dependency list, none of them can be executed. SEA recognizes the condition and informs the user of the fact. Dead sequences of program which cannot be executed are also reported by the SEA.

Acknowledgments

This work is supported in part by a grant from the Scalable I/O Initiative, by Intel Corporation, by National Science Foundation through grants BIR-9301210 and MCR-9527131, and by a grant from Purdue Research Foundation. We thank Chris Bonham who wrote the prototype of the user interface.

References

[1] *CLIPS Reference Manual. Volume I: Basic Programming Guide.* Clips version 6.0, Software Technology Branch, Lyndon B. Johnson Space Center, June 2nd 1993.

[2] *CLIPS Reference Manual. Volume II: Advanced Programming Guide.* Clips version 6.0, Software Technology Branch, Lyndon B. Johnson Space Center, June 2nd 1993.

[3] *CLIPS Reference Manual. Volume III: Interfaces Guide.* Clips version 6.0, Software Technology Branch, Lyndon B. Johnson Space Center, June 2nd 1993.

[4] DARPA Knowledge Sharing Initiative. Specification of the KQML Agent Communication Language. DARPA Knowledge Sharing Initiative, External Interfaces Working Group Draft. June 15, 1993. WWW URL: `http://www.cs.umbc.edu/kqml/papers/kqmlspec.ps`

[5] Oren Etzioni and Daniel Weld, A Softbot-Based Interface to the Internet. *Communications of the ACM*, 37(7):72-76, July 1994.

[6] Stan Franklin and Art Graesser. Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents. WWW URL: `http://www.msci.memphis.edu/~franklin/AgentProg.html`

[7] Don Gilbert, Manny Aparicio, Betty Atkinson, Steve Brady, Joe Ciccarino, Benjamin Grosof, Pat O'Connor, Damian Osisek, Steve Pritko, Rich Spagana, Les Wilson. IBM Intelligent Agents White Paper. WWW URL: `http://activist.gpl.ibm.com:81/White-Paper/ptc2.htm`

[8] Joseph Giarratano. *CLIPS User's Guide.* Clips version 6.0. Software Technology Branch, Lyndon B. Johnson Space Center. May 28th, 1993.

[9] Joseph Giarratano and Gary Riley. *Expert Systems, Principles and Programming.* PWS publishing Company, 1994. ISBN 0-534-93744-6.

[10] Don Libes. *Exploring Expect: A Tcl-based Toolkit for Automating Interactive Programs.* O'Reilly and Associates, Inc., 1995. ISBN 1-56592-090-2.

[11] James Mayfield, Yannis Labrou, and Tim Finin. Evaluation of KQML as an Agent Communication Language. In [WMT96], pages 347-360.

[12] John K. Ousterhout. *Tcl and the Tk toolkit.* Addison-Wesley Publishing Company, 1994. ISBN 0-201-63337-X.

[13] M. D. Rice and S. B. Seidman. A Formal Model for Module Interconnection Languages. *IEEE Transactions on Software Engeneering* 20(1):88-101, January 1994.

[14] Mihai G. Sirbu and Dan C. Marinescu. Bond - A Parallel Virtual Environment. In *Proceedings of HPCN Europe '96*, pages 722-728. Lecture Notes in Computer Science, Volume 1067, Springer Verlag, 1996. ISBN 3-540-61142-8.

[15] Michael Wooldridge, Jorg P. Muller, and Milind Tambe. *Intelligent Agents II - Agent Theories, Architectures, and Languages.* Lecture Notes in Artificial Intelligence, Volume 1037, Springer Verlag, 1996. ISBN 3-540-60805-2.

Mihai G. Sirbu is a Ph.D. candidate in Computer Sciences at Purdue University. He received an Electronics Engineering degree in 1987 form Timisoara Polytechnic Institute, Romania. After working for 4 years in industry and academia, he earned in 1993 a M.S. in Computer Science from University of Missouri at Rolla. His research interests are user environments for parallel and distributed computing, intelligent agents, and computer security. He is a member of ACM and UPE.

Dan C. Marinescu is Professor in the Computer Sciences Department. He joined Purdue University in 1984. He had visiting appointments at IBM Research, Yorktown Heights, and the Supercomputer Systems Division of Intel. He is conducting research in: distributed systems, parallel processing and scientific computing.

He has co-authored more than 90 papers in research journals, conference proceedings or chapters of books in these areas. He was the chief architect of a distributed, real-time data acquisition and analysis system used for nuclear and high energy physics experiments. He is currently a principal investigator of a NSF funded Grand Challenge project to compute macromolecular structures using parallel and distributed systems.

Exploiting Multiple Heterogeneous Networks to Reduce Communication Costs in Parallel Programs

JunSeong Kim
jskim@ee.umn.edu

David J. Lilja
lilja@ee.umn.edu

Department of Electrical Engineering
University of Minnesota
200 Union St. SE
Minneapolis, MN 55455

Abstract

The different types of messages used by a parallel application program executing in a distributed system can each have unique characteristics so that no single communication network can produce the lowest latency for all messages. For instance, short control messages may be sent with the lowest overhead on one type of network, such as Ethernet, while bulk data transfers may be better suited to a different type of network, such as Fibre Channel or HiPPI. In this paper, we investigate how to exploit multiple heterogeneous communication networks that interconnect the same set of processing nodes by dynamically selecting the best (lowest latency) network for each message based on the message size. We also show how to aggregate these multiple parallel networks into a single virtual net*work to further reduce the latency and increase the available bandwidth. We test this multiplexing and aggregation on a cluster of SGI multiprocessors interconnected with both Fibre Channel and Ethernet. We find that multiplexing between Ethernet and Fibre Channel can substantially reduce communication overhead in a synthetic benchmark compared to using either network alone. Aggregating these two networks into a single* virtual network *can further reduce communication delays for applications with many large messages. The best choice of either multiplexing or aggregation depends on the mix of message sizes in the application program and the relative overheads of the two networks.*

Keywords: heterogeneous networks; multiplexing; aggregation; virtual networks; communication overhead.

1 Introduction

The importance of efficient communication in distributed parallel systems cannot be overemphasized since communication overhead restricts the sphere of applications that can be efficiently parallelized on these systems. Additionally, communication delays are typically large compared to computation time, so that communication often becomes the performance bottleneck. While an obvious technique for reducing communication overhead is to use a higher bandwidth network, latency-limited applications, as opposed to bandwidth-limited applications, may not benefit from the increased network speed. In fact, several different types of messages are typically used in a single application program, some of which are latency-limited, while others are bandwidth-limited.

These different types of messages each may be better suited to a different type of communication network so that no single network can provide the best performance for all types of communication within a single program. For instance, short control messages sent between processing nodes, such as synchronization or load information, require low-latency connections. Bulk data transfers, such as the transfer of large matrices, on the other hand, require high-bandwidth, but can often tolerate higher latency. As a result, since each type of network makes different trade-offs in latency and bandwidth [10, 12, 13], each of these different types of data transfers may be best suited for transmission on a different type of communication network. Fortunately, many networked parallel computing systems are being assembled with several different types of communication links between the same processing nodes. A common configuration, for instance, is a network of workstations interconnected with both

0-8186-7879-8/97 $10.00 © 1997 IEEE

Ethernet plus some higher-bandwidth network, such as Fibre Channel, ATM, or HiPPI.

This paper presents some of our preliminary investigations into how to effectively utilize these multiple heterogeneous networks to reduce the communication overhead in parallel application programs. Specifically, we examine three different approaches implemented on a cluster of Silicon Graphics Challenge L multiprocessors interconnected with both Ethernet and Fibre Channel communication networks. The first approach is a simple multiplexing strategy that dynamically selects one of the two networks on which to send each message based on the size of each individual message to minimize the total communication latency. This approach introduces a small amount of overhead to select between the networks compared to using only a single network. The second approach *aggregates* the two separate networks into a single *virtual network* whose bandwidth is approximately the sum of the bandwidth of the two individual networks. This approach introduces some additional latency over simple multiplexing due to the segmentation and reassembly required to send a single message over parallel communication paths. Finally, we evaluate the next level of multiplexing that dynamically selects between the Ethernet, the Fibre Channel, or the single *virtual network.*

In the remainder of the paper, Section 2 provides additional background on communication overheads, and measures the performance characteristics of the networks in our testbed. Section 3 then describes how the multiplexing and aggregation strategies are implemented, and compares their raw performance as a function of message size. Section 4 characterizes the communication patterns of several of the NAS benchmarks which are then used to create a synthetic benchmark for evaluating the performance of our new communication strategies at the application level. Finally, Section 5 summarizes our results and conclusions.

2 Background and Network Characteristics

2.1 Communication Overhead

Communication overhead can be decomposed into both hardware and software overhead. Hardware overhead includes both the host interface delay and the signal propagation delay. Software overhead, on the other hand, is incurred by interactions with the host operating system, the actual device driver routines, and the high-level network communication protocols.

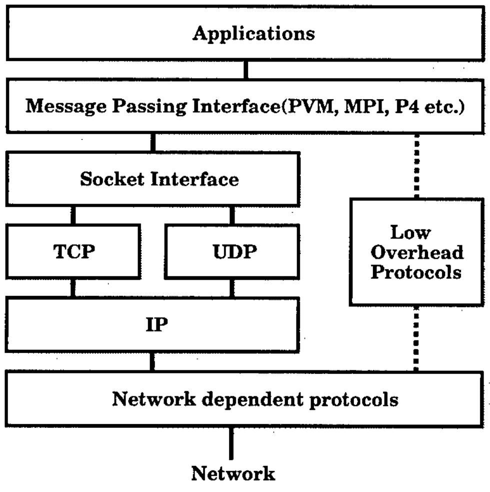

Applications
Message Passing Interface(PVM, MPI, P4 etc.)
Socket Interface
TCP
UDP
IP
Low Overhead Protocols
Network dependent protocols
Network

Figure 1: Communication network protocol hierarchy.

Software communication overhead can be reduced by eliminating the overhead of high-level protocols [4, 13] as shown in the conceptual protocol hierarchy of Figure 1. The overhead of the high-level protocols, and the operating system overhead, can be changed by using different Application Programming Interfaces (APIs). The APIs in the left-hand side of the figure are typical interfaces that pass through the TCP/IP or UDP/IP protocol stack from the application level to the physical level. The right-hand side of the figure shows that special-purpose protocols that reduce the software overhead can be used instead. For example, a 66% throughput improvement and 30% reduction in round-trip latency, compared to using a typical API, has been demonstrated by using PVM as the message passing interface with HiPPI as the communication link [4]. Hewlett Packard's Link Level Access (LLA) was the alternative API. The cost of these low overhead protocols, however, is that they may depend on specific hardware implementations of the network interface.

In addition to reducing the software overhead, there are also system-level techniques for reducing communication latency [12]. For example, there are two common techniques for transferring data between the host processor and the network controller. One technique uses the host memory as a packet buffer with the host and the controller sharing descriptors in the memory. The other technique uses a simple FIFO for buffering packets. The FIFO-based controller is better for small packets than the shared-descriptor technique

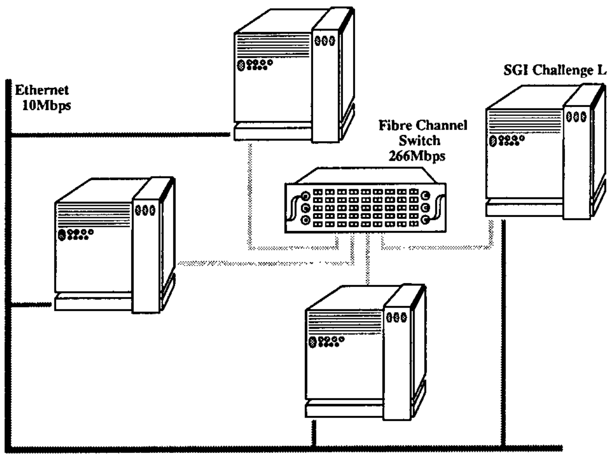


Figure 2: Multiple heterogeneous network configuration used in the experiments.

since it reduces the interrupt handling overhead. The descriptor-based controller is better for large packets in which the memory-to-memory copy time dominates the interrupt handling execution time. A hybrid controller is possible that supports both types of interfaces on the same controller.

2.2 Network Characteristics

There are several different communication networks, such as ATM, Fibre Channel, HiPPI and FDDI, that have been proposed to be used as a communication channel between independent processing nodes to thereby create a scalable parallel computing system [4, 6, 12]. Each of these networks has different latency and bandwidth characteristics. In addition, the different types of communications used by a parallel application have different latency and bandwidth requirements. As a result, we expect that it may be possible to reduce communication overhead in parallel applications by having multiple heterogeneous networks between processing nodes and then matching each message to the most appropriate network.

To demonstrate the differences in network characteristics, we compare an Ethernet and a Fibre Channel network in a system consisting of four Silicon Graphics Challenge L shared-memory multiprocessors, as shown in Figure 2. Each of the nodes in this system contain four or eight R10000 processors running at 196 MHz on a shared bus. The nodes can communicate with each other via either an Ethernet running at a peak transfer rate of 10 Mbps, or a Fibre Channel network, using an Ancor CXT250 16 port switch, running at 266 Mbps. All nodes run version 6.2 of the IRIX operating system.

We measured the latency and bandwidth characteristics of both these networks using the echo program shown in Figure 3. All measurements were made on a dedicated system with no other users to minimize the interference from external traffic. The sending and receiving buffer sizes for both Ethernet and Fibre Channel were set at 61,440 bytes. We repeated the test 3 times, running N=100 iterations of each send and receive, and then chose the best values for Figure 4. The variance in these measurements was minimal since the system was dedicated to these experiments. The average bandwidth is calculated as $\frac{2*m}{average\ latency}$ since there are two m-byte messages transferred, one in the forward direction, and one in the reverse.

Figure 4 shows the measured latency and bandwidth characteristics of these two networks as a function of the message size transferred. The characteristics are compared using both the TCP and UDP transport protocols. As expected, there is little performance difference between the two transport protocols for messages smaller than approximately 32,000 bytes [11]. Comparing Ethernet and Fibre Channel in Figure 4(a), we see that the slope of the latency curve for Ethernet is much steeper than the curve for Fibre Channel, although Ethernet has a lower latency than Fibre Channel for small messages. In fact, Ethernet outperforms Fibre Channel for messages smaller than approximately 900 bytes while Fibre Channel produces a lower total latency for larger messages. The bandwidth of the Ethernet begins to saturate with messages larger than approximately 1,500 bytes while the Fibre Channel bandwidth continues to increase with increases in the message size.

3 Multiplexing and Aggregating Multiple Heterogeneous Networks

The network comparisons in the previous section suggest that there may be some benefit to multiplexing between the two networks as a function of the message size. It is possible to select an appropriate network for a given message using some characteristic other than message size, such as the message type (e.g. synchronization or bulk data transfer), or the network traffic load, but, to demonstrate the basic idea, we focus on selecting an appropriate network based only on the message size. In particular, we extend our net-

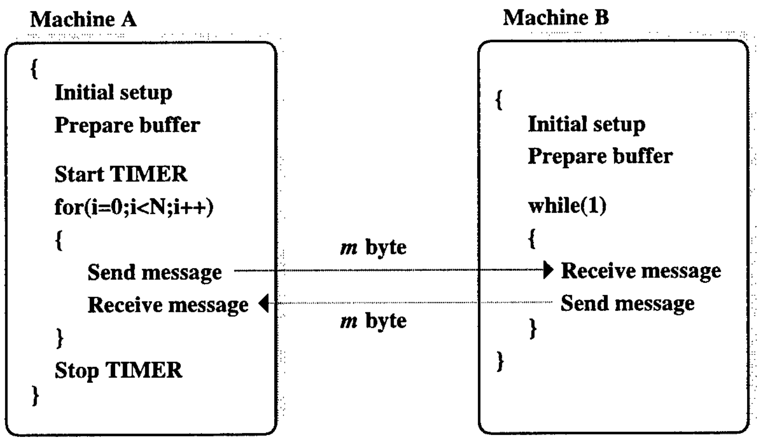
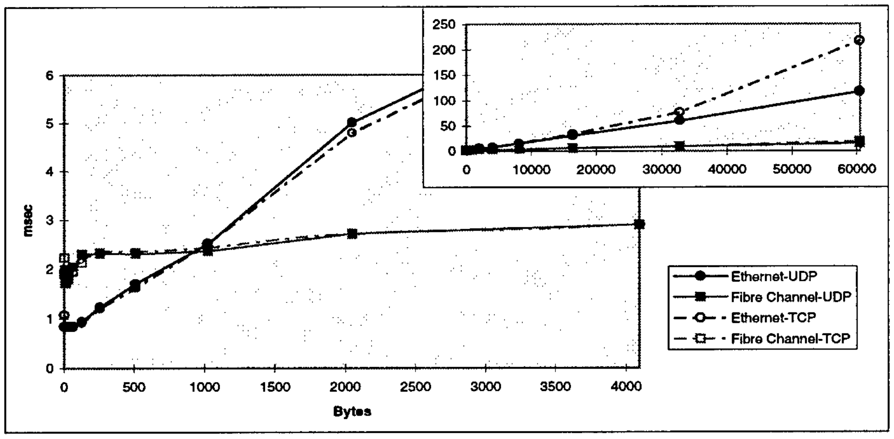
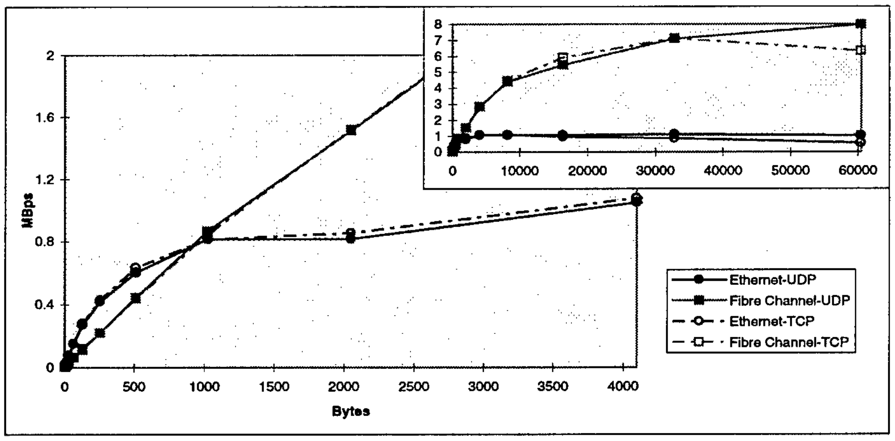


Figure 3: Echo program for network latency and bandwidth measurements.



(a) Latency



(b) Bandwidth

Figure 4: Communication performance of Ethernet and Fibre Channel using TCP and UDP transport protocols.

work characterization echo program described in the previous section to dynamically select for each individual message the Ethernet network if the message is smaller than 900 bytes. Otherwise, the message is sent via the Fibre Channel network. The additional overhead required for this multiplexing compared to using only one network by itself is approximately the time required to execute one conditional statement to compare the message size to the multiplexing threshold.

In addition to multiplexing, we also investigate aggregating the two networks into a single *virtual network*. With this approach, the original message is divided into two smaller submessages. One of these submessages is then transferred over the Ethernet while the other submessage is transferred simultaneously over the Fibre Channel. From the point-of-view of the application program, this aggregated network appears to be a single (*virtual*) network whose bandwidth is approximately the sum of the bandwidths of the individual component networks.

One of the most important considerations for this network aggregation is determining the size of the submessages that should be sent over each of the networks. The original message must be divided into two submessages with the goal of having the two submessages be completely received at the destination at the same time. More precisely, assume that a message consisting of m_e bytes can be sent and received at the destination in time t_e on the Ethernet. Similarly, a message consisting of m_{fc} bytes requires time t_{fc} to be sent and received on the Fibre Channel network. Then m_e and m_{fc} must be chosen so that $t_e = t_{fc}$ and $m_e + m_{fc} = m$, where m is the size of the original message. Given any message of size m, the latency characteristic curves shown in Figure 4 can be used to empirically determine appropriate values for m_e and m_{fc} to minimize the total communication latency. The specific values used in this study are shown in Table 1.

In addition to the two conditional statements required to determine the values of m_e and m_{fc}, this aggregation requires a special message segmentation and reassembly step to first divide the message into the two submessages at the sender, and then reassemble them into a single message at the receiver. As a result, the overhead of this aggregation is considerably higher than the simple multiplexing approach.

We implemented the multiplexing and aggregation using UDP as the application program interface to the transport level protocol. Since UDP does not guarantee delivery, sequencing, or duplicate packet protection [11], we added these services on top of the basic datagram service. We then measured the latency and bandwidth of these communication approaches using the same hardware configuration as shown in Figure 2.

Figure 5 shows the overall latency for multiplexing and aggregation as well as that for Ethernet and Fibre Channel. Figure 5(a) is a magnified version of Figure 5(b) for messages smaller than $10K$ bytes. These figures show that the latency of the multiplexing approach follows the smaller of both Ethernet and Fibre Channel as the message size increases. Also, this simple multiplexing shows better performance than that of the aggregation until the message size is greater than 9,500 bytes because of the software overhead necessary for the segmentation and reassembly of messages with aggregation. However, we can see as much as an 11% improvement with aggregation over simple multiplexing with $55K$ byte messages. The intersections of aggregation and multiplexing at $30K$ bytes and $45K$ bytes are due to the coarse choices of m_e and m_{fc} in Table 1. We expect that they can be eliminated with a finer granularity of message divisions.

Comparing the bandwidth of the different approaches in Figures 5(c) and 5(d), we see that the bandwidth of the multiplexing approach follows the best of Ethernet and Fibre Channel as the message size increases. The bandwidth of the aggregated *virtual network* is almost the sum of the bandwidth of the two networks individually for large message sizes. The slightly reduced bandwidth of this approach is due to the segmentation and reassembly overhead required for the aggregation. These measurements show that, by multiplexing between the two networks based simply on message size, we can obtain a network latency that follows the best of both. Furthermore, we can aggregate the two networks into a single *virtual network* to increase the total bandwidth available to an application when sending large messages.

4 Application-Level Communication Performance

The performance characterizations shown in the previous section demonstrate that multiplexing and aggregating heterogeneous networks can reduce the overall communication latency when sending messages of a specific size. However, it is important to deter-

Message size (m bytes)			Message size sent on	
			Ethernet (m_e)	Fibre Channel (m_{fc})
$0 <$	m	≤ 900	m	0
$900 <$	m	≤ 1800	m-1000	1000
$1800 <$	m	≤ 4000	900	m-900
$4000 <$	m	≤ 6000	1200	m-1200
$6000 <$	m	≤ 15000	1500	m-1500
$15000 <$	m	≤ 25000	2400	m-2400
$25000 <$	m	≤ 40000	3900	m-3900
$40000 <$	m	≤ 60000	5500	m-5500
$60000 <$	m		8500	m-8500

Table 1: Empirically-derived values used to determine the size of the submessages to be sent on the Ethernet (m_e) and the Fibre Channel (m_{fc}) when aggregating the networks into a single *virtual network*. Note that $m = m_e + m_{fc}$.

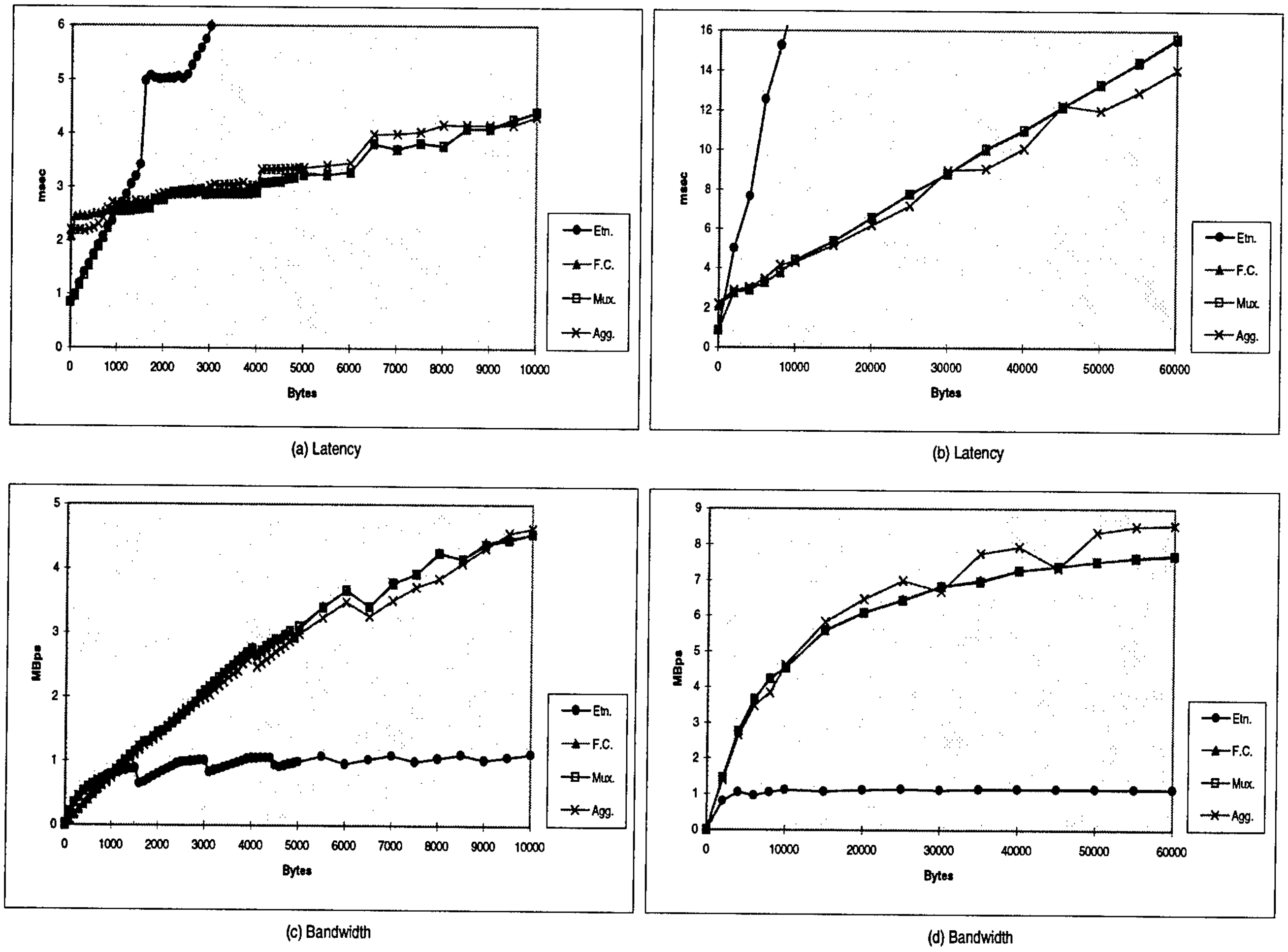


Figure 5: Latency and bandwidth measurements of multiplexing and aggregation. The left figure is a magnification of the right figure in the range from 0 - 10,000 bytes.

		Sample code	Class A	Class B
CG	Conjugate	1400	14000	75000
IS	Integer Sort	2^{16}x2^{11}	2^{23}x2^{19}	2^{25}x2^{21}
MG	Multigrid	32^3, 4 iters	256^3, 4 iters	256^3, 20 iters
BT	BT Simulated CFD application	12x12x12	64x64x64	102x102x102
LU	LU Simulated CFD application	12x12x12	64x64x64	102x102x102

Table 2: Problem sizes for the NAS parallel benchmarks.

mine how these approaches can reduce the total communication time at the application level. To investigate the application-level performance of these latency reduction strategies, we first characterize the program communication patterns of several of the NAS benchmarks [1]. We use these patterns to generate a synthetic benchmark program in which we can vary the mix of message sizes. We then use this synthetic benchmark to determine how the total communication time is affected by our multiplexing and aggregating strategies.

4.1 Characteristics of Program Communication Patterns

This subsection characterizes the communication patterns of several of the NAS benchmarks [1] running on the four-node SGI Challenge system shown in Figure 2. The results provide interesting insights into communication patterns of parallel applications. We chose five of the benchmarks that represent a range of communication characteristics of highly parallel applications. Three of them, CG, IS, and MG, are relatively compact kernel benchmarks that emphasize some particular type of computation. The remaining two benchmarks, BT and LU, are computational fluid dynamics applications that have more data movement than the kernel benchmarks. There are one sample and two standard problem sizes for the NAS Parallel Benchmarks, as shown in Table 2.

In the CG benchmark, a conjugate gradient method is used to compute an approximation to the smallest eigenvalue of a large, sparse, symmetric, positive-definite matrix. This kernel tests irregular long-distance communication, employing unstructured matrix vector multiplication. The IS benchmark is a large integer sort with no floating point arithmetic. It tests both integer computation speed and communication performance. The MG benchmark is a simplified multigrid kernel with a constant coefficient. It solves a 3-D Poisson partial differential equation. This kernel is a good test of both short- and long-distance data communication. The LU code is the lower-upper diagonal benchmark. It does not perform an LU factorization, but instead employs a symmetric successive over-relaxation numerical scheme to solve a regular, sparse, block 5x5 lower and upper triangular system. The block tridiagonal benchmark, BT, solves multiple independent systems of non-diagonally dominant, block tridiagonal equations with a 5x5 block size.

We extend PVM versions of the five NAS benchmarks by inserting monitor operations at points in the programs where message-related activities are executed. This instrumentation allowed us to observe the programs' communication patterns. We tested the Class A problem sizes for the CG, IS, and MG codes, and the Sample problem sizes for the LU and BT codes. However, we believe our results below are somewhat independent of the problem size. For the optional setup of PVM [3], we used the default mode, that is, normal routing using UDP connections, regular *pvm_send()*, *pvm_receive()*, and *pvm_initsend()* commands, and the default XDR (*eXternal Data Representation*) encoding, which is an Internet standard data encoding.

Figure 6 shows the distribution of processors that are message destinations. In the CG, IS, and MG codes, all of the processors perform the same types of tasks. In the BT and LU codes, however, the master processor plays the role of a central controller while the slave processors perform the actual work. From the viewpoint of the individual processors, the destination distribution does not appear to have any special or characteristic pattern. For example, in the CG program, processors 0 and 3 send messages to all of the other processors, while processor 2 sends only to processor 0. Since the distribution of message destinations has a spatial locality which favors nearest neighbors, the destination distribution strongly depends on the algorithm and the network topology[8]. However, we can see that the overall destination distribution is

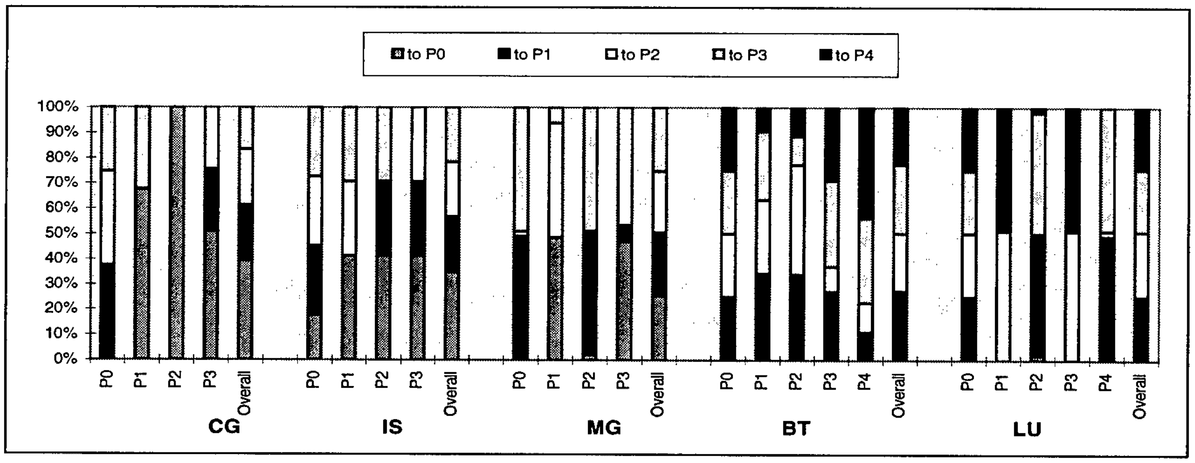


Figure 6: Distribution of processors that are the destinations of messages.

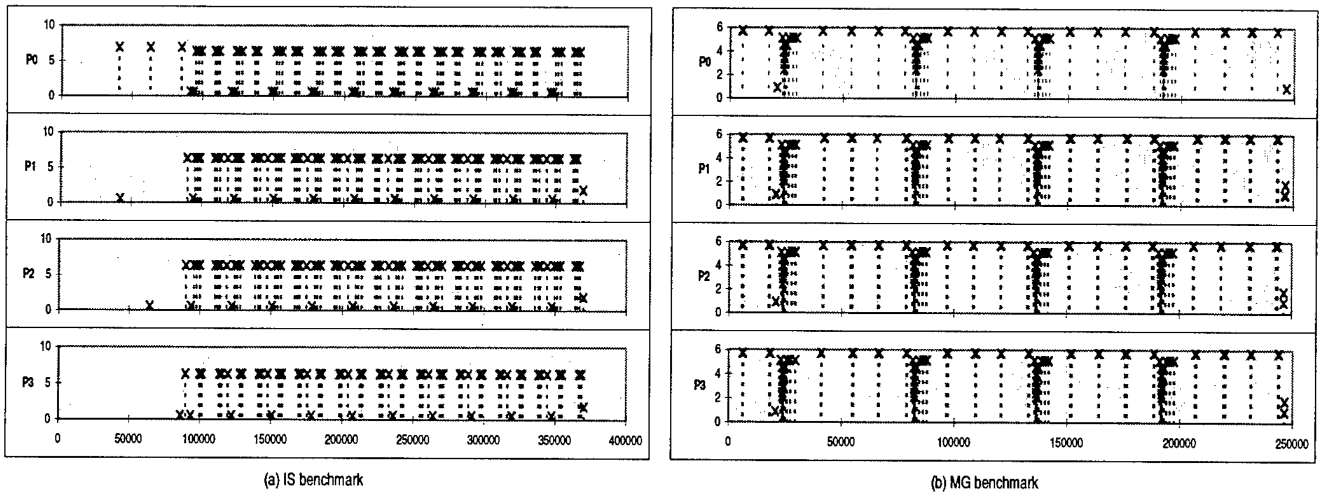


Figure 7: Relative times of communication events for the IS and MG benchmarks.

uniformly distributed among the processors.

Figure 7 shows the occurences in time of communication events for two of the test benchmarks, IS and MG. The X axis represents elapsed time from the beginning of the program execution and the Y axis shows the message size of the corresponding communication event. Note that the Y axis is the log of the actual message size. In both applications, all of the processors show similar communication patterns, except at the beginning of the execution. Specifically, each processor tends to alternate computation and communication at the same time so that they all are likely to communicate at the same time. As a result, communication congestion is likely to occur among the processors even when the system is dedicated to a single application. We also saw similar results from the other three benchmarks.

Figure 8 shows the distribution of message sizes for all five applications. It is clear from the figure that, except for the MG program, all of the messages within an application are one of two distinct sizes. In particular, some fraction of the messages within an application tend to be very large while the remainder tend to be very small. For instance, approximately 70% of the messages in the CG program are 8 bytes in length while the remainder are around $56K$ bytes. Similarly, approximately 40% of the messages in the BT program are 480 bytes, while the remaining 60% are around $2K$ bytes. Similar distributions were also found for the LU and IS programs. Previous researchers' measurements of CAD and numeric applications [5] and scientific applications [2] have shown similar bimodal

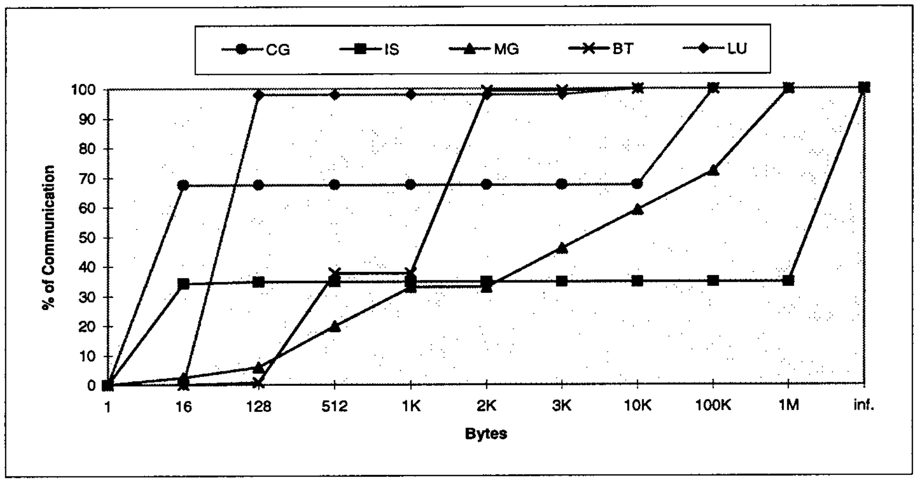


Figure 8: The cumulative distribution of message sizes.

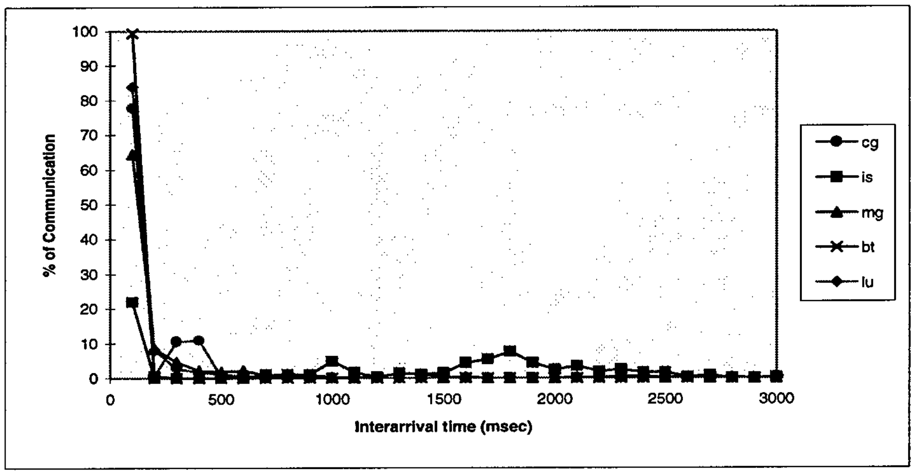


Figure 9: The percentage of messages with a given interarrival time.

distributions.

Figure 9 shows the distribution of the time intervals between two successive communication events in one processor. As is often assumed in analytic studies of network performance [7], these interarrival times tend to look exponentially distributed. However, we can see multiple peaks in the IS and CG benchmarks. These multiple peaks make it difficult to model the distributions with a simple exponential distribution function. Consequently, we need to use a combination of several distribution functions, or multi-stage probability density functions [5], to model the interarrival times more accurately.

4.2 Communication Performance with the Synthetic Benchmark

To estimate the improvement in communication performance that could be obtained at the application level by multiplexing and aggregating the two networks, we develop a synthetic benchmark to execute on the testbed system. This synthetic benchmark is based on the communication patterns observed in the NAS benchmarks and is parameterized so that we may simulate the communication patterns of a variety of different application programs. As shown in Figure 10, the benchmark configures the system into one master processor and $p-1$ slave processors. For each communication event, the master processor generates

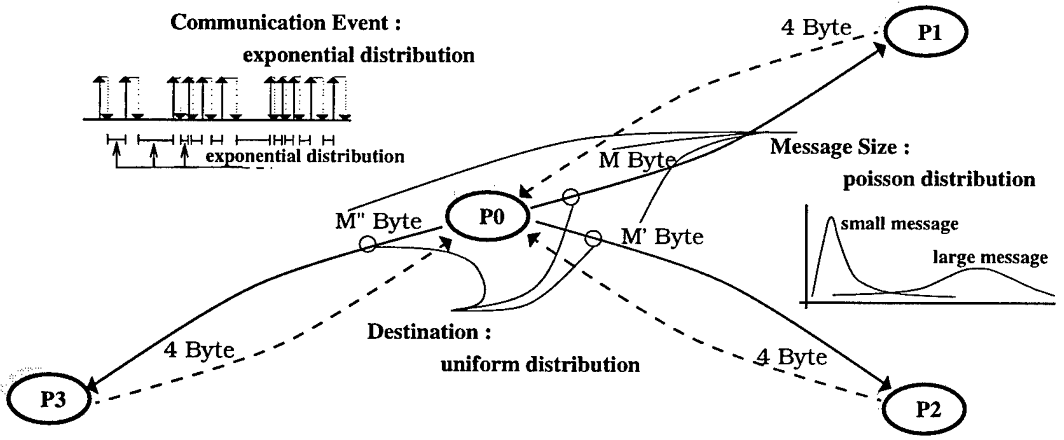


Figure 10: Synthetic benchmark modeling with p=4 processing nodes.

several random numbers. The first of these numbers determines the destination processor number, x. This value is uniformly distributed from 1 to $p-1$ so that each slave processor has an equal chance of becoming the destination of the current message. The second random number follows a Boolean distribution with probability b to select one of two Poisson distributions, one with a mean value of l_1 and the other with a mean value of l_2. The third random number then is used to generate a random value that follows the Poisson distribution selected by b. This value is used as the size of the message that is actually sent, m. The final random number, g, determines the computation time until the next communication event. It also is exponentially distributed.

After generating these random values, the m-byte message is sent to the destination processor using the chosen communication strategy. The receiving processor responds to the sending processor with a four-byte acknowledgement message after it has completely received the message. The master (i.e. sending) processor then idles for a random amount of time, g, to simulate the processors' computation. In these experiments, we set the value of g to zero since the communication behavior is the main focus of this study. Thus, when this synthetic benchmark is executed, the master processor will send an m-byte message to a randomly selected slave processor. The size of the message, m, will follow a Poisson distribution such that $(b * 100)\%$ of the time the mean will be l_1 and the remainder of the time the mean will be l_2. These steps are repeated $N = 10,000$ times for each run of the benchmark program. A pseudocode description of the benchmark is shown in Figure 11.

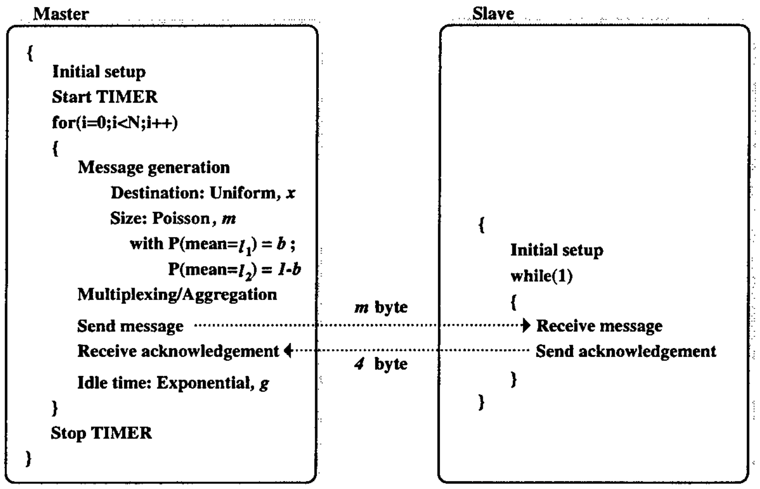
By appropriately choosing the above parameters, we can approximate the communication patterns of several different types of benchmarks. Table 3 shows the parameters actually used in our simulations. The Type A parameters, for instance, simulate an application program in which 50 percent (i.e. $b = 0.5$) of its messages are Poisson distributed with a mean of $l_1 = 8$ bytes, and 50 percent of its messages are Poisson distributed with a mean of $l_2 = 2K$ bytes.

Figure 12 shows the results of executing this synthetic benchmark with the different parameter values shown in Table 3 on the same SGI test system used for the previous experiments. Each data point is the average of 3 different runs of the benchmark with different random seed values. Since we used a dedicated system with no other users when executing the benchmark, and since the loop count, N, was large, the variance in execution times for a specific parameter set was quite small. The different communication network options are: 1) Ethernet (Etn) alone, 2) Fibre Channel (FC) alone, 3) simple multiplexing (Mux) between Ethernet and Fibre Channel, 4) aggregating (Agg) Ethernet and Fibre Channel into a single *virtual network*, or 5) the combination of both multiplexing and aggregating (Mux-Agg). The raw execution times are normalized to the execution time of the benchmark when using only the Ethernet connection.

Figure 12 shows that the simple multiplexing scheme is better than both Ethernet alone and Fibre

Application Type	small message	large message	
	mean $l_1 = 8$ byte	mean $l_2 = 2K$ byte	mean $l_2 = 20K$ byte
A	50%	50%	-
B	10%	90%	-
C	90%	10%	-
D	50%	-	50%
E	10%	-	90%
F	90%	-	10%

Table 3: Parameter values used in the synthetic benchmark.



```
Master
{
    Initial setup
    Start TIMER
    for(i=0;i<N;i++)
    {
        Message generation
            Destination: Uniform, x
            Size: Poisson, m
                with P(mean=l_1) = b ;
                     P(mean=l_2) = 1-b
        Multiplexing/Aggregation
        Send message
        Receive acknowledgement
        Idle time: Exponential, g
    }
    Stop TIMER
}

Slave
{
    Initial setup
    while(1)
    {
        Receive message
        Send acknowledgement
    }
}
```

Figure 11: Synthetic benchmark.

Channel alone for all types of communication patterns, except type E. This set of parameters simulates a program with mostly $20K$ byte messages. When an application consists primarily of large messages, however, there is little benefit to multiplexing. In fact, in this case, the overhead of multiplexing can cause its performance to be worse than no multiplexing. Multiplexing also shows little benefit for the type B communication pattern since it too has mostly large messages.

The aggregation scheme always outperforms both Ethernet and Fibre Channel used alone. However, it is worse than simple multiplexing for the type C and F communication patterns. Both of these benchmarks consist mainly of 8 byte messages with a relatively small fraction (10%) of large messages. Since the overhead of aggregation is too high to be of any benefit for small messages, simple multiplexing is the best solution for these two parameter sets.

For the combination of multiplexing and aggregation (Mux-Agg), the Ethernet is chosen for messages less than 900 bytes, the Fibre Channel is chosen for messages greater than 900 bytes, but less than 9,500 bytes, and the aggregation of both networks is used for messages larger than 9,500 bytes. Since the type A and B parameter sets have essentially no messages larger than $2K$ bytes, aggregation is almost never used with this combined approach. However, these applications do get penalized by the additional overhead of the combined approach. As a result, the combination of multiplexing and aggregation is worse than aggregation alone for these parameter sets. Almost all of the messages in type C are small with only 10% having a mean of $2K$ bytes, so that the combined approach almost always selects the Ethernet with no aggregating. Since aggregating alone has higher overhead than the combined approach for small messages, the combined

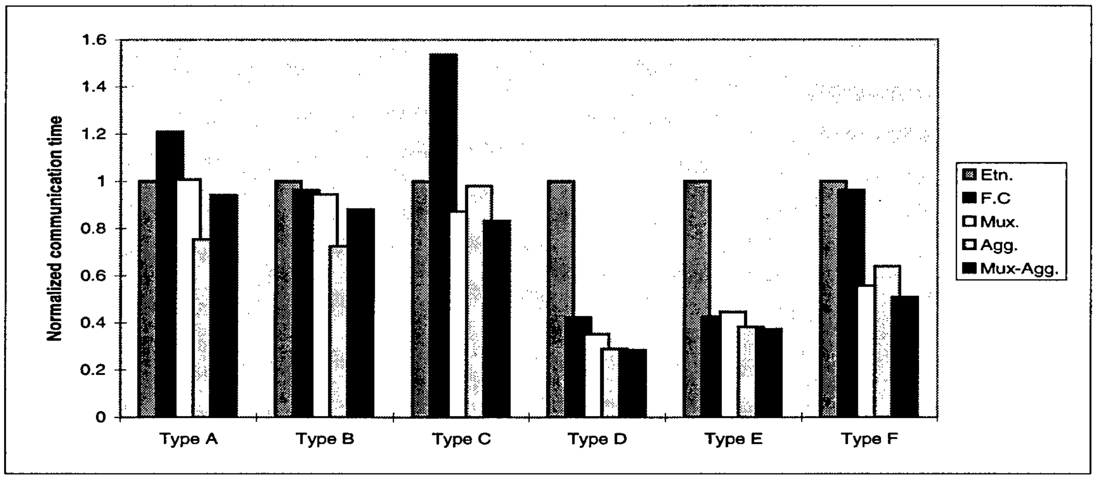


Figure 12: Communication time comparisons for the synthetic benchmark.

approach actually outperforms aggregating alone for this parameter set. For the remaining three parameter sets, multiplexing between Ethernet and Fibre Channel for the small messages ensures that they are always sent on the best of the two available networks. At the same time, the large messages (mean message size = $20K$ bytes) take advantage of the benefit of aggregating the two networks. The net result is that the combined approach can provide the best performance for applications with a high fraction of large messages.

5 Conclusion

The importance of reducing communication overhead in network computing cannot be overemphasized since the communication delays in standardized interconnection networks can often become the performance bottleneck in parallel application programs. Furthermore, the different types of messages used in parallel application programs, such as short control information or bulk data transfers, have affinities for different types of networks. For instance, short messages, such as synchronization, typically require very low latency, while larger messages need high bandwidth, although these larger messages often can tolerate higher latencies.

In this study, we take advantage of these different network and message characteristics to reduce the overall communication delay experienced by parallel application programs by exploiting multiple heterogeneous networks between the same processing nodes. Using a cluster of Silicon Graphics Challenge L multiprocessors interconnected with both Ethernet and Fibre Channel networks, we demonstrated how simple multiplexing can select the best network for each message based on the message size. We also presented an aggregation scheme that combines both networks into a single *virtual network* by dividing a single message into two submessages that are sent on each network simultaneously. Measurements of communication latency versus message size showed that the overhead of the simple multiplexing approach is low enough to allow it to track the performance of the network with the lowest latency. The need for message segmentation and reassembly causes the overhead of the aggregation approach to be significantly higher than the simple multiplexing approach. For messages larger than about 9,500 bytes, however, this aggregation reduces the overall latency to be less than that of using either network alone, and less than multiplexing between them.

Finally, we used a synthetic benchmark to study the effectiveness of these approaches in reducing the communication latency from the perspective of the application program. We modeled the synthetic communication patterns after the communication patterns we measured in the NAS parallel benchmarks. Our experimental measurements show that a combination of aggregation and multiplexing produces the best performance for applications that have a mix of very large messages and small messages. For applications dominated by small messages, however, the simple multiplexing approach is best due to the relatively high overhead of aggregating. Our future work will more precisely evaluate the trade-offs in these various approaches, and will develop approaches for further reducing their overhead. We also plan to investigate characteristics other than the message size for multiplexing between networks, and to develop approaches for adjusting to dynamically varying network loads.

Acknowledgments

This work was supported in part by National Science Foundation Grant No. CDA-9414015, and by a University of Minnesota McKnight Land-Grant Professorship.

References

[1] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Darum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga, "The NAS Parallel Benchmarks," *NAS Report RNR-94-007*, March 1994.

[2] R. Cypher, A. Ho, S. Konstantinidou, and P. Messina, "Architectural Requirements of Parallel Scientific Applications with Explicit Communication," *Intl Symp on Computer Architecture*, 1993, pp. 2-13.

[3] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek and V. Sunderam, *PVM: Parallel Virtual Machine - A Users' Guide and Tutorial for Networked Parallel Computing*, The MIT Press, 1994.

[4] J. Hsieh, D. H. C. Du, N. J. Troullier, and M. Lin, "Enhanced PVM Communications over a HIPPI Networks," *Proceedings of the Second International Workshop on High-Speed Network Computing*, April 1996.

[5] J.-M. Hsu, and P. Banerjee, "Performance Measurement and Trace Driven Simulation of Parallel CAD and Numeric Applications on a Hypercube Multicomputer," *Intl Symp on Computer Architecture*, 1990, pp. 260-269.

[6] C. Huang, E. P. Kasten, and P. K. McKinley, "Design and Implementation of Multicast Operations for ATM-Based High Performance Computing," *Proceedings of Supercomputing'94*, August 1994, pp. 164-173.

[7] P. Kermani, and L. Kleinrock, "Virtual Cut-Through : A New Computer Communication Switching Technique," *Computer Networks*, September 1979, pp. 267-286.

[8] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing - Design and Analysis of Algorithms*, The Benjamin/Cummings Publishing Company, 1994.

[9] D. J. Lilja, "Partitioning Tasks Between a Pair of Interconnected Heterogeneous Processors: A Case Study," *Concurrency: Practice and Experience*, May 1995, pp. 209-223.

[10] S. Nog, and D. Kotz, "A Performance Comparison of TCP/IP and MPI on FDDI, Fast Ethernet, and Ethernet," *Dep't of Computer Science, Dartmouth College, PCS-TR95-273*, 1995.

[11] W. R. Stevens, *UNIX Network Programming*, Prentice Hall, 1994.

[12] C. A. Thekkath, and H. M. Levy, "Limits to Low-Latency Communication on High-Speed Networks," *ACM Transactions on Computer Systems*, May 1993, pp. 179-203.

[13] A. Wolman, G. Voelker, and C. A. Thekkath, "Latency Analysis of TCP on an ATM Network," *Proceedings of USENIX*, 1994, pp. 167-179.

JunSeong Kim received an M.S. and a B.S., both in Electronics Engineering, from the Chung-Ang University, Seoul, Korea. He is a Ph.D. student in Electrical Engineering at the University of Minnesota in Minneapolis. His research interests are in computer architecture, computer networks, parallel processing, and high-performance computing.

David J. Lilja received a Ph.D. and an M.S., both in Electrical Engineering, from the University of Illinois at Urbana-Champaign, and a B.S. in Computer Engineering from Iowa State University in Ames. He is currently an Associate Professor of Electrical Engineering and the Director of Graduate Studies in Computer Engineering at the University of Minnesota in Minneapolis. Previously, he worked as a research assistant at the Center for Supercomputing Research and Development at the University of Illinois, and as a development engineer at Tandem Computers Incorporated in Cupertino, California. His main research interests are in computer architecture, parallel processing, and high-performance computing, with a special emphasis on the interaction of compilation technology and computer architecture. He is a Senior member of the IEEE Computer Society, a member of the ACM, and is a registered Professional Engineer.

On-Line Use of Off-Line Derived Mappings for Iterative Automatic Target Recognition Tasks and a Particular Class of Hardware Platforms

John R. Budenske
Ranga S. Ramanujan
Architecture Technology Corporation
Minneapolis, MN 55424, USA
{budenske, ranga}@atcorp.com

Howard Jay Siegel
Parallel Processing Laboratory
School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN 47907-1285, USA
hj@purdue.edu

Abstract

Heterogeneous computing covers a great variety of situations. This study focuses on a particular application domain (iterative automatic target recognition tasks) and an associated specific class of dedicated heterogeneous hardware platforms. The contribution of this paper is that, for the computational environment considered, it presents a methodology for real-time on-line input-data dependent remappings of the application subtasks to the processors in the heterogeneous hardware platform using previously stored off-line statically determined mappings. That is, the operating system will be able to decide during the execution of the application whether or not to perform a remapping based on information generated by the application from its input data. If the decision is to remap, the operating system will be able to select a previously derived and stored mapping that is appropriate for the given state of the application (e.g., the number of objects it is currently tracking).

This research is supported by the Department of Defense, Small Business Innovative (SBIR) Program, funded by the Army Research Laboratory, under contract number DAAL01-96-C-0031.
H. J. Siegel is a consultant for Architecture Technology Corporation.

Keywords: algorithm mapping, automatic target recognition, distributed computing, genetic algorithms, heterogeneous computing, parallel processing, real-time processing, special-purpose systems.

1: Introduction

Heterogeneous computing (HC) covers a great variety of situations (e.g., see [14], [19], [20]). This study focuses on a particular application domain (iterative automatic target recognition (ATR) tasks) and an associated specific class of dedicated heterogeneous hardware platforms. The contribution of this paper is that, for the computational environment considered, it presents a methodology for real-time on-line input-data dependent remapping of the application subtasks to the processors in the heterogeneous hardware platform using a previously stored off-line statically determined mapping (i.e., a matching of application subtasks to processors and a scheduling for the execution order of these subtasks). That is, the operating system will be able to decide during the execution of the application whether or not to perform a remapping based on information generated by the application from its input data. If the decision is to remap, the operating system will be able to select a previously derived and stored mapping that is appropriate for the given state of the application (e.g., the number of objects it is currently tracking).

The high-level operating system approach presented here for enabling the on-line use of off-line mappings is called the IOS (Intelligent Operating System). The IOS conceptual design, on-line components, and off-line components are depicted in Figures 1, 2, and 3, respectively. Consider the conceptual design (Figure 1). The ATR Kernel makes decisions on how a given ATR application task should be accomplished, including determining the partial ordering of subtasks and which algorithms should be used to accomplish each subtask. The HC Kernel decides how the partially ordered algorithmic suggestions should be implemented and mapped onto the heterogeneous parallel platform. Also, the HC Kernel interacts with the Basic Kernel to execute the application and monitor its execution. Thus, the ATR Kernel deals with application issues, while the HC Kernel deals with implementation issues. Information from the Algorithm Database and the Knowledge Base is used to support the ATR and HC Kernels. This design has its roots in the high-level model presented in [3] for automatic dynamic processor allocation in a partitionable parallel machine with homogeneous processors.

This paper concentrates on the operation of the HC Kernel; the other components will be discussed only to the extent needed for describing the HC Kernel (additional information on the other components is in [2]). In particular, this paper focuses on (1) the application and hardware platform characteristics that are needed to enable the use of the HC Kernel, (2) the overall IOS structure that will include the HC Kernel, (3) the techniques that comprise the HC Kernel, and (4) how to collect the information needed for the HC Kernel to operate. The IOS has not been implemented; such an implementation is a major undertaking and outside the scope of this paper, which is the design concepts for the HC Kernel.

The IOS differs from other real-time HC mapping techniques in that it allows on-line real-time use of off-line precomputed mappings. This is significant because off-line heuristics can produce better mappings because they can have much longer execution times to search for a good solution than what is practical for an on-line heuristic. Thus, the mapping quality of an off-line time-consuming heuristic can be approached at real-time speeds.

The IOS ideas can also be used for other application domains and classes of hardware platforms whose characteristics are similar to those of the iterative ATR applications and platforms considered here. Examples of other such application domains are sensor-based robotics, intelligent vehicle highway systems, air traffic control, nuclear facility maintenance, weather prediction, intruder detection, and manufacturing inspection.

The paper is organized as follows. Section 2 describes the application domain and Section 3 the heterogeneous hardware platform. Overviews of the off-line and on-line components of the IOS are presented in Sections 4 and 5, respectively. More information about the HC Kernel off-line and on-line components are provided in Sections 6 and 7, respectively.

2: Application domain

Simply stated, an ATR system takes a set of images from a group of sensors and produces a description of the scene. The various types of processing required in an ATR system can be broadly classified into three groups: numeric computation, quasi-symbolic computation (e.g., where numeric and symbolic types of operations are used to describe surfaces and shapes of objects in the scene), and symbolic computation (e.g., used to produce the scene description). Heterogeneous parallel architectures are ideal computing platforms for efficiently handling computational tasks with such diverse requirements.

A key technical issue that must be addressed to exploit the inherent potential of heterogeneous parallel computing systems to efficiently implement ATR applications is the development of a high-level operating system that can fully use the architectural flexibility of such a system. Such a high-level operating system must be able to assign each ATR application subtask to the processors where it is best suited for execution. Often, subtasks can execute concurrently, sharing resources. Because the execution time of application subtasks in an ATR system is highly input data dependent (e.g., number of currently located objects), this matching and scheduling of application subtasks to processors must be performed dynamically at run time.

This work is being developed for a class of ATR applications each of which can be modeled as an iterative execution of a set of partially ordered subtasks. Each ATR application in this class is a production job that is executed repeatedly. Thus, it is worthwhile to invest off-line time in preparing an effective mapping of the application onto the hardware platform used to execute it. The ATA (automatic target acquisition) system described in [4] is an example of such an iterative ATR application.

Each application task will be an instantiation of a DDG (data dependency graph), whose nodes are the subtasks that need to be executed to perform the application and whose arcs are the the data dependencies between subtasks. The expected number of subtasks is ten to 50. The DDG will be structured as a directed acyclic graph (DAG). The IOS is being designed for applications that will iteratively execute such a DDG. Note that while the subtasks' dependencies are represented as a DAG, subtasks themselves may contain loops.

For the initial iteration through the set of subtasks, the IOS will use information about the processing environment in its selection of algorithms for the subtasks, and their associated implementations. As part of this, the IOS will decide how to assign processing resources (processors) to the subtasks.

After each execution iteration through the set of subtasks, the values of certain dynamic parameters of the application may change, such as the number of objects detected in the current frame of a real-time image stream being processed. It is expected that the values of these parameters will change slowly. After all subtasks have completed execution for a given iteration, and before the next iteration begins, the latest values of these dynamic parameters will be reported to the on-line HC Kernel. The HC Kernel will use the most recent values of such dynamic parameters to estimate if it is worthwhile to reconfigure the assignment of processing resources to subtasks to reduce execution time of the next iteration. If it is desirable, the HC Kernel will select a new assignment to use for the next execution iteration through the subtasks. If not, the same assignment will continue to be used.

3: Heterogeneous hardware platform

The type of target hardware platforms considered for this study are driven by the expected needs of the kinds of ATR applications that are of interest to the U.S. Army Research Laboratory. Thus, for the intended application environment of the IOS, it is assumed that there will be up to four different types of processors, and up to a total of 64 processors (of all types combined). For example, two types may be SHARC processors and PowerPCs. These processors will comprise the heterogeneous parallel architecture onto which application tasks will be mapped. A system of this size should provide the real-time computing power needed for the intended application domain. The IOS approach is appropriate for larger HC platforms as well.

A small-scale example of the type of hardware platform being considering is the one described in [5]. This system was developed by the Army Research Laboratory for a real-time ATR relational template matching algorithm. Another example, although outdated, is [1], which describes a heterogeneous hardware platform designed to perform ATR research and prototyping.

All the processors of all types will have communication paths to one another. Communications among processors of the same type is assumed to be symmetric in the sense that the conflict-free time for any pair of processors to communicate is the same. For example, a Mercury daughter board can be populated with six SHARC processors that physically share a DRAM. To connect processors of different types, a VME bus can be used to provide communications among different collections of processors. In addition, the hardware platform will include (1) a workstation, for off-line IOS operations to develop an application implementation and for use as the Application User interface, and (2) a Host Processor, which will monitor the application implementation during its execution and implement the on-line HC Kernel.

For simplicity, it is assumed that if an implementation of a given subtask uses multiple processors, all processors will be of the same type. Given this and the symmetry property of the inter-processor communications among processors of the same type, the expected execution time of a particular multiprocessor implementation of a subtask is independent of which fixed-size subset of the processors of a given type are assigned to execute the subtask.

The IOS design should be capable of working with any hardware platform of the type described above. A given hardware platform may be used for many different ATR applications. It is assumed that when a given ATR application is executing on a platform, that platform is dedicated to that application.

4: Overview of IOS off-line components

Figure 3 shows the off-line components of the IOS. The Knowledge Base contains a collection of DDGs. The Algorithm Database contains information about algorithms that can be used to perform the subtasks in the DDGs. The DDGs and algorithm information is supplied by the application domain expert, the Application Developer. The Application Developer, who is responsible for developing the application software, will typically be a different person (or people) from the Application User(s), who may know nothing about software development and may just use the ATR system as a prepackaged tool. An analogy to this in the personal computing field is the programmer who develops a software package for graphics versus the package user who draws figures with the tool without any knowledge of the details of the actual code in the software package.

To design a particular application, the Application Developer first selects a DDG. (The Application Developer can also specify a particular set of DDGs that can execute simultaneously and be treated as a single DDG [2].) Each DDG has associated with it a list of Application Characteristic and Input Data Characteristic names, whose values must be filled in when the DDG is instantiated for a given application and associated environment. (The SmartNet project uses a similar set of characteristics called "Compute Characteristics," whose values when an application is invoked are called the "Compute Characteristics Operating Point" [9].) Each subtask in the DDG is assigned one or more algorithms whose *image processing performance* for that subtask and its associated Application and Input Data Characteristics are above some threshold (note that execution time is not considered by the ATR Kernel). Performing this assignment transforms a DDG into an ODDG (over-instantiated DDG). The ODDG is constructed by the ATR Kernel ODDG Generator.

Each of these sets of Application and Input Data Characteristics can be divided into static and dynamic parameters. Static parameters are those Input Data Characteristics, such as image size, and Application Characteristics, such as type of object of interest (e.g., tank), that will not change during the execution of the application task. Dynamic parameters, in contrast, are those Input Data Characteristics, such as amount of clutter, and Application Characteristics, such as number of located objects to be identified, that will change during run time and can be computed by the application as it executes. When an Application Developer instantiates a DDG to implement an application in a given environment, the values for the static characteristics are known and provided by the Application Developer. For the dynamic characteristics, the Application Developer is expected to provide ranges for these values (i.e., the minimum and maximum value each given dynamic parameter can have). It is the job of the Scenario Generator to use these ranges to derive representative values for the dynamic parameters.

The Scenario Generator subdivides the range of each dynamic parameter into C equal sized intervals (i.e., each dynamic parameter range is transformed into a set of C choices of representative values for that parameter, and these C choices are equally distributed across the range). Assume there are D dynamic parameters. Each set of D values for these D dynamic parameters, one per parameter, is called a scenario. The number of different scenarios that can be generated is $S = C^D$.

For a given application, the ATR Kernel ODDG Generator creates a distinct ODDG for each scenario. Thus, for a single application and associated static environment, one DDG is selected by the Application Developer, which is the basis for S ODDGs generated by the ATR Kernel ODDG Generator, one for each scenario. For each ODDG, only one MDDG (mapped DDG) is generated by the HC Kernel MDDG Generator. The MDDG specifies how the corresponding ODDG will be implemented and mapped onto the HC platform, as discussed further in later sections.

Therefore, the number of MDDGs generated by the HC Kernel MDDG Generator for a given application and its associated static environment is S. These are the S MDDGs that will compose the MDDG Table constructed by the MDDG Table Builder for that application and static environment (the HC Kernel MDDG Generator passes the MDDGs to the MDDG Table Builder). The MDDG Table will be indexed as a D-dimensional array, where each dimension is of size C, i.e., an MDDG Table entry will be accessed by a list of D indices (corresponding to a scenario), where the i-th index corresponds to an allowable representative value for the i-th dynamic parameter. MDDG Tables are stored in the Knowledge Base, as are the associated Application Menus constructed by the Application Menu Builder.

A question that arises is what the value of S should be. The larger S is, the closer a given scenario in the MDDG Table may match a given set of actual dynamic parameter values calculated during execution of the application. It is expected that the closer this match is, the better the mapping specified by the corresponding MDDG Table entry will be. However, the larger S is, the larger the MDDG Table will be and the larger the number of ODDGs and MDDGs for a given application and environment will be, resulting in longer off-line execution time for the ATR Kernel ODDG Generator and HC Kernel MDDG Generator. Thus, the IOS implementor will need to select a value for S that balances these factors based on experience with applications in the intended operating domain.

A variation on the scheme described above (where each dynamic parameter is divided into C equidistant choices) is to allow the Application Developer to specify the number of choices for a given dynamic parameter and do this for all or some subset of the dynamic parameters. Additionally, the Application Developer may wish to specify the exact choices of representative values to use for one or more of the dynamic parameters. To implement such variations, the Scenario Generator would need to be designed to interact with the Application Developer to enforce the given value selected for S.

Three items of information about the hardware platform will be needed by the HC Kernel MDDG Generator: (1) algorithm implementation execution time, (2) number of each type of processor available, and (3) interprocessor communication time. The HC Kernel MDDG Generator uses this information when it applies the heuristic for determining an effective assignment of subtask implementations to processors.

The Algorithm Database will include one or more implementations of each algorithm (e.g., one for each processor type). The Algorithm Database must also contain the expected execution time of each algorithm implementation, typically specified as a function of type and number of processors assigned, interprocessor communication time, and certain static and dynamic Input Data and Application Characteristics. This is an expected time, rather than a definite time, because it may vary depending on the actual values of the input data being processed. This expected execution time information must be provided by the Application Developer, who also supplies the code for each implementation of a given algorithm. It is expected that algorithm implementations will be written using the number of processors (of a given type) as an input parameter whenever possible. The assumption of the availability of expected implementation execution time for each type of processor (or set of processors of the same type) is typically made for the current state of the art in HC systems (e.g., [8], [13], [17], [21]). The Application Developer can determine this information empirically. The HC Kernel MDDG Generator needs to use this information (in conjunction with the other information below) to determine the expected total application task execution time.

The second item needed is the total number and type of the processors in the platform. The HC Kernel MDDG Generator needs this to know how many processors of each type it has available to assign. This is specified by the Platform Architect, who is responsible for the hardware design of the system. Typically, this person (or people) is distinct from the Application Developer(s) and Application User(s), although it is assumed that the Platform Architect(s) will consult with the Application Developer(s). Referring to the earlier analogy with personal computing, the people responsible for designing the system hardware configuration will typically not be the same people who develop the graphics package.

The last item needed is a communication matrix indicating the time it takes for each processor in the platform to communicate with every other processor in the platform. This is also specified by the Platform Architect. Entry (i,j) in this matrix is the information needed to calculate the conflict-free time for processor i to send data to processor j. The communication time for a given pair of processors typically will have two components: a fixed latency time for the first byte to arrive, and a variable time that depends on the length of the message being transmitted (that is based on the bandwidth of the communication path). This type of matrix is used by other researchers in HC (e.g., [13], [17], [21]). This matrix is needed for the HC Kernel MDDG Generator to determine the expected inter-subtask communication times for possible mappings of subtasks onto the platform.

The Platform Architect can also specify faulty variations of a given platform as additional separate platforms. By doing this, the HC Kernel MDDG Generator will construct mappings that can be accessed and used in real time should a potential hardware fault occur. Each MDDG Table will correspond to a given platform variation and a given application with its associated static environment.

5: Overview of IOS on-line components

Figure 2 shows the on-line components of the IOS. Once in the field, the Application User interacts with the ATR Kernel User Interface, including the Application Menus. The Application Menus will be used on-line by the Application User to select a particular initial MDDG to use to invoke an application (with an associated set of Application User specified values for static parameters and initial choices of allowable representative dynamic parameters). That MDDG is passed to the HC Kernel Monitor to use as the initial mapping. Then, as the application is executing, the HC Kernel Monitor monitors the run time values of the dynamic parameters at the end of each iteration through the underlying DDG to decide whether to continue with the current mapping, or to select and instantiate a new mapping (for the next iteration) from among the entries of the relevant MDDG Table (which were determined off-line). Thus, the off-line processing provides a set of predetermined mappings that the on-line processing can index in real time.

6: HC Kernel MDDG Generator

The HC Kernel MDDG Generator is the component of the off-line IOS (Figure 3) that is responsible for mapping each ODDG onto the heterogeneous hardware platform creating a corresponding MDDG. Each MDDG is isomorphic to a given ODDG. For each node in the ODDG, there is a corresponding node in the MDDG that includes: (1) which implementation (stored in the Algorithm Database) will be used for one of the algorithms in that ODDG node; (2) pointers to the needed object code for that implementation; (3) any additional information needed for loading that implementation onto the heterogeneous parallel system; (4) any needed inter-subtask (i.e., inter-node) communications between that given MDDG node and any other node in the MDDG (which will also enforce data-dependency constraints among the subtasks); and (5) the specific set of processors that will be used to execute that MDDG node. In addition to this node specific information, global MDDG information is stored with each MDDG, including: (1) the schedule for the execution order of the MDDG nodes and inter-subtask data transfers; and (2) the expected execution time for one iteration through the MDDG for the scenario (values of dynamic parameters) that was specified along with (and is the basis of) the corresponding ODDG. The execution time information in (2) above is used by the HC Kernel Monitor when deciding whether to change the mapping (i.e., reconfigure), as discussed in the next section. As discussed earlier, the HC Kernel MDDG Generator passes this MDDG to the MDDG Table Builder, to be stored as part of the MDDG Table for the given application and static environment characteristics.

The Application Developer can also specify an estimate of the average overhead time to reconfigure the mapping of the given application (and associated static environment) on the hardware platform. This estimate will represent the time needed to remap the application during execution as a result of changes to the values of the dynamic parameters. The estimated average reconfiguration overhead time will be sent from the Application Developer to the HC Kernel MDDG Generator through the IOS–App Builder software. If the Application Developer does not know how to estimate this value, the Application Developer can provide the IOS-App Builder with a set of scenarios that are expected to occur frequently, and the IOS-App Builder can use these to actually perform remappings among these scenarios to calculate an overhead estimate. Alternatively, the IOS-App Builder can generate a relatively small random subset of scenarios to use to calculate the estimate. Rather than derive a single estimated average overhead value, the reconfiguration time could be calculated for each of the S^2 possible old and new configuration pairs, and stored in the Knowledge Base; however, this would require an excessive amount of space to store and an excessive amount of off-line time to calculate, and, thus, it is not advisable.

The HC Kernel MDDG Generator will pass the average reconfiguration overhead time estimate to the MDDG Table Builder to be stored as part of the MDDG Table for this application. This overhead time will be used by the HC Kernel Monitor when deciding whether or not to perform a reconfiguration.

Thus, the HC Kernel MDDG Generator gets from the ATR Kernel ODDG Generator an ODDG and an associated scenario, it creates an MDDG containing the information specified above, and then passes this MDDG to the MDDG Table Builder that constructs a complete table and stores it in the Knowledge Base. The rest of this section will examine how the HC Kernel MDDG Generator can derive the information that comprises the MDDG.

In the HC field, the node specific items (1), (4), and (5) defined at the beginning of this section are part of the process of matching subtasks in a task graph to processors in the heterogeneous system. The node specific items (3) and (4) above are adaptations of standard operating system functions that will need to be implemented, but will not be discussed further here. The global MDDG item (1) is referred to as the scheduling component of a mapping in an HC environment.

For general HC, deriving an optimal matching and scheduling is intractable (i.e., it is known to be an NP-complete problem that requires exponential execution time to perform an exhaustive search of the space of possible solutions [7]). This is true even when all execution and communication times can be determined statically (i.e., they are not input-data dependent). For the intended application domain and hardware platforms, an exhaustive search will take time proportional to the number of processors in the hardware platform to the power of the number of subtasks in the application. Thus, a heuristic is used, as is common in the heterogeneous field [19], [20].

The structure of the HC Kernel MDDG Generator is such that any good heuristic could be employed. The heuristic that is used could be one that can be executed during run time, such as a levelized-min-time heuristic (e.g., [12]), or an off-line heuristic, such as a genetic algorithm (e.g., [17], [21], [25]). In general, an off-line heuristic can find a better mapping than an on-line run-time heuristic because its execution time can be orders of magnitude longer than that of the run-time algorithm. An example of this difference in quality of matchings is provided in [25]. However, because of the longer execution times of genetic-algorithm-based heuristics, it would not be appropriate to execute a genetic algorithm while a real-time application is running in order to decide how to reconfigure resources based on the actual values of the dynamic parameters at the end of a given iteration through the underlying DDG.

For this application domain, it is possible to use off-line precomputed mappings to reconfigure resources in real time. In particular, the IOS will: (1) allow the HC Kernel MDDG Generator to use genetic algorithms (or other off-line heuristics) to determine a matching and scheduling (i.e., MDDG) off-line for each scenario associated with a production ATR application task; (2) allow the user to select an initial MDDG when the application's execution is initiated (through the ATR Kernel User Interface); and (3) allow the HC Kernel Monitor to select a new MDDG during execution if desired based on the actual values of the dynamic parameters at the end of an iteration through the corresponding DDG. Item (1) above is the subject of this section, item (2) is part of the ATR Kernel User Interface (not the HC Kernel), and item (3) will be covered in the next section.

While the HC Kernel MDDG Generator can incorporate any appropriate off-line mapping heuristic, to describe the design ideas involved in the HC Kernel it will be assumed that a genetic algorithm is used. In this environment, the genetic algorithm is a guided heuristic search through the space of possible matchings and schedulings (called solutions). The genetic algorithm in [25] was found to be very successful, and used an HC model that is quite compatible with the situation here. However, in [25] each subtask was assigned a single machine, so, for the type of platforms described in Section 3, the "chromosome" representing the matching will have to be adapted to allow for multiple processors of the same type to be assigned to a subtask. This genetic algorithm and a method for adapting it for use in the ATR environment are summarized in the appendix.

The approach in [25] differs from other genetic algorithm approaches to matching and scheduling for heterogeneous systems in the literature ([17], [21], [23]) in many ways. The most significant difference from [17] and [21] is that the [25] model of a heterogeneous system is more realistic (e.g., [21] assumes an unlimited number of machines). The main difference between [25] and [23] is that in [25] it is assumed that there is a given target hardware platform, whereas [23] selects processors to be included in the platform and uses processor cost as a co-metric. Thus, for the HC Kernel for the application and platform environments discussed in Sections 2 and 3, it is most appropriate to build on the [25] genetic approach.

For the intended application domain here, the genetic algorithm needs the following information to create a matching and scheduling, i.e., to transform an ODDG into an MDDG: (1) the structure of the underlying DDG; (2) the expected execution time of each subtask on a set of processors (of the same type) assigned, as a function of the type of the processors and the number of processors; (3) the inter-subtask data transfers needed, in terms of formats and expected sizes of the data items to be transferred; (4) the expected time to send data from one processor to another as a function of the size of the data item to be transferred; and (5) the number of each type of processor that is in the hardware platform. The genetic algorithm selects a subset of possible solutions and then evaluates them using the information in items (1) to (5) above. The genetic algorithm uses the results of the evaluations of these possible solutions to generate a new set of possible solutions. This process iterates until some stopping criteria is met (e.g., no improvement in solution quality after a given fixed number of iterations within the genetic algorithm). When the genetic algorithm stops iterating, the best solution found is used as the mapping for the MDDG.

How the HC Kernel MDDG Generator gets each of the above information items is now considered. Much of the relevant information flow is depicted in Figure 3.

The HC Kernel MDDG Generator gets the item (1) from the ODDG that is passed to it from the ATR Kernel ODDG Generator. The genetic algorithm uses a topological sort (i.e., a valid total ordering) of the subtasks in the ODDG to establish the order in which it evaluates the nodes of the ODDG.

Item (2) is provided by the Application Developer and stored in the Algorithm Database. Information about static characteristics may be needed in some cases (e.g., the size of subtask input and output data blocks). The HC Kernel MDDG Generator will receive this from the IOS-App Builder. The expected execution times for subtasks calculated from this information are used by the genetic algorithm in its evaluation of possible solutions.

Recall that each subtask of the underlying DDG corresponds to a node of the ODDG that may contain more than one possible algorithm to perform the subtask. For each of these algorithms, there is at least one implementation that can execute on the hardware platform, and possibly more than one. When the genetic algorithm evaluates a given possible mapping solution (i.e., assignment of resources), it selects the implementation for the subtask that has the smallest expected execution time for the resources assigned to that subtask. If no implementation is available for that assignment, that mapping is considered invalid by the genetic algorithm and is discarded from the set of possible mappings.

There may be cases when there are inter-subtask implementation interactions that must be considered when selecting the implementation for a given subtask. One example of how this can occur is when some subtask A sends a data block, e.g., an image, to a subtask B, and the image is stored using different formats for the two subtasks' implementations selected (e.g., assigning rectangular subimages to processors in a multiprocessor implementation for subtask A and row striping of subimages in a multiprocessor implementation for subtask B). When this occurs, overall implementations based on each data format are considered, and the one with the smallest execution time is used as a basis for selecting the implementations for subtasks A and B for this possible solution mapping. For the intended application domain, this should not cause a significant time penalty relative to the total execution time for the genetic algorithm to generate a mapping for the MDDG being constructed.

An alternate approach for handling mismatched data block storage organizations is to allow each subtask implementation to use its own choice of organization and convert between organizations during run time. For the intended real-time ATR applications, this option will not be considered; however, if it becomes desirable to consider this option in the future, the genetic algorithm framework of [25] will allow it to be included.

Item (3) is provided by the Application Developer and stored in the Algorithm Database as part of the input/output parameter descriptions (see [2]). This inter-subtask data transfer information is used by the genetic algorithm (in conjunction with the information in item (4)) in its evaluation of possible solutions.

Item (4) is provided by the Platform Architect and stored in the Knowledge Base as part of the hardware description. Together with the information from item (3), the genetic algorithm can evaluate the expected inter-subtask communication times for a given possible mapping solution.

Item (5) is provided by the Platform Architect and stored in the Knowledge Base as part of the hardware description. It is used by the genetic algorithm to know the upper bound on the number of each type of processor that can be assigned. It should be noted that in some cases the optimal mapping (in terms of total execution time for an iteration through the ODDG) may not use all available processors (i.e., in some cases the overhead involved in using all of the processors may make it better to use only a subset of the available processors, as discussed in [18]).

Thus, using all of this information, the genetic algorithm can create a matching and scheduling to be included in the MDDG for the given ODDG. As part of the evaluation for the potential solutions, the genetic algorithm computes the expected total execution time per iteration for each solution. For the mapping solution chosen, this expected total time is stored with the MDDG, as mentioned earlier. The completed MDDG is sent to the MDDG Table Builder.

7: HC Kernel Monitor

The HC Kernel Monitor is the on-line component of the IOS (Figure 2) responsible for (1) establishing the initial mapping of the given application onto the hardware platform, and (2) monitoring the execution of the application and at the end of each iteration through the corresponding DDG deciding if and how the mapping of the application onto the hardware platform should be changed based on information about the actual values of the dynamic parameters. To establish the initial mapping, the HC Kernel Monitor uses the MDDG index (scenario) and associated MDDG Table identifier passed to it from the ATR Kernel User Interface. With this index and identifier, the HC Kernel Monitor can access from the appropriate MDDG Table in the Knowledge Base the MDDG entry selected by the Application User (see Figure 2). The HC Kernel Monitor then passes the relevant information to the Basic Kernel (i.e., loading, configuring, and scheduling information). The Basic Kernel accesses the implementation code directly from the Algorithm Database using pointers provided with the loading information. The Basic Kernel can then begin execution of the application task.

During execution of the application, the HC Kernel Monitor receives updated actual values of the dynamic parameters at the end of each iteration through the corresponding DDG. Specifically, after all subtasks' implementations have completed execution for a given iteration of the DDG corresponding to the application, and before the next iteration begins, the latest values of these dynamic parameters will be sent to the HC Kernel Monitor from the Basic Kernel. The HC Kernel Monitor will use the most recent values of these dynamic parameters to estimate if changing the matching and scheduling will reduce the expected execution time of the next iteration through the corresponding DDG. Thus, this decision is made in real time after all subtasks' implementations for the current iteration have finished executing and before any subtask implementations begin to execute for the next iteration.

If it is desirable to change the mapping, then it is the responsibility of the HC Kernel Monitor to select a new matching and scheduling (i.e., a new MDDG entry from the given MDDG Table) to use for the next execution iteration through the corresponding DDG. If not, the same mapping will continue to be used.

To determine if the current mapping should be changed, the HC Kernel Monitor performs the following sequence of steps.

(1) For each dynamic parameter, the representative value included in the allowable choices for that parameter (as specified by the Scenario Generator) is found that is closest in absolute difference to the current actual value for that parameter. This is done for all D dynamic parameters for this application. The resulting vector of D representative values is used as the approximation of the set of current actual dynamic parameters in terms of a precomputed scenario (MDDG Table index). Call this new MDDG Table index $\underline{A}$. If MDDG Table index A is the same MDDG Table index as the one used for the current iteration, the next iteration of the application proceeds with the same mapping as the current iteration. If MDDG Table index A is not the same as the one used for the current iteration, then steps (2) through (5) are performed.

(2) The HC Kernel Monitor then accesses the MDDG Table for this application, using A as the index. The expected execution time for an iteration through the MDDG Table entry corresponding to MDDG Table index A is used as an estimate of what the expected execution time will be for the actual dynamic parameters for the

next iteration of the application if the remapping occurs. Call this time T1.

(3) The actual execution time of the last (current) iteration through the application's corresponding DDG is used as an estimate for the time to execute the next iteration with the current actual dynamic parameters with the current mapping (this is only an estimate because some or all of these parameter values may have been different at the beginning of the current iteration and changed sometime during the execution of this iteration). Call the actual execution time for the current iteration T2.

(4) When making the decision to reconfigure the resources, in addition to comparing T1 and T2, the HC Kernel Monitor must also consider the time required to perform the reconfiguration (e.g., any code and data movements that the reconfiguration will require). The average reconfiguration overhead time estimate stored with the MDDG Table is used. Call this overhead time TO. (The HC Kernel Monitor could start with the estimate provided with the MDDG Table and then, for each time a reconfiguration is performed for this application, measure the actual overhead time and use this experiential information to modify the most recent estimate in some weighted way.)

(5) If $(T1 + TO) < T2$, then the HC Kernel Montior instructs the Basic Kernel to change the mapping to the one corresponding to the MDDG Table entry for MDDG Table index A.

Thus, the HC Kernel Monitor can use the actual values of dynamic parameters at the end of each execution iteration of the application to make remapping decisions and select new mappings derived by a time-consuming off-line heuristic. As can be observed, the execution of the computationally simple steps (1) to (5) above can be done in real time, causing relatively negligible overhead compared to the expected execution time of an iteration through an ATR DDG.

The decisions made by the HC Kernel Monitor are based on heuristics and approximations. Thus, pathological cases could cause a bad decision to be made. In general, the ideas underlying the HC Kernel Monitor will lead to reduced overall application execution time for the environment under consideration.

As an example of a possible pathological case that could occur, if $T1 = T2 - i$ and $TO = i + 1$, then $(T1 + TO) > T2$, and reconfiguration would not be done. If the values of the dynamic parameters do not change for the next 20 iterations, the execution time for those iterations will total approximately $20 \times T2$. If reconfiguration had been done, the execution time for those iterations would total approximately $20 \times T1 = 20 \times T2 - 20 \times i$. Because the values of dynamic parameters depend on the input data, the number of iterations that will occur with no changes to the dynamic parameter values can never be predicted with certainty. If situations such as the above do appear to occur frequently for a given application, it would be possible to instrument the HC Kernel Monitor to collect the relevant statistics and then develop and add special rules. For this example, a new rule may be that if after so many iterations with no change to the dynamic parameter values, if the relationship between TO and $(T2 - T1)$ is smaller than some threshold, perform the remapping. However, there is no guarantee that the dynamic parameter values will not change during the next iteration, so the decision to include such rules must be made carefully.

Thus, by studying properties of the application domain, the IOS builder may decide to fine tune the HC Kernel Monitor in different ways. The HC Kernel design has the flexibility to allow such tuning.

8: Summary

This study focused on a design for an IOS for iterative ATR tasks and an associated specific class of dedicated heterogeneous hardware platforms. For the computational environment considered, an HC Kernel was presented for making real-time on-line input-data dependent remappings of the application subtasks to the processors in the heterogeneous hardware platform using previously stored off-line statically determined mappings. In particular, it was shown that the HC Kernel can be used to create the MDDG Table off-line and then use it to make real-time on-line decisions and selections of mappings. In addition to the HC Kernel, the relevant parts of other components of the IOS were briefly discussed. The overall strategy of the IOS and the interactions of the IOS components are summarized in Figures 1 to 3. The IOS ideas introduced here can also be used for other application domains and classes of hardware platforms whose characteristics are similar to those of the applications and platforms considered here.

Acknowledgments: The authors thank Janet M. Siegel, Min Tan, and Lee Wang for their comments.

References

[1] S. Brandt and J. R. Budenske, "Starcon--a reconfigurable fieldable signal processing system," *SPIE Conf. on Image Understanding in the '90s: Building Systems that Work, SPIE Vol. 1406,* pp. 122-126, Oct. 1990.

[2] J. R. Budenske, H. J. Siegel, R. S. Ramanujan, K. J. Thurber, and M. D. Pritt, *Intelligent Operating System Final Technical Report,* Technical Report ATC-RD-96-

04, Architecture Technology Corp., Minneapolis, MN, 1996.

[3] C. H. Chu, E. J. Delp, L. H. Jamieson, H. J. Siegel, F. J. Weil, and A. B. Whinston, "A model for an intelligent operating system for executing image understanding tasks on a reconfigurable parallel architecture," *Journal of Parallel and Distributed Computing,* Vol. 6, No. 3, pp. 598-622, June 1989.

[4] P. David, S. Balakirsky, and D. Hillis, "A real-time automatic target acquisition system," *Conf. on Unmanned Vehicle Systems,* pp. 183-198, July 1990.

[5] P. David, P. Emmerman, and S. Ho, "A scalable architecture system for automatic target recognition," *13th AIAA/IEEE Digital Avionics Systems Conf.,* pp. 414-420, Oct. 1994.

[6] L. Davis, ed., *Handbook of Genetic Algorithms,* Van Nostrand Reinhold, New York, NY, 1991.

[7] D. Fernandez-Baca, "Allocating modules to processors in a distributed system," *IEEE Transactions on Software Engineering,* Vol. SE-15, No. 11, pp. 1427-1436, Nov. 1989.

[8] R. F. Freund, "The challenges of heterogeneous computing," *Parallel Systems Fair at the 8th Int'l Parallel Processing Symp.,* pp. 84-91, Apr. 1994.

[9] R. F. Freund, T. Kidd, D. Hensgen, and L. Moore, "SmartNet: A scheduling framework for heterogeneous computing," *2nd Int'l Symp. on Parallel Architectures, Algorithms, and Networks (I-SPAN '96),* pp. 514-521, June 1996.

[10] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning,* Addison-Wesley, Reading, MA, 1989.

[11] J. H. Holland, *Adaptation in Natural and Artificial Systems,* Univ. of Michigan Press, Ann Arbor, MI, 1975.

[12] M. A. Iverson, F. Ozguner, and G. J. Follen, "Parallelizing existing applications in a distributed heterogeneous environment," *1995 Heterogeneous Computing Workshop (HCW '95),* pp. 93-100, Apr. 1995.

[13] A. Ghafoor and J. Yang, "Distributed heterogeneous supercomputing management system," *IEEE Computer,* Vol. 26, No. 6, pp. 78-86, June 1993.

[14] A. Khokhar, V. K. Prasanna, M. Shaaban, and C. L. Wang, "Heterogeneous computing: Challenges and opportunities," *IEEE Computer,* Vol. 26, No. 6, pp. 18-27, June 1993.

[15] J. L. Ribeiro Filho and P. C. Treleaven, "Genetic-algorithm programming environments," *IEEE Computer,* Vol. 27, No. 6, pp. 28-43, June 1994.

[16] G. Rudolph, "Convergence analysis of canonical genetic algorithms," *IEEE Trans. Neural Networks,* Vol. 5, No. 1, pp. 96-101, Jan. 1994.

[17] P. Shroff, D. W. Watson, N. S. Flann, and R. F. Freund, "Genetic simulated annealing for scheduling data-dependent tasks in heterogeneous environments," *5th Heterogeneous Computing Workshop (HCW '96),* pp. 98-117, Apr. 1996.

[18] H. J. Siegel, J. B. Armstrong, and D. W. Watson, "Mapping computer-vision-related tasks onto reconfigurable parallel processing systems," *IEEE Computer,* Vol. 25, No. 2, pp. 54-63, Feb. 1992.

[19] H. J. Siegel, J. K. Antonio, R. C. Metzger, M. Tan, and Y. Alexander Li, "Heterogeneous computing," in *Parallel and Distributed Computing Handbook,* ed. A. Y. Zomaya, McGraw-Hill, New York, NY, pp. 725-761, 1996.

[20] H. J. Siegel, H. G. Dietz, and J. K. Antonio, "Software support for heterogeneous computing," in *The Computer Science and Engineering Handbook,* ed. A. B. Tucker, Jr., CRC Press, Boca Raton, FL, pp. 1886-1909, 1997.

[21] H. Singh and A. Youssef, "Mapping and scheduling heterogeneous task graphs using genetic algorithms," *5th Heterogeneous Computing Workshop (HCW '96),* pp. 86-97, Apr. 1996.

[22] M. Srinivas and L. M. Patnaik, "Genetic algorithms: A survey," *IEEE Computer,* Vol. 27, No. 6, pp. 17-26, June 1994.

[23] Y. G. Tirat-Gefen and A. C. Parker, "MEGA: An approach to system-level design of application specific heterogeneous multiprocessors," *5th Heterogeneous Computing Workshop (HCW '96),* pp. 105-117, Apr. 1996.

[24] D. Tolmie and J. Renwick, "HiPPI: Simplicity yields success," *IEEE Network,* Vol. 7, No. 1, pp. 28-32, Jan. 1993.

[25] L. Wang, H. J. Siegel, and V. Roychowdhury, "A genetic-algorithm-based approach for task matching and scheduling in heterogeneous environments" *5th Heterogeneous Computing Workshop (HCW '96),* pp. 72-85, Apr. 1996.

AUTHOR BIOGRAPHIES

John R. Budenske was born in Faribault, Minnesota. He earned his B.S. in Computer Science at the University of Minnesota in 1983. From 1982 to 1992, John worked at the Honeywell Systems and Research Center (later to become the Alliant Techsystems Research and Technology Center) and was involved in research in artificial intelligence, signal image processing, robotics, simulation, and database systems. During that time, in 1987, he received his M.S. in Computer Science from the University of Minnesota. From 1992 to 1996, John worked on advanced methods for software reengineering and object oriented programming at Loral Defense Systems in Eagan, Minnesota. In 1993, he earned his Ph.D. in Computer Science, also from the University of Minnesota, where he researched methods for intelligent plan execution for robotic systems. Currently, he is performing research in intelligent control, software engineering, electronic collaboration, and intelligent agents for Architecture Technology Corporation in Minneapolis, Minnesota. In intelligent control he is investigating methods for controlling heterogeneous (multi-processor, multi-sensor, multi-driver) systems to execute tasks that require information sensed from the current environment, as well as domain knowledge, in order determine subsequent processing steps (e.g., complex robotics tasks).

Ranga S. Ramanujan is the Director of Research and Development at Architecture Technology Corporation. He received the B.S. degree from Annamalai University, India, the M.S. degree from Clarkson University, New York, and the Ph.D. degree from the University of Iowa, all in Electrical Engineering. He has over 12 years of industrial research and development experience in the areas of parallel and distributed computing, software engineering, networking, and fault-tolerant computing. Dr. Ramanujan has served as the principal investigator of several research efforts sponsored by the US Department of Defense, the National Science Foundation (NSF), and the National Aeronautics and Space Administration (NASA) covering the areas of real-time embedded systems, parallel processing, distributed computing, and network architectures.

Howard Jay Siegel is a Professor and Coordinator of the Parallel Processing Laboratory in the School of Electrical and Computer Engineering at Purdue University. He received two BS degrees from the Massachusetts Institute of Technology (MIT), and the MA, MSE, and PhD degrees from Princeton University. He has coauthored over 230 technical papers, has coedited seven volumes, and wrote the book *Interconnection Networks for Large-Scale Parallel Processing* (second edition 1990). He is a Fellow of the IEEE, was a Coeditor-in-Chief of the *Journal of Parallel and Distributed Computing*, and is currently on the Editorial Boards of both the *IEEE Transactions on Parallel and Distributed Systems* and the *IEEE Transactions on Computers*. He is an international keynote speaker and tutorial lecturer. Prof. Siegel's research and consulting interests include heterogeneous computing, parallel algorithms, interconnection networks, and reconfigurable parallel computer systems. In the area of heterogeneous computing, he is examining ways to match segments of a task to different machines in a heterogeneous suite to exploit the varied computational capabilities available. His algorithm work explores the factors involved in mapping a problem onto a parallel processing system to minimize execution time. Topological properties and fault tolerance are the focus of his research on interconnection networks for large-scale parallel machines. He is analytically and experimentally investigating the utility of the three dimensions of dynamic reconfigurability supported by the PASM design ideas and the small-scale proof-of-concept prototype: mixed-mode parallelism, switchable inter- processor communications, and system partitionability. Prof. Siegel has been Program Chair/Co-Chair or General Chair/Co-Chair of seven international conferences and Chair/Co-Chair of four workshops.

APPENDIX:

This appendix summarizes the genetic-algorithm-based approach for subtask matching and scheduling in HC environments that was presented in [25]. It is followed by a discussion of modifications needed to make it suitable for dealing with multiple processors of the same type being assigned to a given subtask. In the summary below, a machine corresponds to a processor type in the ATR environment.

An application task is decomposed into a set of subtasks $\underline{B}$ of size $\underline{|B|}$. Let $\underline{b_i}$ be the i-th subtask. An HC suite consists of a set of machines $\underline{M}$ of size $\underline{|M|}$. Let $\underline{m_j}$ be the j-th machine. Each machine can be a different type. The estimated expected execution time of subtask b_i on machine m_j is $\underline{T_{ij}}$. The global data items (gdis), i.e., data items that need to be transferred between subtasks, form a set $\underline{G}$ of size $\underline{|G|}$. Let $\underline{gdi_k}$ be the k-th global data item. The following assumptions about the applications and HC environment are made. The data dependencies among the subtasks are known and are represented by a directed acyclic graph (DAG). For each global data item, there is a single subtask that produces it (producer) and there are some subtasks that need this data item (consumers). Each edge goes from a producer to a consumer and is labeled by the global data item that is transferred over it. This application task has exclusive use of the HC environment, and the genetic-algorithm-based matcher/scheduler controls the HC machine suite (hardware platform). Subtask execution is non-preemptive. The heuristic assumes that all input data items of a subtask must be received before its execution can begin, and none of its output data items is available until the execution of this subtask is finished.

Genetic algorithms (GAs) are a promising heuristic approach to optimization problems that are intractable [6], [10], [11]. There are a great variety of approaches to GAs; many are surveyed in [15], [22]. The first step is to encode some of the possible solutions as chromosomes, the set of which is referred to as a population. In the [25] approach, each chromosome consists of two parts: the matching string and the scheduling string.

Let mat be the matching string, which is a vector of length |B|, where mat(i) = m_j (i.e., subtask b_i is assigned to machine m_j). Typically, multiple subtasks will be assigned to the same machine, and then executed in a non-preemptive manner based on an ordering that obeys the precedence constraints (data dependencies) specified in the application task DAG.

The scheduling string is a topological sort of the DAG (i.e., a valid total ordering of the partially orered DAG). Define ss to be the scheduling string, which is a vector of length |B|, where ss(k) = b_i, $0 \le i,k < |B|$ (i.e., subtask b_i is the k-th subtask in the scheduling string). Because it is a topological sort, if ss(k) is a consumer of a gdi produced by ss(j), then $j < k$. The scheduling string gives an ordering of subtasks that is used by the evaluation step. Thus, in this approach, each chromosome is a two-tuple <mat, ss>.

In the initial population generation step, a predefined number of chromosomes are created. A new matching string is obtained by assigning each subtask to a machine randomly. The DAG is first topologically sorted to form a basis scheduling string. Then, for each chromosome to be generated, this basis string is mutated multiple times using the scheduling string mutation operator (defined below) to generate a valid ss vector. The solution from a non-evolutionary baseline (BL) heuristic is also included in the initial population. It is common in GA applications to incorporate solutions from some non-evolutionary heuristics into the initial population, which may reduce the time needed for finding a satisfactory solution [6]. It is guaranteed that the chromosomes in the initial population are distinct from each other.

After the initial population is determined, the genetic algorithm iterates until a predefined termination criteria is met. Each iteration consists of the selection, crossover, mutation, and evaluation steps.

Each chromosome is associated with a fitness value, which is the completion time of the solution (i.e., matching and scheduling) represented by this chromosome (i.e., the expected execution time of the application task if the matching and scheduling specified by this chromosome were used). Overlapping among all of the computations and communications performed is limited only by inter-subtask data dependencies and the availability of the machines and the inter-machine network. The fitness values are determined in the evaluation step (discussed later).

In the selection step, all of the chromosomes in the population are ordered (ranked) by their fitness values. Then a rank-based roulette wheel selection scheme is used to implement proportionate selection [11]. The population size is kept constant and a chromosome representing a better solution has a higher probability of having one or more copies in the next generation population. This GA-based approach also incorporates elitism, i.e., the best solution found so far is always maintained in the population [16].

The selection step is followed by the crossover step, where some chromosomes are paired and corresponding components of the paired chromosomes are exchanged. The crossover operator for the scheduling strings randomly chooses some pairs of the scheduling strings in the current population. For each pair, it randomly generates a cutoff point, and divides the scheduling strings of the pair into top and bottom parts. Then, the subtasks in each bottom part are reordered. The new ordering of the subtasks in the bottom part of one string is the relative positions of these subtasks in the other original scheduling string, thus guaranteeing that the newly generated scheduling strings are valid schedules. The crossover operator for the matching strings randomly chooses some pairs of the matching strings in the current population. For each pair, it randomly generates a cutoff point, and divides both matching strings of the pair into two parts. Then the machine assignments of the bottom parts are exchanged.

The next step is mutation. The scheduling string mutation operator randomly chooses some scheduling strings in the current population. Then for each chosen scheduling string, it randomly selects a victim subtask. The valid range of the victim subtask is the set of the positions in the scheduling string at which this victim subtask can be placed and still have a valid topological sort of the DAG. After a victim subtask is chosen, it is moved randomly to another position in the scheduling string within its valid range. The matching string mutation operator randomly chooses some matching strings in the current population. For each chosen matching string, it randomly selects a subtask entry. Then the machine assignment for the selected entry is changed randomly to another machine.

The last step of an evolution iteration is the evaluation step to determine the fitness value of each chromosome in the current population (discussed earlier). The computation characteristics of the subtasks are obtained from the array T described above. The communication characteristics of the given HC system are also needed. To demonstrate the evaluation process, a communication subsystem that is modeled after a HiPPI LAN with a central crossbar switch [24] was assumed for the tests that were conducted (discussed further below). (For the ATR environment, the inter-processor communication matrix described in Section 4 will be used in the evaluation of the fitness value for each chromosome in the current population.) As stated earlier, the above steps of selection, crossover, mutation, and evaluation are repeated until one of the stopping criteria are met: (1) the number of iterations reaches some limit (e.g., 1000), (2) the population converged (all the chromosomes had the same fitness value), or (3) the best solution found was not improved after some number of iterations (e.g., 150).

In the tests of this GA approach in [25], simulated program behaviors were used. GA simulation studies were conducted using the following parameters. The probabilities for scheduling and matching string crossovers and scheduling and matching string mutations were 0.4, 0.4, 0.1, 0.1, respectively. This set of numbers was selected by experimentation. Small-scale tests were conducted with up to ten subtasks, three machines, seven global data items, and population size 50. For each test, the GA-based approach found a solution (matching and scheduling) that had the same expected completion time for the task as that of the optimal solution found by exhaustive search. Larger tests with up to 100 subtasks and 20 machines were conducted. Each of them had its number of global data items in the range $(2/3)*|B| < |G| < |B|$. The population size for these larger tests was chosen to be 200. This GA approach produced solutions (matchings and schedulings) that averaged from 150% to 200% better than those produced by the non-evolutionary levelized min-time (LMT) heuristic proposed in [12]. The heuristic in [12] was selected for comparison because it used a similar model of HC.

Now consider one way to adapt this GA heuristic for the case of allowing multiple processors to be assigned to a subtask. There are different ways in which this can be done; the method discussed below is just one example.

Recall that each machine m_j will represent a processor type in the ATR environment. After the initial population is generated, for each subtask the following is done. If the subtask is assigned to machine m_j, then a random number is generated that is from one to the total number of processors corresponding to type m_j that are in the hardware platform. This will be the number of processors of type m_j that are assigned to this subtask. As described earlier, in the case for [25], if multiple subtasks are assigned to the same machine, then the subtasks are executed on that machine is some order. In the ATR environment, if multiple subtasks are assigned to a total number of processors of type m_j that exceeds the number of that type that are in the hardware platform, then the subtasks that are assigned to the same processors must be executed on those processors in the order specified by the scheduling string. Because of the symmetry assumption about the communications among processors of the same type (see Section 3), it does not matter which processors of the same type are used for a subtask. If the Algorithm Database implementation information (Section 6) does not contain an implementation for some algorithm associated with that subtask in the ODDG that will execute on the number of type m_j processors assigned to that subtask, then that assignment is considered invalid, and another choice for number of processors is made (or processor type if necessary). Thus, each matching string now matches a subtask with one or more processors of the same type.

Performing mutations on the matching strings is done as specified earlier, except now in addition to randomly selecting a new machine (processor type), a number of processors of that type is randomly assigned (in the same manner as done for the initial population). Crossovers are done in the same way as before, except now instead of swapping machine assignments, the processor type and number of processors are swapped together.

Thus, with these modifications, the successful GA in [25] can be used in the ATR environment. The various GA parameters, such as population size and probability of performing a mutation, will be set based on experiments conducted with the ATR problem domain.

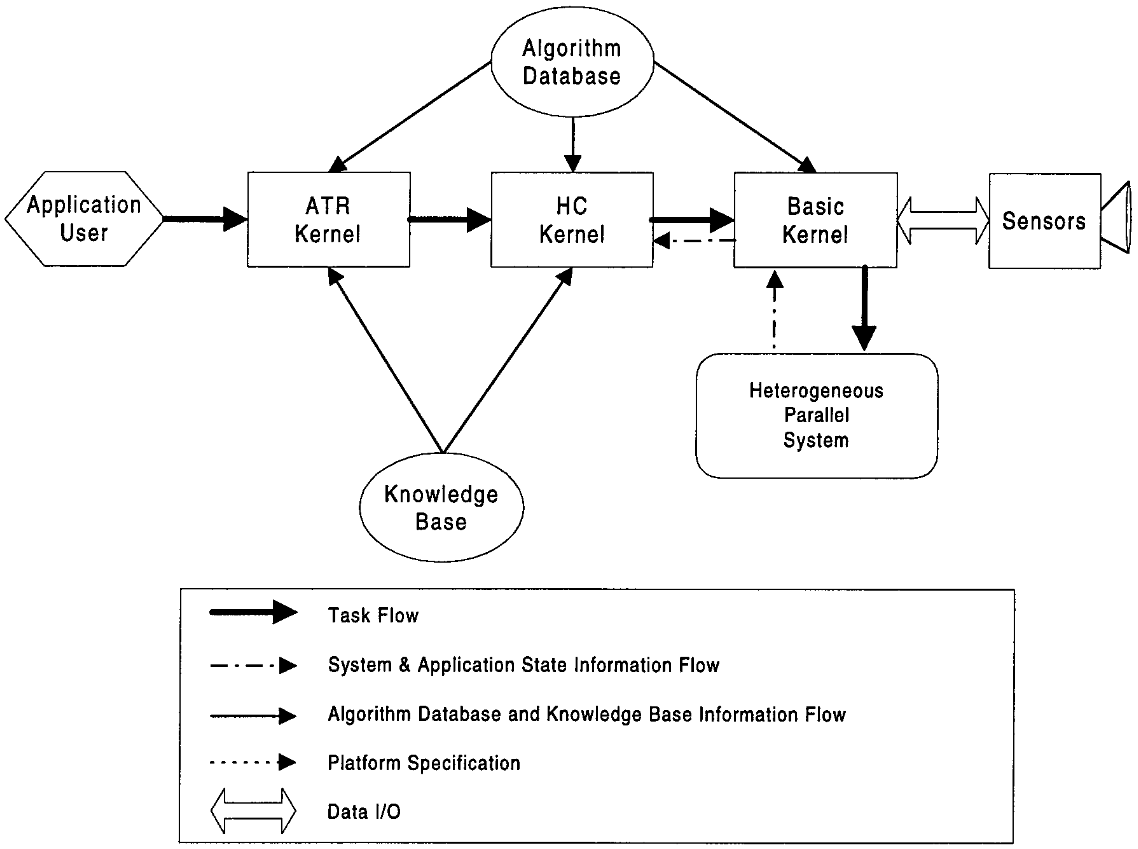

Algorithm Database
Application User
ATR Kernel
HC Kernel
Basic Kernel
Sensors
Heterogeneous Parallel System
Knowledge Base
Task Flow
System & Application State Information Flow
Algorithm Database and Knowledge Base Information Flow
Platform Specification
Data I/O

Figure 1: IOS Conceptual Design

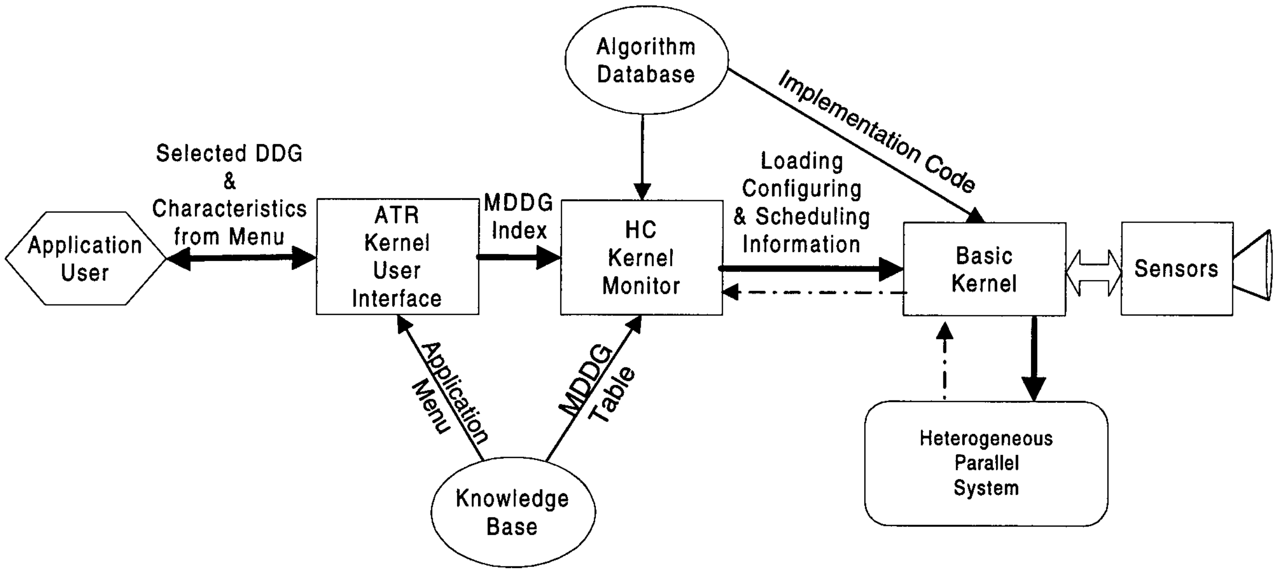


Figure 2: IOS On-line Components

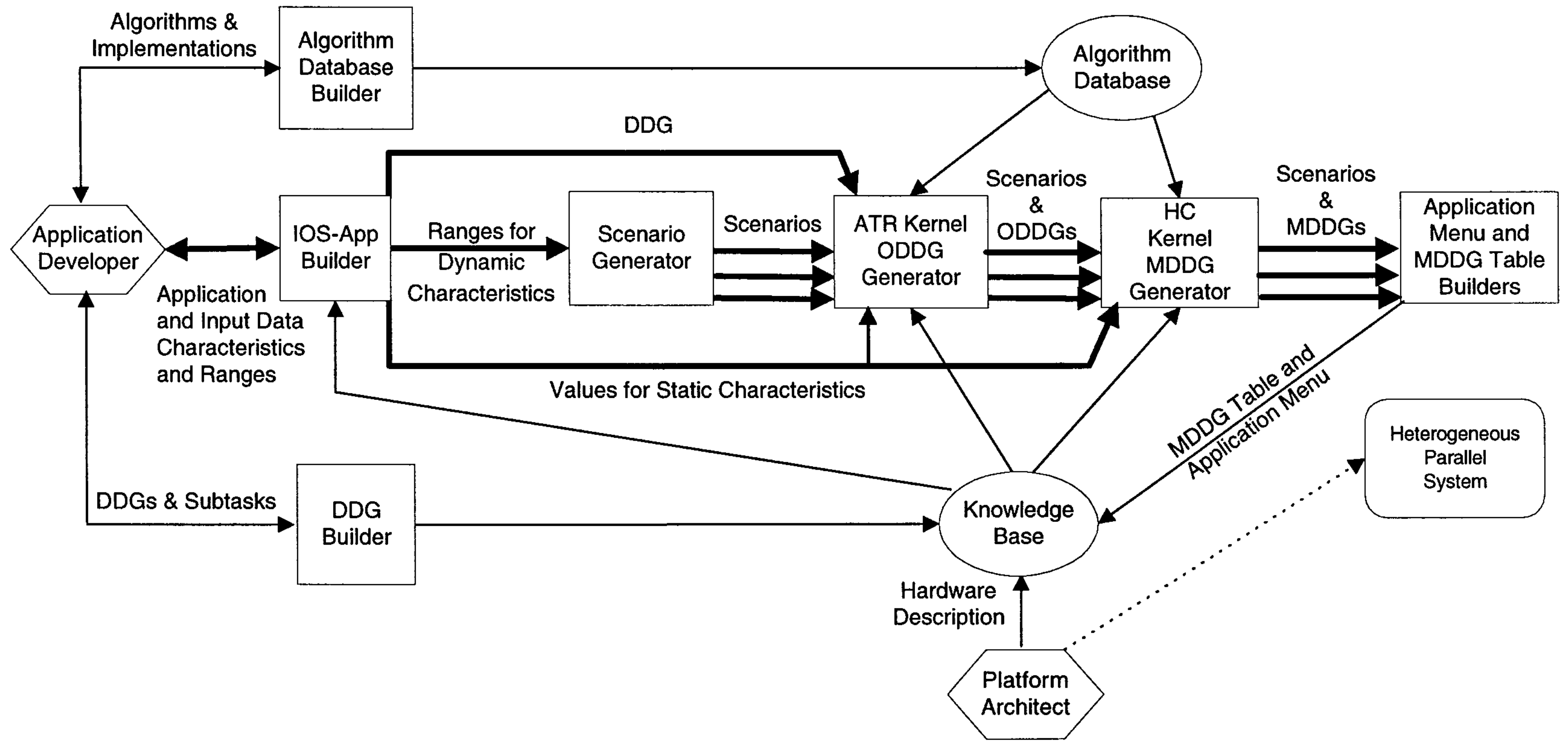

Algorithms & Implementations
Algorithm Database Builder
Algorithm Database
DDG
Application Developer
IOS-App Builder
Ranges for Dynamic Characteristics
Scenario Generator
Scenarios
ATR Kernel ODDG Generator
Scenarios & ODDGs
HC Kernel MDDG Generator
Scenarios & MDDGs
Application Menu and MDDG Table Builders
Application and Input Data Characteristics and Ranges
Values for Static Characteristics
MDDG Table and Application Menu
Heterogeneous Parallel System
DDGs & Subtasks
DDG Builder
Knowledge Base
Hardware Description
Platform Architect

Figure 3: IOS Off-line Components

Case Study

Distributed Interactive Simulation for Synthetic Forces

Paul Messina, Sharon Brunett, Dan Davis, Tom Gottschalk
California Institute of Technology, Pasadena, CA, USA

David Curkendall, Laura Ekroot, Herb Siegel
Jet Propulsion Laboratory, Pasadena, CA, USA

Distributed Interactive Simulation for Synthetic Forces *

P. Messina, S. Brunett, D. Davis, T. Gottschalk
Center for Advanced Computing Research
California Institute of Technology
Pasadena, California 91125

D. Curkendall, L. Ekroot, and H. Siegel
The ALPHA Group
Jet Propulsion Laboratory
Pasadena, California 91109

Abstract

Interactive simulation of battles is a valuable tool for training. The behavior and movement of hundreds or thousands of entities (tanks, trucks, airplanes, missiles, etc.) is currently simulated using dozens or more workstations on geographically distributed LANs connected by WANs. The simulated entities can move, fire weapons, receive "radio" messages, etc. The terrain that they traverse may change dynamically, for example due to rains turning dirt roads into mud or bombs forming craters. Thus the entities need to receive frequent information about the state of the terrain and the location and state of other entities. Typically, information is updated several times a second. As the number of simulated entities grows, the number of messages that need to be sent per unit of time can grow to unmanageable numbers. One approach to reducing the number of messages is to keep track of what entities need to know about which other entities and only send information to the entities that need to know. For example, tanks in Germany need not know about a change of course of a ship in the Pacific. This technique for reducing messages is known as interest management.

Caltech and its Jet Propulsion Laboratory have implemented a simulation of this type on several large-scale parallel computers, exploiting both the compute power and the fast messaging fabric of such systems. The application is implemented using a heterogeneous approach. Some nodes are used to simulate entities, some to manage a database of terrain information, some to provide interest management functions, and some to route messages to the entities that do need to receive the information. Some of these tasks require more memory than others, some require faster processing capability. Thus the application is heterogeneous both in its functional decomposition and to a smaller extent in the characteristics of the hardware that is used to run each function. In addition, workstations are used to run the Graphical User Interface (GUI) that is used to control the simulation and to visualize the simulation as it is running. This approach has been used to run an exercise with over twice the previous record number of vehicles simulated.

A near-term goal is to simulate 50,000 entities. To do so, it will be necessary to run the simulation on several geographically distributed SPPs. For pragmatic reasons (availability of sufficiently large systems), the machines employed will have different architectures.

1 Introduction

Simulation of synthetic environments and activities for training of military personnel is routinely carried out on distributed, homogeneous computing assets. Caltech has undertaken a project whose goal is to increase substantially the size and fidelity of these simulations. Our approach of using large-scale parallel computers has led to a heterogeneous computing strategy. This paper describes our software architecture, our motivation for using a heterogeneous approach, and preliminary experience with the implementation of the simulation program on parallel systems.

2 Background

The United States Department of Defense has found it increasingly useful to train individuals and commands using simulated environments. These simulations have become more realistic and effective with the advent of computer-generated scenarios, visualizations, and battlefield entity behaviors. Of particular importance has been the development and use of Distributed Interactive Simulation (DIS). A large implementation of the DIS was conducted by several units located in Europe in November of 1994. It was called Synthetic Theater of War—Europe (STOW-E). It combined the classic manned simulator entities (as originally developed under SIMNET) with Modular SemiAutomated Forces (ModSAF) simulation soft-

*Support for this research was supplied by the Information Technology Office, DARPA, with contract and technical monitoring via Naval Research and Development Laboratory (NRaD)

0-8186-7879-8/97 $10.00 © 1997 IEEE

ware executing on networks of workstations; the individual ethernet networks were themselves interconnected by Wide Area Network (WAN) links. The total number of simulators and ModSAF entities used in this exercise was about 2,000. Stimulated in part by this successful exercise, current simulation initiatives have vehicle count goals in the 10,000–50,000 range. A vehicle is defined in the military argot to be any substantial entity—ground, air vehicles, autonomous personnel, etc. In addition to a desire to simulate more entities, the trainers and the trainees are constantly asking for more resolution, faster refresh rates, higher fidelity, more automatic behaviors, increased training environment responsiveness, and overall improvements in the training environment. Finally, there is the emergent realization that faster than real-time analytic simulations will be required in the future to support the operational use of simulations in the battlefield itself. This latter capability is essential if the simulation software is to be used for planning as opposed to training. It should be noted that this class of simulation has applications in other fields, for example for emergency response to natural disasters.

These demands for increased capability and capacity lead one naturally to consider devising a software architecture and computer platform strategy that will support a wide range of requirements. In other words, a scalable approach is needed.

Caltech and its Jet Propulsion Laboratory (JPL) have a long history of using parallel computer architectures for scalability of scientific and engineering simulations, including discrete event simulations. In addition, in the CASA gigabit testbed project [1], we performed experiments with distributed, heterogeneous implementations of several applications executing on parallel supercomputers connected by a high-speed wide-area network. Hence when we decided to tackle the challenge of supporting more ambitious simulations, we quickly decided to apply the large-scale capabilities of High Performance Computing and Communications (HPCC) assets as an alternative to WAN-linked sub networks of workstations in order to develop and demonstrate the software architectures needed to reach these goals.

The Caltech/JPL project, called Synthetic Forces Express (SF Express), selected ModSAF as the base software to enhance and use to carry out scaling experiments. The SF Express project has a two-year goal to achieve a 50,000 vehicle count simulation via:

The efficient operation of the ModSAF software on individual, large, SPP platforms and, the networking of two or more of these large platforms together as a single metacomputer for the largest runs. These WAN's will include connectivity to more conventional ModSAF assets of workstations and simulators.

At present, the SF Express Team has pilot versions of its emerging software architecture operational on Intel Paragon platforms at Caltech and Oak Ridge National Laboratory (ORNL) and on IBM SP2 systems at Caltech and Ames Research Center (ARC). Use of the much larger SP2 at the Cornell Theory Center's SP2 is about to begin. Efforts are also underway to port the SF Express software to the CRAY T3D and T3E class of machines.

At this writing a full 10,000 vehicle scenario, approximately twice the size achieved previously, has been demonstrated on several occasions using the 1,024-node ORNL machine. Indeed, one of these demonstrations took place live during Supercomputing '96 from the floor of the Pittsburgh Convention Center. Software adapted to the SP2 has achieved runs of up to 8,000 vehicles on the 143-node SP2 at ARC.

To date, these simulations have been run using scenarios created by NRAD and executed using the simulated ground environment of that of the Fort Knox Terrain Database. Larger scenarios—up to 50,000 vehicles—are actively being constructed, this time on the much larger playing field afforded by Southwest USA Terrain Database (SWUSA), centered near 29 Palms and spanning much of the surrounding territory of Southern California.

Based on measured performance of our variant of the ModSAF code, we have determined that no single available SPP can execute the full 50,000 vehicle scenario; indeed, the near term 50,000 goal was selected in part so as to require the involvement of two or more supercomputers. Accordingly, our SPP architecture includes provisions for networking several large SPPs together, creating a meta-supercomputing network.

In what follows, we discuss some of the key architectural concepts being explored to make ModSAF suitable for SPP machines and to improve its overall scalability. While ModSAF is the basis for all of our current work, we intend that the applicability of this research to be much broader. ModSAF, then, is the current focus serving both as a convenient tool and as a familiar yardstick for measuring progress familiar to a large community.

3 Interest Management

We take as axiomatic that to enable dramatic scalability of entity level simulations, "interest management" must be central to the software architecture.

Using the language of ModSAF, beyond a certain (rather small) limit, it is necessary to abandon broadcast style inter-entity messaging schemes and insert rather precise interest management techniques. This arises because of two separate but related notions:

An entity's behavior is shaped partly by an awareness of other entities around it (local perceived ground truth). Since not all entities of interest are computed by the same local CPU, the need arises for "remote entities" to signal their presence and activities to that local CPU via messaging. But if each individual CPU attempts to deal with all of these incoming messages (global ground truth), all CPU's will be overwhelmed both in memory and in performing bookkeeping duties. Interest management must be performed more globally to permit scalability.

As the number of entities increases, an all to all protocol eventually overwhelms the physical SPP messaging fabric. The same conclusion is obtained: a global interest management scheme is critical.

Accordingly, the SF Express Team has been experimenting with two variants of global interest management: one a server based notion and a second router based scheme.

Space does not permit their detailed exposition here [2] but the main ideas are easily grasped. See Figure 1.

In Figure 1, the top squares represent nodes executing the ModSAF entity behavior codes known as SAFSIMs. As part of this behavior, each vehicle asserts its interest in what in effect are "regions of interest spaces." There are several of these—e.g., a high and a low resolution terrain space, vehicle i.d., signal frequency—but to grasp the basic ideas it suffices to consider interest to be a function of geographic location. In the server interest management scheme, this interest is registered in one of the interest management nodes, nodes which themselves are decomposed over the index of that interest space. Messages (known as PDUs in ModSAF) generated by any vehicle are sent (registered) to the coordinate of that interest space corresponding to the coordinate of the sending vehicle. For example, if a PDU is sent from a vehicle whose location is (x, y), it is sent to the (x, y) coordinate of the Interest Management (IM) nodes. The IM node then forwards the message back to each SAFSIM that has registered an interest in that coordinate.

Looking at the process from the point of view of the IM nodes, each maintains queues of messages to be sent to each SAFSIM, looping over all SAFSIMS, and sending a single bundled message for each traversal of that loop. In this relatively straightforward manner, messages arrive at only the SAFSIMs that have explicitly asserted interest. The remote entities represented at each SAFSIM node and the volume of individual PDUs processed are thus kept to a minimum.

In this IM scheme, communications channels are associated with interest classes, and a single simulator node will generally exchange data with more than one IM node. In the alternative Router model, each simulator node has a single communications channel to the "outside world."

The basic building block of the Router architecture is a fixed collection of SAFSIM nodes associated with a Primary Router, as seen in the bottom of Figure 2. The SAFSIM nodes send data and interest declarations up to their associated Primary Routers, and only the appropriate, interest selected data flow back down. Data communications among the (SAFSIMs+Primary) building blocks are accomplished through additional layers of data collection and data distribution router nodes shown in the top part of Figure 2. Communications within the upper layers occur in parallel with those in the Primary↔SAFSIM layer. This means that there are no significant additional time costs for data messages which take the longer (5 hop) path through the full communications network.

The use of (few) fixed communications channels in the Router architecture allows extremely efficient bundling of data messages. During the communications-intensive initialization phases of ModSAF, individual messages flowing down to the SAFSIM nodes routinely contain 40 or more PDUs, and total data rates through the Primary Routers in excess of 16K PDU/second have been observed. Once initializations are completed, the "steady-state" Primary↔SAFSIM communications account for only about 3% of a SAFSIM's (wall clock) time.

A system-wide evolving picture of interest declarations and payloads can be obtained from the Router architecture. Tracing performance and program behavior, along with general purpose logging capabilities, are facilitated by the very nature of the Router clusters.

4 Functional Decomposition

Vanilla ModSAF normally executes completely within a single workstation, replicating workstations until enough are employed to execute the desired size of the simulation. There are two basic modules in ModSAF: the SAFSIM, already identified, and the GUI which is only activated on a workstation if it is desired to input to the scenario or observe the simulation's progress. In building SF Express, we have already migrated some of the sub elements away from

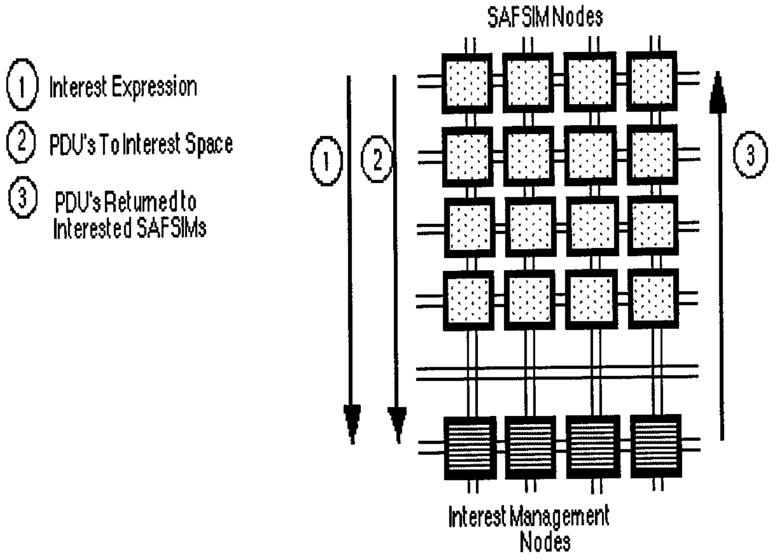

SAFSIM Nodes
1 Interest Expression
2 PDU's To Interest Space
3 PDU's Returned to Interested SAFSIMs
1
2
3
Interest Management Nodes

Figure 1: Interest Management Server

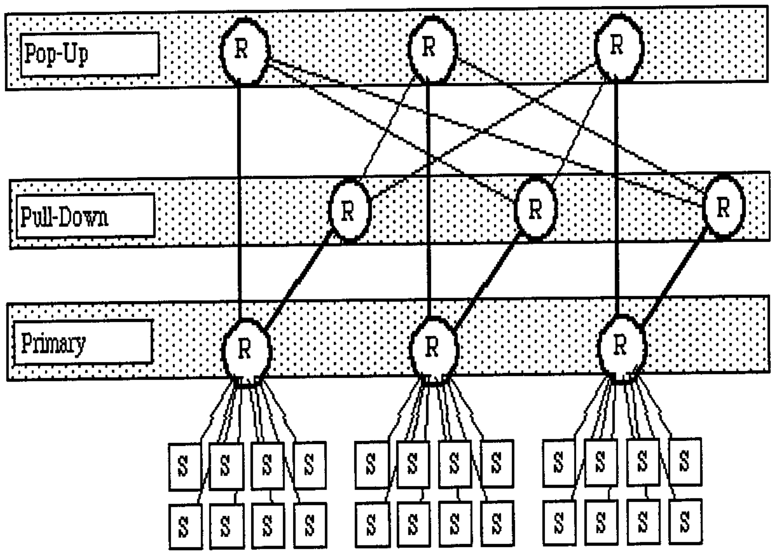


Figure 2: Router-Based Interest Management

the SAFSIM and are planning to migrate others. In addition, we add others, such as the interest management just discussed, as separate and new functions not present in vanilla ModSAF.

Functional decomposition is natural in the quest for scalability. When a resource (such as a terrain data base) must be accessed simultaneously by hundreds or thousands of processors, one replicates it. If the data base is large, computers with large memories should be used. If the computational cost of simulating a complex type of vehicle is high, one spins off that task to separate nodes; if to achieve fidelity the simulation requires a lot of floating-point computation, nodes with suitable CPUs should be chosen. Router nodes on the other hand will do few if any floating-point computations to carry out their role; routers can therefore be hosted on systems that excel at logic and integer operations. Data logging for subsequent replay of the simulation might require processors with ample attached disk storage.

The need to keep up with real time also dictates a functional decomposition. Furthermore, in some simulations sensor data from real instruments must be read and processed to guide parts of the simulation. Visualization of the ongoing simulation is essential and it also requires a different type of computer resource.

An indication of how this approach is used in practice can be gleaned from our experience with a 10,174 Vehicle synthetic forces simulation that was run by on the Oak Ridge National Lab 1024-node Intel Paragon. The run, approximately twice as large as the largest previous such simulation, utilized a scenario set on the Ft. Knox Compact Terrain Data Base, with "blue and red" forces made up of battle tanks, fighting vehicles, armored personnel carriers and trucks. This run employed 784 Paragon processors, of which 640 were devoted to simulating vehicles, 48 processors acted as routers in a communications network that provides the good scalability demonstrated; 90 processors were used as terrain data base servers; and six processors were used as servers to load the program and data. In addition, a GUI proxy node was used, as is described in the next section.

In short, heterogeneous functional decomposition is a natural strategy for coping with the evolving needs of synthetic forces simulations.

4.1 Graphics user interface and visualization

We have experimented with a number of approaches to providing GUI functionality. The most straightforward method on the SP2 is simply to take advantage of its X Windowing system and devote one or more nodes hosting a complete ModSAF with an X Window output being sent to a remote workstation. This is an attractive option, particularly when it is desired to interact with the simulation during its progress: e.g., vehicles can be created and instantiated on the GUI node as in workstation based ModSAF, a function not otherwise readily available with the SPPs. It is also easy to "interest manage" the display, by attaching the GUI node directly to the Interest Management nodes. Interest is geographically expressed by turning the screen display corner coordinates into an interest expression. PDUs only from vehicles within the covered region will be transmitted to the GUI node, a key circumscription if that node is not to be overwhelmed with irrelevant information.

A second technique removes the GUI from the SPP entirely, substituting there instead a GUI Proxy, and executing a workstation GUI as a stand-alone unit on the outside. This workstation then transmits interest declarations to the Proxy, which in turn interfaces with the interest management machinery in a manner similar to a SAFSIM. This technique is less demanding of connection bandwidth but sacrifices some of the portability of the X Windows approach.

A third approach, and one which ultimately may prove more powerful, is to send the PDUs themselves out of the SPP to external devices. These data can be compressed and limited in various ways, but current experience indicates that the entire PDU stream can be issued by the SPP and assimilated by a high performance workstation in real time. A current experiment [3] describes progress in processing the PDU stream on external devices either for more scalable real time display or for after action analysis. The post processing can subdivide the PDU stream, redirecting the PDUs to multiple processes and to, for example, a matrix of coordinated screens, giving an overall view of the battlefield.

4.2 Replacing routine disk access

Frequent retrieval of data from disk storage is too slow to be practical on the SPPs. Instead, reader files common to all SAFSIMs are held in RAM in one or more file server nodes. Supplying each SAFSIM with its required information then takes place at RAM access and SPP messaging rates, greatly reducing initialization time. We are currently experimenting with compiling these reader files into binary prior to any single simulation. This compacts the files and further speeds up their delivery to the individual SAFSIM nodes.

The simulation terrain in ModSAF is represented through a fairly elaborate, memory-efficient scheme built from small terrain elements ("pages" and "patches"). Arbitrarily large terrains are supported through a caching scheme in which a SAFSIM maintains only a modest fraction of the full terrain in memory, requesting new pages and patches as they are needed.

In the parallel implementation, the disk-read data retrievals of conventional ModSAF are replaced by message exchanges with database server partitions. Each partition consists of a sufficient number of nodes to hold the entire terrain database in memory. Multiple replicas of the database partition are used for runs with large numbers of SAFSIM nodes.

4.3 Some future possibilities

While not currently implemented, the above terrain serving scheme is consistent with ultimately providing for dynamic terrain. Since only a few terrain servers are needed, it is practical to keep these synchronously updated with terrain changes and, via cache coherence methods, ensure that the SAFSIMs receive cached updates as well.

In the future we expect to migrate more functionality away from the individual SAFSIMs. Terrain reasoning is a good candidate. High level and complex functions such as path planning are currently handled within the SAFSIMs on a lower priority basis than the fundamental activity loops. The computation takes many cycles to complete and its performance is hard to predict. Migrating that function to the terrain server nodes has great appeal.

It may even be helpful to migrate lower level functions like intervisibility calculations there as well. In workstation based ModSAF many intervisibility calculations are unnecessarily duplicated. Vehicle A calculates its visibility to remote vehicle B, while in B's local workstation, the reciprocal calculation is being made to its remote vehicle A. Doing this calculation once in a server can gain important economies.

Finally, decomposing the ModSAF functionalities and switching to a server perspective paves the way for higher fidelity reasoning and environmental calculations, since more CPU power can be deployed to any one function when it is needed without interfering with the tightly controlled and repetitive tasks within each SAFSIM.

5 SPP Portability

SF Express has been built around MPI messaging libraries, a necessary but by no means sufficient condition to ensure portability. Machines that have been addressed so far with various degrees of completeness are:

- Intel Paragon
- IBM SP2
- Cray T3D
- SGI Origin 2000
- SGI Power Onyx/Challenge Series
- Beowulf

The codes on the Paragon and SP2 are by far the most mature. The major difficulty encountered with the Paragon was the reversed endianess as compared to all other machines on the list, save Beowulf. The port to the SP2 was smooth and uneventful. Unfortunately the Cray T3D has proved the most difficult of all, almost entirely because of the lack of a 32 bit Cray C compiler. ModSAF was definitely not written with portability to 64 bit machines in mind. Our current approach is to work with the AC compiler authored by Bill Carlson and available on both the T3D and the T3E. Success here would give the Project access to this important class of machines.

An informal port to the SGI Origin 2000 was performed and demonstrated during the Supercomputing '96 Convention in Pittsburgh. The Power Onyx/Challenge Series of machines are listed, even though they are shared memory machines, because they offer an MPI library. The shared memory machines, then, emulate the message passing architectures and the SF Express concepts port without difficulty. Since ModSAF itself is native to the SGI's, the port was uneventful.

A Beowulf "pile of PCs" cluster, has been built by the California Institute of Technology and the Jet Propulsion Laboratory. The cluster consists of 16 Intel Pentium Pro (200MHz) processors running Parallel Linux connected via a 100Mb/sec ethernet switch. Out of the box ModSAF has been ported to Beowulf. We are experimenting various MPI extensions and profiling libraries to maximize efficiency and properly characterize the performance of the SF Express port.This kind of cluster shows very good price-performance ratios and may be a viable platform for future uses of SF Express.

In summary, we are pleased with the considerable—but incomplete—progress made towards our portability goals. We believe that offering options to be an important aspect of enabling the continuing applicability of this research.

6 Interoperability and Metasupercomputing

Implementing SF Express on multiple machines is additionally important to achieving the project goal of 50,000 entities. As mentioned in the introduction, no single SPP is likely to be able to achieve this goal and it will be necessary to utilize two or more SPPs together connected by wide area networks to achieve this result.

Fortunately, the essential information that needs to be shared among the participating SPPs is exchanged using ModSAF PDUs and their data structures were designed to interoperate with different machines. Endianess and machine word lengths will not pose difficult problems.

Also, the key to scalability is once again, precise interest management. And this can be accomplished between SPPs as an extension of the interest schemes already described.

In an unconstrained world, a uniform messaging structure would be established across the whole meta-supercomputer and the structures we have been discussing would need no modifications at all—a node on a distant machine would be different only in that it had a unique node identification. Unfortunately, this would require the WAN network to be as high in bandwidth and message handling capabilities as the SPP messaging fabrics themselves. Since we will attempt the metacomputing runs with at best OC-3 networks, an approach more parsimonious of bandwidth resources is required.

Referring to Figure 2, one can think of the interface between the geographically-distributed SPPs as being done by connecting the Pop-Up routers with WAN connections. The time delays for PDUs sent through the upper router layer are modest (e.g., less than 50 msec) and thus likely to be small compared to the delays introduced by WAN access.

This approach has not been fully implemented but its broad outlines are clear. To establish a global interest manager, each SPP would need to create periodically (once every ~1–5 sec) a complete interest expression across the entire range of interest coordinates. The remote SPP returns only the PDUs responsive to those interests.

7 Conclusions and Plans for '97

At this writing, the project is consolidating the progress made thus far which culminated in the 10,000 and 8,000 vehicle runs at ORNL and ARC respectively. Implementations are being cleaned up and more comprehensive attention paid to instrumentation and measurement.

Near term developments include the design of the meta-supercomputing interfaces to enable the employment of two or more SPPs in a single exercise.

In addition, little attention has been paid thus far to how to make the large simulations thus enabled available to conventional ModSAF cluster workstation networks and simulators. In the sense that everyone speaks DIS protocol, the interface is easy and assured. But once again, interest management must be enabled as a two way interface between the parties, else the workstations will be overwhelmed and the influence of the entities modeled within the conventional workstations will not be properly represented to the SF Express Forces within the SPP. There are several choices available; perhaps the best is to treat the SF Express as an HLA federate and implement a standard HLA/RTI interface to the outside world.

We are being asked to reach the 50,000 goal this year and in pursuit of this are setting up the necessary cooperations between several major national SPP assets. In addition to the assets at JPL/CIT, we are enlisting support in pursuit of the meta-supercomputing goals from ORNL, ARC, CTC, and the San Diego Supercomputing Center (SDSC).

Acknowledgments

Access to various computational facilities was essential to the work performed. The Intel Paragon and SP2 at Caltech were made available by the Caltech Center for Advanced Computing Research and the Oak Ridge Intel Paragon by the Oak Ridge Center for Computational Sciences. The Cray T3D at JPL and the JPL Dual SGI Power Onyx Visualization Laboratory were made available by the JPL/Caltech Supercomputing Project. Finally, access to the IBM SP2 at ARC was provided by the Numerical Aerodynamic Simulation Systems Division at NASA Ames Research Center.

References

[1] Messina, P., and Mihaly-Pauna, T., "CASA Gigabit Network Testbed: Final Report." California Institute of Technology Technical Report CACR-123, July 1996.

[2] Craymer, L. and Lawson, C. "A Scalable, RTI Compatible Interest Manager for Parallel Processors." 1997 Spring Simulation Interoperability Workshop.

[3] Plesea, L. and Ekroot, L., "Data Logging and Visualization of Large Scale Simulations (SF Express)." 1997 Spring Simulation Interoperability Workshop.

Sharon Brunett is a computing analyst for Caltech's Center for Advanced Computing Research. She is currently involved with optimization and characterization of applications on shared and distributed memory

MPPs. She is also involved with modifying application programs to exploit new systems software and language features. Current interests include analyzing the various I/O requirements and interfaces necessary for applications communicating across multiple MPPs. She received her B.S. in Computer Science and Applied Mathematics from the University of California Riverside in 1983.

David W. Curkendall received a BS in electrical engineering from Cornell University and a PhD in engineering and science from University of California, Los Angeles. He is manager of the Advanced Parallel Processing program at the Jet Propulsion Laboratory. His early technical specialization were in interplanetary navigation and the development of radiometric and optical tracking system technology. For the past several years, he has been active in all areas of computational science, including the leadership of the technology teams that developed the early Hypercube Concurrent Computers, their software and applications. He is currently active in several applications of MPP machines including parallel rendering, military simulation, and leads a NASA multidisciplinary Grand Challenge Science team in spaceborne Synthetic Aperture Radar processing.

Dan Davis received a B.A. in Quantitative Psychology from the University of Colorado in Boulder in 1973. He entered graduate study there in Business Administration, but transferred to the University of Colorado School of Law. He received his Juris Doctors degree in December of 1975 and was admitted to the bar of the Supreme Court of Colorado and the U.S. Supreme Court, among others. He is currently the Assistant Director of The Center for Advanced Computing Research at Caltech. He is currently involved in implementing large scale simulations on Scalable Parallel Processing computers. These simulations range from military battlefields to VLSI designs. His computer research activities have run from his early work in computational analyses for the behavioral sciences at C.U., Boulder through defense work in the U.S. Navy, in which he holds a reserve commission as a Commander, Cryptologic Specialty.

Laura L. Ekroot received the B.S. degree in electrical engineering from the California Institute of Technology, Pasadena, in 1986, and the M.S. and Ph.D. degrees in electrical engineering from Stanford University, Stanford, CA, in 1988 and 1991 respectively. She is currently working as a Member of the Technical Staff at the Jet Propulsion Laboratory in Pasadena, CA. Her research interests include Shannon theory, coding theory, image compression, and parallel simulation.

Tom Gottschalk is a Lecturer in Physics and a Member of the Professional Staff at the California Institute of Technology. In 1978 he received a Ph.D. in Theoretical Physics from the University of Wisconsin. In 1974, he received a BS. in Astrophysics from Michigan State University. He is currently researching new architectures and algorithms for parallel processing of various simulations. His work is directed at applying the power of large computers, capable of computing in the range of tens of GigaFLOPS. In the electronics area, he has developed a new tool for analyzing VLSI design verification on very large circuits, VLSIs of 5 million transistors or more. In the defense area, he has worked on battlefield simulators and missile defense tracking algorithms. His early work was in simulating sub-atomic particles using Monte Carlo models. He is a frequent speaker on the subject of computer and internet usage at primary and secondary schools.

Paul Messina is Assistant Vice President for Scientific Computing at Caltech, Faculty Associate in Scientific Computing, Director of Caltech's Center for Advanced Computing Research, and serves on the executive committee of the Center for Research on Parallel Computation. His recent interests focus on advanced computer architectures, especially their application to large-scale computations in science and engineering. He also is interested in high-speed networks and computer performance evaluation. He heads the Scalable I/O Initiative, a multi-institution, multi-agency project aimed at making I/O scalable for high-performance computing environments. Messina has a joint appointment at the Jet Propulsion Laboratory as Manager of High-Performance Computing. Messina received his PhD in mathematics in 1972 and his MS in applied mathematics in 1967, both from the University of Cincinnati, and his BA in mathematics in 1965 from the College of Wooster.

Herb Siegel received a B.A. in mathematics from the University of California at Berkeley in 1962 and an MA in mathematics from the California State University at Los Angeles in 1965. His early work at the Jet Propulsion Laboratory included the calibration of interplanetary radiometric range and development of high performance software correlators for VLBI. He led the development of a commercial multiprocessor for Action Computer Enterprise. Currently he is the computational leader for a NASA Grand Challenge effort in high performance computing and Earth-Space science to processes interferometric SAR over high performance networks and using national assets of MPP machines.

Session 3

Mapping and Scheduling Algorithms

Session Chair

Ranga S. Ramanujan
Architecture Technology Corporation,
Minneapolis, MN, USA

A Stochastic Model of a Dedicated Heterogeneous Computing System for Establishing a Greedy Approach to Developing Data Relocation Heuristics

Min Tan and *Howard Jay Siegel*

Parallel Processing Laboratory
School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN 47907-1285, USA
{mtan, hj}@ecn.purdue.edu

Abstract

In a dedicated mixed-machine heterogeneous computing (HC) system, an application program may be decomposed into subtasks, then each subtask assigned to the machine where it is best suited for execution. Subtask data relocation is defined as selecting the sources for their needed data items. This study focuses on theoretical issues for data relocation using a stochastic HC model. It is assumed that multiple independent subtasks of an application program can be executed concurrently on different machines whenever possible. A stochastic model for HC is proposed, in which the computation times of subtasks and communication times for inter-machine data transfers can be random variables. The optimization problem for finding the optimal matching, scheduling, and data relocation schemes to minimize the total execution time of an application program is defined based on this stochastic HC model. The optimization criteria and search space for the above optimization problem are described. It is proven that a greedy algorithm based approach will generate the optimal data relocation scheme with respect to any fixed matching and scheduling schemes. This result indicates that a greedy algorithm based approach is the best strategy for developing data relocation heuristics in practice.

This research was supported in part by NRaD under contract number N66001-96-M-2277.

Keywords: data relocation, greedy algorithm, heterogeneous computing, mapping, matching, optimization, scheduling, stochastic modeling.

1: Introduction

A single application program often requires many different types of computation that result in different needs for machine capabilities. Heterogeneous computing (HC) is the effective use of the diverse hardware and software components in a heterogeneous suite of machines connected by a high-speed network to meet the varied computational requirements of a given application [8]. One goal of HC is to decompose an application program into subtasks, each of which is computationally homogeneous, and then assign each subtask to the machine where it is best suited for execution.

Subtask matching, scheduling, and data relocation are three critical steps for implementing an HC application on an HC system. Matching involves assigning subtasks to machines. Scheduling includes ordering the execution of the subtasks assigned to each machine and ordering the inter-machine communication steps for data transfers. Data relocation is the scheme for selecting the sources for needed data items. This study focuses on theoretical issues for data relocation using a stochastic HC model. It is assumed that multiple independent subtasks of an application program can be executed concurrently on different machines whenever possible (e.g.,

0-8186-7879-8/97 $10.00 © 1997 IEEE

when the machines are available for subtask execution).

The contribution of this paper can be summarized as follows. A general stochastic HC model is proposed, in which the computation times of subtasks and communication times for inter-machine data transfers are random variables. The optimization problem for finding the optimal matching, scheduling, and data relocation schemes to minimize the total execution time of an application program executed in a dedicated HC system is defined based on this proposed stochastic HC model. The optimization criterion and search space for the above optimization problem in HC are described. It is proven that a greedy algorithm based approach will generate the optimal data relocation scheme with respect to any fixed matching and scheduling schemes. This result indicates that a greedy algorithm based approach is the best strategy for developing data relocation heuristics in practice.

The inter-machine communication time between subtasks can be substantial and is one of the major factors that degrade the performance of an HC system. This paper focuses on potential methods for minimizing the inter-machine communication time of an application program when the concurrent execution of different subtasks on different machines is considered whenever possible. In particular, the impact of the data relocation scheme on the total execution time of the subtasks executed in a dedicated HC system is examined.

In most of the mathematical models for HC in the literature (e.g., [5, 9]), the computation times and inter-machine data transfer times of data items for different subtasks in the application program are assumed to be deterministic quantities. This is valid when the inter-machine network is completely controlled by the scheduler and all execution times and inter-machine communication needs are known *a priori* (not dependent on input data). However, there are elements of uncertainty (e.g., input data dependent looping and conditional constructs) that impact the deterministic nature of both the computation and inter-machine communication times for different subtasks. Such uncertainties can create others, e.g., network contention among different inter-machine data transfer steps. They are unpredictable prior to execution time. An approach to modeling these computation and communication times is to represent them as random variables with assumed probability distribution functions.

To use a dedicated HC system to execute an application program efficiently, the optimization problem of using matching, scheduling, and data relocation schemes to minimize the total execution time must be defined. Section 2 provides the background and terminology needed for the rest of this paper. In Section 3, a stochastic HC model for matching, scheduling, and data relocation is introduced. A topological sort based procedure is presented in Section 4 for defining the execution time of an application program executed in a dedicated HC system where the execution of the subtasks is partially ordered, and when matching, scheduling, and data relocation schemes are known. In Section 5, a method is devised to enumerate all the valid options in choosing the data relocation scheme for a given arbitrary matching. Thus, Sections 3, 4, and 5 collectively define the above optimization problem in HC with a stochastic model. Because of the complexity of this defined optimization problem in HC, guidelines for devising heuristics must be provided. It is proven in Section 6 that a greedy algorithm based approach will generate the optimal data relocation scheme with respect to any fixed matching and scheduling schemes. This result indicates that a greedy algorithm based approach is the best strategy for developing data relocation heuristics in practice.

Most of the literature for HC has concentrated on addressing the practical aspects and heuristics for matching and scheduling. This paper emphasizes instead the theoretical issues involved in data relocation using a stochastic HC model. The practical implication on data relocation heuristic design of the theoretical result derived is explained.

2: Background and terminology

The material in this subsection is summarized from [9]. It provides the background and terminology needed for the rest of this paper. In general, the goal for HC is to assign each subtask to one of the machines in the system such that the total execution time (computation time and inter-machine communication time) of the application program is minimized [3]. The subtask to machine assignment problem is referred to as matching in HC. When a subset of subtasks can be executed in any order, varying the order of the computation of these subtasks (while maintaining the data dependencies

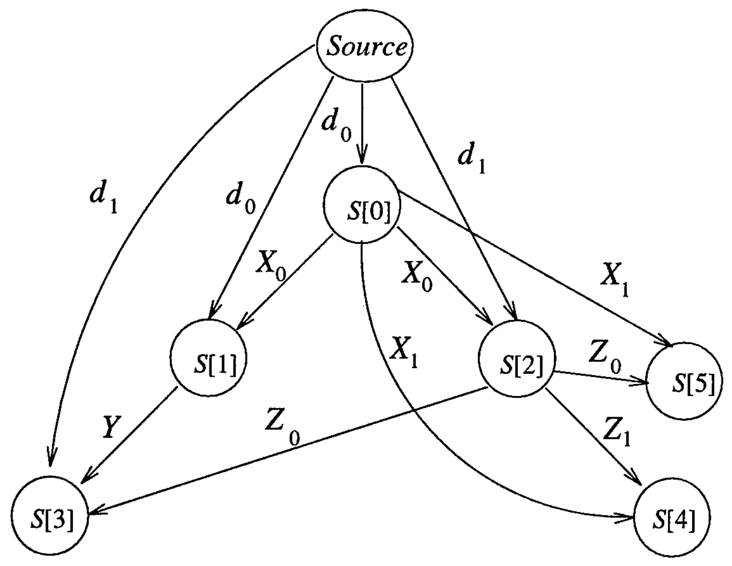


Figure 1: Subtask flow graph for the example application program.

among all subtasks) can impact the total execution time of the application program. Determining the order of computation for the subtasks is referred to as scheduling in this paper. In most of the literature for HC, a subtask flow graph is used to describe the data dependencies among subtasks in an application program (e.g., [5, 9]). In Figure 1, each vertex of the subtask flow graph represents a subtask. Let $S[k]$ denote the k-th subtask. For each data element that $S[k]$ transfers to $S[j]$ during execution, there is an edge from $S[k]$ to $S[j]$ labeled with the corresponding variable name. An extra vertex labeled *Source* denotes the locations where the initial data elements of the program are stored.

Let a data item be a block of information that can be transferred between subtasks. Using information from the subtask flow graph, a data item is denoted by the two-tuple (s, d), where $s \geq 0$ is the number of the subtask that generates the needed value of variable d upon completion of computation of that subtask. If the needed value of d is an initial data element to the program, then $s = -1$. Two data items are the same if and only if they are both associated with the same variable name in an application program and the corresponding value of the data is generated by the same subtask (which implies that the two data items have the same value).

In general, most of the graph-based algorithms for matching-related problems assume that the pattern of data transfers among subtasks is known *a priori* and can be illustrated using a subtask flow graph (e.g., [5, 10]). Thus, no matter which machine is used for executing each subtask of a specific application program, the locations (subtasks) from which each subtask obtains its corresponding input data items are determined by the subtask flow graph and are independent of any particular matching scheme between machines and subtasks.

The above assumption generally needs refinement in the case of HC. In [9], two data-distribution situations, namely data locality and multiple data-copies, are identified for addressing refinements of the above assumption. It is assumed that each subtask $S[i]$ keeps a copy of each of its individual input data items and output data items on the machine to which $S[i]$ is assigned by the matching scheme. Furthermore, it is also assumed that all input data items are received for a subtask prior to that subtask's computation.

Data locality arises when two subtasks, $S[j]$ and $S[k]$ that are assigned to the same machine, need the same data item e from $S[i]$ (assigned to a different machine). Because a machine can fetch a data item from its local storage faster than fetching it from other machines, if $S[j]$ is executed after $S[k]$, then $S[j]$ should obtain e locally from $S[k]$ instead of from the machine assigned to $S[i]$. If a subtask flow graph is used to compute inter-subtask communication cost, then without considering machine assignments, the impact of data locality might be ignored.

The multiple data–copies situation arises when two subtasks, $S[j]$ and $S[k]$, need the same data item e from $S[i]$, where $S[i]$, $S[j]$, and $S[k]$ are assigned to three different machines. If $S[k]$ is executed after $S[j]$ obtains e, then the machine assigned to $S[k]$ can get data item e from either the machine assigned to $S[i]$ or the machine assigned to $S[j]$. The choice that results in the shorter time should be selected. Selecting the sources for needed data items is referred to as data relocation (because the data relocation scheme determines the source machines from which the data items will be relocated to the destination machines). In general, when using information only from the subtask flow graph, the possibility of having multiple sources for a needed data item is not considered. Data locality can be viewed as a special case of having multiple data copies (i.e., one copy is on the machine to which the receiving subtask is assigned by the matching scheme).

In [9], it is assumed that, at any instant in time during the execution of an application program, only one computation or inter-machine data transfer step for a specific subtask is being executed. Based on this assumption, a minimum spanning tree based algorithm is presented in [9] that finds, for a given matching, the optimal scheduling scheme for inter-machine data transfer steps and the optimal data relocation scheme for each subtask. Data locality and multiple data-copies are all considered in the above algorithm. The mathematical model for HC presented in this paper differs from the one in [9] in that the possible concurrent execution of both the computation and inter-machine communication steps of different subtasks in an application program is considered. Also, the computation times of subtasks and communication times for inter-machine data transfers can be random variables. It is proven in this paper that a greedy algorithm based approach will generate the optimal data relocation scheme with respect to any fixed matching and scheduling schemes. This result indicates that a greedy algorithm based approach is the best strategy for developing data relocation heuristics in practice and attempts to solve a much more general problem in HC than addressed in [9].

3: A stochastic model for matching, scheduling, and data relocation in HC

A stochastic model of matching, scheduling, and data relocation for HC is formalized in this section. This model is an extension of the one presented in [9]. The possible concurrent execution of both the computation of subtasks and inter-machine communication steps in an application program is considered. The issues related to using a stochastic HC model are addressed. When the computation time of each subtask on each machine and the communication times of transferring data items have stochastic properties, those timing parameters must be modeled as random variables. This paper examines an underlying theoretical issue with respect to data relocation. Due to the theoretical nature of the proof of the main result in this paper, it is not necessary to know the actual distribution functions of those random variables. The mathematical model presented in this section allows the material in the rest of this paper to be given in unambiguous terms.

(1) An application program $\underline{P}$ is composed of a set of $\underline{n}$ subtasks

$$\underline{S} = \{S[0], S[1], ..., S[n-1]\}.$$

There are a set of $\underline{Q}$ initial data elements

$$\{d_0, d_1, ..., d_{Q-1}\}.$$

(2) The set of $\underline{NI[i]}$ input data items required by $S[i]$ is

$$\underline{I[i]} = \{Id[i, 0], Id[i, 1], ..., Id[i, NI[i]-1]\},$$

and the set of $\underline{NG[i]}$ output data items generated by $S[i]$ is

$$\underline{G[i]} = \{Gd[i, 0], Gd[i, 1], ..., Gd[i, NG[i]-1]\}.$$

The program structure of P is specified by a subtask flow graph.

In this paper, the subtask flow graph of any application program P is assumed to be acyclic. A cycle in a graph represents a loop containing one or more subtasks. With the presence of looping constructs, an appropriate statistical approach can be used to determine the distribution for the number of iterations each looping construct will execute and the maximum number of iterations each looping construct has [10]. Then, the existent subtask flow graph can be transformed into an acyclic one by unrolling each looping construct with the known or estimated maximum number of iterations. The above approach potentially will increase the number of subtasks present in the acyclic subtask flow graph significantly. Also, the distribution for the number of iterations each looping construct will execute and the maximum number of iterations each looping construct has can be difficult to estimate in reality. A possibly more practical approach is to group a fixed number of consecutive iterations of the unrolled looping constructs together to decrease the number of subtasks present. Another approach is to view each looping construct as part of a single subtask and the boundaries for decomposing an application program into subtasks are not allowed to be in the middle of a looping construct.

(3) An HC system consists of a heterogeneous suite of $\underline{m}$ machines

$$\underline{M} = \{M[0], M[1], ..., M[m-1]\},$$

M includes the devices where all the initial data elements are stored before the execution of P.

(4) There is a computation matrix $\underline{C} = \{C[i, j]\}$, where $\underline{C[i, j]}$ denotes the computation time of $S[i]$ on machine $M[j]$ (e.g. [4]). For the reason stated in Section 1, $C[i, j]$ is assumed to be a random variable with a known distribution. It can be computed from empirical information or by applying two characterization techniques in HC, namely task profiling and analytical benchmarking (see [8] for a survey of these techniques). In [7], a methodology is introduced for estimating the distribution of execution time for a given data parallel program that is to be executed on a single hybrid SIMD/SPMD mixed-mode machine. However, as mentioned earlier, for the results mentioned here, it is not necessary to determine the distribution functions for the random variables.

(5) An assignment (matching) function $\underline{Af}$: $S \rightarrow M$ is such that if $Af(i) = j$, then $S[i]$ is assigned to be executed on machine $M[j]$. $\underline{NS[j]}$ is defined as the number of subtasks assigned to be executed on machine $M[j]$. Thus, $\sum_{j=0}^{m-1} NS[j] = n$.

(6) A scheduling function $\underline{Sf}$ indicates the execution order of a subtask with respect to the other subtasks assigned to the same machine. If $Sf(i) = k$, then $S[i]$ is the k-th subtask whose computation is executed on machine $M[Af(i)]$, where $0 \le k < NS[Af(i)]$. Readers should notice that the scheduling function Sf schedules only the order of the computation for different subtasks (*not* the order for executing the inter-machine communication steps).

(7) The set of data–source functions is

$$\underline{DS} = \{DS[0], DS[1], ..., DS[n-1]\},$$

where $DS[i](j) = [k_1, k_2]$ ($0 \le i < n$, $0 \le j < NI[i]$, $0 \le k_1 < n$, and $0 \le k_2 < m$) means that $S[i]$ obtains the input data item $Id[i, j]$ from $S[k_1]$ and $k_2 = Af(k_1)$. If $DS[i](j) = [k_1, k_2]$ and $k_1 = -1$, then $Id[i, j] = (-1, d_x)$ and $S[i]$ obtains the associated initial data element from machine $M[k_2]$ where d_x is initially stored. Readers should notice that, when $k_1 \ne -1$, the augmented information k_2 can be obtained with the known Af and is redundant. But the information from k_2 is necessary to specify the source of an initial data element when $k_1 = -1$. The above definition of DS gives a unified way of specifying the values of a data-source function. For different assignment and scheduling functions, with consideration of the impact of data locality and multiple data-copies, there are different choices for sets of the data-source functions. This choice of DS corresponds to the data relocation problem discussed in Section 2.

It is assumed that each subtask $S[i]$ will submit a copy of its input data item $Id[i, j]$ to the network for forwarding to other destination machines (based on DS) immediately after $Id[i, j]$ is available on machine $M[Af(i)]$. Each subtask will also submit copies of all of its output data items to the network to be transferred to the proper destination machines (based on DS) after the completion of its entire computation. Thus, Af, Sf, and DS together completely specify the computation and inter-machine communication steps needed at any time to execute the application program P in a dedicated HC system.

(8) The communication time estimator $\underline{D[s, r, e]}$ denotes the length of the communication time interval between the time when a data item e is available on $M[s]$ and the time when e is obtained by $M[r]$ (assuming this transfer is required for the given Af, Sf, and DS). For the reason stated in Section 1, $D[s,r, e]$ is assumed to be a random variable (again recall that the distribution of this random variable is not needed to derive the results of this paper). $D[s, r, e]$ includes all the various hardware and software related times of the inter-machine communication process (e.g., network latency and the time for data format conversion between $M[s]$ and $M[r]$ when necessary).

Most of the literature for HC (e.g., [4, 9]) assumes that the inter-machine communication time for sending a data item e from $M[s]$ to $M[r]$ is only a function of s, r, and e. But in reality, even in a dedicated HC system, when an application program is executed, the traffic pattern for inter-machine communication can be impacted by subtask computation and other inter-machine communication times that are all input data dependent (and represented as random variables). The choice of Af, Sf, and DS impacts all of these computation and communication times and, hence, the communication time interval between the time when e is available on $M[s]$ and the time when e is obtained by $M[r]$. Thus, the communication time estimator $D[s, r, e]$ is dependent on Af, Sf, DS, s, r, and e.

In general, it will be extremely difficult (if not impossible) to estimate the distribution function of $D[s, r, e]$ as a function of Af, Sf, DS, s, r, and e. The purpose of defining $D[s, r, e]$ here is to address the factors that impact the inter-machine communication times for the application programs executed in a dedicated HC system. It also helps to establish a theoretical model for defining the optimization criterion of the optimization problem for HC. With this well-defined theoretical model and optimization criterion, the greedy algorithm based approach introduced in Section 6 can provide potential data relocation heuristics with a sound local optimization strategy based on a solid theoretical derivation. Future data relocation heuristics can follow the local optimization strategy in Section 6 to achieve a reasonable level of global optimization without the information about the exact distribution function of $D[s, r, e]$.

4: A topological sort based algorithm for calculating the execution time of an application program in an HC system

For a given computation matrix C and communication time estimator $D[s, r, e]$, the total execution time of the application program P associated with an assignment function Af, a scheduling function Sf, and a set of data-source functions DS is defined by the following procedure. A data relocation graph (denoted as $\underline{Gr}$) corresponding to a particular Af, Sf, and DS is generated using the steps specified below. When the impact of data locality and multiple data-copies is considered, the concept of a valid set of data-source functions DS of the application program P can be defined according to the properties of Gr. There may be many valid sets for P, each corresponding to a unique graph for P, and each resulting in possibly different execution time of P. An invalid DS would correspond to a set of data-source functions that does not result in an operational program.

The steps for constructing Gr are as follows.

Step 1: A *Source* vertex is generated that represents the locations of all the initial data elements (which may be on different machines).

Step 2: For each $S[i]$, $NI[i] + 1$ vertices are created, one for each of the $NI[i]$ input data items and one for all of the generated output data items of $S[i]$. These are the set of input data vertices, labeled $\underline{V[i, j]}$ $(0 \le j < NI[i])$ and the output data vertex $\underline{V_g[i]}$. $V[i, j]$ represents the operation for subtask $S[i]$ to receive its j-th input data item. $V_g[i]$ represents the computation for $S[i]$ to generate all of its output data items. $\underline{V}$ is a set that contains all of the above vertices associated with the application program P in Steps 1 and 2. Each $V[i, j]$ is associated with a weight zero and each $V_g[i]$ is associated with a weight $C[i, Af(i)]$, the computation time of subtask $S[i]$ on the machine assigned by the assignment function Af.

Step 3: For any input data vertex $V[i_1, j_1]$, suppose that $DS[i_1](j_1) = [i_2, k_2]$ where $-1 \le i_2 < n$ and $0 \le k_2 < m$, and if $0 \le i_2 < n$, then $k_2 = Af(i_2)$.

Case A: $S[i_1]$ obtains its required input-data item $Id[i_1, j_1]$ by copying it from the *Source* vertex if $Id[i_1, j_1] = (-1, d_k)$ and d_k is one of the initial data elements.

If $i_2 = -1$, then there exists k $(0 \le k < Q)$, such that $Id[i_1, j_1] = (-1, d_k)$, and a directed edge with weight $D[k_2, Af(i_1), Id[i_1, j_1]]$ is added from the *Source* vertex to $V[i_1, j_1]$ (recall that $DS[i_1](j_1) = [i_2, k_2]$ implies that d_k is received from machine $M[k_2]$). That is, if subtask $S[i_1]$'s j_1-th input data item $Id[i_1, j_1]$ is one of the initial data elements and is obtained from one of the initial locations where d_k is stored before program execution, then add an edge from the *Source* vertex to $V[i_1, j_1]$ whose weight is the communication time interval needed to transfer that initial data element from the initial location $M[k_2]$ where it is stored to the machine assigned to $S[i_1]$.

Case B: $S[i_1]$ obtains its required input-data item $Id[i_1, j_1]$ by copying it from the subtask that generates $Id[i_1, j_1]$.
If $0 \le i_2 < n$ and there is j_2, such that $Id[i_1, j_1] = Gd[i_2, j_2]$, then a directed edge with weight $D[k_2, Af(i_1), Id[i_1, j_1]]$ is added from $V_g[i_2]$ to $V[i_1, j_1]$. That is, if subtask $S[i_1]$'s j_1-th input data item $Id[i_1, j_1]$ is subtask $S[i_2]$'s j_2-th output data item $Gd[i_2, j_2]$, then add an edge from $V_g[i_2]$ to $V[i_1, j_1]$ whose weight is the communication time interval needed to transfer that data item from $M[k_2]$ to the machine assigned to $S[i_1]$.

Case C: $S[i_1]$ obtains its required input-data item $Id[i_1, j_1]$ by copying it from one of the other subtasks that have obtained that input-data item already.
If $0 \le i_2 < n$, and there is a j_2, such that $Id[i_1, j_1] = Id[i_2, j_2]$, then a directed edge with weight $D[k_2, Af(i_1), Id[i_1, j_1]]$ is added from $V[i_2, j_2]$ to $V[i_1, j_1]$. That is, if subtask $S[i_1]$'s j_1-th input data item $Id[i_1, j_1]$ is obtained by copying subtask $S[i_2]$'s j_2-th input data item $Id[i_2, j_2]$, then add an edge from $V[i_2, j_2]$ to $V[i_1, j_1]$ whose weight is the communication time interval needed to transfer that data item from $M[k_2]$ to the machine assigned to $S[i_1]$.

For any input data vertex $V[i_1, j_1]$ ($0 \le i_1 < n$ and $0 \le j_1 < NI[i_1]$) for a given DS, one and only one case of A, B, or C can occur. Thus, any vertex $V[i_1, j_1]$ has one and only one parent vertex, which is specified by the given DS. Also, the weight of the edge between $V[i_1, j_1]$ and its unique parent vertex is the communication time interval needed for $S[i_1]$ to obtain $Id[i_1, j_1]$ from its source with respect to the given *Af*, *Sf*, and *DS*.

Step 4: For every $0 \le i < n$, a directed edge with weight zero is added from $V[i, j]$ to $V_g[i]$ ($0 \le j < NI[i]$).

If the *Gr* generated above is an acyclic graph, then the corresponding *DS* is defined as a valid set of data–source functions for the application program *P*. If the graph had a cycle, then deadlock would arise in the application program *P*, which makes *P* unschedulable. Readers should notice that the weight of each edge or vertex depends on *Af*, *Sf*, and *DS*. The validity of a particular *DS* is based on the subtask flow graph and is independent of the underlying *Af* and *Sf* for generating the specific *Gr*. For the rest of this paper, only valid sets of data-source functions will be considered.

Step 5: For each i_1 and i_2 ($0 \le i_1 < n$ and $0 \le i_2 < n$), if $Af(i_1) = Af(i_2)$ and $Sf(i_1) = Sf(i_2) - 1$ (i.e., $S[i_1]$ and $S[i_2]$ are assigned to the same machine and $S[i_1]$ is executed immediately before $S[i_2]$), a directed edge with weight zero is added from $V_g[i_1]$ to $V_g[i_2]$. The extended graph based on *Gr* and *Sf* after this step is defined as the execution graph *Ex* of *P*. If the generated execution graph *Ex* is acyclic, then the corresponding scheduling function generates an operational program and is defined as a valid scheduling function. For the rest of this paper, only valid scheduling functions will be considered.

Step 6: Each vertex *v* of *Ex* is associated with a starting time *ST(v)* and a finishing time *FT(v)* (*ST(v)* and *FT(v)* are random variables). From the definitions in Steps 4 and 5, the execution graph *Ex* generated is acyclic. Thus, there exists a topological sort [2] of the vertices in *V*. Set *ST(Source)* = 0. *W(v)* is the weight of *v* (recall that each $V[i, j]$ is associated with a weight zero and each $V_g[i]$ is associated with a weight $C[i, Af(i)]$). Suppose that v_i is one of the immediate predecessors of v, $W(v_i, v)$ is the weight of the direct edge from v_i to v. Then $ST(v)$ and $FT(v)$ can be derived inductively one by one in the order specified by the topological sort according to the following formula:

$$ST(v) = \max_i \{FT(v_i) + W(v_i, v)\} \qquad (1)$$

$$FT(v) = ST(v) + W(v). \qquad (2)$$

Step 7: The total execution time of the application program *P* associated with an assignment function *Af*, a valid scheduling function *Sf*, and a valid set of data-source functions *DS* is defined by the following formula:

$$\text{Execution\_time}_P(Af, Sf, DS) = \max_{v \in V}\{FT(v)\}. \qquad (3)$$

Suppose that $E\{x\}$ denotes the expected value of a random variable x. The objective of matching, scheduling, and data relocation for HC is to find an assignment function Af^*, a valid scheduling function Sf^*, and a valid set of data-source functions DS^*, such that
$E\{\text{Execution\_time}_P(Af^*, Sf^*, DS^*)\} =$

$$\min_{Af,Sf,DS} \mathrm{E}\{\mathrm{Execution\_time}_P(Af, Sf, DS)\}. \quad (4)$$

Thus, the minimization of the expected value of the total execution time of an application program is the optimization criterion of the optimization problem for HC described in Section 1 with respect to the stochastic model defined in Section 3.

It is assumed in this mathematical model that, if there is no data dependency between two subtasks $S[i]$ and $S[j]$, and they are assigned to be executed on two different machines by the assignment function Af, then $S[i]$ and $S[j]$ can be executed concurrently. Furthermore, the inter-machine communication step for one subtask to obtain one of its input data items can be overlapped with (a) inter-machine communication step(s) to obtain its other input data item(s), (b) the inter-machine communication steps of other subtasks to obtain their input data items, and (c) the computation steps of other subtasks. The distribution of each random variable $D[s, r, e]$ indicates any time delay resulting from network or machine I/O conflicts.

As stated in Section 3, it is extremely difficult to obtain the exact distribution of $D[s, r, e]$. The purpose of the above topological sort based procedure is not for calculating Execution_time$_P(Af, Sf, DS)$ in practice due to this difficulty. Rather it is to define the optimization criterion theoretically for the optimization problem of HC. The theorem presented in Section 6 is based on this defined Execution_time$_P(Af, Sf, DS)$ with a known Af, Sf, and DS and provides a practical local optimization strategy for future data relocation heuristics.

5: A procedure for enumerating the valid options in choosing data relocation schemes

In this subsection, a procedure for enumerating all the valid options available in choosing data relocation schemes (with respect to an arbitrary matching) is described for subtask flow graphs without data-dependent conditional and looping constructs. Due to space limitations, the case of having data-dependent conditional and looping constructs inside the subtask flow graph is not described in this paper. When data-dependent conditional and looping constructs are present inside the subtask flow graph, the same procedure presented in this section can be modified to enumerate the valid options in choosing data relocation schemes as well. The material presented in this section defines the search space for the optimization problem of HC mentioned in Section 1. This defined search space also helps future data relocation heuristic developers to know all the valid options in choosing a data relocation scheme.

A directed graph $\underline{Dg[Af]}$ corresponding to a specific assignment function Af can be generated by connecting the vertices in V as follows (recall that V is a set that contains all the vertices generated for any specific application program P according to Steps 1 and 2 described in Section 4):

(a) For every i_1, j_1, i_2, and j_2, where $0 \le i_1 < n$, $0 \le i_2 < n$, $0 \le j_1 < NI[i_1]$, $0 \le j_2 < NI[i_2]$, and $i_1 \ne i_2$, such that $Id[i_1, j_1] = Id[i_2, j_2] = e$, a directed edge from $V[i_1, j_1]$ to $V[i_2, j_2]$ and a directed edge from $V[i_2, j_2]$ to $V[i_1, j_1]$ are added.

(b) For every i_1, j_1, i_2, and j_2, where $0 \le i_1 < n$, $0 \le i_2 < n$, $0 \le j_1 < NG[i_1]$, and $0 \le j_2 < NI[i_2]$, such that $Gd[i_1, j_1] = Id[i_2, j_2] = e$, a directed edge from $V_g[i_1]$ to $V[i_2, j_2]$ is added.

After (a) and (b), each generated data item $Gd[i_1, j_1]$ of P corresponds to a fully connected graph of the set of vertices $\underline{VG\ [i_1, j_1]} = \{V[i_2, j_2] \mid Gd[i_1, j_1] = Id[i_2, j_2], 0 \le i_2 < n, 0 \le j_2 < NI[i_2]\}$. This corresponds to the set of input data vertices that need the generated data item $Gd[i_1, j_1]$. Also, $V_g[i_1]$ is connected unidirectionally (i.e., $V_g[i_1]$ is the starting point of each directed edge) to all the vertices in $VG[i_1, j_1]$.

(c) For every i, j, and k, such that $Id[i, j] = (-1, d_k)$, where $0 \le i < n$, $0 \le j < NI[i]$, and $0 \le k < Q$, a directed edge from the *Source* vertex to $V[i, j]$ is added.

After (c), each initial data item $(-1, d_k)$ $(0 \le k < Q)$ of P corresponds to a fully connected graph of the set of vertices $\underline{VI[k]} = \{V[i, j] \mid Id[i, j] = (-1, d_k)\}$ (i.e., the input data vertices that need the initial data element d_k). There is also a directed edge from the *Source* vertex to each vertex in $VI[k]$. All the edges generated in (a), (b), and (c) are called fetch edges.

Figure 2 illustrates components of $Dg[Af]$ for the subtask flow graph shown in Figure 1 (d_0 is an initial data element and X_0 and Z_0 are generated data items). The notation relevant to d_0, X_0, and Z_0 is as follows.

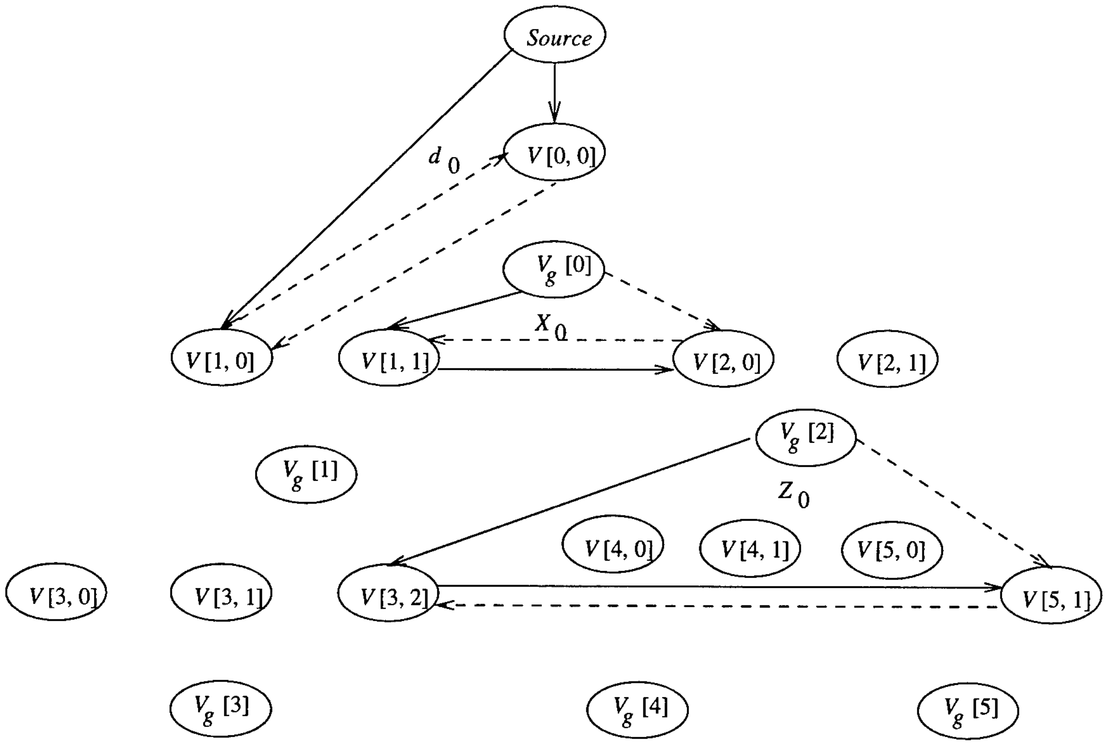


Figure 2: The d_0, X_0, and Z_0 components of $Dg[Af]$, based on the subtask flow graph in Figure 1.

$S[0]$: $NI[0] = 1$, $Id[0, 0] = (-1, d_0)$; $NG[0] = 2$, $Gd[0, 0] = (0, X_0)$.
$S[1]$: $NI[1] = 2$, $Id[1, 0] = (-1, d_0)$, $Id[1, 1] = (0, X_0)$.
$S[2]$: $NI[2] = 2$, $Id[2, 0] = (0, X_0)$; $NG[2] = 2$, $Gd[2, 0] = (2, Z_0)$.
$S[3]$: $NI[3] = 3$, $Id[3, 2] = (2, Z_0)$.
$S[5]$: $NI[5] = 2$, $Id[5, 1] = (2, Z_0)$; and $NG[5] = 0$.

Suppose that the assignment function Af for this current example is defined such that: $Af(0) = 1$, $Af(1) = 2$, $Af(2) = 2$, $Af(3) = 1$, $Af(4) = 3$, and $Af(5) = 0$. After applying above Steps (a), (b), and (c), the edges (both solid and dashed lines) of $Dg[Af]$ in Figure 2 are fetch edges corresponding to the initial data elements d_0 and the generated data items X_0 and Z_0.

A directed graph $Dg[Af]$ can be generated by knowing only P and Af. After generating $Dg[Af]$, any algorithm for enumerating the spanning trees of a directed graph [2] can be applied to the subgraphs of $Dg[Af]$ for (1) the set of vertices $\{V_g[i]\} \cup VG[i, j]$ $(0 \leq i < n$ and $0 \leq j < NG[i])$ and (2) the set of vertices $\{Source\} \cup VI[k]$ $(0 \leq k < Q)$. The roots of all possible spanning trees are $V_g[i]$ $(0 \leq i < n)$ or the *Source* vertex, respectively. Each spanning tree corresponding to the set of vertices $\{V_g[i]\} \cup VG[i, j]$ specifies a valid data relocation scheme for the generated data item $Gd[i, j]$. Because the *Source* vertex can denote multiple locations where each initial data element d_k is stored before the execution of P, each spanning tree corresponding to the set of vertices $\{Source\} \cup VI[k]$ can specify a suite of valid data relocation schemes for the initial data element d_k. In the above generated spanning trees, if the parent vertex of $V[i_1, j_1]$ is $V[i_2, j_2]$ or $V_g[i_2]$, then $DS[i_1](j_1) = [i_2, Af(i_2)]$; and if the parent vertex of $V[i_1, j_1]$ is the *Source* vertex, then $DS[i_1](j_1) = [-1, q]$, where $M[q]$ is one of the initial locations of the corresponding initial data element. The solid lines in Figure 2 illustrate one spanning tree for each of d_0, X_0, and Z_0, respectively.

6: A greedy algorithm based approach to developing data relocation heuristics

In this section, a greedy algorithm based approach to developing data relocation heuristics is presented. This greedy strategy is established based on the mathematical model, optimization criterion, and search space described in Sections 3, 4, and 5, respectively, for the optimization problem in HC. Choosing *Af*, *Sf*, and *DS* to minimize the expected value of the total execution time based on a stochastic HC model is a complex optimization problem. However, developing heuristics to find suboptimal *Af*, *Sf*, and *DS* is necessary to use HC systems efficiently.

A greedy algorithm based approach to developing data relocation heuristics is to find a data relocation scheme DS^*, such that for the same *Af* and *Sf*, each subtask can obtain its individual input data item as early as possible in terms of its expected receiving time. Readers should recall that, based on a stochastic HC model, the receiving time of each input data item of a subtask is a random variable. In general, it is difficult to compare two random variables without referring to a particular statistic (e.g., the expected value). The following theorem shows that, if the expected receiving time for each input data item of a subtask can be minimized, then the expected time when each subtask has received all of its input data items can be minimized based on the assumptions of the distributions of those receiving times stated later. This conclusion demonstrated by the following theorem is not obvious because the expected value of the maximum of a set of random variables is *not* necessarily equal to the maximum of the corresponding expected values of the same set of random variables.

Suppose that $\underline{rt(V[i, j])}$ and $\underline{rt'(V[i, j])}$ are the random variables that specify the receiving times of input data item $Id[i, j]$ for subtask $S[i]$, corresponding to two different data relocation schemes *DS* and DS' for the same *Af* and *Sf*. The following assumptions about $rt(V[i, j])$ and $rt'(V[i, j])$ are made:

(1) $rt(V[i, j]) + k$ and $rt'(V[i, j]) + k'$ for a fixed i and j (where k and k' are arbitrary constants) belong to the same two-parameter family of random variables [1] such that their probability distribution functions can be completely determined by their corresponding means and variances. Most of the common families of distributions for random variables, such as normal distribution, Gamma distribution, and Beta distribution, have this property.

(2) The variance of $rt(V[i, j])$ is equal to the variance of $rt'(V[i, j])$ for fixed i and j.

(3) For any data relocation scheme *DS*, $rt(V[i, j_1]) + c_1$ is independent of $rt(V[i, j_2]) + c_2$ ($j_1 \neq j_2$ and c_1 and c_2 are arbitrary constants).

Readers should notice that assumptions (1), (2), and (3) are all related to the statistical properties of $rt(V[i, j])$ and $rt'(V[i, j])$. As long as they are approximately satisfied in reality, the theorem that follows based on those assumptions still has practical as well as theoretical significance. The following discussion provides the rationale behind the above three assumptions. For assumptions (1) and (2), because $rt(V[i, j])$ and $rt'(V[i, j])$ are two random variables for specifying the receiving times of the *same* data item (i.e., $Id[i, j]$) for $S[i]$ corresponding to two different data relocation schemes and the same *Af* and *Sf*, it is quite reasonable to assume that they have certain similar statistical properties (e.g., their variances, their families of distribution). For assumption (3), although $rt(V[i, j_1])$ and $rt(V[i, j_2])$ are defined for two different data items, but if the inter-machine data transfer steps for $Id[i, j_1]$ and $Id[i, j_2]$ will impact each other or those two data items are generated by the same subtask, their corresponding receiving times by $S[i]$ can be correlated to each other. However, conditions exist under which the random variables can be treated as independent with each other despite this type of correlation. The Kleinrock independence approximation for a data network in which there are many interacting transmission queues [6] is a well-known condition for describing this situation. This Kleinrock independence approximation is used here as the basis for assuming independence between $rt(V[i, j]) + c_1$ and $rt'(V[i, j]) + c_2$ that may technically be correlated.

Theorem: For two different data relocation schemes *DS* and DS', with the same *Af* and *Sf*, and a fixed i ($0 \leq i < n$), suppose $\underline{X_j} = rt(V[i, j])$, $\underline{Y_j} = rt'(V[i, j])$, $\underline{X} = \max_j[X_j]$, and $\underline{Y} = \max_j[Y_j]$ ($0 \leq j < NI[i]$), where X and Y are random variables for specifying the times when $S[i]$ receives all of its input data items with respect to *DS* and DS'. If $E\{X_j\} \leq E\{Y_j\}$ for $0 \leq j < NI[i]$, then $E\{X\} \leq E\{Y\}$.

<u>Proof</u>: Suppose that the distribution function of a random variable w is F_w. Because $E\{X_j\} \leq E\{Y_j\}$ for all j, there exists $c_j \geq 0$, such that $E\{X_j\} + c_j = E\{Y_j\}$. It is true that

$$E\{X_j + c_j\} = E\{Y_j\}$$

and

$$\mathrm{Var}\{X_j + c_j\} = \mathrm{Var}\{X_j\} = \mathrm{Var}\{Y_j\}$$

due to assumption (2). Then, because of assumption (1),

$$F_{X_j + c_j} = F_{Y_j}.$$

From assumption (3),

$$\{X_j + c_j \mid 0 \le j < NI[i]\}$$

is a set of independent random variables and

$$\{Y_j \mid 0 \le j < NI[i]\}$$

is another set of independent random variables. With the properties associated with the ''max'' operator over multiple independent random variables [1], it can be shown that

$$F_{\max\{X_j + c_j\}} = \prod_{j=0}^{NI[i]-1} F_{X_j + c_j} = \prod_{j=0}^{NI[i]-1} F_{Y_j} = F_{\max\{Y_j\}}.$$

Therefore,

$$\mathrm{E}\{Y\} = \mathrm{E}\{\max_j[Y_j]\} = \mathrm{E}\{\max_j[X_j + c_j]\}.$$

Without loss of generality, suppose $c_0 = \min_j[c_j]$, then

$$\begin{aligned} \mathrm{E}\{Y\} &= \mathrm{E}\{\max_j[X_j + c_j]\} \\ &\ge \mathrm{E}\{\max_j[X_j + c_0]\} \\ &\ge \mathrm{E}\{\max_j[X_j]\} \\ &= \mathrm{E}\{X\}. \end{aligned}$$

Thus, $\mathrm{E}\{X\} \le \mathrm{E}\{Y\}$.

Based on the above theorem, the greedy algorithm based approach that finds DS^* to minimize $\mathrm{E}\{rt(V[i, j])\}$ for $S[i]$ to obtain $Id[i, j]$ with respect to the same Af and Sf for all $0 \le i < n$ can also minimize the expected time when each subtask receives all of its input data items (i.e., $\mathrm{E}\{X\}$) and is ready for its computation. The exact starting time and the cost of the computation for $S[i]$ (i.e., $ST(V_g[i])$ and $C[i, Af(i)]$) depend on the choice of Af and Sf. But with respect to the same given Af and Sf, DS^* is the optimal data relocation scheme that minimizes the expected value of the probability distribution of the execution time (as defined in Section 4).

The significance of the above theorem is that it shows a greedy algorithm based approach is the best for data relocation heuristics. Based on the above conclusion, in order to minimize the expected total execution time of an application program executed in a dedicated HC system, data relocation heuristics should select the source for each input data item of $S[i]$, among all the valid options described in Section 5, such that its receiving time by $S[i]$ is as small as possible. Referring to the above theorem, for a specific subtask, there exists a DS that is better than all other DS'. But the inter-machine communication steps specified by the selected DS for one subtask may impact the expected receiving time of input data items for other subtasks. Thus, the DS^* that minimizes $E\{rt(V[i, j])\}$ for every $S[i]$ may be hard to find or may not exist. Trade-offs must be made to choose a suboptimal data relocation scheme, such that more input data items can be obtained by more subtasks as quickly as possible.

6: Summary

In an HC system, the subtasks of an application program P must be assigned to a suite of heterogeneous machines (the matching problem) and ordered (the scheduling problem) to utilize computational resources effectively. The matching and scheduling solutions presented in the literature, in general, concentrate on decreasing the computation time of P. The inter-machine communication time of P is impacted by the scheme for distributing the initial data elements and the generated data items of P to different subtasks (the data relocation problem).

The inter-machine communication time in an HC system can have a significant impact on overall system performance, so that any technique that can be used to reduce this time is important. This paper focused on the data relocation scheme to decrease the inter-machine communication time for given matching and scheduling schemes, when the possible concurrent execution of multiple subtasks on different machines is considered.

This paper concentrates on theoretical aspects of matching, scheduling, and data relocation for stochastic HC. The optimization problem for minimizing the total execution time of an application program executed in a dedicated HC system with respect to the above three factors is completely defined based on a stochastic mathematical model, optimization criterion, and the search space described in Sections 3, 4, and 5. The practical application of the above theoretical results is demonstrated by the theorem shown in Section 6 that proves a greedy algorithm based approach is the best strategy for developing data relocation heuristics. The greedy local optimization strategy, coupled with the search space defined for choosing the data relocation schemes, can help developers of future data relocation heuristics.

Acknowledgments: The authors thank J. K. Antonio, M. B. Kulaczewski, Y. A. Li, and J. M. Siegel for their valuable comments.

References:

[1] G. Casella and R. L. Berger, *Statistical Inference*, Wadsworth & Brooks/Cole Advanced Books & Software, Pacific Grove, CA, 1990.

[2] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, MA, 1992.

[3] R. F. Freund, "Optimal selection theory for superconcurrency," *Supercomputing '89*, Nov. 1989, pp. 699-703.

[4] A. Ghafoor and J. Yang, "Distributed heterogeneous supercomputing management system," *IEEE Computer*, Vol. 26, No. 6, June 1993, pp. 78-86.

[5] M. A. Iverson, F. Ozguner, and G. J. Follen, "Parallelizing existing applications in a distributed heterogeneous environment," *Heterogeneous Computing Workshop*, Apr. 1995, pp. 93-100.

[6] L. Kleinrock, *Communication Nets: Stochastic Message Flow and Delay*, McGraw-Hill, New York, NY, 1964.

[7] Y. A. Li, J. K. Antonio, H. J. Siegel, and M. Tan, D. W. Watson, "Estimating the distribution of execution times for SIMD/SPMD mixed-mode programs," *Heterogeneous Computing Workshop*, Apr. 1995, pp. 35-46.

[8] H. J. Siegel, J. K. Antonio, R. C. Metzger, M. Tan, and Y. A. Li, "Heterogeneous computing," in *Parallel and Distributed Computing Handbook*, A. Y. Zomaya, ed., McGraw-Hill, New York, NY, 1996, pp. 725-761.

[9] M. Tan, J. K. Antonio, H. J. Siegel, and Y. A. Li, "Scheduling and data relocation for sequentially executed subtasks in a heterogeneous computing system," *Heterogeneous Computing Workshop*, Apr. 1995, pp. 109-120.

[10] D. Towsley, "Allocating programs containing branches and loops within a multiple processor system," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 10, Oct. 1986, pp. 1018-1024.

AUTHOR BIOGRAPHIES

Min Tan is a Ph.D. candidate in the School of Electrical and Computer Engineering at Purdue University, West Lafayette, Indiana, USA. His research interests include data source management in heterogeneous computing, data staging issues for network communication, video compression and financial applications on parallel and distributed systems, and dynamic partitionability for reconfigurable parallel processing machines. He has authored or coauthored ten conference papers, one book chapter, and two technical reports.

Mr. Tan attended Shanghai Jiao Tong University, Shanghai, People's Republic of China, in 1988. In 1991, he went to Western Maryland College, Maryland, USA, and received a BA degree in Mathematics and Physics in 1993. In 1994, he received an MS degree in Electrical Engineering from Purdue University. While at Purdue, he received the "Estus H. and Vashti L. Magoon Outstanding Teaching Assistant Award" in 1996. He also worked as a software engineer for Dupont Photomasks, Inc., and for Hughes Network Systems, Inc., during the summers of 1995 and 1996, respectively. Mr. Tan is a member of IEEE, the IEEE Computer Society, and the Eta Kappa Nu honorary society.

Howard Jay Siegel is a Professor and Coordinator of the Parallel Processing Laboratory in the School of Electrical and Computer Engineering at Purdue University. He received two B.S. degrees from the Massachusetts Institute of Technology (MIT), and the M.A., M.S.E., and Ph.D. degrees from Princeton University. He has coauthored over 230 technical papers, has coedited seven volumes, and wrote the book "Interconnection Networks for Large-Scale Parallel Processing" (second edition 1990). He is a Fellow of the IEEE, was a Coeditor-in-Chief of the Journal of Parallel and Distributed Computing, and is currently on the Editorial Boards of both the IEEE Transactions on Parallel and Distributed Systems and the IEEE Transactions on Computers. He is an international keynote speaker and tutorial lecturer, as well as a consultant.

Prof. Siegel's research interests include heterogeneous computing, parallel algorithms, interconnection networks, and the PASM reconfigurable parallel computer system. In the area of heterogeneous computing, he is examining ways to match segments of a task to different machines in a heterogeneous suite to exploit the varied computational capabilities available. His algorithm work explores the factors involved in mapping a problem onto a parallel processing system to minimize execution time. Topological properties and fault tolerance are the focus of his research on interconnection networks for large-scale parallel machines. He is analytically and experimentally investigating the utility of the three dimensions of dynamic reconfigurability supported by the PASM design ideas and the small-scale proof-of-concept prototype: mixed-mode parallelism, switchable inter-processor communications, and system partitionability.

Prof. Siegel was Program Co-Chair of the ''1983 International Conference on Parallel Processing,'' Program Chair of ''Frontiers '92: The 4th Symposium on the Frontiers of Massively Parallel Computation,'' and Program Chair of the ''8th International Parallel Processing Symposium.'' In addition, he has been General Chair/Co-Chair of four international conferences and Chair/Co-Chair of four workshops.

Optimal Task Assignment in Heterogeneous Computing Systems

Muhammad Kafil and Ishfaq Ahmad

Department of Computer Science

The Hong Kong University of Science and Technology, Hong Kong.

Abstract[1]

Distributed systems comprising networked heterogeneous workstations are now considered to be a viable choice for high-performance computing. For achieving a fast response time from such systems, an efficient assignment of the application tasks to the processors is imperative. The general assignment problem is known to be NP-hard, except in a few special cases with strict assumptions. While a large number of heuristic techniques have been suggested in the literature that can yield sub-optimal solutions in a reasonable amount of time, we aim to develop techniques for optimal solutions under relaxed assumptions. The basis of our research is a best-first search technique known as the A algorithm from the area of artificial intelligence. The original search technique guarantees an optimal solution but is not feasible for problems of practically large sizes due to its high time and space complexity. We propose a number of algorithms based around the A* technique. The proposed algorithms also yield optimal solutions but are considerably faster. The first algorithm solves the assignment problem by using parallel processing. Parallelizing the assignment algorithm is a natural way to lower the time complexity, and we believe our algorithm to be novel in this regard. The second algorithm is based on a clustering based pre-processing technique that merges the high affinity tasks. Clustering reduces the problem size, which in turn reduces the state-space for the assignment algorithm. We also propose three heuristics which do not guarantee optimal solutions but provide near-optimal solutions and are considerably faster. By using our parallel formulation, the proposed clustering technique and the heuristics can also be parallelized to further improve their time complexity.*

Keywords: Best-first search, parallel processing, task assignment. mapping, distributed systems.

1 Introduction

The fast progress of network technologies and sequential processors has made distributed computing systems, such as networks of heterogeneous workstations or PCs, an attractive alternative to massively parallel machines. To exploit the capabilities of these systems for an effective parallelism, the tasks of an application must be properly assigned to the processors.

Given a parallel program represented by a task graph and a network of processors also represented as a graph, the assignment problem is to find an allocation of the tasks to the processors that results in the minimum turnaround time. This is usually done by assigning an equal amount of load to all processors and by reducing the overhead of interaction among them. An assignment can be *static* or *dynamic*, depending upon on the time at which the allocation or assignment decisions are made. In a static assignment the information about the tasks and processors in the systems is assumed to be known in advance, and the tasks are allocated to the processors before starting the execution. The task assignment problem, also known as the *allocation problem* or the *mapping problem* [4], is well known to be NP-hard [6], but continues to be regarded as an interesting and important problem.

Most of the algorithms proposed in the past yield sub-optimal solutions while optimal algorithms exist only for restricted cases or small problem sizes. Optimal solutions, however, are required in many situations where performance is the primary goal. Also, once an optimal assignment of a program is determined, one can reuse this information for future mappings.

The simplest approach to finding an optimal solution is an exhaustive search. But since there are n^m ways for assigning m tasks to n processors, an exhaustive search is impractical. Another possibility is to reduce the size of the state-space using an *informed search*. The A* algorithm from the area of artificial intelligence is one such informed search algorithm. The algorithm, despite guaranteeing an optimal solution, is not feasible for problems of practically large sizes because of its high time and space complexity. Thus, we need ways to either further reduce the size of the state-space, or speedup the search process using parallel processing — or do both.

Since a parallel program is executed on multiple processors, it is natural to utilize the same processors to speedup the mapping of the program. Parallel processing can help in reducing the search time and allows to find optimal assignments for larger problem sizes, as compared to the serial algorithms. Even for a sub-optimal solution, parallel processing can help in solving a problem of larger size. However, very little work has been done on using parallel processing in solving the assignment problem; a few exceptions are the parallel heuristic for the scheduling problem proposed by Ahmad and Kwok [2] and the parallel heuristics for the assignment problem proposed by Bultan and Akyanat [5]. To the best of our knowledge, no prior work on finding an optimal assignment using parallel processing has been reported.

1. This research was supported by the Hong Kong Research Grants Council under contract number HKUST 619/94E.

We propose a parallel algorithm that generates an optimal solution for assigning an arbitrary task graph to an arbitrary network of heterogeneous processors. The algorithm, running on the Intel Paragon parallel machine, gives optimal assignments for small to medium size problems, with a reasonable speedup. We also propose a clustering based pre-processing algorithm that merges the high affinity tasks before starting the search. This reduces the problem size which in turn reduces the size of the state-space for the assignment algorithm. We also propose three heuristics which do not guarantee optimal solutions but yield near-optimal solutions and take considerably less execution time. The proposed heuristics and the clustering-based approach can also be parallelized using the proposed parallel formulation.

2 Problem Definition

A parallel program can be partitioned into a set of m communicating tasks represented by an undirected graph $G_T = (V_T, E_T)$ where V_T is the set of vertices, $\{t_1, t_2,.., t_m\}$, and E_T is a set of edges labelled by the communication costs between the vertices. The interconnection network of n processors, $\{p_1,p_2,..,p_n\}$, is represented by an $n*n$ matrix L, where an entry L_{ij} is 1 if the processors i and j are connected, and 0 otherwise.

A task t_i from the set V_T can be executed on any one of the n processors of the system. In a heterogeneous system [16], each task has an execution cost associated with it on a given processor. The execution costs of tasks are given by a matrix X, where the matrix entry X_{ip} is the execution cost of task i on processor p. When two tasks t_i and t_j executing on two different processors need to exchange data, a communication cost is incurred. Communication among the tasks is represented by a matrix C, where C_{ij} is the communication cost between task i and j if they reside on two different processors. The load on a processor is the combination of all the execution and communication costs associated with the tasks assigned to it. The total completion time of the entire program will be the time needed by the heaviest loaded processor.

Task assignment problem is to find a mapping of the set of m tasks to n processors such that the total completion time is minimized. The mapping or assignment of tasks to processors is given by a matrix A, where A_{ip} is 1 if task i is assigned to processor p and 0 otherwise. The load on a processor p is given by

$$\sum_{i=1}^{m} X_{ip} \bullet A_{ip} + \sum_{\substack{q=1 \\ (p \neq q)}}^{n} \sum_{i=1}^{m} \sum_{j=1}^{m} (C_{ij} A_{ip} A_{jq} L_{pq}) .$$

The first part of the equation represents the total execution cost of the tasks assigned to processor p, and the second part is the communication overhead on p. To find the processor with the heaviest load, the load on each of the n processors needs to be computed. The optimal assignment is the one that results in the minimum load on the heaviest loaded processor among all the assignments.

3 Related Work

A large number of task assignment algorithms have been proposed using various techniques such as network flow [17], integer programming [12], state-space search [14, 15, 18], clustering [3], bin-packing [19], randomized optimization [1, 5, 7, 8], etc. Most of these algorithms can be classified according to the taxonomy given in Figure 1. At the first level of the hierarchy these algorithms can be classified as *optimal* and *sub-optimal* categories, where the optimal algorithms can be further classified as *restricted* or *non-restricted* categories. Restricted algorithms yield optimal solutions in a polynomial time by restricting the structure of the program and/or the multicomputer system. Non-restricted algorithms, on the other hand, consider the problem in a more general context; they give optimal solutions but not necessarily in a polynomial time.

Sub-optimal algorithms can be divided into *approximate* or *heuristics* classes. Approximate algorithms [9] assume the same computational model used by the optimal algorithm. But instead of searching the complete solution space for optimal solution, approximate algorithms guarantee a solution that is within a certain range from the optimal solution. Heuristic algorithms make use of special parameters which affect the system in indirect ways, for example, clustering the groups of heavily communicating tasks together. A greedy heuristic starts from a partial assignment and assigns one task at each step until a complete assignment is obtained; in general, backtracking is not allowed. Bin-packing techniques use a sizing policy, an ordering policy, and a placement policy for the tasks to be assigned. Randomize optimization methods start from a complete assignment and search for an improvement in the assignment by exchanging and moving tasks among different processors.

Because of the intractable nature of the problem most of the research is focused on the development of heuristic algorithms. There are also some optimal algorithms available either for restricted cases of the problem or for very small problem sizes.

4 Overview of the A* Technique

The A* algorithm is a *best first* search algorithm [13]. It has been extensively used for problem solving in artificial intelligence. The algorithm is used to search efficiently in a search-space (which is a tree in our case but can be some other type of graph). It searches the nodes of the tree starting from the root called the *start node* (usually a null solution of the problem). Intermediate nodes represent the partial solutions while the leaf nodes represent the complete solutions or *goals*.

Associated with each node is a cost which is computed by a cost function f. The nodes are ordered for search according to this cost, that is, the node with the minimum cost is searched first. The value of f for a node n is computed as:

$$f(n) = g(n) + h(n)$$

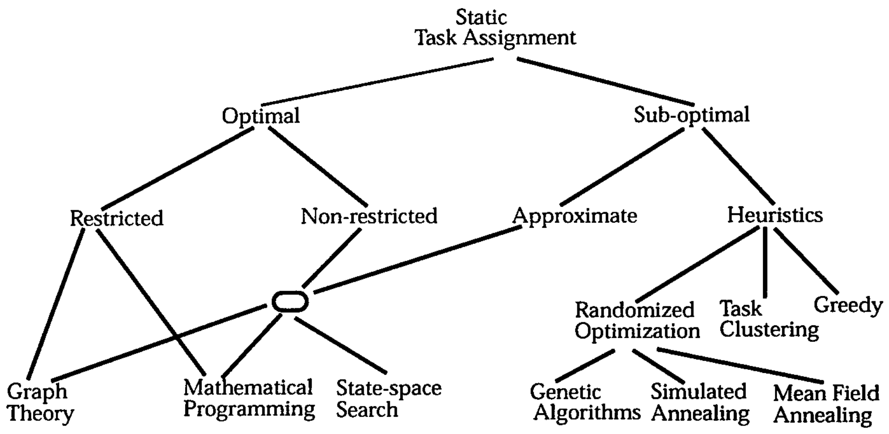


Figure 6: A classification of task assignment algorithms.

where $g(n)$ is the cost of the search path from the start node to the current node n; $h(n)$ is a lower bound estimate of the path cost from node n to the goal node (solution). Expansion of a node is to generate all of its successors or children and compute the f value for each of them. The algorithm maintains a sorted list, called *OPEN*, of nodes (according to their f values) and always selects a node with the best cost for expansion. Since the algorithm always selects the best cost node, it guarantees an optimal solution. Since for a leaf node n, $h(n)$ is 0, we will set the value of $f(n)$ equal to $g(n)$ for all leaf nodes.

4.1 Application to Task Assignment

For the task assignment problem under consideration, the search space is a tree. The initial node (the root) is a node with null assignment, i.e., no task is assigned; intermediate nodes are nodes with partial assignments, i.e., some tasks are assigned while others are still unassigned at this stage. A solution (goal) node is a node with a complete assignment (all task are assigned). For the computation of the cost function, g(n) is the cost of partial assignment (A) at node n, that is, the load on the heaviest loaded processor. For the computation of $h(n)$, two sets T_p (the set of tasks which are assigned to the heaviest loaded processor p) and U (the set of tasks which are unassigned at this stage of the search and have a communication link with any task in set T_p) are defined. Now each task t_i in U will be assigned to either processor p or any other processor q which has a direct communication link with p. Thus, there can be two kinds of costs associated with the assignment of each t_i: $X_{i,p}$ (the execution cost of t_i on processor p) and the sum of communication cost with all the tasks in set T_p. Let cost (t_i) be the minimum of these two costs, then $h(n)$ is computed as;

$$h(n) = \sum_{t_i \varepsilon U} \text{cost}(t_i)$$

The algorithm A* is described as follows:

The A* Algorithm

```
(1) Build the initial node s and insert it into the list OPEN
(2) Set f(s) = 0
(3) Repeat
(4)     Select the node n with the smallest f value.
(5)   if (n is not a solution)
(6)     Generate successors of n
(7)     for each successor node n' do
(8)   if (n' is not at the last level in the search tree)
(9)   f(n') = g(n') + h(n')
(10)  else f(n') = g(n')
(11)    Insert n' into OPEN
(12)    end for
(13) end if
(14)if (n is a solution)
(15)  Report the Solution and stop
(16)Until (n is a Solution) or (OPEN is empty)
```

A study by Ramakrishnan *et al.* [14] showed that the order in which the tasks are considered for allocation has a great impact on the performance of the algorithm (for the same cost function used). Their study indicated that a significant performance improvement could be achieved by first considering the tasks with larger weights in the computation of the optimal cost at the shallow levels of the tree. They proposed a number of heuristics for ordering the tasks. Out of these heuristics the so called *minimax sequencing* heuristic has been shown to perform the best. The minimax sequencing works as follows. Consider a matrix H of m rows and n columns where m is the number of tasks and n is the number of processors. The entry $H(i, k)$ of the matrix is given by

$$H(i, k) = X_{ik} + h(v),$$

where $h(v)$ is given by

$$h(v) = \sum_{j \in U} min(X_{jk}, C_{ij}),$$

where U is the set of unassigned tasks which communicate with t_i. The minimax value, $mm(t_i)$ of task t_i is defined as

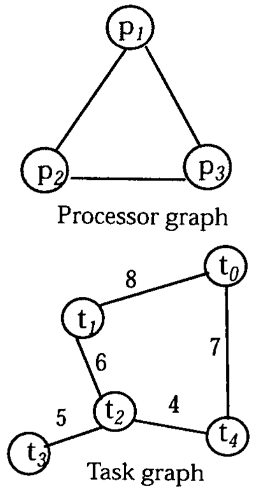
$$mm(t_i) = min\{H(i,k), 1 \le k \le n\}.$$

The minimax sequence is then defined as:

$$\Pi = \{\tau_1, \tau_2, \ldots, \tau_m\}, mm(\tau_i) \ge mm(\tau_{i+1}), \forall i.$$

4.2 An Illustrative Example

Given a set of 5 tasks, $\{t_0, t_1, t_2, t_3, t_4\}$ and a set of 3 processors $\{p_0, p_1, p_2\}$ as shown in Figure 2, the algorithm first generates the minimax sequence $\{t_0, t_1, t_2, t_4, t_3\}$.



	p_0	p_1	p_2
t_0	15	11	9
t_1	14	12	8
t_2	16	13	6
t_3	5	4	3
t_4	10	9	7

Execution cost matrix

Figure 2: An example task graph and a processor and the network, execution costs of the tasks on various processors.

Figure 2 illustrates the search tree for finding the assignment for this example.

A node in the search tree includes the partial assignment of tasks to processors as well as the value of f (the cost of partial assignment). The assignment of m tasks to n processors is indicated by an m digit string '$a_0a_1 \ldots a_{m-1}$', where a_i ($0 \le i \le m-1$) represents the processor (0 to $n-1$) to which ith task has been assigned. A partial assignment means that some tasks are unassigned; the value of a_i equal to 'X' indicates that ith task has not been assigned yet. Each level of the tree corresponds to a task, thus replacing an 'X' value in the assignment string with some processor number. Node expansion is to add the assignment of a new task to the partial assignment. Thus the depth (d) of the search tree is equal to the number of tasks m, and any node of the tree can have a maximum of n (no of processors) successors.

The root node includes the set of all unassigned tasks 'XXXXX'. For example in Figure 2, the allocations of t_0 to p_0 ('0XXXX'), t_0 to p_1 ('1XXXX'), and t_0 to p_2 ('2XXXX') are considered by determining the costs of assignments at the first level of the tree. The assignment of t_0 to p_0 ('0XXXX') results in the total cost $f(n)$ being equal to 30. The $g(n)$ in this case equals 15 which is the cost of executing t_0 on p_0. The $h(n)$ in this case also equals 15 which is the sum of minimum of the execution or communication costs of t_1 and t_4 (tasks communicating with t_0). The costs of assigning t_0 to p_1 (26) and t_0 to p_2 (24) are calculated in a similar fashion. These three nodes are inserted to the list *OPEN*. Since 24 is the minimum cost, the node '2XXXX' is selected for expansion. The search continues until the node with the complete assignment ('20112') is selected for expansion

At this point since this is the node with a complete assignment and the minimum cost, it is the goal node. Notice that all assignment strings are unique. A total of 39 nodes are generated and 13 nodes are expanded. In comparison, an exhaustive search will generate $n^m = 243$ nodes in order to find the optimal solution.

5 The Proposed Algorithms

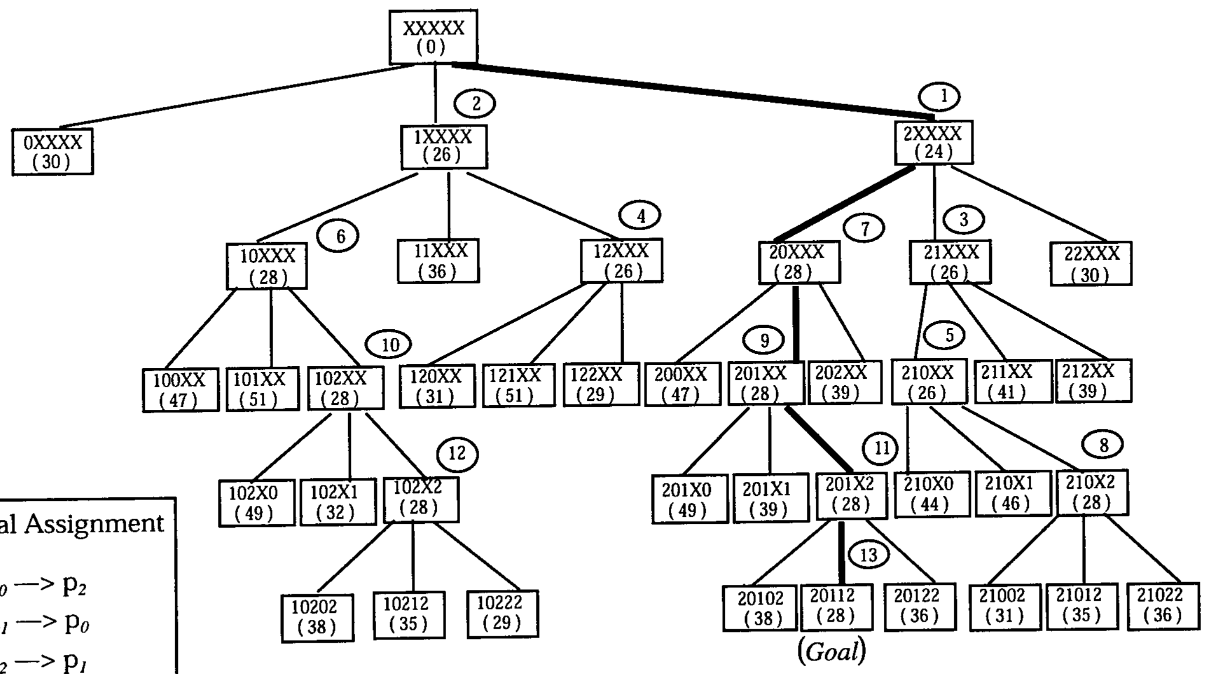
In this section, we describe our proposed parallel and clustering algorithms for optimal solutions. The sub-optimal algorithms are also explained in this section.

5.1 The Parallel Algorithm

The objective of the parallel algorithm is to divide the search tree among the processing elements (PEs) as evenly as possible and to avoid the expansions of non-essential nodes, that is, the nodes which are not expanded by the sequential algorithm. A good overview of parallel depth-first and best-first search algorithms are given in [10][11]. To distinguish the processors on which the parallel task assignment algorithm is running from the processors in the problem domain, we will denote the former with the abbreviation PE (processing element which in our case is the Intel Paragon processor). We call this parallel algorithm the *Optimal Assignment with Parallel Search* (OAPS) algorithm.

The OAPS Algorithm:

```
(1) Init- Partition()
(2) SetUp-Neighborhood()
(3) Repeat
(4)    Expand the best cost node from OPEN
(5)    if (a solution found)
(6)      if (it's better than previously received Solutions)
(7)      Broadcast the Solution to all PEs
(8)    else
(9)      Inform neighbors that I am done
(10)   end if
(11)   Record the solution and stop
(12)   end if
(13)   If (OPEN's length increases by a threshold u)
(14)     Select a neighbor PE j using RR
(15)     Send the current best node from OPEN to j
(16)   end if
(17)   If (Received a node from a neighbor)
(18)     Insert it to OPEN
(19)   if (Received a solution from a PE)
(20)       Insert it to OPEN
(21)     if (Sender is a neighbor)
(22)       Remove this from neighborhood list
(23)   end if
(24)Until (OPEN is empty) OR (OPEN is full)
```



Final Assignment

$t_0 \longrightarrow p_2$

$t_1 \longrightarrow p_0$

$t_2 \longrightarrow p_1$

$t_3 \longrightarrow p_1$

$t_4 \longrightarrow p_2$

Figure 3: The search tree for the example problem (nodes generated = 39, nodes expanded = 13).

Initially the search tree is divided statically based on the number of processing elements (PEs) P in the system and the maximum number of successors, S, of a node in the search tree. There could be three situations:

Case 1) $P< S$: Each PE will expand only the initial node which results in S new nodes. Each PE will get one node and additional nodes are distributed in a round robin (*RR*) fashion.

Case 2) $P = S$: Only the initial node will be expanded and each PE will get one node.

Case 3) $P > S$: Each PE will keep expanding nodes starting from the initial node (the null assignment) until the number of nodes in the list is greater than or equal to P. List is sorted in an increasing order of cost values of the nodes. The first node in the list will go to PE_1, the second node will go to PE_p, the third node goes to PE_2, the fourth node goes to PE_{p-1}, and so on. Extra nodes will be distributed in RR fashion. Although there is no guarantee that a best cost node at the initial levels of the tree will lead to a good cost node after some expansions, the algorithm still tries to distribute the good nodes as evenly as possible among all the PEs.

If a solution is found during the search, the algorithm terminates. Note that there is no master PE which is responsible for generating and distributing nodes among the PEs. Therefore, the overhead of the static node assignment is negligible as compared to the host-node style because the whole process is done in parallel. To illustrate this, we consider the example of the task assignment problem of assigning 10 tasks to 4 processors using 2 PEs (PE1 and PE2). Here S is 4 since a node in the search tree can have a maximum of 4 successors. Each PE, therefore, generates 4 nodes numbered from 1 to 4 (as shown in Figure 4 where the number in a box is the *f* value of the node). PE1 will then get the first and third node 3, while PE2 will get the second and fourth node.

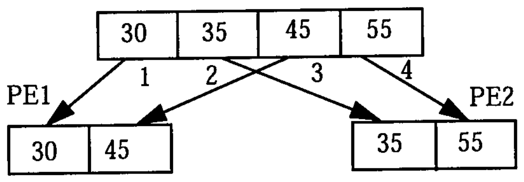


Figure 4: An initial static assignment.

If there is no communication among the PEs after the initial static assignment (i.e., every PE just searches its own tree), some of them may work on a good part of the search space, while others may expand unnecessary nodes (i.e., the nodes which the serial algorithm will not expand). This can result in a poor speedup. To avoid this, PEs need to communicate to share the best part of the search space and to avoid unnecessary work. This communication can be global (a PE broadcast its nodes to all other PEs) or local (a PE communicates only with its neighbors).

In our formulation we have used a round robin (RR) within neighborhood communication strategy. With this communication strategy a PE can share the best part of the

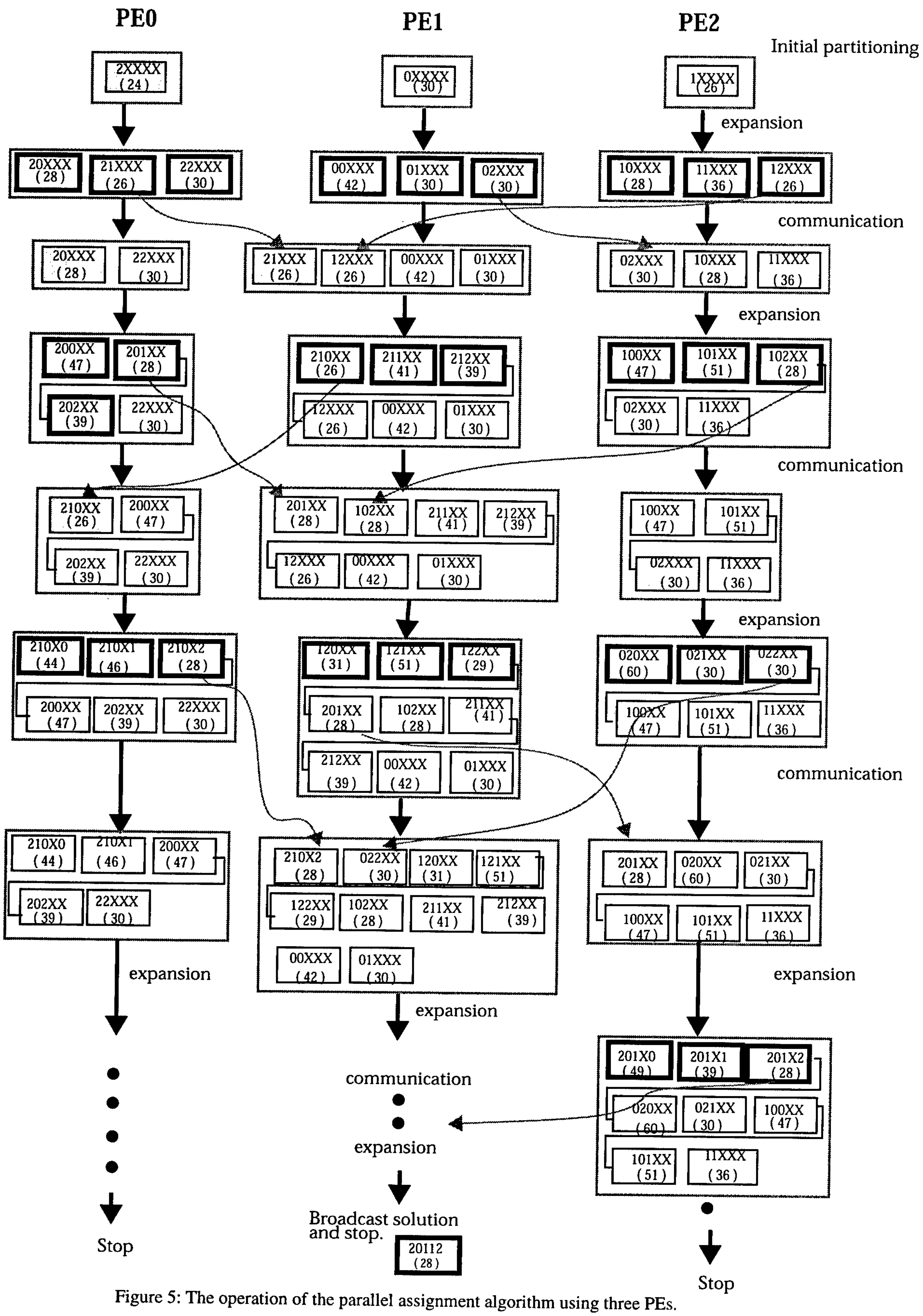


Figure 5: The operation of the parallel assignment algorithm using three PEs.

search space. Further, a PE can avoid unnecessary work explicitly by communicating with its neighbors and implicitly by broadcasting its solution to all other PEs. Since the Paragon PEs are connected together with a mesh topology, a PE can have a maximum of 4 neighbors. Since most of the time a PE communicates only with its neighbor, a low communication overhead is incurred making the algorithm more scalable as compared to a global communication strategy.

A PE periodically (when *OPEN* increases beyond a threshold *u*) selects a neighbor in a RR fashion and then sends its best node to that neighbor. As a result, the load is balanced and the best part of the search space is shared within the neighborhood of a PE. At finding a solution, a PE broadcasts it to all the PEs, thus helping in avoiding the unnecessary work for a PE that is working on the bad part of the search space. Once a node receives a better cost solution than its current best node, it stops expanding the unnecessary nodes. The PE that finds the first solution broadcasts its result to all other PEs, and from that point each PE broadcasts its solution only if its cost is better than a previously received solution.

With an initial partitioning, every PE has one or more nodes in its list *OPEN*. Each PE then determines the PEs in its neighbor by using its own position in the mesh (topology of the Intel Paragon). A PE starts expanding new nodes starting from the initial nodes. PEs then interact with each other for exchanging their best nodes and to broadcast their solutions. When a PE finds a solution, it records it in a common file (opened by all PEs) and stops. The optimal solution is the solution with the minimum costs among all PEs.

To illustrate the operation (see Figure 5) of the OAPS algorithm, we consider the example used earlier for the sequential assignment algorithm. Here we assume that the parallel algorithm runs on three PEs connected together as a linear chain. Initially three nodes are generated as in the sequential case. Then, through the initial partitioning, these nodes are assigned to the three PEs. Each PE then goes through a number of steps. In each step, there are two phases: the expansion phase and the communication phase[1]. In the expansion phase, a PE sequentially expands its nodes (the newly created nodes are shown with thick borders). It will keep on expanding until it reaches the threshold (*u*) (which is set to be 3 in this example). In the communication phase, a PE selects a neighbor and then sends its best cost node to it. The selection of the neighbors is done in a RR fashion. In Figure 5, the exchange of the best cost nodes among the neighbors is shown by dashed arrows. In the 5th step, PE1 finds its solution, broadcasts it to other PEs, and then stops. In the final step, PE0 also broadcasts (not shown here for the sake of simplicity) its solution to PE2 which finally records its solution and stops.

1. The synchronous operation of PEs shown here is just to illustrate the concept; the actual algorithm is fully asynchronous and thus may follow a different sequence — the final result will of course be the same.

5.2 The Preprocessing Clustering Algorithm

The algorithm starts by clustering (or merging) the tasks in the task graph. Two tasks are merged if the communication cost among them is so high that they will never be assigned to two different processors in the optimal assignment; Equations 5.1 and 5.2 given below ensure that the two tasks under consideration are never assigned to two different processors. Clustering reduces the size of the task graph and hence the depth (*d*) of the resulting search tree.

The algorithm first sorts the edges of the task graph, and then selects the largest edge (*i, j*), where task *i* and *j* are the tasks connected with the edge. The cost of an edge when mapped onto an edge of the processor graph is defined as the sum of the edge cost and the minimum execution cost of task *i* or *j* on the processors of the processor edge. The cost is computed using the following equation:

$$\min_{p,\,q\,=\,1\text{ to }n}\begin{pmatrix}\min\{(X_{ip}+C_{ij})\bullet L_{pq},\ (X_{jq}+C_{ij})\bullet L_{pq}\}\\ \min\{(X_{jp}+C_{ij})\bullet L_{pq},\ (X_{iq}+C_{ij})\bullet L_{pq}\}\end{pmatrix}\quad(5.1)$$

The cost of assigning tasks *i* and *j* to the same processor is the minimum execution cost of two tasks on either of the two processors of the processor edge. This cost is given by the following equation.

$$\min_{p,\,q\,=\,1\text{ to }n}\ \{(X_{ip}+X_{jq}),\ (X_{iq}+X_{jq})\}\qquad(5.2)$$

A selected edge is merged if the cost of mapping it onto all of the processor edges is higher than the cost of assigning the two tasks on the same processor. The clustering process is repeated for all the edges of the processor graph.

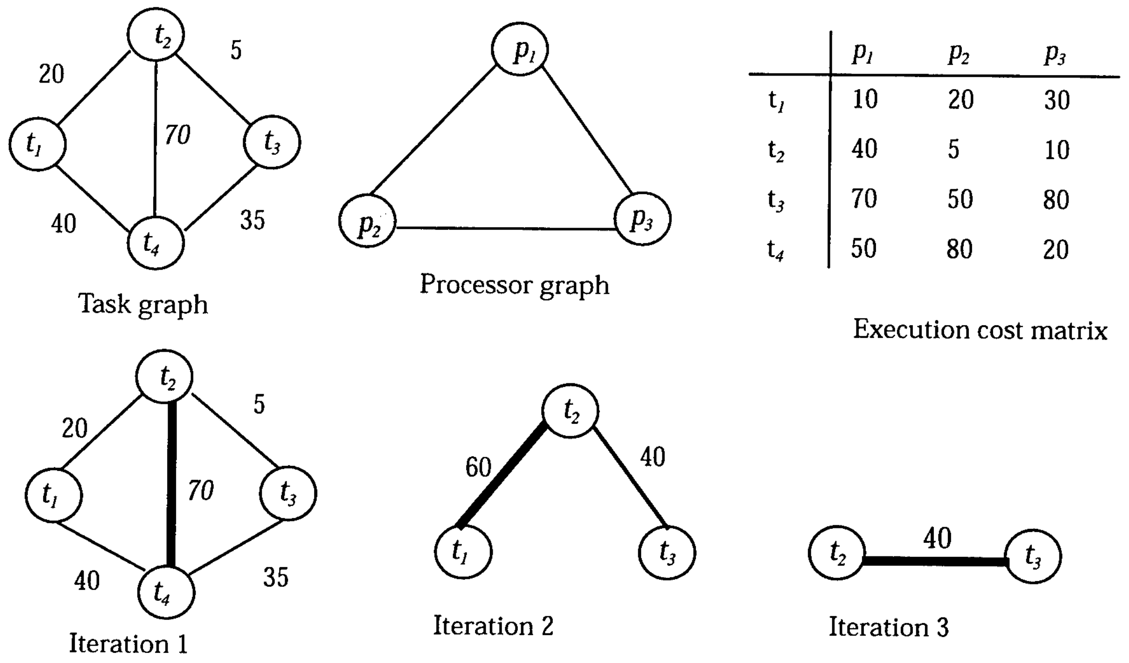
The clustering process is illustrated by an example, given in Figure 6, where the largest edges selected are shown as thick edges. In the first iteration the edge (t_2, t_4) is selected and task t_4 is merged with t_2 and its communication links with other tasks are added to t_2. In the second iteration t_1 is merged. In the third iteration, the selected edge is not merged, and the algorithm stops.

After clustering, the tasks are reordered using the minimax sequencing as discussed in Section 4.1. Now the tasks are selected for the assignment using this sequence.

The clustering procedure guarantees an optimal assignment only when the processors are fully-connected since the searching algorithm assigns two communicating tasks only to the directly connected processors.

5.3 Sub-optimal Algorithm

The sub-optimal algorithm, henceforth referred to as the *Sub-Optimal Assignments* (**SA**) algorithm, is designed to obtain the solution faster and to overcome the high memory requirements of A*. The basic idea in this algorithm is that when the search process reaches a certain level deep in the search tree, some search can be avoided (some tree nodes can be discarded) without moving far from the optimal solution. Based on this reasoning, we



	p_1	p_2	p_3
t_1	10	20	30
t_2	40	5	10
t_3	70	50	80
t_4	50	80	20

Figure 6: Illustration of the clustering procedure.

propose three heuristics, SA1, SA2 and SA3. The first heuristic (SA1) is explained as follows. When the algorithm selects a node for expansion and that node belongs to a level R or deeper than that in the search tree, it generates only its best successor instead of generating all the successors (i.e., it discards all successors except the best one). The second heuristic (SA2) is similar to the first: when the search reaches at level R for the first time, the algorithm starts discarding all successors except the best node among all the nodes selected for expansion. The third heuristic (SA3) is similar to the second heuristic except the nodes are discarded from the global list (*OPEN*). For example, if n nodes are generated, then all of them are inserted to *OPEN* and $n - 1$ high cost nodes are discarded.

There is a little chance of running out of memory for the above mentioned heuristics. This is because when a node at level R is selected, the algorithm inserts only one node to *OPEN* for expansion and takes one node from it. Thus, no extra memory is required. Moreover, the running time of the algorithm is reduced by a large factor since the algorithm explores fewer nodes once it reaches the level R.

6 Experimental Results

We first discuss the workload used in our study and then present the experimental results obtained by the proposed algorithms.

6.1 Workload Generation

A realistic workload is important to validate an assignment algorithm but very little information is available about process communication patterns encountered in distributed systems. In distributed systems, there is usually a number of process groups with heavy interaction within the group, and almost no interaction with the processes outside the group [3]. With this intuition, we first generated a number of primitive task graph structures such as the pipeline, the ring, the server, and the interference graphs, all consisting of 2 to 8 nodes. The complete task graphs, consisting of 10-28 nodes, were generated by randomly selecting these primitives structures and combining them until the desired number of tasks was reached. This was done by first selecting a primitive graph and then combining it with a newly selected graph through a link labelled with cost 1; the last node was connected back to the first node.

Since we assume the processors to be heterogeneous (a homogeneous processor system is a special case of a heterogeneous processor system), the execution cost varies from processor to processor in the execution cost matrix (X); the average value, however, remains the same. To generate the execution costs for the nodes and the communication costs for the edges, we used a parameter called the *communication-to-cost ratio* (CCR) which is the value of the average computation cost divide by the average communication cost per node. For example, if the total communication cost (sum of the cost of all of the edges connected to this task) of task i is equal to 16.0 and the CCR is equal to 0.2, then the average execution cost of i will be given by: 16.0 /0.2 = 80. We used the following values of CCR: 0.1, 0.2, 1.0, 5.0, and 10.0.

For the processor graphs, we used 3 topologies each comprising 4 nodes. For the parallel algorithm OAPS, we used 2, 4, 8, and 16 Paragon PEs.

6.2 Running Times of the Serial Algorithm

In this section we present the running times of various versions of the serial assignment algorithm. Table 1 and 2 include the running times for different variations of the serial algorithm for the fully-connected topology comprising 4 processors. The running times of the serial algorithm without any task ordering or clustering are given in column 2; we will refer to it as A* in these tables. An entry '**' in a column means the algorithm could not generate the solution for this case using 50 MB of memory, i.e., it ran out of memory after a few hours (usually 5 to 6 hours). The third column shows the running times of the algorithm with the task ordering; we will refer to this technique as A*R. The fourth column shows the running times of the algorithm with clustering and then ordering; we will refer to this technique as A*C. The fifth column is the ratio of the running times of the two algorithms.

For the fully-connected topology of 4 processors and with CCR equal to 1.0 (see Table 1), the clustering algorithm is on the average 3.95 times faster than A*R. Table 2 presents the running times for the same topology but with CCR equal to 5.0. The clustering algorithm is on the average 281 times faster. The clustering algorithm performs well when the value of CCR is high because for these cases the optimal algorithm also assigns highly communicating tasks to the same processor. For lower values of CCR the algorithm does no merging for most of the cases.

It is observed that for most of the cases, task graphs with CCR equal to 0.1 and 0.2 result in larger search trees as compared to the graphs with CCR equal to 1.0, 5.0, and 10. The task graphs with CCR equal to 10.0 take the lowest running times. This is because the cost of the optimal solution for a higher CCR is less than a lower CCR and thus the algorithm finds the optimal solution quickly starting from an initial cost 0. For example, a task graph consisting of 10 tasks with the CCR equal to 10.0 has the solution cost equal to 7.36, while the same graph with the CCR equal to 0.1 has the solution cost equal to 374.00. Thus, the former takes only 0.40 seconds to find the solution while the latter takes 4.30 seconds.

The processor topology also has a great impact on the size of the search tree as well as on the running time. This is because the algorithm assigns two communicating task to two different processors only if the processors are directly connected. So, in case of the line or ring topology, the algorithm prunes some of the nodes in the search tree based on this constraint. On the other hand, no such pruning is done for the fully-connected case.

6.3 Speedup Using the Parallel Algorithm

In this section, we present the speedup of the parallel algorithm using various number of processors. The speedup is defined as the running time of the serial algorithm over the running time of the parallel algorithm.

Table 3 presents the speedup data for the fully-connected topology comprising 4 processors and the task graphs with CCR equal to 0.1. The second column includes the running time of the serial algorithm while the third, fourth, fifth, and sixth columns include the speedup of the parallel algorithm over the serial algorithm using 2, 4, 8 and 16 Paragon PEs, respectively. The bottom row of the table indicates the average speedup of all the task graphs.

We can observe that the speedup increases with an increase in the problem size. Also the problems with a lower value of CCR yield a better speedup in most of the cases, since the running times of the serial algorithm in those cases are much longer compared to the parallel algorithm.

Table 1: The running times using the fully-connected topology (CCR = 1.0)

No. of Tasks	T(A*) (sec)	T(A*R) (sec)	T(A*C) (sec)	T(A*R) / T(A*C)
10	3.35	0.87	0.17	5.12
12	139.54	0.73	0.77	0.95
14	270.70	4.82	3.77	1.28
16	822.08	36.08	1.67	21.60
18	**	31.62	30.76	1.03
20	**	55.78	22.19	2.51
22	**	67.70	67.78	1.00
24	**	191.27	55.21	3.46
26	**	206.63	143.06	1.44
28	**	2451.56	2124.08	1.15
Avg				3.95

Table 2: The running times using the fully-connected topology (CCR=5.0).

No. of Tasks	T(A*) (sec)	T(A*R) (sec)	T(A*C) (sec)	T(A*R) / T(A*C)
10	0.24	0.27	0.08	3.37
12	1.53	0.49	0.12	4.08
14	35.98	1.73	0.25	6.92
16	10.29	1.67	0.27	6.19
18	6195.63	29.79	0.55	54.16
20	**	21.96	1.14	19.26
22	**	3.98	3.18	1.25
24	**	3387.58	4.15	816.28
26	**	4134.28	2.19	1887.80
28	**	52.86	3.87	13.66
Avg				281.30

The values of the average speedup for the fully-connected, ring, and line topologies are shown graphically in Figure 7.

6.4 Results of the Heuristics

In this section, we present the result of comparing the three proposed heuristics (SA1, SA2, SA3) with the optimal algorithm. We make two kinds of comparisons. First, we compare the percentage deviation of the solution produced by SA1, SA2 and SA3 from that of OASS. This deviation is defined as follows:

$$\%D = (\text{Cost(SA)} - \text{Cost(OASS)} * 100) / \text{Cost(OASS)}$$

Second, we compare ratios of the running times of SA1, SA2 and SA3 to those of OASS. Optimal solutions are first obtained for the five task sets discussed in Section 5.2 and then sub-optimal solution are obtained using SA1, SA2 and SA3 for the same task sets. Heuristic tree level used is:

$$R = \left\lfloor \frac{d}{3} \right\rfloor,$$

where d is the maximum depth of the search tree. Table 4 presents the results for the ring topology with 4 processors and the task graphs with CCR equal to 0.2. Each entry in the table is the average of five runs of each algorithm for 5 task graphs generated using various permutations of the pipeline, the ring, the server and the interference sub-graphs. The average values of the percentage deviation in the solution and the ratios of the running times are indicated in the bottom row.

The results indicate that SA3 always gives good solutions in terms of the percentage cost deviation from the optimal. This is because SA3 discards high cost nodes from the global list *OPEN*, so good nodes are always prevented from deletion. SA2 deviates more than SA3 but is faster.

The average cost deviation and the ratio of time improvement for the fully-connected topology (with different values of CCR) is shown in Figure 8. It can be noted that the average percentage cost deviation for the cases with CCR equal to 5.0 and 10.0 is quite high as compared to the cases with lower values of CCR. This is because when the task graph has a larger value of CCR the optimal algorithm assigns more tasks to a single processor (for some cases all the tasks goes to one processor). Therefore, the optimal algorithm follows a rather straight path in the search tree considering less options. If the sub-optimal algorithm discards a node on this path, it will deviate far from the optimal.

The availability of the optimal algorithm, sub-optimal heuristics, and the parallel algorithm gives a choice to the user to select a suitable algorithm depending upon the objective. If the objective is to find a solution in a short time, then SA2 can be used. To obtain a near-optimal assignments for a task graphs with higher values of CCR, SA3 can be used. If finding the optimal solution is the main objective without any regard to the algorithm running time, then the sequential A* can be used. If the resources, such as a parallel machine, are available, then OAPS can be used to speedup the running time of the optimal algorithm.

7 Conclusions and Future Work

We proposed algorithms for optimal and sub-optimal assignments of tasks to processors. We considered the problem under relaxed assumptions such as an arbitrary task graph with arbitrary costs on the nodes and edges of the graph, and processors connected through an interconnection network. Our algorithms can be used for homogeneous as well as heterogeneous processors, although in this paper we considered only the heterogeneous cases. We believe that to the best of our knowledge, ours is the first attempt in proposing a parallel algorithm for the optimal task-to-processor assignment problem. Although we kept the mapping of the algorithm on the Paragon PEs simple, some fine refinements are possible to further improve the performance.

A further study is required to understand the behavior of the parallel algorithm. One possibility is to implement quantitative load balancing of the tree nodes after a processor finds its solution, i.e., let the processor find more than one solution. Also, additional experimentation is required to find the ideal value of the threshold u. The clustering algorithm and the sub-optimal heuristic SA3 may be combined in order to obtain faster and close-to-optimal assignments for task graphs with high values of CCR. Our future plans also include a parallelization and analysis of the heuristic algorithms (for an ideal tree level R) to start applying the heuristics would also require more future works.

References

[1] I. Ahmad and M. K. Dhodhi, "Task Assignment using Problem-Space Genetic Algorithm," *Concurrency: Practice and Experience*, vol. 7, no. 5. pp. 411-428, August 1995.

[2] I. Ahmad and Yu-Kwong Kwok, "A Parallel Approach to Multiprocessor Scheduling," *International Parallel Processing Symposium*, Santa Barbra, CA, April 1995, pp. 289-293.

[3] N. S. Bowen, C. N. Nikolaou, and A. Ghafoor, "On the Assignment Problem of Arbitrary Process Systems to Heterogeneous Distributed Computer Systems," *IEEE Trans on Computers*, vol. 41, no. 3, pp. 197-203, March 1992.

[4] S. H. Bokhari, ''On the Mapping Problem," *IEEE Trans. on Computers* vol. c–30, March 1981, pp. 207–214.

[5] T. Bultan and C. Aykanat, "A New Heuristic Based on Mean Field Annealing," *Journal of Parallel and Distributed Computing*, vol. 16, no. 4, pp. 292-305, Dec 1992.

[6] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, (Freeman, San Francisco, CA, 1979).

[7] D. E. Goldberg, "Genetic Algorithms in Search, Optimization, and Machine Learning," (Addison, Wesely, Reading, MA 1989)

[8] S. M. Hart and Chuen-Lung S. Chen, "Simulated Annealing and the Mapping Problem: A

Computational Study," *Computers and Operations Research,* vol. 21, no. 4, pp 455-461, 1994.

[9] M. A. Iqbal and S. H. Bokhari, "Efficient Algorithms for a Class of Partitioning Problems," *IEEE Trans. on Parallel and Distributed Systems,* vol. 6, no. 2, Feb. 1995.

[10] A. Grama and Vipin Kumar, "Parallel Search Algorithms for Discrete Optimization Problems," *ORSA Journal on Computing,* vol.7, no.4 (Fall 1995) pp 365-385.

[11] V. Kumar, K. Ramesh, and V. Nageshwara Rao. "Parallel best-first search of state-space graphs: A summary of results," *Proceedings of the 1988 National Conference on Artificial Intelligence*, pp. 122-126, Aug. 1988.

[12] P.-Yio R. Ma, E. Y. S Lee, "A Task Allocation Model for Distributed Computing Systems," *IEEE Trans. on Computers*, vol. c-31, no. 1, Jan. 1982.

[13] N. J. Nilson, *Problem Solving Methods in Artificial Intelligence.* New York: McGraw-Hill, 1971.

[14] S. Ramakrishnan, H. Chao, and L.A. Dunning, "A Close Look at Task Assignment in Distributed Systems," *IEEE INFOCOM '91*, pp. 806-812, 1991.

[15] C.-Ch. Shen and W.-H. Tsai, "A Graph Matching Approach to Optimal Task Assignment in Distributed Computing System Using a Minimax Criterion," *IEEE Trans. on Computers*, vol. c-34, no. 3, pp. 197-203, March 1985.

[16] H. J Siegel, J. K. Antonio, R. C. Metzger, Min Tan, and Yan A Li, "Heterogeneous Computing", Parallel and Distributed Computing Handbook, pp. 725-761, McGraw-Hill, New York.

[17] H. S. Stone, "Multiprocessor Scheduling with the Aid of Network Flow Algorithms," *IEEE Trans. on Software Engineering*, SE-3, vol. 1, pp. 85-93, Jan. 1977.

[18] J. B. Sinclair, "Efficient Computation of Optimal Assignments for Distributed Tasks," *Journal of Parallel and Distributed Computing*, vol. 4, 1987, pp. 342-362.

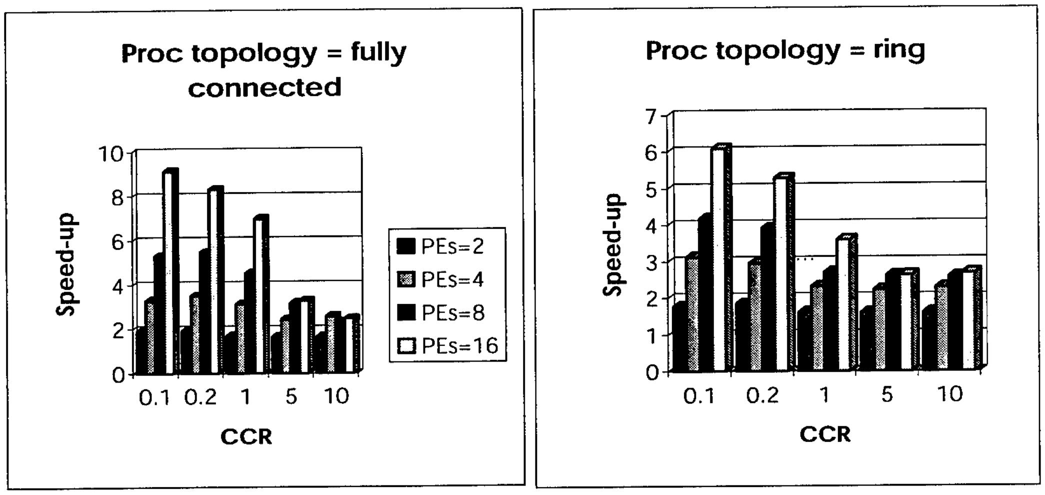
[19] C. Woodside and G. Monforton, "Fast Allocation of Processes in Distributed and Parallel Systems," *IEEE Transactions on Parallel and Distributed Systems,* vol. 4, no. 2, Feb. 1993.

Table 3: The speedup using the fully-connected topology (CCR=0.1).

No. of Tasks	T(A*R) (sec)	$\frac{T(A*R)}{T(OPAS)}$			
		PEs=2	PEs=4	PEs=8	PEs=16
10	30.14	1.87	3.48	5.72	7.63
12	58.96	1.96	3.68	3.60	12.85
14	105.05	1.70	2.02	4.58	4.64
16	1550.46	2.00	2.94	4.72	6.71
18	3839.00	2.00	3.86	7.59	13.16
20	3191.86	1.78	3.72	5.62	9.97
Avg		1.89	3.28	5.30	9.13

Table 4: The time and cost comparison using the ring topology (CCR=0.2).

No. of Tasks	$\frac{C(SA1) — C(OASS) *100}{C(OASS)}$	$\frac{C(SA3) — C(OASS) *100}{C(OASS}$	$\frac{C(SA3) — C(OASS) *100}{C(OASS}$	$\frac{T(O)}{T(SA1)}$	$\frac{T(O)}{T(SA2)}$	$\frac{T(O)}{T(SA3)}$
10	7.20	8.82	0.00	1.72	2.68	1.95
12	1.96	2.24	1.49	1.54	3.18	1.90
14	4.08	4.21	0.55	1.75	3.94	2.48
16	2.68	3.41	0.55	4.17	8.65	4.24
18	1.86	2.07	1.15	3.53	7.22	2.81
20	6.04	6.31	2.30	2.35	5.11	2.91
22	2.81	4.15	3.27	7.13	37.97	22.59
24	1.53	2.51	0.94	6.19	25.89	10.19
26	3.52	4.39	3.88	15.72	108.75	40.81
Avg	3.52	3.53	1.67	4.90	22.60	9.99



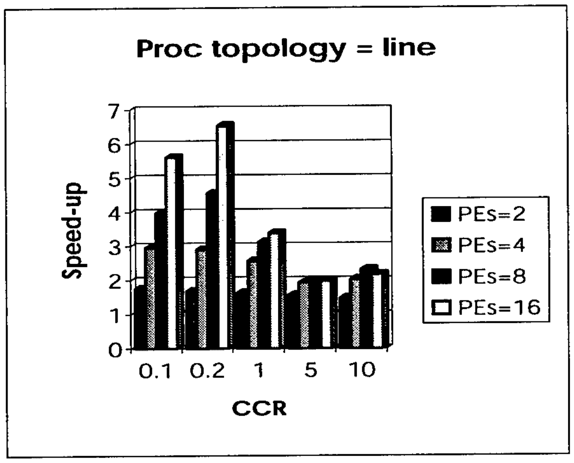


Figure 7: The average speedup of the parallel algorithm.

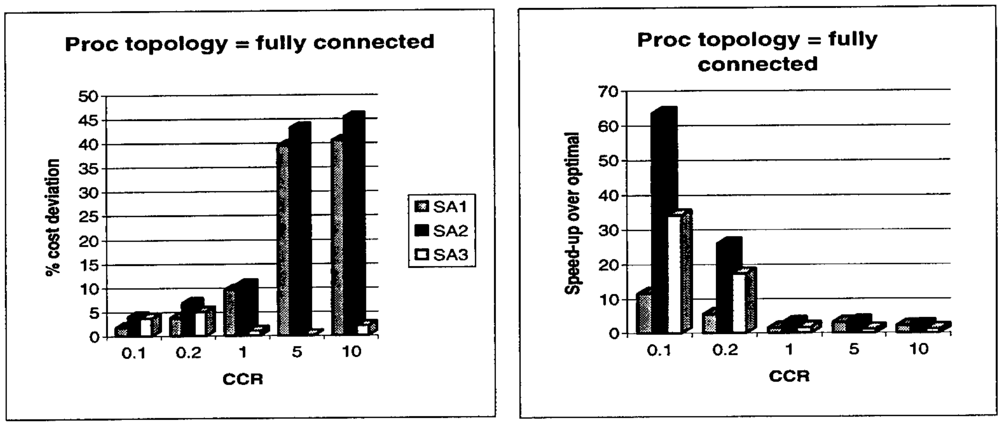


Figure 8: The percentage cost deviation and speedup of the sub-optimal algorithms over the optimal algorithm.

Mapping Heterogeneous Task Graphs onto Heterogeneous System Graphs *

M. M. Eshaghian

Dept. of Computer and Information Science
New Jersey Institute of Technology
Newark, NJ 07102

Y. C. Wu

SyncSort, Inc.
Woodcliff Lake, NJ 07675

Abstract

In this paper, a generic technique for mapping heterogeneous task graphs onto heterogeneous system graphs is presented. The task and system graphs studied in this paper have nonuniform computation and communication weights associated with the nodes and the edges. Two clustering algorithms have been proposed which can be used to obtain a multilayer clustered graph called a Spec graph from a given task graph and a multilayer clustered graph called a Rep graph from a given system graph. We present a mapping algorithm which produces a suboptimal matching of a given Spec graph containing M task modules, onto a Rep graph of N processors, in $O(MP)$ time, where $P = \max(M, N)$. Our experimental results indicate that our mapping algorithm is the fastest one and generates results which are better than, or similar to, those of other leading techniques which work only for restricted task or system graphs.

1 Introduction

The mapping problem is one of the most challenging problems in parallel and distributed computing. It is known to be NP-complete in its general form as well as several restricted forms [7]. In the mapping problem, a program task is divided into a number of task modules and these task modules are to be assigned to a parallel computer system with a set of homogeneous or heterogeneous processors for execution. A program task can be represented by a task graph, with each node representing a task module and each edge representing data communication between two modules. In the task graph, each node is associated with a weight representing the computation amount of the corresponding task module, while the weight of an edge represents the communication amount between the two task modules it is connecting. Similarly, a parallel computer system can be modeled as a weighted, undirected system graph with each node representing a processing unit and each edge representing a communication channel. In the system graph, each node is associated with a weight representing the computation speed of the corresponding processing unit while the weight of an edge represents the transmission rate of the two processing units which it connects.

In static mapping, the assignments of the nodes of the task graphs onto the system graphs are determined prior to the execution and are not changed until the end of the execution. Static mapping can be classified in two general ways. The first classification is based on the topology of task and/or system graphs [3]. Based on this, the mappings can be classified into four groups: (1) mapping specialized tasks onto specialized systems, (2) mapping specialized tasks onto arbitrary systems, (3) mapping arbitrary tasks onto specialized systems and (4) mapping arbitrary tasks onto arbitrary systems. The second classification can be based on the uniformity of the weights of the nodes and the edges of the task and/or the system graphs. Based on this, the mappings can be categorized into the following four groups: (1) mapping uniform tasks onto uniform systems [3, 2, 12, 1, 8], (2) mapping uniform tasks onto nonuniform systems, (3) mapping nonuniform tasks onto uniform systems [17, 14, 15, 6, 18], and (4) mapping nonuniform tasks onto nonuniform systems [16, 13].

In this paper, we concentrate on static mapping of arbitrary nonuniform task graphs onto arbitrary nonuniform system graphs. The existing mapping techniques in this group include El-Rewini and Lewis' mapping heuristic algorithm [6] and Lo's max flow min cut mapping heuristic [13]. The time complexity of these two heuristics are $O(M^2N^3)$ and $O(M^4N \log M)$, respectively, where M is the number of task modules and N is the number of processors. In this paper, we present an algorithm which can map arbitrary, nonuniform, architecturally inde-

0-8186-7879-8/97 $10.00 © 1997 IEEE

pendent task graphs onto arbitrary, nonuniform, task-independent system graphs in $O(MP)$ time, where $P = \max(M, N)$. This technique is based on the mapping methodology used in the Cluster-M portable parallel programming tool and consists of two clustering algorithms and a mapping algorithm, which are extensions to those presented in [3]. The experimental studies indicate that our mapping results are better than or similar to those of other leading techniques.

The rest of the paper is organized as follows. We present the Cluster-M preliminaries in Section 2. In Section 3, we present two clustering algorithms. The mapping algorithm is detailed in Section 4. We show our experimental results in comparing our algorithm with several other existing mapping algorithms in Section 5. A brief conclusion is given in Section 6.

2 Cluster-M Preliminaries

The nonuniform clustering and mapping algorithms presented in this paper are part of the mapping module of the Cluster-M portable parallel programming tool [4, 9]. In an earlier publication [3] a set of clustering and mapping algorithms was presented for the preliminary version of the Cluster-M mapping module. Those algorithms can handle only "uniform" arbitrary task and system graphs. The algorithms presented in this paper are nontrivial extensions of the Cluster-M uniform algorithms for mapping "nonuniform" arbitrary task graphs onto "nonuniform" arbitrary system graphs. In the following, we first give an overview of the Cluster-M tool and then present basic concepts used both in uniform and nonuniform Cluster-M clustering and mapping algorithms. A set of parameters used in the nonuniform clustering and mapping algorithms is presented in Section 2.3.

2.1 Cluster-M

Cluster-M is a programming tool that facilitates the design and mapping of portable parallel programs [3]. Cluster-M has three main components: the specification module, the representation module and the mapping module. In the specification module, machine-independent algorithms are specified and coded using the Program Composition Notation (PCN [11]) programming language [9]. Cluster-M specifications are represented in the form of a multilayer clustered task graph called Spec graph. Each clustering layer in the Spec graph represents a set of concurrent computations, called Spec clusters. A Cluster-M Representation represents a multilayer partitioning of a system graph called Rep graph. At every partitioning layer of the Rep graph, there are a number of clusters called Rep clusters. Each Rep cluster represents a set of processors with a certain degree of connectivity. Given a task (system) graph, a Spec (Rep) graph can be generated using one of the Cluster-M clustering algorithms. The clustering is done only once for a given task (system) graph independent of any system (task) graphs. It is a machine-independent (application-independent) clustering, therefore it is not necessary to be repeated for different mappings. For this reason, the time complexities of the clustering algorithms are not included in the time complexity of the Cluster-M mapping algorithm. In the mapping module, a given Spec graph is mapped onto a given Rep graph. This process is shown in Figure 1. In an earlier publication [3] two Cluster-M clustering algorithms and a mapping algorithm were presented for uniform graphs. Next, the basic concepts used in Cluster-M clustering and mapping will be explained. Using these concepts we present a set of parameters which is going to be used in the nonuniform clustering and mapping algorithms presented in Sections 3 and 4.

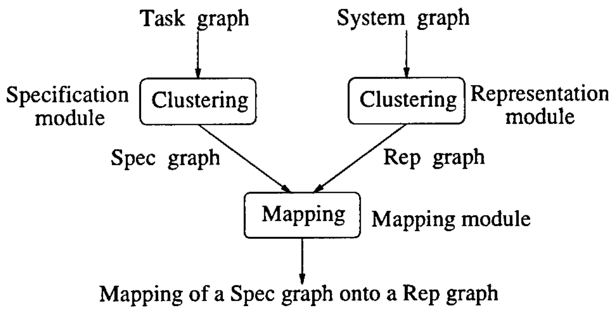
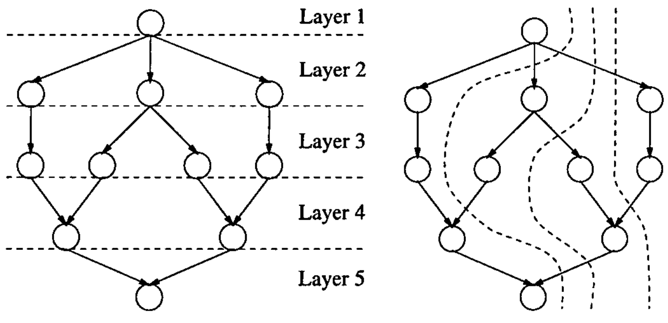


Figure 1: Cluster-M mapping process.

2.2 Basic Concepts

There are a number of reasons and benefits in clustering task and system graphs in the Cluster-M fashion. Basically Cluster-M clustering causes both task and system graphs be partitioned so that the complexity of the mapping problem is simplified and good mapping results can be obtained. In clustering an undirected graph, completely connected nodes are grouped together forming a set of clusters [3, 9]. Clusters are then grouped together again if they are completely connected. This is continued until no more clustering is possible. When an undirected graph is a task graph, then doing this clustering essentially identifies and groups communication-intensive sets of task nodes into a number of clusters called Spec clusters. Similarly for a system graph, doing the clustering identifies well-connected sets of processors into a number of clusters called Rep clusters. In the mapping process, each of the communication intensive sets of task nodes (Spec clusters) is to be mapped onto a communication-efficient subsystem (Rep cluster) of suitable size. Note that mapping of undirected task graphs onto undirected system graphs is referred to as the allocation problem. An earlier publication [3] showed that Cluster-M clustering and mapping algorithms can lead to good allocation results. It compared its results with Bokhari's $O(N^3)$ algorithm and showed that its algorithm has a lower time complexity of $O(MN)$, where M and N are the number of nodes in the task and system graphs, respectively.

Clustering directed graphs (i.e., directed task graphs) produces two types of graph partitioning: horizontal and vertical. Horizontal partitioning is obtained because, as part of clustering, we divide a directed graph into a layered graph such that each layer consists of a number of computation nodes that can be executed in parallel and a number of communication edges incoming to these nodes. This is shown in Figure 2(a). The layers are to be executed one at a time. Therefore, the mapping is done one layer at a time. This significantly reduces the complexity of the mapping problem since the entire task graph need not to be matched against the entire system graph.



(a) Horizontal (b) Vertical

Figure 2: Horizontal and vertical partitioning of a task graph.

Vertical graph partitioning is obtained because as part of the clustering the nodes from consecutive layers are merged or embedded. All the nodes in a layer are merged to form a cluster if they have a common parent node in the layer above or a common child node in the layer below. Doing this traces the flow of data. This information will be used later as part of the mapping so that the tasks are placed onto the processors in a way that total communication overhead is minimized. For example, to avoid unnecessary communication overhead, the task nodes along a path may be embedded into one another so that they are assigned to the same processor. The effect of this type of partitioning is shown in Figure 2(b).

Both horizontal and vertical graph partitionings are accomplished by performing the clustering in a bottom-up fashion. The Cluster-M mapping will then be performed in a top-down fashion by mapping the Spec clusters one layer at a time onto the Rep clusters. The next two sections show how these clustering and mapping ideas work for nonuniformly weighted graphs. The nonuniform algorithms shown in this chapter are nontrivial extensions of the Cluster-M uniform algorithms presented in an earlier publication [3].

2.3 Clustering Parameters

In the following, we present a set of parameters needed for nonuniform version of Cluster-M clustering and mapping. The first set is for representing a portable parallel program and the other for specifying the organization of the underlying heterogeneous architecture or suite.

2.3.1 Machine-Independent Program Parameters

A given parallel program consists of a sequence of steps such that in each step a number of computations can be done concurrently. Each step is called a layer. These concurrent computations for a given step (layer) can each be presented by a cluster called a Spec cluster. The mth Spec cluster at layer u is denoted by S^u_m and associated with the following parameters.

σS^u_m The size of S^u_m which is the maximum number of nodes in this cluster that can be computed in parallel.

δS^u_m The maximum sequential computation amounts (i.e., the maximum number of clock cycles required to execute all the instructions sequentially using a baseline computer) in S^u_m.

ΠS^u_m The total amount of communication from layer 1 to layer u of S^u_m.

πS^u_m The average communication amount at the layer u in S^u_m.

ρS^u_m The computational type of S^u_m. Its value is set to 0 for single instruction stream, multiple data stream (SIMD) type and 1 for multiple instruction stream, multiple data stream (MIMD) type[1].

[1]All the examples of the problems and systems studied in this paper are assumed to be of MIMD-type. However, in heterogeneous computing, it is possible to have a mix of SIMD and MIMD nodes both in the task and the system.

2.3.2 Program-Independent Machine Parameters

Any heterogeneous architecture can similarly be represented in a multilayered format such that each layer presents a set of processing units which are completely connected. Each processing unit is represented by a cluster called a Rep cluster. The nth Rep cluster at layer v is denoted by R_n^v and associated with the following parameters.

σR_n^v The number of processors contained in R_n^v.

δR_n^v The average computation speed of the processors in R_n^v.

ΠR_n^v The total data transmission rate including the transmission rate over the links (communication bandwidth) and over the nodes (switching latency) from layer 1 to v in R_n^v.

πR_n^v The average data transmission rate at layer v of R_n^v.

ρR_n^v The computational type of the Rep cluster. Its value is set to 0 for SIMD type and 1 for MIMD type.

3 Non-Uniform Clustering

This section first presents a clustering algorithm to be used for directed task graphs independent of any system graphs and then present another one for undirected system graphs independent of any task graphs. Both algorithms are done only once for any given task or system graph and are not repeated as part of the mapping process.

3.1 Clustering Directed Task Graphs

A task can be represented by a directed graph $G_t(V_t, E_t)$, where $V_t = \{t_1, ..., t_M\}$ is a set of task modules to be executed and E_t is a set of edges representing the partial orders and communication directions between task modules. A directed edge (t_i, t_j) represents that a data communication exists from module t_i to t_j and that t_i must be completed before t_j can begin, where $1 \leq i, j \leq M$. Each edge (t_i, t_j) is associated with D_{ij}, the amount of data required to be transmitted from module t_i to module t_j, where $D_{ij} \geq 1$. Each task module t_i is associated with its amount of computation A_i, that is, the number of instructions contained in t_i. Note that $A_i \geq 1$ and $D_{ij} \geq 1$ if there exits an edge (t_i, t_j), for $1 \leq i, j \leq M$. If a directed edge (t_i, t_j) exists, t_i is called a parent node (module) of t_j and t_j a child node (module) of t_i. If a node has more than one child, it is called a fork-node. If a node has more than one parent, it is called a join-node. A task graph is divided into a number of layers, so that all nodes in a layer can be executed concurrently.

```
Algorithm CNDG
Divide the directed graph into a number of layers
for each node at layer 1 do
    Make it into a cluster and calculate its parameters
For each of the other layers do
begin
    for all edges $(t_i, t_j)$ do
    begin if $t_i$ is a fork-node then
            begin Embed the child node with the largest edge
                      weight to $t_i$
                  if the child nodes of $t_i$ are not in a cluster then
                  begin Merge them with $t_i$ into a cluster
                            Calculate parameters of the new cluster
                  end
            end
            if $t_j$ is a join-node then
            begin Embed the child node with the largest edge
                      weight to $t_i$
                  if the parent nodes of $t_j$ are not in a cluster then
                  begin Merge them with $t_j$ into a cluster
                            Calculate parameters of the new cluster
                  end
            end
    end
end
```

Figure 3: Clustering Nonuniform Directed Graphs (CNDG) algorithm.

A clustering algorithm called clustering nonuniform directed graphs (CNDG) is shown in detail in Figure 3. This nonuniform algorithm is designed as an extension to the uniform clustering algorithm presented in an earlier publication [3]. The nonuniform algorithm has been designed in such a way that it is a generalization of the uniform algorithm. For clustering nonuniform directed graphs, a quintuple of parameters (σS_m^u, δS_m^u, ΠS_m^u, πS_m^u, ρS_m^u) from the Cluster-M model described in Section 2.3 is associated with the mth Spec cluster at layer u denoted by S_m^u. The clustering is done layer by layer. At layer 1, a node with computation amount A_i is a cluster by itself with parameters $(1, A_i, 0, 0, 0)$ for SIMD type or $(1, A_i, 0, 0, 1)$ for MIMD type. Then for other layers, the nodes are clustered as follows. If a node is a join-node, we first embed it onto one of its parent nodes that has the largest weighted edge connecting to this join-node. If multiple parent nodes have edges with the same largest weight, we randomly select one of them. When a node with a computation amount A is to be embedded to S_m^u, then these parameters are updated to σS_m^u, $\delta S_m^u + A_i$, ΠS_m^u, πS_m^u, and ρS_m^u. We then merge all its parent nodes into a new cluster denoted by S_1^{u+1}. This is shown in Figure 4, where a join-node at layer $(u+1)$ with computation amount A has n parent nodes $S_1^u, S_2^u, \cdots, S_n^u$ at layer u. The communication amount between the join-node and one of its parent nodes S_i^u is denoted by D_i, where $1 \leq i \leq n$. Also, $D_1 = \max_{1 \leq i \leq n} D_i$. The new cluster S_1^{u+1} is generated by embedding the join-node to S_1^u and merging it with all the other parent nodes. The first four parameters of S_1^{u+1} can be computed as follows.

$$\sigma S_1^{u+1} = \sum_{i=1}^{n} \sigma S_i^u \quad (1)$$

$$\delta S_1^{u+1} = max(\delta S_1^u + A, \delta S_2^u, \cdots, \delta S_n^u) \quad (2)$$

$$\Pi S_1^{u+1} = \sum_{i=1}^{n} (\Pi S_i^u + D_i) - D_1 \quad (3)$$

$$\pi S_1^{u+1} = \frac{\sum_{i=2}^{n} D_i}{n-1} \quad (4)$$

If a node is a fork-node, we will embed one of its child nodes to this fork-node. The child node is selected so that it has the largest weighted edge connecting to the fork-node. If multiple child nodes have edges with the same largest weight, we randomly select one of them. We then merge the rest of the child nodes with the fork-node into a new cluster. As shown in Figure 5, a fork-node S_1^u at layer u has n child nodes at layer $(u+1)$. These child nodes have computation amounts $A_1, A_2, \cdots, A_n$, and the communication amounts between the fork-node and each of them are $D_1, D_2, \cdots, D_n$, respectively. Similar to the case of join-node, $D_1 = \max_{1 \leq i \leq n} D_i$. Then the node with the computation amount A_1 is embedded to the fork-node before we merge the fork-node with all the other child nodes to generate the new cluster S_1^{u+1}. The first four parameters of S_1^{u+1} is then computed as follows.

$$\sigma S_1^{u+1} = \max(\sigma S_1^u, n-1) + 1 \quad (5)$$

$$\delta S_1^{u+1} = \max(\delta S_1^u + A_1, A_2, \cdots, A_n) \quad (6)$$

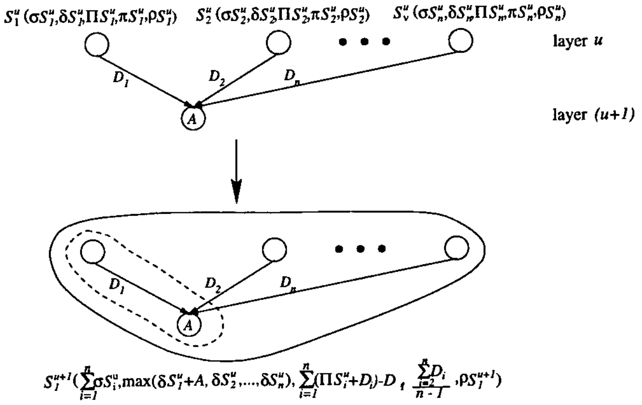
$$\Pi S_1^{u+1} = \Pi S_1^u + \sum_{i=2}^{n} D_i \quad (7)$$

$$\pi S_1^{u+1} = \frac{\sum_{i=2}^{n} D_i}{n-1} \quad (8)$$

For both fork and join nodes, the fifth parameter, ρS_m^u, is determined as follows. As an MIMD cluster is merged with an SIMD or MIMD cluster, the computation type of the new generated cluster is MIMD. When two SIMD clusters are merged then the computation type of the new cluster is decided by their computational form (addition, subtraction, multiplication, etc.). If the two SIMD clusters have exact the same computation form then the computational type of the new cluster is SIMD, otherwise, it is MIMD. We denote the computation form of S_m^u by $CF(S_m^u)$. Then the computational type of a new cluster S_m^u generated from embedding or merging n clusters, $S_1^u, S_2^u, \cdots, S_n^u$, can be formulated as follows.

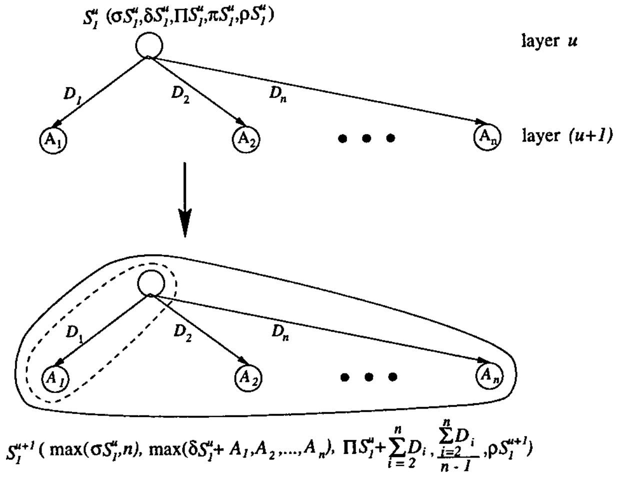
$$\rho S_m^u = \begin{cases} 0 & \text{if } (\rho S_i^u = 0, \text{ for all } i) \text{ and } (CF(S_1^u) = \\ & CF(S_2^u) = \cdots = CF(S_n^u)) \\ 1 & \text{otherwise} \end{cases} \quad (9)$$

Note that since our task graphs are independent of any system graphs (unlike [17, 15, 18]), they do not contain the information about computation time and communication delay. Therefore, we can only embed one node into another as part of clustering for reducing communication overhead. The embedding of multiple nodes onto one node is done as part of the mapping, as explained in the next section.



$S_1^u(\sigma S_1^u, \delta S_1^u, \Pi S_1^u, \pi S_1^u, \rho S_1^u)$
$S_2^u(\sigma S_2^u, \delta S_2^u, \Pi S_2^u, \pi S_2^u, \rho S_2^u)$
$S_v^u(\sigma S_n^u, \delta S_n^u, \Pi S_m^u, \pi S_n^u, \rho S_n^u)$
layer u
D_1
D_2
D_n
A
layer $(u+1)$
$S_1^{u+1}(\sum_{i=1}^{n}\sigma S_i^u, \max(\delta S_1^u + A, \delta S_2^u, ..., \delta S_n^u), \sum_{i=1}^{n}(\Pi S_i^u + D_i) - D_1, \frac{\sum_{i=2}^{n} D_i}{n-1}, \rho S_1^{u+1})$

Figure 4: Clustering on a join-node: a general case.



$S_1^u(\sigma S_1^u, \delta S_1^u, \Pi S_1^u, \pi S_1^u, \rho S_1^u)$
layer u
D_1
D_2
D_n
A_1
A_2
A_n
layer $(u+1)$
$S_1^{u+1}(\max(\sigma S_1^u, n), \max(\delta S_1^u + A_1, A_2, ..., A_n), \Pi S_1^u + \sum_{i=2}^{n} D_i, \frac{\sum_{i=2}^{n} D_i}{n-1}, \rho S_1^{u+1})$

Figure 5: Clustering on a fork-node: a general case.

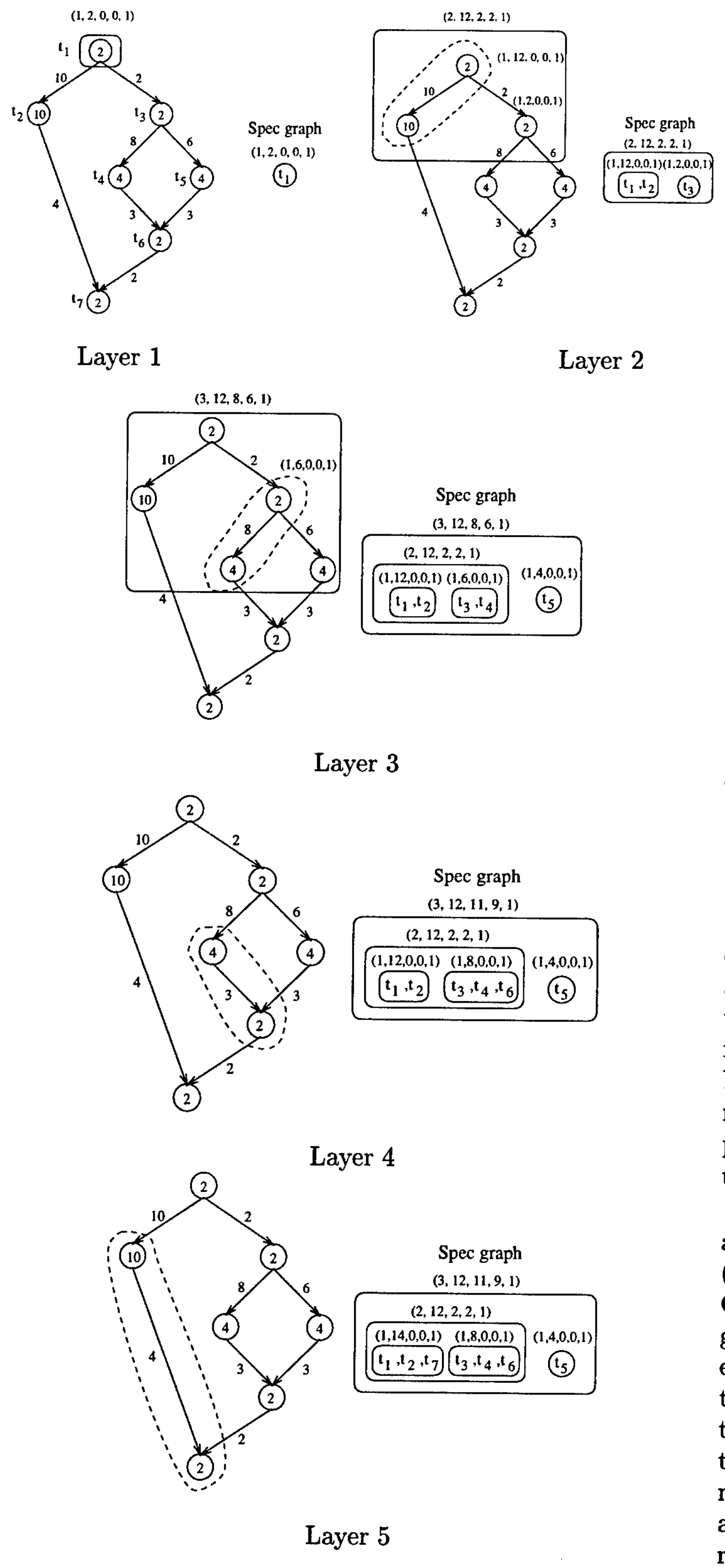


Figure 6: A task graph and steps for obtaining the Spec graph.

The time complexity of the CNDG algorithm is bounded by the number of edges in the task graph, which is $O(|E_t|)$. For the worst case, we have an upper bound for this algorithm, that is, $O(M^2)$, where M is the number of nodes. However, note that most graphs are not completely connected, therefore, in practice, the time complexity of this algorithm will be $O(M)$ if the number of edges is proportional to the number of nodes. To illustrate this algorithm, consider the task graph of seven modules and its Spec graph, as shown in Figure 6. Each module is labeled with its computation amount and each edge is labeled with the amount of data communication. The Spec graph is constructed by embedding/merging the clusters layer by layer and is a multi-layer clustered graph as shown.

3.2 Clustering Undirected System Graphs

A parallel system that can be modeled as an undirected system graph $G_p(V_p, E_p)$. In G_p, $V_p = \{p_1, ..., p_N\}$ is the set of processors forming the underlying architecture, while E_p is the set of edges representing the interconnection topology of the parallel system. We assume that the connections between adjacent processors are bidirectional. Therefore, an edge (p_i, p_j) represents that there is a direct connection between processor p_i and p_j. The computational speed of processor p_i is denoted by B_i, and the communication bandwidth between two processors p_i and p_j is denoted by C_{ij}. The transmission rate is a function of the communication bandwidth between p_i and p_j and the node latencies at p_i and p_j. Both the computational speeds of different processors and the transmission rates of different communication links may be nonuniform. This makes the Cluster-M approach more general than approaches such as PYRROS, Hypertool, and PARSA, which assume fully connected uniform systems.

Similar to Spec clusters, the nth Rep cluster at layer v, R_n^v, is associated with the quintuple $(\sigma R_n^v, \delta R_n^v, \Pi R_n^v, \pi R_n^v, \rho R_n^v)$ defined as part of the Cluster-M model in Section 2.3. To construct a Rep graph from an undirected system graph, initially, every node with computation speed of B_i forms a cluster by itself with parameters (1, B_i, 0, 0, 1), assuming that these nodes are all MIMD type. Then clusters that are completely connected are merged to form a new cluster, and the parameters of the new cluster are calculated, as explained below. This process is repeated until no further merging is possible. Three clusters R_x^v, R_y^v, and R_z^v are completely connected if R_x^v contains a node p_x, R_y^v contains a node p_y, and R_z^v contains a node p_z, so that nodes p_x, p_y, and p_z

form a clique. This definition can be extended for N completed connected clusters. To calculate the values of the first four parameters for a new cluster, consider a new cluster R_n^{v+1}, which is generated at layer $(v+1)$ by merging N completely connected clusters $R_1^v, R_2^v, \cdots, R_N^v$ at layer v. Then the values of σR_n^{v+1} and δR_n^{v+1} can be easily computed as follows.

$$\sigma R_n^{v+1} = \sum_{i=1}^{N} \sigma R_i^v \tag{10}$$

$$\delta R_n^{v+1} = \frac{\sum_{i=1}^{N} \sigma R_i^v \delta R_i^v}{\sigma R_n^{v+1}} = \frac{\sum_{i=1}^{N} \sigma R_i^v \delta R_i^v}{\sum_{i=1}^{N} \sigma R_i^v} \tag{11}$$

We denote the transmission rate between R_i^v and R_j^v to be C_{ij}^v, which is defined as the sum of the transmission rate (as a function of communication bandwidth and switching latency) of each pair of processors (sub-clusters) p_i and p_j such that p_i is in R_i^v and p_j is in R_j^v, that is, $C_{ij}^v = \sum_{p_i \in R_i^v, p_j \in R_j^v} C_{ij}$. Then ΠR_n^{v+1} and πR_n^{v+1} can be calculated as follows.

$$\Pi R_n^{v+1} = \sum_{i=1}^{N} \Pi R_i^v + \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} C_{ij}^v \tag{12}$$

$$\pi R_n^{v+1} = \frac{\sum_{i=1}^{N-1} \sum_{j=i+1}^{N} C_{ij}^v}{\frac{N(N-1)}{2}} = \frac{2(\sum_{i=1}^{N-1} \sum_{j=i+1}^{N} C_{ij}^v)}{N(N-1)} \tag{13}$$

The fifth parameter, ρR_n^{v+1}, is computed per (9).

The algorithm for clustering undirected graphs is shown in Figure 7. Instead of using an optimal algorithm for finding cliques, we use a heuristic so that, for every cluster, we examine the set of edges connected to it in the following manner. The edges are sorted in descending order based on the value of C_{ij}. The edges are then examined one at a time from this list. If more than one of the edges have the same weight, then an arbitrary one is selected. A simple example is shown in Figure 8.

We now analyze the running time of this implementation. For each layer, we first sort all the edges between clusters that take $O(|E_p| \log |E_p|)$, where $|E_p|$ is the number of edges in the system graph. Then, we keep merging clusters into the next layers. Suppose at a certain layer, there are m clusters $c_1, \cdots, c_m$. The time for finding cliques among these clusters is at most $m \times m \leq N^2$, where N is the number of processors in the system graph. The most number of layers there can be is $N-1$. Therefore the total time complexity of this algorithm is $O(N(|E_p| \log |E_p| + N^2))$.

```
Algorithm CNUG
for all nodes $p_i$ do
begin Make a cluster for $p_i$ at clustering layer 1
          Set the parameters of the cluster to be (1, $B_i$, 0, 0)
end
Set cluster layer to be 1
while there is at least one edge linking two clusters do
begin Sort all edges linking any two clusters
          while sorted edge list is not empty, do
          begin Take the first edge $(c_i, c_j)$ from sorted edge list
                    Delete the edge from the list
                    Merge $c_i$ and $c_j$ into cluster $c'$ at next layer
                    Calculate the parameters of $c'$
                    Delete clusters $c_i$ and $c_j$ from current layer
                    for each edge $(c_x, c_y)$ in sorted edge list
                    if $c_x$ is a sub-cluster of $c'$ and
                    $c_y$ is not a sub-cluster of any cluster and
                    $c_y$ is connected to all other sub-clusters of $c'$, then
                    begin Merge $c_y$ into $c'$
                              Recalculate the parameters of $c'$
                              Delete $(c_x, c_y)$ from edge list
                    end
                    else if $c_x$ and $c_y$ are sub-clusters of
                    two different clusters at next layer, then
                    begin Add the weight of $(c_x, c_y)$ to
                              the edge between the two super-clusters
                              Delete $(c_x, c_y)$ from edge list
                    end
          end
          Increment clustering layer by 1
end
```

Figure 7: Clustering Nonuniform Undirected Graphs (CNUG) algorithm.

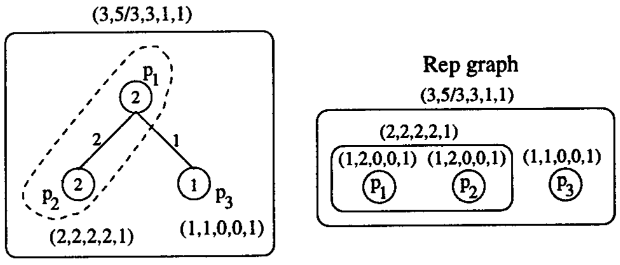


Figure 8: A nonuniform system graph and its Rep graph.

Consider the worst case, where the system graph is completely connected (i.e., $|E_p| = O(N^2)$), then the time complexity of this algorithm will be $O(N^3 \log N)$. Note that most system graphs are not completed connected. Therefore, in practice the time complexity of this algorithm will be $O(N^3)$ if the number of edges is proportional to the number of nodes.

4 Cluster-M Mapping Algorithm

A Spec graph and a Rep graph can be generated directly from a given task graph and system graph, using the clustering algorithms presented in the previous section. Given a Spec graph and a Rep graph, this section presents an efficient mapping algorithm that produces a suboptimal matching of the two graphs in $O(MP)$ time, where $P = \max(M, N)$. Note that the mapping algorithm maps the Spec graph one layer at a time as explained in Section 2.2. Every layer of the Spec graph represents a computational step in which a number of concurrent computations are represented by a number of Spec clusters. These clusters are formed by tracing the data dependency of other subcomputations from a previous step. We are interested in mapping the Spec clusters at each layer to the appropriate Rep clusters. In the following, we first present a set of preliminaries and then give a high-level description of the mapping algorithm. In Section 4.3, a few examples are given to illustrate the mapping algorithm.

4.1 Preliminaries

We first define the mapping function $f_m : V_t \overset{\text{onto}}{\longrightarrow} V_p$. Following the precedence constraints and the computation and communication requirements of the original task graph, a schedule can be obtained by assigning each task module t_i to the processor $f_m(t_i)$. We assume that the communication time for a task graph edge (t_i, t_j) is equal to $\sum_{(p_x,p_y)\in \text{path}(f_m(t_i),f_m(t_j))} \frac{D_{t_i t_j}}{C_{xy}}$, where $\text{path}(p_i, p_j)$ is the shortest path between processor p_i and p_j.

A schedule can be illustrated with a Gantt chart that consists of a list of all processors and a list of all task modules allocated to each of the processors ordered by their execution time [7]. We define the total execution time of a schedule, T_m, to be the latest finishing computation time of the last scheduled task module on any processor. Obviously, T_m is equal to the total execution time of a given task on a given system. As we consider the shortest execution time of a given task on a system to be the ultimate goal in scheduling, we take T_m as our measure of quality to scale how good a mapping is.

4.2 The Algorithm

A detailed description of the mapping algorithm is presented in Figure 9. In the following, we give an overview of the algorithm. The mapping is done recursively at each clustering layer, where we try to find the best matching between Spec clusters and Rep clusters. Assume that at a certain step of mapping, m Spec clusters of layer u, $S_1^u, S_2^u, \cdots, S_m^u$, are to be mapped onto n Rep clusters of layer v, $R_1^v, R_2^v, \cdots, R_n^v$. We denote the estimated total execution time of mapping the Spec cluster S_i^u onto the Rep cluster R_j^v by $\tau(S_i^u, R_j^v)$, which includes computation time and communication time. The total computation amount of S_i^u is estimated to be $\sigma S_i^u \times \delta S_i^u$, and the total computation power of R_j^v can be calculated as $\sigma R_j^v \times \delta R_j^v$. Therefore, the computation time for executing S_i^u on R_j^v is estimated to be $(\sigma S_i^u \times \delta S_i^u)/(\sigma R_j^v \times \delta R_j^v)$. Similarly, the total communication requirement of S_i^u is ΠS_i^u and the total communication capacity of R_j^v is ΠR_j^v, hence the estimated communication time for mapping S_i^u on R_j^v will be $\Pi S_i^u / \Pi R_j^v$. A slow-down factor, d, is defined that indicates the factor of slow down due to mismatch of the computation type between S_i^u and R_j^v. This leads to an estimated execution time in (14). Note that the estimated execution time does not take into consideration the memory requirements of a given problem and the memory space available in the underlying organization. This is mainly due to the fact that the model does not contain any parameters for memory size requirements and availabilities.

$$\tau(S_i^u, R_j^v) = d \times \frac{\sigma S_i^u \times \delta S_i^u}{\sigma R_j^v \times \delta R_j^v} + \frac{\Pi S_i^u}{\Pi R_j^v},$$
$$d = \begin{cases} \sigma S_i^u & \text{if } \rho S_i^u = 1 \text{ and } \rho R_j^v = 0 \\ 1 & \text{otherwise} \end{cases} \tag{14}$$

Then the mapping process at each layer can be viewed as an optimization problem as follows.

$$Min \sum_{i=1}^{m} \tau(S_i^u, f_m(S_i^u)) \tag{15}$$

The time complexity of finding an optimal solution to the above formula can be costly [10]. Therefore, we propose the following greedy algorithm for finding a near-optimal solution to the formula for each layer. In this greedy algorithm, we assume that all the computations are MIMD. Therefore, we only deal with four of the five parameters in the process. The greedy algorithm continues as follows. First, the Spec and Rep clusters are sorted in descending order with respect to the order of the four parameters (σ, δ, Π,

π). For example, Spec clusters with larger sizes are sorted before those with smaller sizes, and for Spec clusters with the same size, those with larger amount of sequential computation are sorted first.

Secondly, we compute a reduction factor denoted by $f_{(u,v)}$, which is the ratio of the total size of the Rep clusters over the total size of the Spec clusters and is used to estimate how many computation nodes to share a processor. This is essential for mapping task graphs of size M onto system graphs of size N, where $M > N$. The value of $f_{(u,v)}$ is computed as:

$$f_{(u,v)} = \frac{\sum_{j=1}^{n} \sigma R_j^v}{\sum_{i=1}^{m} \sigma S_i^u} \tag{16}$$

Third, we map each of the Spec clusters S_i^u, $1 \leq i \leq m$, as follows. We first search for a Rep cluster R_j^v, $1 \leq j \leq n$, with the best matched size, that is, closest to $f_{(u,v)} \times \sigma S_i^u$. Therefore, we try to minimize the function in Equation (17). If multiple Rep clusters with the matching size are found, we select the one with the minimum estimated execution time. If no Rep cluster with a matching size can be found for a Spec cluster, we either merge or split (unmerge) Rep clusters until a matching Rep cluster is found.

$$|f_m| = \sum_{i=1}^{m} |f_{(u,v)} \times \sigma S_i^u - \sigma[f_m(S_i^u)]| \tag{17}$$

Finally, for every matched pair of the Spec and Rep clusters, we do the following to embed communication intensive nodes together. This is similar to the clustering process in [17, 15, 18]. However, in this chapter, we only do it in the mapping step so that the clustering of the task graph is kept independent of the system graph, as described in the previous section. Assume that a Spec cluster S_i^u having k subclusters, $S_1^{u-1}, S_2^{u-1}, \cdots, S_k^{u-1}$, is mapped to a Rep cluster R_j^v. If the communication overhead for processing the subclusters in parallel is greater than the computation overhead for processing the subclusters sequentially, then we embed all subclusters into one subcluster having the largest size so that they will be executed sequentially. We then calculate the parameter quadruple for the new cluster. In Inequality (18), $\pi S_i^u / \pi R_j^v$ is the communication time if the subclusters are executed in parallel and

$$\frac{1}{f_{(u,v)}} \times \frac{\min(\sigma S_1^{u-1} \delta S_1^{u-1}, \sigma S_2^{u-1} \delta S_2^{u-1}, \cdots, \sigma S_k^{u-1} \delta S_k^{u-1})}{\delta R_j^v}$$

is the computation time for executing the subclusters sequentially on R_j^v. The embedded cluster is inserted back in the proper position in the sorted list of Spec clusters for mapping, and the matching process is repeated for the remaining Spec clusters in the list. If no embedding is necessary, then the mapping of this Spec cluster onto a Rep cluster is done for this layer, and, therefore, this Spec cluster is removed from the list.

$$\frac{\pi S_i^u}{\pi R_j^v} > \frac{1}{f_{(u,v)}} \times \frac{\min(\sigma S_1^{u-1} \delta S_1^{u-1}, \sigma S_2^{u-1} \delta S_2^{u-1}, \cdots, \sigma S_k^{u-1} \delta S_k^{u-1})}{\delta R_j^v} \tag{18}$$

In the above mapping algorithm, the worst case of the time complexity of the mapping algorithm at layer i occurs in one of the following two cases. In case 1, for each Spec cluster, all the remaining Rep clusters have the matching size, thus (14) is used to select the best Rep cluster. In case 2, for each Spec cluster, no Rep cluster of matching size is found, thus Rep clusters are merged or split recursively until a Rep cluster of matching size is obtained. Suppose the number of Spec clusters at layer i is K_i. In both cases described above, or in any combination of the two cases, it takes $O(K_i N)$ time to find the best matches for all K_i Spec clusters, as the total number of clusters in the Rep graph is $O(N)$, where N is the number of processors. For each pair of matching Spec and Rep clusters, if Inequality (18) is satisfied, then an extra $O(M)$ time for embedding will be needed. The total number of Spec clusters is $O(M)$, that is, $\sum_i K_i = O(M)$, where M is the number of nodes in original task graph. Therefore, the total time complexity of this mapping algorithm is $\sum_i (K_i N + M) = O(MN) + O(M^2) = O(MP)$, where $P = \max(M, N)$.

4.3 A Mapping Example

In Section 3, we constructed a Spec graph and a Rep graph from the original task graph and system graph, as shown in Figure 6 and 8. Figure 10 shows the snapshot of the mapping process. Figure 11 shows the final schedule obtained from the above mapping by following the data and operational precedence of the task graph. As shown in the Gantt chart, $T_m = 10$.

To show that the same task graph can be mapped onto various system graphs, three different system graphs are chosen and shown in Figure 12. Figure 12(a) is the same task graph as shown in Figure 6. Figure 12(b) shows a uniform, fully connected system graph and its clustering. The computation speed of each processor and communication bandwidth of each communication link are equal to 2. The result of Cluster-M mapping onto this graph is shown in Figure 12(c). In Figure 12(d), the system is fully connected with computation speed of 1 at each processor, but the communication bandwidths are nonuniform.

```
Mapping Algorithm
for each layer of Spec graph do
begin
        Sort all Spec clusters at top layer in descending order of
        $\sigma S_i^u$, $\delta S_i^u$, $\Pi S_i^u$, and $\pi S_i^u$.
        Sort all Rep clusters at top layer in descending order of
        $\sigma R_j^v$ $\delta R_j^v$, $\Pi R_j^v$, and $\pi R_j^v$.
        Calculate $f_{(u,v)}$, if $f_{(u,v)} > 1$, let $f_{(u,v)} = 1$.
        Calculate the required size of the Rep cluster matching
        $S_i^u$ to be $f_{(u,v)} \times \sigma S_i^u$
        for each Spec cluster at top layer sorted list, do
        begin if the cluster has only one sub-cluster, then
                Go to a lower layer containing multiple or no
                sub-clusters
                if at least a Rep cluster of required size is found, then
                begin Select the Rep cluster of required size with
                        minimum estimated execution time according
                        to Equation (14)
                        Match the Spec cluster to the Rep cluster
                        Delete the Spec and Rep clusters from Spec
                        and Rep lists
                end
        end
        for each unmatched Spec cluster, do
        begin if size of the first Rep cluster > than required size
                begin Split the Rep cluster into two parts with one
                        part of the required size
                        Match the Spec cluster to this part
                        Insert the other part to proper position of the
                        sorted Rep cluster list
                end
                else begin
                        Merge Rep clusters until sum of sizes $\geq$ the
                        required size
                        if =, then
                        Match the Spec cluster to merged Rep cluster
                        else
                        begin Split the merged Rep cluster into two
                                parts with one of required size
                                match the Spec cluster to this part
                                Insert the other part to sorted Rep list
                        end
                end
        end
        for each matching pair of Spec cluster and Rep cluster, do
        begin if the Rep cluster contains only one processor, then
                Map all modules in the Spec cluster to the processor
                else if Inequality (18) is satisfied, then
                        begin Select the sub-cluster of the Spec cluster
                                with the largest size
                                Embed the nodes of other sub-clusters to
                                connected nodes of selected sub-cluster
                                Calculate parameters of the new cluster
                                Insert it into the sorted Spec cluster list
                        end
                        else
                        begin Delete the Spec cluster from Spec list
                                Delete the Rep cluster from Rep list
                                Go to sub-clusters of the Spec and Rep
                                clusters (so they are pushed to top layer)
                                Call the same mapping algorithm for
                                these clusters
                        end
        end
end
```

Figure 9: Mapping algorithm.

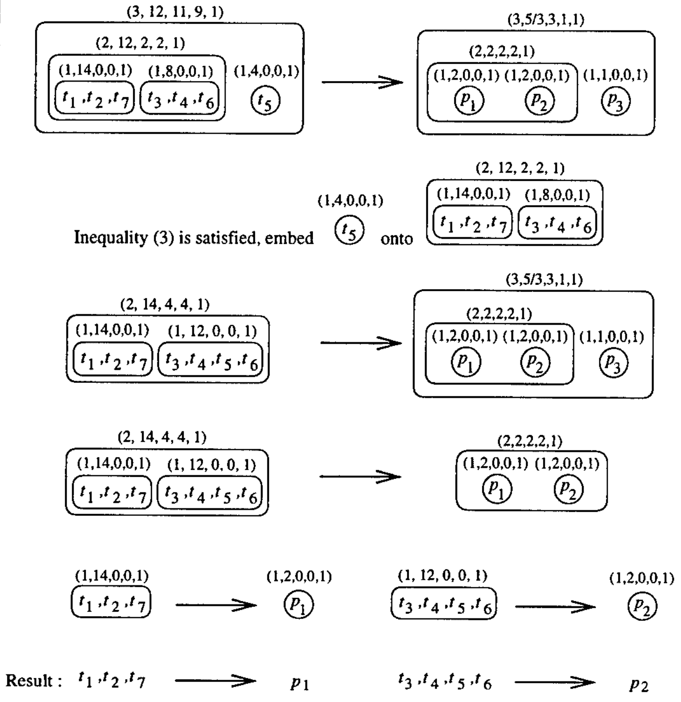


Figure 10: A mapping example.

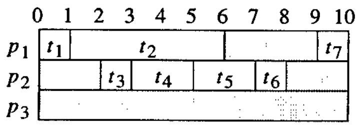


Figure 11: Gantt chart of the obtained schedule.

In this case, the Cluster-M algorithm distributes the task modules to all three processors, as shown in Figure 12(e), to utilize the relatively high communication bandwidth available. If the system is fully connected with uniform communication bandwidth and nonuniform computation speeds as shown in Figure 12(f), Cluster-M mapping algorithm maps all the task modules onto the processor with the highest speed to avoid the relatively expensive communication cost. This is shown in Figure 12(g). For more examples, see the full version of the paper [5].

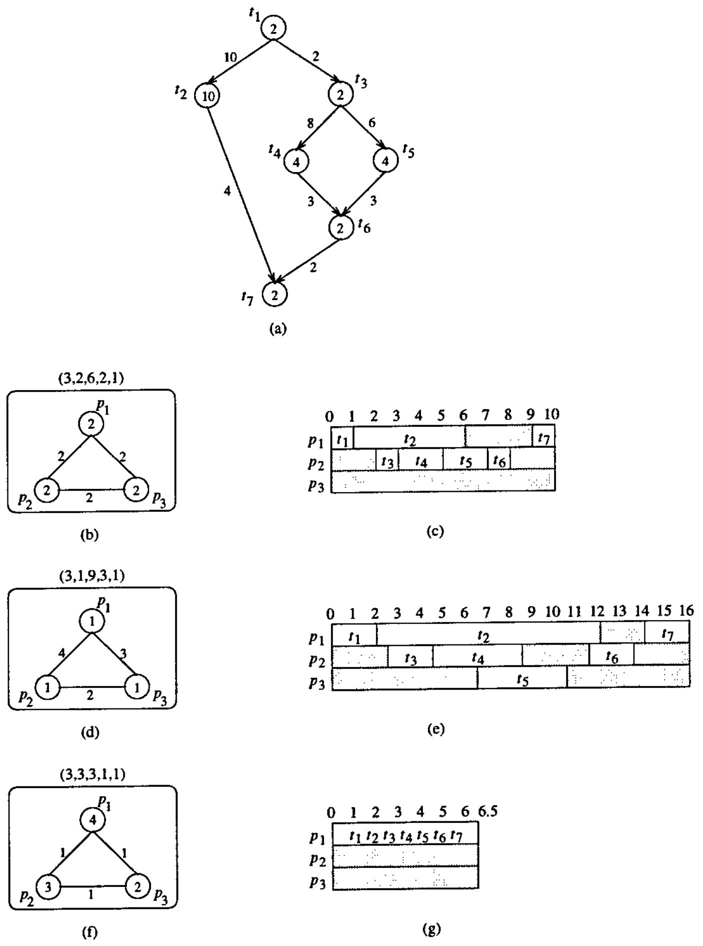


Figure 12: Mappings on different system graphs.

5 Comparison Results

In the full version of the paper [5], we present a set of experimental results we have obtained in comparing our algorithm with other leading techniques. The comparisons presented in the full version of the paper [5], are classified into two categories: (1) mapping arbitrary nonuniform task graphs onto arbitrary nonuniform system graphs, and (2) mapping arbitrary nonuniform task graphs onto uniform fully connected system graphs. We first present the comparison for the first category and then the second one. The following three criteria are used for comparing the performance of our algorithm with other leading techniques: (1) the total time complexity of executing the mapping algorithm, T_c; (2) the total execution time of the generated mappings, T_m; and (3) the number of processors used, N_m. From (2) and (3), we can obtain the speedup $S_m = \frac{T_s}{T_m}$ and efficiency $\eta = \frac{S_m}{N_m}$, where T_s is the sequential execution time of the task. In this paper, we present a summary of our results for mapping arbitrary nonuniform task graphs onto arbitrary nonuniform system graphs, only.

The mapping techniques in this category include El-Rewini and Lewis' mapping heuristic (MH) [6] and Lo's Max Flow/Min Cut (MFMC) algorithm [13]. To the best of our knowledge, they are the only known efficient mapping techniques that can map arbitrary nonuniform task graphs onto arbitrary nonuniform system graphs in polynomial time. The experimental results shown in this section are obtained by running a set of simulations on a SUN SPARCstation 20 workstation, and all running times are measured in second on this machine. The nonuniform task graphs are randomly generated. In the full version of the paper [5], we map these task graphs onto four different nonuniform systems[2]: (1) a randomly gener-

[2]For comparing against MFMC, we use three system configurations, system (2)-(4). The time complexity of MFMC in practice is too high and for the first system configuration, each experiment takes several days. For more detail, see Section 5.1.2.

ated system graph with 100 nodes, where the computation speed of the nodes and the communication bandwidth of the edges range from 1 to 5, (2) a randomly generated system graph with five nodes, where the computation speed of the nodes and the communication bandwidth of the edges range from 1 to 5, (3) a completely connected system graph with four nodes as shown in Figure 13, and (4) a hypercube with eight nodes as shown in Figure 14. In the following, we present a summary of our comparison results using system configuration 2, system configuration 3, and system configuration 4. As shown in Tables 1, 2, and 3, Cluster-M produces similarly good results but in significantly less time.

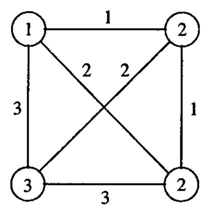


Figure 13: System (2): A completedly connected system.

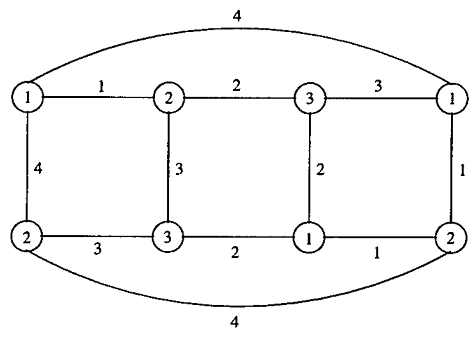


Figure 14: System (3): A hypercube system.

6 Conclusion

In this paper, we have presented a generic algorithm for mapping non-uniform arbitrary task graphs onto non-uniform arbitrary system graphs. Given a task graph and system graph, we have shown efficient techniques for producing two clustered graphs called Spec graph and Rep graph, which are the input to the mapping algorithm. The clustering is done only once for a given task graph (system graph) independent of any system graphs (task graphs). It is a machine-independent (application-independent) clustering and is not repeated for different mappings. The complexity of the mapping algorithm is $O(MP)$, where M is the number of task modules, N is the number of processors, and $P = \max(M, N)$. We presented our experimental results in comparing the performance of our generic algorithm with other leading ones. We have shown that we can obtain similar results in less time. The presented mapping algorithm can be efficiently integrated as part of portable parallel programming tools.

References

[1] F. Berman and L. Snyder. On mapping parallel algorithms into parallel architectures. *Journal of Parallel and Distributed Computing*, pages 439–458, April 1987.

[2] S. H. Bokhari. On the mapping problem. *IEEE Trans. on Computers*, c-30(3):207–214, March 1981.

[3] S. Chen and M. Eshaghian. A fast recursive mapping algorithm. *Concurrency: Practice and Experience*, 7(5):391–409, August 1995.

[4] S. Chen, M. M. Eshaghian, R. F. Freund, J. L. Potter, and Y. Wu. Evaluation of two programming paradigms for heterogeneous computing. *Journal of Parallel and Distributed Computing*, 31(1):41–55, November 1995.

[5] S. Chen, M. M. Eshaghian, and Y. Wu. Mapping arbitrary nonuniform task graphs onto arbitrary nonuniform system graphs. *in revision for possible publication at IEEE Transactions on Parallel and Distributed Systems*, 1996.

[6] H. El-Rewini and T. G. Lewis. Scheduling parallel program tasks onto arbitrary target machines. *Journal of Parallel and Distributed Computing*, pages 138–153, September 1990.

[7] H. El-Rewini, T. G. Lewis, and H. H. Ali. *Task Scheduling in Parallel and Distributed Systems*. Prentice Hall, Englewood Cliffs, NJ, 1994.

[8] F. Ercal, J. Ramanujam, and P. Sadayappan. Task allocation onto a hypercube by recursive mincut bipartitioning. *Journal of Parallel and Distributed Computing*, pages 33–44, October 1990.

[9] M. Eshaghian and M. Shaaban. Cluster-M parallel programming paradigm. *International Journal of High Speed Computing*, 6(2):287–309, June 1994.

[10] M. A. Driscoll et al. Sensitivity analysis and mapping programs to parallel architectures. In *1991 International Conference on Parallel Processing*, volume II, pages 272–273, August 1991.

[11] I. Foster and S. Tuecke. Parallel programming with PCN. Technical report, Argonne National Laboratory, University of Chicago, January 1993.

[12] S. Lee and J. K. Aggarwal. A mapping strategy for parallel processing. *IEEE Trans. on Computers*, 36:433–442, April 1987.

[13] V. M. Lo. Heuristic algorithms for task assignment in distributed systems. *IEEE Trans. on Computers*, C-37(11):1384–1397, November 1988.

[14] C. McCreary and H. Gill. Automatic determination of grain size for efficient parallel processing. *Communications of ACM*, 32(9):1073–1078, September 1989.

[15] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors.* The MIT Press, Cambridge, MA, 1989.

[16] C. Shen and W. Tsai. A graph matching approach to optimal task assignment in distributed computing systems using a minmax criterion. *IEEE Trans. on Computers*, c-34(3):197–203, March 1985.

[17] M. Y. Wu and D. Gajski. Hypertool: A programming aid for message-passing systems. *IEEE Trans. on Parallel and Distributed Systems*, 1(3):101–119, 1990.

[18] T. Yang and A. Gerasoulis. DSC: Scheduling parallel tasks on an unbounded number of processors. *IEEE Trans. on Parallel and Distributed Systems*, 5(9):951–967, September 1994.

Table 1: Comparison of Cluster-M, MFMC, and MH on system (2).

Size of Random Graph	T_s	Cluster-M [$O(MN)$]			MFMC [$O(M^4N \log M)$]			MH [$O(M^2N^3)$]		
		T_m	S_m	T_c	T_m	S_m	T_c	T_m	S_m	T_c
10	27	7.93	3.40	0.01	8.10	3.33	0.8	11.13	2.43	0.1
12	33	8.23	4.00	0.01	16.85	1.96	4.1	9.03	3.65	0.1
14	45	8.20	5.49	0.01	18.25	2.47	23.9	16.87	2.67	0.1
16	46	12.50	3.68	0.01	23.70	1.94	109.1	14.05	3.27	0.1
18	54	20.33	2.66	0.01	27.90	1.94	556.3	19.98	2.70	0.1
20	64	19.00	3.37	0.01	34.70	1.84	2762.3	26.33	2.43	0.1
22	60	23.40	2.56	0.01	33.20	1.80	13430.0	28.29	2.12	0.1
24	86	16.00	5.38	0.01	39.65	2.17	21323.0	32.75	2.63	0.1

Table 2: Comparison of Cluster-M, MFMC, and MH on system (3).

Size of Random Graph	T_s	Cluster-M [$O(MN)$]			MFMC [$O(M^4N \log M)$]			MH [$O(M^2N^3)$]		
		T_m	S_m	T_c	T_m	S_m	T_c	T_m	S_m	T_c
10	27	9.00	3.00	0.01	15.33	1.76	0.8	17.33	1.56	0.1
12	33	13.50	2.44	0.01	17.83	1.85	3.7	17.00	1.94	0.1
14	45	13.67	3.29	0.01	19.00	2.37	21.8	20.67	2.18	0.1
16	46	21.00	2.19	0.01	22.50	2.04	99.6	20.50	2.24	0.1
18	54	19.33	2.79	0.01	26.83	2.01	503.8	32.00	1.69	0.1
20	64	19.00	3.37	0.01	31.17	2.05	2504.8	33.83	1.89	0.1
22	60	24.50	2.45	0.01	35.83	1.67	13445.3	39.17	1.53	0.1
24	86	26.67	3.23	0.01	39.83	2.16	15225.2	48.17	1.79	0.1

Table 3: Comparison of Cluster-M, MFMC, and MH on system (4).

Size of Random Graph	T_s	Cluster-M [$O(MN)$]			MFMC [$O(M^4N \log M)$]			MH [$O(M^2N^3)$]		
		T_m	S_m	T_c	T_m	S_m	T_c	T_m	S_m	T_c
10	27	9.83	2.75	0.01	18.66	1.45	1.1	17.92	1.51	0.1
12	33	21.33	1.54	0.01	19.33	1.71	5.3	17.08	1.93	0.1
14	45	13.67	3.29	0.01	39.00	1.15	29.3	16.17	2.78	0.1
16	46	21.00	2.19	0.01	45.83	1.00	141.2	25.83	1.78	0.1
18	54	19.33	2.79	0.01	29.50	1.83	715.4	33.58	1.61	0.1
20	64	19.00	3.37	0.01	60.17	1.06	3579.5	44.83	1.43	0.1
22	60	26.00	2.31	0.01	40.83	1.47	17298.8	51.00	1.18	0.2
24	86	26.67	3.23	0.01	71.83	1.20	30081.7	41.17	2.09	0.2

Case Study

Practical Issues in Heterogeneous Processing Systems for Military Applications

Glenn O. Ladd, Jr.
Hughes Aircraft Company, El Segundo, CA, USA

Practical Issues in Heterogeneous Processing Systems for Military Applications

Glenn O. Ladd, Jr.
Hughes Aircraft Company, El Segundo, CA

Abstract

Heterogeneous parallel processing systems have been extensively used in embedded military applications due to their advantages in size, weight, power, and hardware cost. This paper reviews the evolution of some of these systems and discusses design factors and tradeoffs which affect their application. As military systems have become more cost sensitive, and initial development more common than long term production, the use of commercial hardware and software has become more common. The rapid advances of computer technology seem likely to accelerate that trend in the future.

Introduction

This paper was motivated by observations by the author that heterogeneous processing design for many military applications is significantly different than is generally treated in the literature. There are many publications on the types of embedded military applications that are discussed herein, but a review of the practical considerations that have driven the design of fielded military processing systems may be of value to the reader. For our purposes, the term embedded is used in the sense of a focused, mission critical system design as opposed to a system for multiple user programming. The term is not intended to denote anything with respect to physical configuration. This paper reviews the more obvious features of several types of fielded systems, including more recent systems based on parallel processing architectures, sometimes termed embedded high performance computing designs after the work sponsored by DARPA in this area.

Motivation

The heterogeneous processing technical area is very broad, as is well developed in a recent paper by Ekmecic, et al[1]. The heterogeneity of the basic application, with respect to execution mode, is of great importance in developing the algorithms and application design for a specific compute problem. The concerns include detection of fine grain and course grain parallelism, the choice of a specific machine type to run each part of the application, and the allocation of the application code among various numbers of heterogeneous processing nodes. This statement already assumes the concept of parallelism, which one may observe is a consequence of the use of different machine or node types in the system, or of the use of a homogeneous machine on multiple execution modes at different times, i.e., temporal parallelism. The paper cited covers this discussion in some detail.

In this paper, the focus is on the development of a practical heterogeneous processing system for military applications and the tradeoffs that must be made for an optimal system platform solution. The basic design of the applications that will be used is assumed to have been determined, though this is by no means an obvious step in the overall software system or software architecture design. The application design can have a profound effect on the hardware efficiency achieved and the choice of heterogeneous elements to incorporate in the system. The focus here is on the implications of such application designs for the processing system design rather than on the details of application algorithms and partitioning.

The designs of practical embedded military processing systems tend to be driven strongly by considerations of platform system cost and required functionality. This is contrasted with decisions that might be made for a heterogeneous processing system for support of scientific or so-called "grand challenge" applications. A key notion for all embedded military processing systems is resource constraints: size, weight, volume; recurring cost; available memory and numbers of compute nodes and types, leading to throughput, latency, memory, interconnect bandwidth and similar constraints; application development cost; upgrade costs, etc. The notion of resource constraints pervades the entire system design and system application behavior, leading to demanding requirements for all levels of the hardware and software in the embedded processing system, and the hardware and software which supports application development and system integration.

0-8186-7879-8/97 $10.00 © 1997 IEEE

Heterogeneous processing categorization

Reference 1 develops a taxonomy of machine types which, when applied to military systems, reveals that such systems have embraced heterogeneous processing from fairly early designs. It also serves to focus attention on the diversity of heterogeneous processing system designs, and the on the resulting complexity of these systems. The taxonomy is described as follows, with the interpretations used in this paper:

1. SESM: Single Execution Mode/Single Machine Model (Single application execution mode, single processing element/machine design)
2. SEMM: Single Execution Mode/Multiple Machine Model (Single execution mode per processing element, multiple processing element/machine designs)
3. MESM: Multiple Execution Mode/Single Machine Model (Multiple execution modes on any single processing element type, single processing element/machine design)
4. MEMM: Multiple Execution Mode/Multiple Machines (Multiple execution modes, Multiple processing element/machine designs)

The execution mode as interpreted here refers to the type of parallelism exhibited by the application, not the processing node or processing element (PE), and assumes that various execution modes can be implemented through software on a variety of machine models. The machine model refers to the fundamental design of the processing node or element. Examples of different execution modes for applications include vector, scalar, dataflow, systolic, SIMD, MIMD, etc. Examples of PE types, that is, hardware architectures, include general purpose microprocessors, digital signal processing (DSP) microprocessors, systolic array PEs, vector processing machines, and unique digital processing machine designs (common in military systems). Different machine models also include machines which are operated at different clock rates such that applications running on them execute with different performance and timing.

The purpose of Table 1 is to show not only how high a degree of heterogeneity has been used in military processing, but also to show how long it has been a part of military systems. This observation will, of course, come as no surprise to those involved in these systems over the years, but the evolving complexity of some of the recent systems may be of some interest. The most early digital processing systems were uniprocessors which did some signal processing and some control and/or display functions. SESM was the paradigm of examples 1 and 2 in the table. Example 3 is a surprisingly early example of an MEMM system—the digital signal processor did both vector and scalar computations for digital filtering, Kahlman filtering, and control synchronization. The general purpose processor was a unique design of the day used for sensor control and display and post processing on the radar mode applications. This was one of the very earliest programmable digital signal processing systems, and was motivated by the need for tuning the signal processing applications in response to actual flight data and the need to support multiple mode applications within severe hardware constraints. The hardware design was tailored to the application need. At that time, no commercial DSP machines existed, and funding for unique machines was available. Now we see this type of architecture used widely, but the programmable DSP and the GP are commercially available items.

Example 5 in the table represents a generic type of integrated processing system. This system is uniquely designed to support the requirements of the platform. Further in this paper the type of requirements that such a system may need to support are reviewed, but as an example of an embedded heterogeneous processing system, this is of high complexity, yet highly unified in concept. Six machine types are suggested. The lower case n is some multiplier below 10 for practical cases. Each machine type supports one or more execution modes. The nodes might be connected by one or more communication paths including a bus, high bandwidth interconnect such as a switched network, and perhaps even a multiport memory. The system may support several sensors as well as mission management applications and user control console displays. Not only is the system hard real-time, but is also required to be multi-level secure, supporting applications at multiple security levels. This property fortunately supports separation within a complex application program suite which runs on the heterogeneous machine resources. This processing system serves as an example of a type of highly complex heterogeneous, parallel processing system of custom design.

By contrast, current embedded military systems are beginning to be fielded with parallel processing systems that are commercially available from a number of companies. The most compact of these are supplied in the VME format by companies such as Mercury, SKY, and CSPI, while some military platforms are using parallel systems from IBM (SP-2), SGI, Digital, and others. While the IBM and similar systems would be classified as MESM, the Mercury, etc. systems are supplying multiple node types and are being operated as MEMM, as noted in the table.

A final word regarding Table 1 – there are numerous other examples of the types of processing systems shown from multiple military suppliers. The author hopes to be

Table 1. Examples of military heterogeneous processing systems which have been or are planned to be fielded

No.	*Heterog. Arch. Type*	*Architecture Description*	*Computing Commun.*	*Control Commun.*	*Processing Element Types: Typical number used*	*Application, Developer and Approx. Dates*
1	SESM	Bus-Oriented	Multidrop Bus	Same	GP type: 1–3	Typical Airborne, Naval Mission Computers of 70s, 80s
2	SESM	In-line hardwired preprocessor and GP	Unique parallel interfaces	Unique Interfaces	One GP and one preprocessor	Typical radar processor of 70s and early 80s.
3	MEMM	In-line Programmable Signal Processor and GP	Internal to PES	Unique Interfaces	One GP and one Signal Processor	Hughes programmable signal processor for airborne radar, mid-70s.
4	MEMM	Bus and Multi-Port Memory	Multiport Memory	Bus	PES: 1–4 MPM: 4 way 1750A: 1–2	Hughes Aircraft processors for F-14, F-15 and F/A-18 airborne radars of 1980s
5	MEMM	Bus, MPM, and SWN	MPM, SWN	Bus	PES-Radar: 4n PES-Communic.: n PES-EO: n PES-Display/Graphics: n PES-Encryption: n GP: 10n	Integrated avionics or integrated sensor systems processing complex
6	MEMM	Bus, SWN	SWN	Bus and SWN	Intel I860: 32–128 GP: 1–2	Mercury RACE systems, 1995
7	MESM	Commercial High-End Server, Multiprocessor	Various parallel network designs	Usually the network	GP SMP: 1–32 (SMP = 2–4, or up to 8)	Recent parallel commercial servers by DEC, H-P, Sun, SGI, etc.

Glossary for Table 1

PES	PE for digital Signal Processing, unique design.
MD Bus	Multi-drop bus, e.g., VME, PI bus, etc.
GP	General Purpose uniprocessor design, e.g., 1750A, Commodity microprocessors of 80s.
MPM	Multiport memory.
SWN	Switched Network (not differentiated as to type of switching or message protocol).
SMP	Symmetric Multi-Processor.

forgiven for not providing a comprehensive list, and does not intend to imply by omission that the examples cited are necessarily the best that might be cited!

Application requirements issues and drivers

The demands (and opportunities) of military platform requirements have had a high impact on the implementation of embedded heterogeneous processing systems. Table 2 adds information about the application and software requirements for systems shown in Table 1.

The purpose of Table 2 is to add information about application or military platform/system requirements that add complexity to the system design and/or support of embedded heterogeneous systems. The focus is on the complexity of the system software architecture. Note that with example 3, the requirement for low latency, preemptive, deadline scheduled operating systems appeared. This was driven by the fact that sensor control loops began to be handled by the digital processing system. Synchronization of the sensor input with data availability for starting the application was required in some systems.

By the time systems such as example 4 appeared, the use of multiple parallel signal PEs and multiple application programs which could be preempted by the operator resulted in the requirement for support for preemption of running programs in the operating systems (OS). This requirement alone begins to divide the practice of heterogeneous computing in embedded military

Table 2. Comparison of military system application latency and control requirements

No.	*Application, Developer and Approximate Dates*	*Principle System Application*	*Heterog. Arch. Type*	*Application Program Support Reqmnts.*	*Interrupt-Driven Computing Reqmnts.*	*Hard Real-Time Control Reqmnts.*	*Special Run-Time SW Reqmnts.*
1	Typical Airborne, Naval Mission Computers of 70s, 80s	General purpose processing and control	SESM	Single program	NO	NO	Custom RT OS design
2	Typical radar processor of 70s and early 80s.	Single sensor, few modes	SESM	1–3 programs	NO	NO. Synch to sensor front-end	Custom RT OS design
3	Hughes programmable signal processor for airborne radar, mid-70s.	Single sensor, few modes.	MEMM	Multi-program, sensor control loops	YES	YES	Custom RT OS design
4	Hughes Aircraft processors for F-14, F-15 and F/A-18 airborne radars of 1980s	Single sensor, multiple, interrupt driven modes	MEMM	Multi-program, interrupt driven	YES	YES	Custom RT OS design
5	Integrated avionics or integrated sensor systems processing complex	Multiple sensor, multiple, interrupt driven modes per sensor	MEMM	Multi-program, interrupt driven, reconfigurable, fault-tolerant, secure	YES	YES	RT OS design with MLS, fault-toler. support, resource mgmnt.
6	Mercury RACE systems, 1995	Various system applications	MESM	Multi-program, interrupt driven	YES	YES	Custom RT OS design, POSIX interface.
7	Recent parallel commercial servers by DEC, H-P, Sun, SGI, etc.	Military command and control, ground processing stations	MESM	Multiple programs, not hard RT	NO	NO	High performance UNIX OS, commercial middleware
8	Future integrated heterog. parallel processing systems	Fully integrated sensor & C4I systems/ platforms	MESM	Multi-program, interrupt driven, dynamically scaling and repartitioning, reconfigurable, fault-tolerant, secure	YES	YES	RT OS with POSIX API, MLS, fault-toler., high performance resource management, interface to DISA COE.

Glossary for Table 2:

RT OS – Real-Time Operating System

MLS – Multi-Level Security: Separation of applications and files as well as message separation, audit and logging, etc.

DISA COE – Defense Information Security Agency, Common Operating Environment

systems from heterogeneous applications in scientific processing.

Example 5 exhibits the most demanding application and software requirements in this table. Sensor control loops must be supported with low latencies. Applications and files must be protected while supporting hard real-time, preemptive priority interrupt and context switching performance. Special maintenance hardware may be available to report failures at run time, in much the same manner as for commercial mainframes, and run-time software can reconfigure the applications to hot spare modules. Error logging and instrumentation must be supported at run-time. Run-time management of application faults must be supported, with restart. All of these capabilities place high demands on the hardware and software design, as stringent as may be found in today's military systems.

In the past few years, the application of systems like example 6 has become popular because of the high performance per watt-cubic inch-pound-dollar. The Mercury and similar systems provide a high performance real-time OS but with significantly less robust application support than those in example 7. On the other hand, the technical community that grew up with the systems in examples 1–5 are quite able to design and field systems with high efficiency for parallel application code, though at higher cost. Highly complex, multiprogram applications similar to example 5 have not appeared publicly at this time, but may be anticipated. The only barriers are sufficiently capable operating systems and middleware similar to what has been developed for example 5.

With example 7 in the table, the requirement for preemption is usually a soft requirement, in that system operation can be achieved within the capabilities of modern high performance UNIX operating systems, and modified commercial middleware. These systems are not operating with typical commercial client-server performance parameters, however. They do operate with commercial application programming interfaces (API). The commercial parallel server hardware is very cost-effective and offers high value when size/weight/power requirements are not demanding. Typical installations are in ground, ship, and cabin-mounted aircraft environments.

Example 8 suggests what may be expected in future heterogeneous parallel processing systems. The hardware designs to support these systems exists in part in several existing commercial products, but not all in one product. This is discussed further in the conclusion section.

Design trades for heterogeneous systems

While the design of a heterogeneous system will be greatly affected by the nature of the applications and their performance on specific PEs or machine types, the practical requirements play such large role that they cannot be ignored. The figure of merit [Performance ÷ [Watt * Cubic Inch * Pound * Cost ($)]] is always a strong consideration for any embedded military processing system. The impact can cause large differences in the choice of hardware, run-time software, and application design. This section attempts to illustrate these issues in order to highlight their importance.

Table 3 shows three systems configured with up to three different PE or machine types. The prototype for such systems are those processors being supplied commercially by VME-based suppliers such as CSPI, Mercury, SKY, and others. The example PEs are the popular Motorola PowerPC and Analog Devices 21060 devices, which are typically supplied in the configurations shown. Other configurations, such as 9U VME format, are available, and offer other possibilities in a design trade. An assumption is also made as to the replacement of PowerPC code in system 1 with PE-B code in system 2, specifically, that the source lines of code (SLOC) will be the same in either processor, which is a simplification. The performance replacement is as noted in the table, and is confirmed by actual benchmark results. Again, this replacement ratio can be quite different for different applications. The point, however, is that the use of special purpose PEs can significantly increase the FOM defined above. For systems which demand the lowest recurring cost, volume and power, heterogeneous systems are highly advantageous.

The downside of heterogeneous systems is the additional complexity and software cost. Table 4 attempts to demonstrate the impact of software costs on the system design trades. The table assumes an application base of 100,000 SLOC, which is not atypical of such systems, and can be much larger. Software development costs vary widely from place to place but the figure is at least illustrative. As in Table 3, the DSPs replace GP PE code on a one-one basis, and the cost of developing such code is considered to be about 25% higher. The Software Development Environment (SDE) cost is higher for the GP due to it's richness, and that for the DSP is less expensive. Note that DSP SDE costs can ultimately be much higher if the developer chooses to add and support unique tools. Running the numbers shows that that the total expense is about the same for the three example systems, while the volume from Table 3 is much smaller as the DSP nodes are added.

Note, however, that SW costs greatly dominate the total system cost for the first article, validated system. At this point, considerations as to how this cost will be recovered become paramount. If the system is for development only, then the example would tend to support the homogeneous case, because the application should be more portable to upgraded hardware later. If the system is for reasonable production volumes, then the lower recurring cost of system 3 is desired. If the military platform requires minimum size, system 3 is also favored by a large margin. But what of other considerations?

Table 5 presents a list of design factors that affect the choice of a processing system design or supplier. Each of these factors, if not more, will be evaluated by a system architect, or will be a factor in the cost and performance that a supplier can provide. To somewhat better appreciate the effect of architecture layout on application partitioning, reconfiguration, resource management, etc.,

Table 3. Notional illustration of a heterogeneous parallel system design tradeoff

System	*CPU Type*	*Typical example*	*No. of CPUs*	*No. of Modules*	*Cost/ Module, $K*	*Total Module Cost, $K*	*Total Cost, $K*	*Volume = Total Modules*	*Approximate Ratios, $K–Vol*
1	A	PPC 604, 200 MHz	144	36	45	1620	1620	36	1–1
2	A	PPC 604, 200 MHz	96	24	45	1080	1142	25	0.70–0.69
	B	AD21060, 40 MHz	12	1	62	62			
3	A	PPC 604, 200 MHz	48	12	45	540	672	14	0.41–0.39
	B	AD21060, 40 MHz	12	1	62	62			
	C	Like AD21060, different ISA	12	1	62	62			

NOTES: Assumes 12 AD21060s are about 4X faster than 4 PPCs (127 MOPS/SHARC, 91 MOPS/PPC on FFTs).

Table 4. Notional illustration of a heterogeneous parallel system design tradeoff—software costs added

System	*CPU Type*	*Typical Example*	*No. of Modules*	*No. of Applic. SLOCs, K*	*SW Devel. Cost, $K (2)*	*SDE Fixed Cost, $K*	*Total SW Costs, $M*	*Total HW Cost, $M*	*Grand Total Cost, $M*	*Approx. Ratios, Total $K–Vol*
1	A	PPC 604, 200 MHz	36	100(1)	10,000	1000	10.1	1.620	11.7	1–1
2	A	PPC 604, 200 MHz	24	90	9,000	1000	11.5	1.142	12.6	1.1–0.69
	B	AD21060, 40 MHz	1	10	1,250	300				
3	A	PPC 604, 200 MHz	12	80	8,000	1000	12.1	0.672	12.7	1.1–0.39
	B	AD21060, 40 MHz	1	10	1,250	300				
	C	Like SHARC, different ISA	1	10	1,250	300				

NOTES:

1. The estimate of 100,000 SLOCs for this problem is on the low side; practical systems could be 3 times as large.
2. Development cost: GP code – 100 SLOC/MM divided by 10K/MM = 10 SLOC/1K. DSP code – 8 SLOC/1K.

consider Figure 1. Three types of nodes are shown—two DSPs and one GP node type. In the table (row 7 and 8), the application is assumed to support scaling from a few to many nodes either at design or at run-time. The degree of scaling is limited by the number of physical nodes per type, which will be fewer for the heterogeneous system, though percentage scaling will be similar. For reconfiguration on detected faults, there is less flexibility for the heterogeneous case.

Another consideration for the choice of PEs is the life cycle costs of upgrading and replacing both the hardware and software. GP microprocessors tend to be upgraded every 18 months and are generally instruction set architecture (ISA) compatible. DSPs evolve on something more like a 3 year cycle, and are not necessarily ISA compatible. For military systems, upgrades tend to be costly and complex due to the need to validate correct system operation, a process which has usually required field trials involving expensive equipment, instrumented test ranges, and months of effort. ISA compatibility is a typical measure of reduced risk in such upgrades. In the past, such "DSP" machines were uniquely built and were designed to be ISA compatible. This factor alone should be enough to mitigate against heterogeneous designs using current commercial DSPs. The fact that such designs are nevertheless being widely used may be attributed to the high impact of the small footprint of the heterogeneous designs.

Summary

The use of heterogeneous processing systems in embedded military systems is well entrenched and will continue. The key driver is the need to conserve size, weight, and power for many military systems. A key supporting capability which is little acknowledged is that a pool of highly capable analysts and programmer exists in the defense industry for whom the programming of high performance DSP machines is a known art, albeit more expensive. This allows designers to choose heterogeneous machines in some cases where they might otherwise be rejected.

Table 5. Comparison of heterogeneous and homogeneous parallel system design factors

No.	*Comparison Factors*	*Homogeneous Parallel*	*Heterogeneous Parallel*	*Life Cycle Cost Impact*	*Heterog. Advantage*
1	Hardware Arch Design	Baseline, simplest	Complicated by PE type placement in net	Initial	None
2	HW Interfaces	Baseline	Multiple network and I/O interfaces	Upgrade	None
3	Software Architecture Design	Least Complex	Partitioning of applications, multiple PE targets for middleware, resource management, etc.	Design, Maintenance	None
4	SW Interface Drivers	Fewer types	Increased due to more PE types	Design	None
5	SW Engineering Environment	Single PE target	Multiple SDE	Upgrade	None
6	SW Programmer Training	Single PE target	Multiple types	Upgrade, Maintenance	None
7	Applic. Behavior: Scaling	Most simple case	Complicated by multiple types	Design	None
8	Applic. Behavior: Reconfiguration.	Most simple case	Complicated by multiple types	Design	None
9	Multilevel Security	Most simple case	Harder – DSPs do not host secure OS, must be protected by GP-hosted OS	Design	None
10	Fault Tolerance Implementation	Most simple case	Complicated by multiple PE types	Design	None
11	Upgradeability: HW Cost	Baseline	Multiple PE types = multiple generations, not concurrent	Upgrade, Maintenance	None
12	Upgradeability: SW Cost	Baseline	Portability less likely on DSP designs	Upgrade	None
13	Perform.-HW: Unit Cost	Baseline	Lower due to DSP efficiencies	Recurring Cost	Significant
14	Perform.-HW: W*cu.in.*lb.	Baseline	Lower due to DSP efficiencies	High FOM	Significant
15	Perform. – SW: Devel. Cost	Baseline	Higher due to more difficult DSP programmability	Design, Maintenance	None
16	Perform. – SW: SDE Cost	Single PE target	Multiple SDE types for multiple PE types	Design, Maintenance, Upgrades	None
17	HW Spares Cost	Most simple, least costly	Higher cost, less commonality, more types of spares	Recurring	None

In addition, technology is emerging which will make the development of systems noted in line 8 of Table 2 readily achievable. The major computer industry suppliers are moving to parallel systems for high end servers, and will incorporate high bandwidth networks. Industry standard high bandwidth networks, if adopted would allow DSP modules to be attached or incorporated for some military systems. As these new servers mature, the maintenance capabilities and run-time software will become more powerful in response to commercial application drivers such as telecommunications and video on demand. Unfortunately, internal network standards are the exception in current machines, not the norm.

At the same time, those companies specializing in the embedded market will add PE types and more powerful software to their offerings, thus maintaining the advantages of performance per power/size/weight/cost that they have historically enjoyed. The introduction of new network concepts such as System Area Network (SAN), of which the Myrinet currently being supplied by Myricom, Incorporated is an example, will also allow such systems to achieve large numbers of nodes which can include both commercial workstations and highly compact, high performance heterogeneous rack systems.

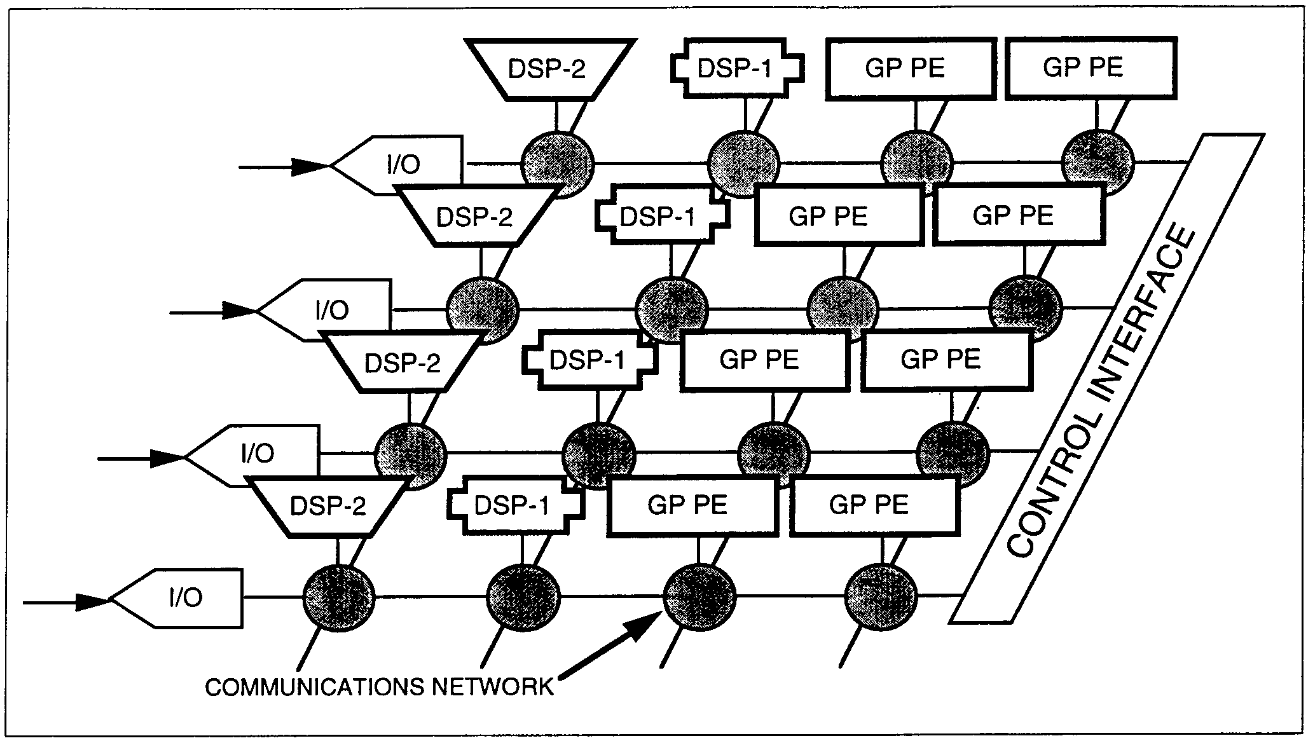


Figure 1. Illustrative heterogeneous parallel processing system

Acknowledgments

The author would like to acknowledge contributions and discussions with his many colleagues at Hughes, where much of the work reported has been done, as well as support of the management of the Electronics Division of the Sensor & Communications Systems Segment.

Biography

Glenn Ladd is a Program Manager at Hughes Aircraft Company, where most recently he leads various projects in development of advanced parallel processing systems and software. He has also developed VLSI chips and design tools, digital and analog GaAs device technology, and sensor systems. He received a Ph.D. in Electrical Engineering from Carnegie-Mellon University.

References

1. "A Survey of Heterogeneous Computing: Concepts and Systems," Ilija Ekmecic, Igor Tartalja, and Veljko Milutinovic, Proceedings of IEEE, Vol. 84, No. 8, August 1996

Session 4

Performance Evaluation and Reliability and Security

Session Chair

Domenico Laforenza,
CNUCE - Institute of the Italian National Research Council, Italy

Estimating the Execution Time Distribution for a Task Graph in a Heterogeneous Computing System*

Yan Alexander Li
Intel Corporation, SC9-15
2200 Mission College Blvd.
Santa Clara, CA 95052-8119 USA
ali2@mipos3.intel.com

John K. Antonio
Department of Computer Science
Texas Tech University
Lubbock, TX 79409-3104 USA
antonio@cs.ttu.edu

Abstract

The problem of statically estimating the execution time distribution for a task graph consisting of a collection of subtasks to be executed in a heterogeneous computing (HC) system is considered. Execution time distributions for the individual subtasks are assumed to be known. A mathematical model for the communication network that interconnects the machines of the HC system is introduced, and a probabilistic approach is developed to estimate the overall execution time distribution of the task graph. It is shown that, for a given matching and scheduling, computing the exact distribution of the overall execution time of a task graph is very difficult, and thus impractical. The proposed approach approximates the exact distribution and requires a relatively small amount of calculation time. The accuracy of the proposed approach is demonstrated mathematically through the derivation of bounds that quantify the difference between the exact distribution and that provided by the proposed approach. Numerical studies are also included to further validate the utility of the proposed methodology.

1 Introduction

A heterogeneous computing (HC) system provides a variety of architectural capabilities, orchestrated to perform an application whose subtasks have diverse execution requirements [1]. HC has become a subject of intensive research within the high-performance computing community in the quest of systems that are versatile and provide good performance. For a description of example HC applications and a list of related references, refer to [1].

Throughout this paper, an HC system is assumed to consist of a suite of independent machines of different types interconnected by a high-speed network. HC requires the effective use of diverse hardware and software components to meet the distinct and varied computational requirements of a given application. Implicit in the concept of HC is the idea that subtasks with different machine architectural requirements are embedded in the applications executed by the HC system. The concept of HC is to decompose a task into computationally homogeneous subtasks, and then assign each subtask to the machine where it is best suited for execution [1].

Unlike in distributed homogeneous systems (e.g., a network of workstations of the same type and configuration), it is generally difficult and impractical to suspend the execution of a subtask on one machine and resume that subtask's execution on another machine of a different type in an HC system. Thus, a challenge in making effective use of an HC system is to minimize the need for such dynamic subtask migration, which implies an increased importance on the static problems of assigning subtasks to machines (matching) and ordering the execution of subtasks assigned to the same machine (scheduling).

Performance prediction is the basis of matching and scheduling techniques for HC systems. Many matching and scheduling algorithms make the simplifying assumption that the execution time for each subtask is a known constant for each machine in the system (e.g., [2, 3]). However, there are elements of uncertainty, such as the uncertainty in input data values or in inter-machine communication time, which can impact the execution times. Machine choices for executing subtasks can also affect the execution time and its degree of uncertainty.

In this paper, the task to be executed on the HC system is modeled as a task graph consisting of a collection of subtasks. A mathematical model for the

*This work was supported by Rome Laboratory under grant number F30602-96-1-0098.

communication network that interconnects the machines of the HC system is introduced, and a probabilistic approach is developed to estimate the overall execution time distribution of the task graph. This overall distribution depends on the individual subtask execution time distributions, the inter-machine communication time distributions, the data dependency structure among the subtasks, the matching of subtasks to machines, and the scheduling of subtasks matched to a common machine. It is shown that, for a given matching and scheduling, computing the exact distribution of the overall execution time of a task graph is very difficult, and thus impractical. The proposed approach approximates the exact distribution and requires a relatively small amount of calculation time. The accuracy of the proposed approach is demonstrated mathematically through the derivation of bounds that quantify the difference between the exact distribution and that provided by the proposed approach. Numerical studies are also included to further validate the utility of the proposed methodology.

Section 2 presents the basic assumptions and an overview of the approach. A mathematical framework for the approach is presented in Section 3. Section 4 demonstrates the generic difficulty associated with calculating the exact execution time distribution for a task graph. An approximate approach is then proposed based on the conditions set forth by the Kleinrock independence approximation [4]. Section 4 concludes with a mathematical derivation of a bound for quantifying the difference between the exact distribution and that associated with the proposed approach. In Section 5, execution time distributions determined using the proposed approach are compared with corresponding distributions obtained by simulation of example task graphs. These studies indicate that the proposed approach predicts the execution time distribution for a large class of practical task graphs with high accuracy.

2 Assumptions and Overview

It is assumed that the HC system consists of a dedicated network of machines under the control of a single matching/scheduling mechanism. The type of application task considered is composed of a number of subtasks, each of which is to be executed on a particular machine in the HC system. The execution time distribution for each individual subtask is assumed to be known or estimated for the machine on which it is to be executed. Previous approaches for determining the execution time distribution of (parallel) programs (e.g., [5, 6]) could be applied for estimating the execution time distribution of subtasks. Estimates of subtask execution time distributions based on empirical measurements could also be utilized in the framework assumed here.

The subtask-to-machine matching and the order of execution for multiple subtasks assigned to the same machine (i.e., the subtask scheduling for each machine) are assumed to be static and known. The problems of optimal matching and scheduling represent large bodies of research in the field of HC, e.g., see [3, 7]. How to determine good solutions to the matching and scheduling problems is beyond the scope of this paper. The goal here is to develop a new probabilistic approach for analyzing the overall task execution time for given matching and scheduling policies. From this probabilistic modeling foundation, future matching and scheduling techniques may be developed that are based on probabilistic metrics for performance.

For each subtask, all input data items must be present at the subtask's designated machine before computation starts, and output data items can be transferred to other machines only after computation of the subtask is completed. Data-dependencies among the subtasks are represented by a task graph. A task graph is a directed acyclic graph in which nodes represent subtasks and arcs represent the data-dependencies among the subtasks.

To compute the execution time distribution of the entire task, the assumed execution time distributions of all subtasks (on their designated machines) are utilized. These distributions correspond to the computation time of the subtasks only; distributions for the times required to input and output any data structures are defined separately.

For each subtask, the machine from which each required input data item is fetched is assumed to be specified. In general, these machines will depend on the subtask-to-machine matching that is used. For example, fetching a data item from the machine at which it was first generated (or was initially stored) is a simple rule that is often assumed for this type of analysis. However, the model devised here allows for more general refinements in how the data is distributed and retrieved. For example, the model is general enough to account for the data-reuse and multiple data-copies situations [8], which allow the fetching of data items from a machine other than the one where it was generated.

Network I/O at each machine is assumed to be nonblocking (e.g., handled by a stand-alone I/O processor). Therefore, computation and inter-machine communication can be overlapped in time. Each subtask is assumed to start computation when its designated machine is ready and all its input data items are available on this machine. Immediately after computation for a subtask completes, the machine is made available for executing the next subtask scheduled for execution on that machine. Also, the output data items produced by the completed subtask are made available for all subsequent subtasks to be executed on that machine. If multiple data items produced by a subtask are to be transferred to other machines, the order in which they are sent is assumed to be specified. All outgoing data items at a given machine are assumed to be buffered when the machine is transmitting another data item.

In general, the network transmission time for a given data item (including the uncertain delay caused by network contention) depends on factors such as the size of the data item being transferred, the topology and bandwidth of the network, the type of switching used, the types of machine-network interfaces used, and the overall network load. For the purpose of this analysis, a network model in which the transmission time is modeled according to a probabilistic distribution is assumed. In this model, the shape of the distribution is fixed (e.g., the variance is fixed), and the mean of the distribution is defined to be the sum of a fixed overhead and a term proportional to the size of data item transfered. The fixed overhead corresponds to the latency of the network, and the coefficient for the second term corresponds to the inverse of the channel bandwidth. This represents one possible model for the transmission time for a network. Other network models are possible and could be used in place of the one assumed here. Furthermore, the transmission time can be source-destination dependent. The only requirement is that the transmission time be modeled according to a probabilistic distribution.

Three separate random variables are used to represent the start time, execution time, and completion time for each subtask. The values of the start and finish times are defined with respect to a global time-line, and the subtask execution time represents the length of an interval on this time-line. The task is assumed to start execution at time 0. A subtask is called a terminal subtask if it corresponds to a node with no successors in the task graph and is the last subtask executed on its designated machine. The maximum of the completion times over all terminal subtasks defines the finish time of the entire task.

3 Mathematical Model of Task Graph Execution in an HC System

In this section, random variables are used to model the data communication times among the machines and the start time, execution time, and finish time for each subtask. The relationships among these random variables are derived. These are used in the next section to compute the overall task execution time distribution by performing appropriate operations to the distribution functions of these random variables.

It is assumed that there are $\underline{m}$ machines in the HC system, labeled $\underline{M_i}$, $i = 0, 1, \ldots, m-1$. The task consists of $\underline{n}$ subtasks, labeled $\underline{S_j}$, $j = 0, 1, \ldots, n-1$. The subtask-to-machine matching is defined by the function

$$\underline{\mathcal{M}} : \{0, \ldots, n-1\} \rightarrow \{0, \ldots, m-1\}. \qquad (1)$$

Thus, subtask S_j is to be executed on machine $M_{\mathcal{M}(j)}$. For each machine M_i, the number of subtasks assigned to M_i is denoted as $\underline{\alpha_i}$, and the execution schedule for these α_i subtasks is defined by the function $\underline{\mathcal{X}_i}$: $\{0, \ldots, \alpha_i - 1\} \rightarrow \mathcal{M}^{-1}(i)$, which is a bijection. Thus, $\mathcal{X}_i(k)$, $0 \leq k < \alpha_i$, defines the $(k+1)$-th task to be executed; the sequence of execution on machine M_i is from subtask $S_{\mathcal{X}_i(0)}$ to $S_{\mathcal{X}_i(\alpha_i - 1)}$.

For each subtask S_j, $0 \leq j < n$, define $\underline{n_j^I}$ and $\underline{n_j^O}$ to be the number of associated input and output data items, respectively. Input data items of S_j are labeled $\underline{D_{j,v}^I}$, $0 \leq v < n_j^I$. Output data items of S_j are labeled $\underline{D_{j,u}^O}$, $0 \leq u < n_j^O$. If multiple output data items of a subtask need to be transmitted to other machine(s), then they are transmitted serially in ascending index order, i.e., $D_{j,u}^O$ is transmitted before $D_{j,\ell}^O$, for $u < \ell$.

For each subtask S_j, the times at which computation starts and finishes is modeled by random variables $\underline{T_j^S}$ and $\underline{T_j^F}$, respectively. The execution time of S_j is modeled by random variable $\underline{\tau_j^E}$, defined as $\tau_j^E = T_j^F - T_j^S$. Note that throughout this paper, values of random variables involving the letter "T" correspond to points on the global time-line, and those that use "τ" represent lengths of intervals on this time-line. It is assumed that the execution times of all subtasks are independent, i.e., τ_j^E, $0 \leq j < n$, form a set of mutually independent random variables. This has been an assumption made by other researchers as well, e.g., [9]. Based on the definition of τ_j^E, a useful expression for T_j^F is given by:

$$T_j^F = T_j^S + \tau_j^E. \qquad (2)$$

For each subtask S_j, the time at which input data item $D^I_{j,v}$, $0 \le v < n^I_j$, becomes available on machine $M_{\mathcal{M}(j)}$ is defined by $\underline{T^I_{j,v}}$. It is assumed that all initial data items are pre-loaded to the machines that will first use them, i.e., the time required to load these data items is not considered in the analysis. Thus, the available time for all pre-loaded data items is defined as 0. The sum of the queuing time, denoted by $\underline{\tau^Q_{j,u}}$, and the network time, denoted by $\underline{\tau^N_{j,u}}$, defines the time period starting at time T^F_j and ending when data item $D^O_{j,u}$ is available at its destination subtask. If the destination subtask of $D^O_{j,u}$ is on the same machine as S_j, then both $\tau^Q_{j,u}$ and $\tau^N_{j,u}$ are defined to be 0. Otherwise, $\tau^Q_{j,u}$ represents the time $D^O_{j,u}$ waits in the queue before machine $M_{\mathcal{M}(j)}$ begins to transmit it, and $\tau^N_{j,u}$ represents the amount of time (including any delay caused by network contention) to transfer $D^O_{j,u}$ through the network to its destination. Network times for different output data items are assumed to be independent (i.e., the random variables $\tau^N_{j,u}$ are independent).

In the following discussion, let output data item u of S_j (i.e., $D^O_{j,u}$) be input data item v of subtask S_g (i.e., $D^O_{j,u} = D^I_{g,v}$). Also, if S_j and S_g are assigned to the same machine, i.e., $\mathcal{M}(g) = \mathcal{M}(j)$, then $\tau^Q_{j,u} = 0$ for all u. If $u = 0$ (i.e., $D^O_{j,u} = D^O_{j,0}$ is the first data item transmitted), then the queuing time is zero, i.e., $\tau^Q_{j,0} = 0$. Hence, the general expression for $\tau^Q_{j,u}$ is:

$$\tau^Q_{j,u} = \begin{cases} 0 & \text{if } u = 0 \\ & \text{or } \mathcal{M}(j) = \mathcal{M}(g), \\ \tau^Q_{j,u-1} + \tau^N_{j,u-1} & \text{otherwise.} \end{cases} \tag{3}$$

It is assumed that $D^O_{j,u}$ is available to S_g immediately upon arriving at machine $M_{\mathcal{M}(g)}$. Let $\underline{T^A_{j,u}}$ define this arrival time, then

$$T^A_{j,u} = T^I_{g,v} = T^F_j + \tau^Q_{j,u} + \tau^N_{j,u}. \tag{4}$$

Assume now that subtask S_j is to be executed on machine M_i, i.e., $\mathcal{M}(j) = i$, and is the $(k+1)$-th subtask scheduled for execution, i.e., $\mathcal{X}_i(k) = j$. Let $\underline{T^M_j}$ denote the time that M_i becomes available for executing S_j. If $k = 0$, i.e., S_j is the first subtask scheduled to execute on M_i, then T^M_j is defined to be 0. Otherwise, $S_{\mathcal{X}_i(k-1)}$ is the previous subtask that executes on M_i, and T^M_j is defined to be the finish time of $S_{\mathcal{X}_i(k-1)}$. Therefore, the general relation for the time when machine M_i becomes available for executing subtask $S_{\mathcal{X}_i(k)}$ is:

$$T^M_{\mathcal{X}_i(k)} = \begin{cases} 0 & \text{if } k = 0, \\ T^F_{\mathcal{X}_i(k-1)} & \text{otherwise.} \end{cases} \tag{5}$$

The start time of a subtask is the maximum of the available time of the designated machine and the maximum of all arrival times of its input data items:

$$T^S_j = \max\left\{ T^M_j, \max_{v=0}^{n^I_j - 1} \{T^I_{j,v}\} \right\}. \tag{6}$$

Equations (2) through (6) establish the relationships among the defined random variables, and are used to derive their associated probability distribution and/or density functions. In particular, distributions for the random variables τ^E_j and $\tau^N_{j,u}$ are assumed to be specified, and distributions for the other random variables are defined based on the relationships derived in this section. The overall execution time distribution of a task graph is analyzed in the next section.

4 Calculating the Execution Time Distribution for a Task Graph

4.1 Difficulty with exact calculation

In a task graph, subtasks that require input data items from a common subtask have correlated start and finish times, and thus their associated random variables are not independent. This correlation can propagate to the start and finish times of subsequent subtasks that get data from these subtasks. All such subtasks correspond to nodes in the task graph that have a common ancestor. It is shown in this subsection that this type of correlation generally makes the exact derivation of the overall execution time distribution of a task graph difficult and impractical.

Before demonstrating the difficulty of performing basic operations on correlated random variables, the summation and maximum operations for independent random variables are first reviewed. From basic probability theory, recall that the density function of the summation of independent random variables is the convolution of the density functions of the individual random variables [10]. Thus, for two independent random variables, say R and V with density functions $f_R(\cdot)$ and $f_V(\cdot)$, the density function of $Y = R + V$ is given by the convolution of $f_R(\cdot)$ and $f_V(\cdot)$, denoted by $\underline{f_Y(\cdot) = f_R(\cdot) * f_V(\cdot)}$, which is defined by:

$$f_Y(y) = \int_{-\infty}^{\infty} f_R(y-t) f_V(t) dt. \tag{7}$$

Also recall that the distribution function of the maximum of independent discrete random variables is the product of the distribution functions of the individual random variables [10]. Thus, for two independent discrete random variables, say R and V with distribution functions $F_R(\cdot)$ and $F_V(\cdot)$, the distribution function of $Z = \max\{R, V\}$ is given by

$$F_Z(z) = F_R(z) \cdot F_V(z). \tag{8}$$

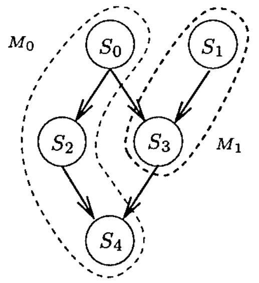


Figure 1: An example task graph whose overall execution time distribution is difficult to derive.

Consider the task graph with five subtasks shown in Fig. 1. Assume there are two machines in the HC system. Subtasks S_0, S_2 and S_4 are assigned to machine M_0, and subtasks S_1 and S_3 are assigned to machine M_1. To simplify the presentation, the network communication times are assumed to be zero, i.e., $\tau^N_{j,u} = 0$ for all j and u. Recall that τ^E_j denotes the given execution time distribution of S_j on its designated machine. The start time of subtask S_4 can be derived by using Equations (2)–(6) as follows.

$$\begin{aligned}
T^F_0 &= \tau^E_0, \\
T^F_1 &= \tau^E_1, \\
T^S_2 &= T^F_0 = \tau^E_0, \\
T^S_3 &= \max\{T^F_0, T^F_1\} = \max\{\tau^E_0, \tau^E_1\}, \\
T^F_2 &= \tau^E_0 + \tau^E_2, \\
T^F_3 &= \max\{\tau^E_0, \tau^E_1\} + \tau^E_3, \\
T^S_4 &= \max\{T^F_2, T^F_3\} \\
&= \max\{\tau^E_0 + \tau^E_2, \max\{\tau^E_0, \tau^E_1\} + \tau^E_3\}. \qquad (9)
\end{aligned}$$

Because T^F_2 and T^F_3 are not independent, Equation (8) is not applicable for computing the distribution of T^S_4. The only way to compute the exact distribution for T^S_4 is to consider T^S_4 as a function of τ^E_0, τ^E_1, τ^E_2, and τ^E_3 (which are assumed to be independent random variables) and use direct integration. To simplify the notation, let $\tau^E_0 = A$, $\tau^E_1 = B$, $\tau^E_2 = C$, $\tau^E_3 = D$, and $T^S_4 = X$. With these substitutions, Equation (9) is

$$X = \max\{A + C, \max\{A, B\} + D\}.$$

An exact derivation of the distribution function for X (i.e., T^S_4) based on basic probability theory is as follows.

$$\begin{aligned}
&F_X(x) \\
&= \Pr[\max\{(A + C), \max\{A, B\} + D\} \le x] \\
&= \Pr[A + C \le x, \max\{A, B\} + D \le x] \\
&= \int \Pr[A + C \le x, \max\{A, B\} + D \le x | D = d] \\
&\qquad dF_D(d) \\
&= \int \Pr[A + C \le x, \max\{A, B\} \le x - d] \; dF_D(d) \\
&= \iint \Pr[A + C \le x, \max\{A, B\} \le x - d | B = b] \\
&\qquad dF_B(b) \; dF_D(d) \\
&= \iint \Pr[A + C \le x, \max\{A, b\} \le x - d] \\
&\qquad dF_B(b) \; dF_D(d) \\
&= \iiint \Pr[A + C \le x, \max\{A, b\} \le x - d | A = a] \\
&\qquad dF_A(a) \; dF_B(b) \; dF_D(d) \\
&= \iiint \Pr[C \le x - a, \max\{a, b\} \le x - d] \\
&\qquad dF_A(a) \; dF_B(b) \; dF_D(d) \\
&= \iiint F_C(x - a) I(\max\{a, b\} \le x - d) \\
&\qquad dF_A(a) \; dF_B(b) \; dF_D(d), \qquad (10)
\end{aligned}$$

where $I(\cdot)$ is the "indicator function," defined for this case as follows: if $\max\{a, b\} \le x - d$, then $I(\max\{a, b\} \le x - d) = 1$; otherwise $I(\max\{a, b\} \le x - d) = 0$.

The above example illustrates that even for a simplified model (i.e., ignoring the communication overhead) of the considered task graph, the derivation of the exact execution time distribution is non-trivial. In particular, the production of a string of equalities is required (based on basic principles of probability theory) in order to derive the final expression given in Equation (10). Thus, although the final expression can be evaluated in this case, it was not straightforward to derive.

Practical task graphs will be more complicated than that of Fig. 1, and dependencies imposed by machine availability could further complicate the relationships among the start and finish times of subtasks. Although exact derivation for general task graphs may be possible, there is no clear systematic approach for automating such a derivation. A goal of this paper is to devise an approach for systematically determining (or suitably approximating) the execution time distribution of a task graph. In the remainder of this section, such a technique is proposed for estimating the overall execution time distribution based on the Kleinrock independence approximation. This approximation enables the usage of the simple formulas for summation and maximum of random variables (i.e., Equations (7) and (8)).

4.2 Independence assumption

As demonstrated in the previous subsection, subtasks corresponding to nodes in the task graph that have a common ancestor can have correlated random variables associated with the start and finish times. In such cases, deriving an expression for the exact distribution of the overall execution time distribution can become unrealistic for general task graphs. However, conditions exist for which the associated random variables can nevertheless be treated as uncorrelated despite this type of interaction. The Kleinrock independence approximation [4] is a well-known condition for describing this situation, and is used here as the basis for assuming independence among random variables that may technically be correlated.

To understand the original rationale of the Kleinrock independence approximation, consider a data network in which there are many interacting transmission queues. A traffic stream departing from one queue enters one or more other queues, perhaps after merging with portions of other traffic streams departing from yet other queues. Although packet inter-arrival times of data packets can become dependent (i.e., correlated) after a traffic stream leaves the first queue, the Kleinrock independence approximation concludes that the merging of many different packet streams on a transmission line has an effect similar to restoring the independence of inter-arrival times and packet lengths [11].

Similarly, in a task graph, a subtask may take input from many other subtasks, and multiple subtasks may be assigned to the same machine, where they must wait for the machine to become available before execution. The effect is analogous to that of merging many traffic streams in a data network. Thus, it is assumed that the input of data from many other subtasks has the effect of restoring independence among the start and finish times of subtasks that have a common ancestor in the task graph. This approximation of independence is the basis for justifying the use of Equations (7) and (8) to compute the distribution of start and finish times of subtasks. The degree to which this independence approximation is violated (or not) will influence the resulting accuracy of the calculated distribution. The estimation error is analyzed mathematically in Subsection 4.4, and is investigated further in Section 5 through numerical simulation studies.

4.3 Proposed approach for calculating the execution time distribution

Subtask start and finish time distributions are calculated in an order determined by the data dependency structure of the task graph and machine availability, which depends on the given matching of subtasks to machines and local scheduling for each machine. The key to calculating the start and finish time distributions is to partition the subtasks into layers, which requires the definition of an immediate predecessor. Subtask S_j is an immediate predecessor of subtask S_g if either of the following conditions is satisfied: (1) S_g requires data from S_j (i.e., there is an arc in the task graph from S_j to S_g) or (2) S_j and S_g are assigned to the same machine and S_j is scheduled to execute immediately before S_g. Those subtasks without an immediate predecessor are put into layer 1. A subtask is put into layer $k+1$ if the highest layer number of its immediate predecessors is k. Based on this definition (which implies a constructive procedure), there is no data dependence among subtasks of the same layer. The main difference between this layering approach and those found in the literature (e.g., Cluster-M model in [12]) is the resource dependence determined by scheduling is also considered, i.e., condition (2) above, is also considered.

Subtask start and finish time distributions are first computed for subtasks in layer 1. Distributions for the time each output data item is available on its target machine are then calculated. These steps are repeated for subtasks in layer 2, and so on. In this way, when the start time distribution of each subtask is computed, the distributions for available times of the designated machine and all input data items are known.

For each subtask S_j considered in the "layered" order, the following four calculations are performed.

1. Compute distribution function for subtask start time:

$$F_{T_j^S}(\cdot) = F_{T_j^M}(\cdot) \prod_{v=0}^{n_j^I-1} F_{T_{j,v}^I}(\cdot). \qquad (11)$$

2. Compute density function for subtask completion time:
$$f_{T_j^F}(\cdot) = f_{T_j^S}(\cdot) * f_{T_j^E}(\cdot). \quad (12)$$

3. Let the next subtask to be executed on machine $M_{\mathcal{M}(j)}$ be S_g. Define the density function for machine available time for S_g:
$$f_{T_g^M}(\cdot) = f_{T_j^F}(\cdot). \quad (13)$$

4. For each u from 0 to $n_j^O - 1$, let output data item $D_{j,u}^O$ be the input data item h of subtask S_g (i.e., $D_{g,h}^I$). Compute the distributions for queuing time, arrival time for $D_{j,u}^O$, and the available time for $D_{g,h}^I$:
$$\begin{aligned} & f_{T_{j,u}^Q}(\cdot) \\ &= \begin{cases} \delta(\cdot) & \text{if } u = 0 \text{ or } \mathcal{M}(j) = \mathcal{M}(g), \\ f_{T_{j,u-1}^Q}(\cdot) * f_{T_{j,u-1}^N}(\cdot) & \text{otherwise.} \end{cases} \quad (14) \\ & f_{T_{j,u}^A}(\cdot) = f_{T_{g,h}^I}(\cdot) \\ &= f_{T_j^F}(\cdot) * f_{T_{j,u}^Q}(\cdot) * f_{T_{j,u}^N}(\cdot). \quad (15) \end{aligned}$$
($\delta(\cdot)$ represents the Dirac delta function [13].)

After completing the above four steps for each subtask (ordered according to the layer numbers), the overall distribution of the task execution time is computed. Let ϕ be the number of terminal subtasks, and let $\{S_{j_0}, S_{j_1}, \ldots, S_{j_{\phi-1}}\}$ be the set of terminal subtasks. Then the probability distribution function of the completion time of the entire task is:
$$F_{T^C}(\cdot) = \prod_{\psi=0}^{\phi-1} F_{T_{j_\psi}^F}(\cdot). \quad (16)$$

4.4 Error analysis

In this subsection, the difference between the distribution computed by the proposed approach (Subsection 4.3) and the exact distribution is analyzed for a special class of task graphs for which the Kleinrock independence assumption is (apparently) violated. An analytical expression for the difference of the means of these distributions is derived. Based on this expression, it is shown that the proposed approach always overestimates the actual mean of the execution time. A bound for the difference of the means is also derived that depends on the parameters of the assumed subtask distributions involved. Finally, conditions are determined for which the distribution of the proposed approach equals the exact distribution for this class of task graph.

In a task graph, a fork-join structure between two subtasks S_f and S_j contains a set of two or more disjoint paths from S_f to S_j ("f" is for fork and "j" is for join). Let $\underline{W}$ denote the set of subtasks in a fork-join structure, excluding S_f and S_j. For a given subtask-to-machine matching and a given scheduling for each machine, a fork-join structure is called an isolated fork-join structure if every immediate predecessor of a subtask in W belongs to $W \cup \{S_\mathrm{f}\}$. Examples of isolated fork-join structures are shown in Fig. 2, in which each subtask is assumed to be assigned to a distinct machine. The conditions of the Kleinrock independence approximation are clearly violated in an isolated fork-join structure. This is because the data that flows from S_f to S_j (e.g., S_0 to S_3 in Fig. 2(a)) does not merge with other data originating from subtasks outside that fork-join structure. Therefore, the effect of restoring independence of arrival times of input data items for S_j by merging data flows from different subtasks/machines does not occur. Perhaps surprisingly, it is shown that even for this "worst case" structure (i.e., the isolated fork-join structure), the proposed approach still can provide reasonably accurate estimate of the exact distribution. Under certain conditions, it is shown that the distribution from the proposed approach actually coincides with the exact distribution.

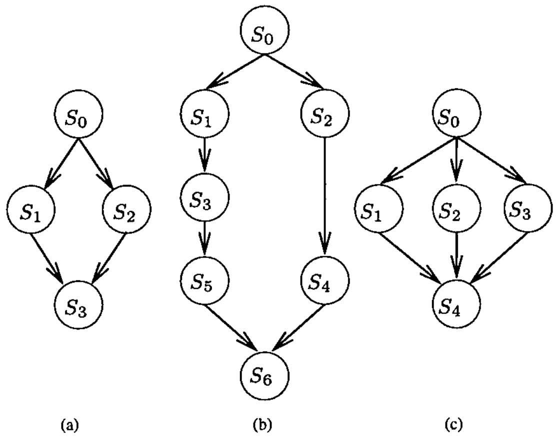


Figure 2: Examples of isolated fork-join structures. Each subtask is assumed to be assigned to a distinct machine for each example.

Isolated fork-join structures are characterized by the number of disjoint paths that connect S_f to S_j. In Fig. 2(b), note that the chains $S_1 \rightarrow S_3 \rightarrow S_5$ and $S_2 \rightarrow S_4$ can each be reduced to a single subtask, resulting in a structure identical to that in Fig. 2(a).

Therefore, without loss of generality, only structures in which there is exactly one subtask on each path between S_f and S_j (such as Figs. 2(a) and (c)) are studied here.

In general, for multiple subtasks matched to the same machine, a fork-join structure may not be an isolated fork-join structure. For example, consider Fig. 3 in which the subtasks are matched to machines as indicated by the ovals. Although S_0 through S_3 form a fork-join structure, the scheduling of S_1 and S_5 on M_1 determines whether it is an isolated fork-join structure. In particular, if S_5 is scheduled before S_1, then it is not an isolated fork-join structure, because S_5 is an immediate predecessor of S_1 and $S_5 \notin W \cup \{S_0\} = \{S_0, S_1, S_2\}$.

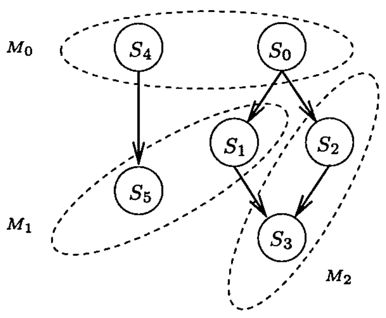


Figure 3: Example task graph in which its characterization as an isolated fork-join structure depends on the scheduling.

Consider the isolated fork-join structure shown in Fig. 2(a). For clarity of presentation and without loss of generality, network communication times are ignored. Let A, B, and C denote the execution time distribution of S_0, S_1, and S_2 on their designated machines, respectively. The start time of S_3 can be derived as:

$$\begin{aligned} T_0^F &= T_1^S = T_2^S = A, \\ T_1^F &= A + B, \\ T_2^F &= A + C, \\ T_3^S &= \max\{T_1^F, T_2^F\} = \max\{A + B,\ A + C\}. \end{aligned}$$

Note that the distributions for T_1^F and T_2^F are obtained by convolving the appropriate density functions (i.e., $f_{T_1^F}(\cdot) = f_A(\cdot) * f_B(\cdot)$ and $f_{T_2^F}(\cdot) = f_A(\cdot) * f_C(\cdot)$). The proposed approach estimates the distribution of T_3^S as $F_{T_1^F}(\cdot)F_{T_2^F}(\cdot)$. For notational convenience, let X denote the random variable with distribution function $F_{T_1^F}(\cdot)F_{T_2^F}(\cdot)$, and let $X^* = T_3^S$, i.e., X and X^* represent the estimated and exact value of T_3^S, respectively.

In [6], the exact distribution for X^* is derived and the difference between X and X^* is analyzed mathematically. Due to space limitations, only the results of this analysis is included here (refer to [6] for the detailed derivation). To state the results, some notation is needed for quantifying the ranges of the random variables A, B, and C. Because these random represent the execution times of S_0, S_1, and S_2, it is assumed that they are finite and have finite range. Thus, there exists constants $0 \le a_1 \le a_2$, $0 \le b_1 \le b_2$, and $0 \le c_1 \le c_2$, such that $\Pr[A < a_1] = \Pr[A > a_2] = 0$, $\Pr[B < b_1] = \Pr[B > b_2] = 0$, $\Pr[C < c_1] = \Pr[C > c_2] = 0$.

The following is a summary of the results proven in [6].

1. The proposed approach always overestimates the mean, and the estimation error for the mean is upper-bounded by the length of the range of A:
$$0 \le EX - EX^* \le a_2 - a_1, \qquad (17)$$
where EX and EX^* denote the expected values (i.e., means) of the approximate and exact distributions, respectively.

2. The proposed approach yields the exact distribution if the length of the range of A is shorter than the length of combined range of B and C. Mathematically, this condition on the length of the range of A is stated as:
$$a_2 - a_1 < \max\{b_2, c_2\} - \min\{b_1, c_1\}.$$
Thus, if the above inequality is satisfied, then the distribution produced by the proposed approach equals the exact distribution (i.e., $X = X^*$).

These two results share a common theme – the smaller the range of possible values for A, the smaller the estimation error. The second result is most interesting, and perhaps most surprising. It states that if the range of values for A is sufficiently small (with respect to the corresponding ranges for B and C), then the estimated distribution actually equals the exact distribution (i.e., there is no estimation error). An important key in deriving these results is the finite range assumption for the random variables A, B, and C.

5 Numerical Studies

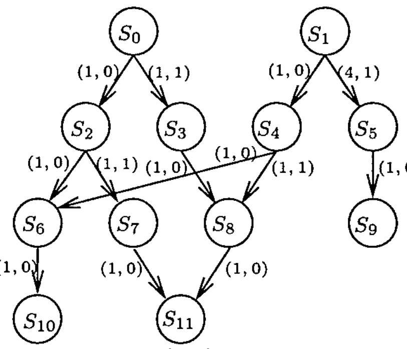
5.1 Overview

In this section, the accuracy of the execution time distributions determined from the proposed approach

(Subsection 4.3) is evaluated through numerical studies. Due to the size and complexity of the task graphs considered, derivation of the exact distributions (as done for the simple task graphs considered in the previous section) is not feasible. Thus, detailed simulations of the task graphs are performed as a means of determining the actual distributions. The results show that the proposed approach predicts simulated task graph execution time distribution with high accuracy. Task graph structures for which the independence among the random variables is apparently violated to a substantial degree are also studied. Even for these task graph structures, the distributions computed by the proposed approach match those from the simulation reasonably well. On a Sun SPARCstation 5, the time required to compute distributions based on the proposed approach is about 10 times less than that based on simulating the task graph over 4000 instances.

For this study, each subtask execution time distribution (associated with the random variable τ_j^E for subtask S_j) is assumed to be either a uniform or a normal distribution. (This is for convenience only; any distribution could be used in this framework.) Each network transmission time distribution (associated with the random variable $\tau_{j,u}^N$) is assumed to be a normal distribution with a fixed standard deviation, and the mean defined as the sum of a fixed overhead and a term proportional to the size of data item to be transfered. (Each normal distribution $N(\mu, \sigma)$ with mean μ and standard deviation σ is discretized in the range of $(\max\{\mu - 4\sigma, 0\}, \mu + 4\sigma)$.) The parameters of these distributions are included in an input file. Also included in this file is the subtask-to-machine matching and execution schedule of each individual machine. A program was written in C to parse this input file and output a Matlab program for simulating the execution time of task graph (see [6] for details).

For each instance of the simulation, the execution time of a subtask is determined by generating a random number according to its assumed distribution, and network transmission times are determined similarly. The overall execution time of the task graph is calculated according to data dependency structure of the task graph and execution schedule of the individual machines. Subtask start and finish times are first computed for subtasks in layer 1. The time each output data item of layer 1 subtasks is available on its target machine are then calculated. These steps are repeated for subtasks in layer 2, and so on. In this way, when the start time of each subtask is computed, the available times of the designated machine and all



Key for arc labels (d, k):

d: amount of data to be transfered

k: order index of output data item

Figure 4: Example task graph.

input data items are known. The overall execution time of the task graph is the maximum of the finish times over all terminal subtasks. For each task graph studied, 4000 simulation instances were performed to collect the sample distribution of the overall execution time of the task graph.

5.2 Comparison of estimated distribution and simulated sample distribution

The first task graph studied is shown in Fig. 4. There are 12 subtasks in the graph, labeled from S_0 to S_{11}. Each arc is labeled with an ordered paired (d, k), where d is the amount of data to be transfered and k is the output data index for the source subtask. (Recall if a subtask has multiple output data items to be transmitted to other machine(s), they are transmitted in ascending index order.) It is assumed that there are four machines in the HC system, labeled from M_0 to M_3. The assumed subtask-to-machine matching, execution schedule of each machine, and execution time distribution of each subtask on its designated machine are defined in Table 1. Three subtasks are assigned to each machine. Each row corresponds to parameters for a subtask; the first column is the label of the subtask, the second column is the label of its designated machine, the third column is the order of execution on that machine, and the forth column parameterizes its execution time distribution.

The network transmission time is modeled by a normal distribution with a standard deviation of 3. Two separate models for the mean of the network network transmission time were used. In the first, the mean is equal to $10 + (5 \times d)$, and in the second, the mean is equal to $20 + (30 \times d)$. These two models represent different computation to communication ratios. For

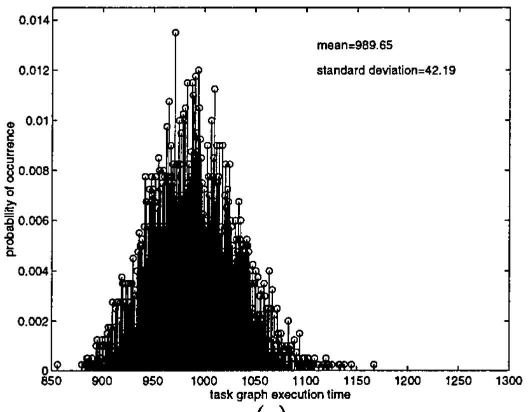
subtask	machine	schedule	exe. time distr.
0	0	0	$U(125, 146)$
1	1	0	$N(203, 10.3)$
2	2	0	$U(244, 325)$
3	0	1	$N(301, 24.3)$
4	3	0	$U(203, 248)$
5	1	1	$N(350, 27.3)$
6	2	1	$U(271, 324)$
7	0	2	$N(283, 26.1)$
8	3	1	$N(201, 34.1)$
9	1	2	$U(278, 321)$
10	2	2	$U(130, 183)$
11	3	2	$N(231, 29.4)$

Table 1: Assumed matching, scheduling, and execution time distribution for subtasks of Fig. 4. $U(a, b)$: uniform distribution between a and b. $N(\mu, \sigma)$: normal distribution with mean μ and standard deviation σ.

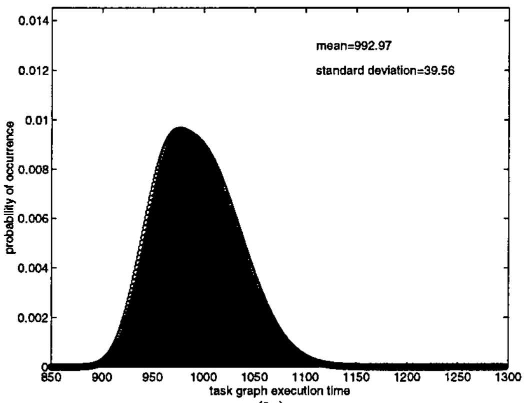
each model, the distribution for the execution time of the entire task graph are obtained through simulation, and the corresponding distribution is also computed by the proposed approach.

The results of these studies are shown in Figs. 5 and 6. For each study, the difference between the simulated mean execution time and the estimated mean execution time is less than 0.4%, and the difference for standard deviation is less than 6.3%. From this, it is concluded that the proposed approach accurately estimates the distribution of execution time for the task graph considered.

Simulation was also performed for the task graph shown in Fig. 7. The task graph is nearly the same structure as that of Fig. 4. The only difference is the arc $S_4 \rightarrow S_6$ (in Fig. 4) has been changed to become arc $S_6 \rightarrow S_{11}$ (in Fig. 7). However, this small change in the structure of the task graph creates an isolated fork-join structure (S_2, S_6, S_7, S_{11}). The subtask-to-machine matching, subtask execution time distribution, and execution schedules of individual machines remain unchanged. Simulations are performed using the same two network communication models as used for Fig. 4. The resulting distributions are compared with estimates in Figs. 8 and 9. Still, the proposed approach provides a good approximation for the task graph execution time distribution. The error between estimated and simulated results is less than 1.08% for the mean execution time and less than 10.4% for the standard deviation. This example demonstrates the robustness of the proposed approach when some assumptions are violated. (Recall that even for isolated fork-join structure, the estimation error for the mean execution time is upper-bounded by the width of the density of execution time of the fork node, and conditions exists under which exact results could be obtained.)



(a)



(b)

Figure 5: Distributions of execution time for task graph in Fig. 4 where the mean of network transmission time is equal to $10 + (5 \times d)$. (a) Sample distribution. (b) Estimated distribution.

6 Summary

In this paper, a methodology for estimating the execution time distribution for a task graph executed in an HC system is introduced. Individual subtask execution time distributions are assumed to be known or estimated using analytical or empirical techniques. A probabilistic model for the data transmission time is developed. Random variables are used to represent

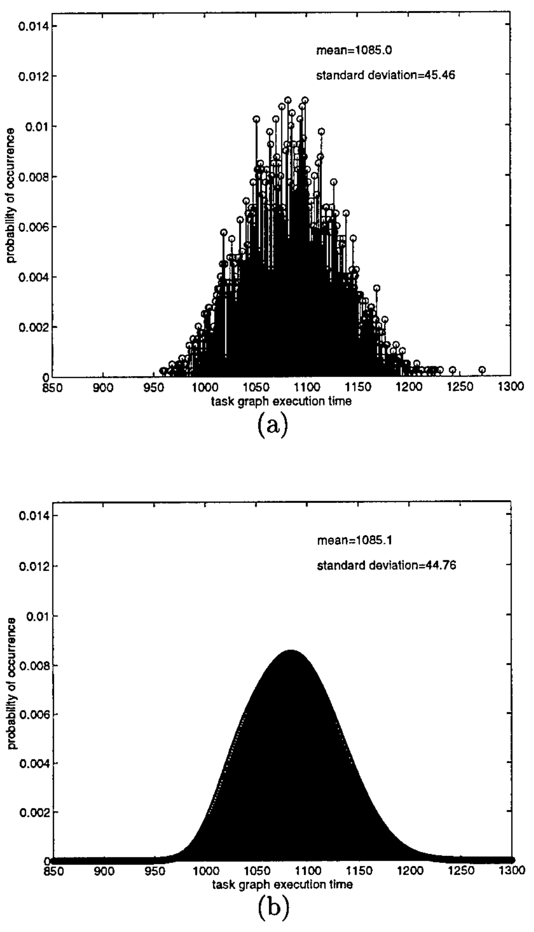
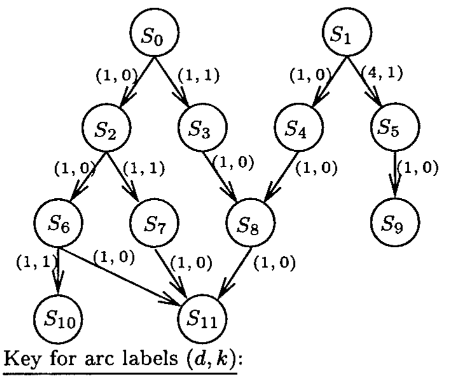


Figure 6: Distributions of execution time for task graph in Fig. 4 where the mean of network transmission time is equal to $20 + (30 \times d)$. (a) Sample distribution. (b) Estimated distribution.



Key for arc labels (d, k):

d: amount of data to be transfered

k: order index of output data item

Figure 7: Example task graph in Fig. 4 with arc $S_4 \rightarrow S_6$ moved to $S_6 \rightarrow S_{11}$.

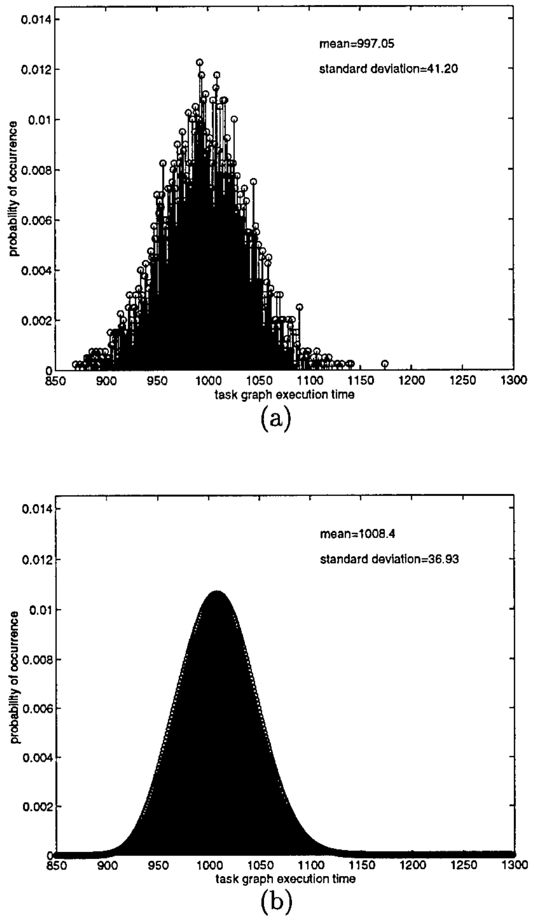
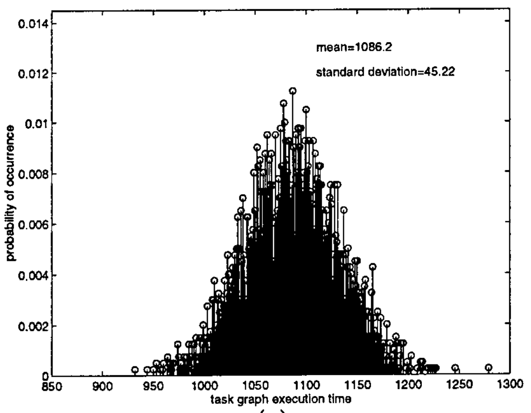


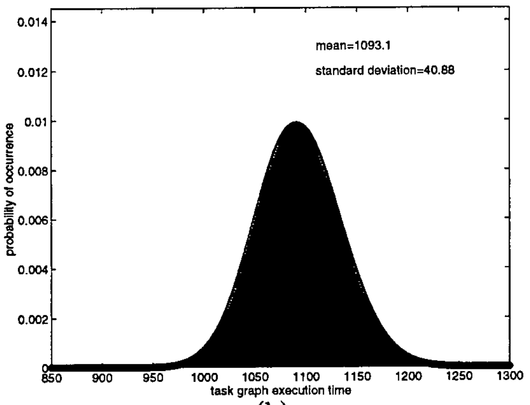
Figure 8: Distributions of execution time for task graph in Fig. 7 where the mean of network transmission time is equal to $10 + (5 \times d)$. (a) Sample distribution. (b) Estimated distribution.

the duration of subtask executions and network transmissions, as well as the start and finish times. Data dependency and machine availability are used to derive the relationships among these random variables.

It is demonstrated that deriving the exact execution time distribution for general task graphs is extremely difficult. The Kleinrock independence approximation is applied to make the computation of associated probability distributions tractable. Graph structures for which the independence assumption is violated are identified, and an upper bound of the estimation error for the mean execution time is derived for this case. Simulations were performed for various task graphs. The simulation results indicate that the proposed approach provides accurate estimates for execution time distribution.



(a)



(b)

Figure 9: Distributions of execution time for task graph in Fig. 7 where the mean of network transmission time is equal to $20 + (30 \times d)$. (a) Sample distribution. (b) Estimated distribution.

References

[1] H. J. Siegel, J. K. Antonio, R. C. Metzger, M. Tan, and Y. A. Li, "Heterogeneous Computing," in *Handbook of Parallel and Distributed Computing*, A. Y. Zomaya, ed., pp. 725–761, McGraw-Hill, New York, NY, 1996, (also Purdue EE School technical report TR-EE 94-37).

[2] D. W. Watson, J. K. Antonio, H. J. Siegel, and M. J. Atallah, "Static Program Decomposition Among Machines in an SIMD/SPMD Heterogeneous Environment with Non-Constant Mode Switching Costs," in *Proceedings of the Heterogeneous Computing Workshop (HCW '94)*, pp. 58–65, Apr. 1994.

[3] R. F. Freund, "Optimal Selection Theory for Superconcurrency," in *Proceedings of Supercomputing '89*, pp. 47–50, Nov. 1989.

[4] L. Kleinrock, *Communication Nets: Stochastic Message Flow and Delay*, McGraw-Hill, New York, NY, 1964.

[5] Y. A. Li, J. K. Antonio, H. J. Siegel, M. Tan, D. W. Watson, "Estimating the Distribution of Execution Times for SIMD/SPMD Mixed-Mode Programs," in *Proceedings of the Heterogeneous Computing Workshop (HCW '95)*, pp. 35–46, Apr. 1995.

[6] Y. A. Li, *A Probabilistic Framework for Estimation of Execution Time in Heterogeneous Computing Systems*, Ph.D. Dissertation, School of Electrical and Computer Engineering, Purdue University, Aug. 1996.

[7] S. Chen, M. M. Eshaghian, A. Khokhar, and M. E. Shaaban, "A Selection Theory and Methodology for Heterogeneous Supercomputing," in *Proceedings of the Workshop on Heterogeneous Processing*, pp. 15–22, Apr. 1993.

[8] M. Tan, J. K. Antonio, H. J. Siegel, and Y. A. Li, "Scheduling and Data Relocation for Sequentially Executed Subtasks in a Heterogeneous Computing System," in *Proceedings of the Heterogeneous Computing Workshop (HCW '95)*, pp. 109–120, Apr. 1995.

[9] A. B. Tayyab and J. G. Kuhl, "Stochastic Performance Models of Parallel Task Systems," in *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 284–285, May 1994.

[10] A. M. Mood, F. A. Graybill, and D. C. Boes, *Introduction to the Theory of Statistics*, McGraw-Hill, New York, NY, 1974.

[11] D. Bertsekas and R. Gallager, *Data Networks*, Prentice-Hall, Englewood Cliffs, NJ, 1987.

[12] M. M. Eshaghian and R. F. Freund, "Cluster-M Paradigms for High-Order Heterogeneous Procedural Specification Computing," in *Proceedings of the Workshop on Heterogeneous Processing*, pp. 47–49, May 1992.

[13] J. B. Thomas, *An Introduction to Applied Probability and Random Processes*, Robert E. Krieger Publishing Company, Huntington, NY, 1981.

Biographies

Yan A. Li received his B.E. degree from Tsinghua University, Beijing, China, in 1991, and his MSEE and Ph.D. degrees, both from Purdue University, West Lafayette, Indiana, U.S.A., in 1993 and 1996, respectively. He is currently a Senior System Architect at Intel Corporation. He is a member of IEEE and Eta Kappa Nu. His major research interest includes parallel processing, high-performance heterogeneous computing, computer architecture, and computer systems simulation.

John K. Antonio received the B.S., M.S., and Ph.D. degrees in electrical engineering from Texas A&M University, College Station, TX. He currently holds the position of Associate Professor of Computer Science within the College of Engineering at Texas Tech University. Before joining Texas Tech, he was with the School of Electrical and Computer Engineering at Purdue University. During the summers of 1991-94 he participated in a faculty research program at Rome Laboratory, Rome, NY, where he conducted research in the area of high performance computing. His current research interests include heterogeneous systems, configuration techniques for embedded parallel systems, and computational aspects of control and optimization. He has co-authored over 50 publications in these and related areas. For the past four years, he has organized the Industrial Track and Commercial Exhibits portions of the International Parallel Processing Symposium. He is a member of the IEEE computer society and is also a member of the Tau Beta Pi, Eta Kappa Nu, and Phi Kappa Phi honorary societies. Organizations that have supported his research include the Air Force Office of Scientific Research, National Science Foundation, Naval Research Laboratory, Orincon, Inc., and Rome Laboratory.

Stochastic Petri Nets Applied to the Performance Evaluation of Static Task Allocations in Heterogeneous Computing Environments

Albert R. McSpadden and Noé Lopez-Benitez
Department of Computer Science
College of Engineering
Texas Tech University
Lubbock, Texas 79409-3104
{nlb, amcspadd}@cs.ttu.edu

Abstract

A Stochastic Petri Net (SPN) is systematically constructed from a task graph whose component subtasks are statically allocated onto the processor suite of a Heterogeneous Computing System (HCS). Given that subtask execution times are exponentially distributed, an exponential distribution can be generated for the overall completion time. In particular, the enabling functions and rate functions used to specify the SPN model provide needed versatility to integrate processor heterogeneity, task priorities, allocation schemes, communication costs, and other factors characteristic of a HCS into a comprehensive performance analysis. The manner in which these parameters are incorporated into the SPN allows the model to be transformed into a testbed for optimization schemes and heuristics. The proposed approach can be applied to arbitrary task graphs including non-series-parallel.

1. Introduction

Stochastic Petri Nets (SPN's) can be used as a versatile analytic tool for evaluating task graphs composed of tasks with exponentially distributed execution times allocated onto a finite set of heterogeneous distributed processors, i.e. Heterogeneous Computing System (HCS). HCS's are complex systems exhibiting diverse architectural capabilities which perform applications composed of subtasks with diverse execution paradigms [1]. In HCS theory, the motivation of qualitative matching of machine type with task type can be more important than the quantitative balancing of the load of tasks among the processors [2]. Optimally allocating application subtasks to HCS components as well as the specification of initiation times is known as the *task scheduling problem.* Hence, HCS is a rich field requiring qualitative and quantitative analysis of both the task at hand and the processing resources available to achieve an optimal allocation of tasks to the system. In this paper, a SPN-based methodology, augmented with enabling and rate functions, is discussed; it is shown that the modeling approach presented provides the flexibility and versatility necessary to meet the challenge of representing and analyzing complex HCS's.

Task graphs represent general computation jobs which have been decomposed into modules called tasks which must be executed according to some precedence constraints. Direct evaluation of task graphs provides an average completion time of the overall job assuming no restrictions exist on the number and architecture of processing units and with no regard to allocation schemes. When the task graph is executed on the processing elements of a HCS, estimating overall completion time becomes an optimization problem involving allocation of tasks to processors such that completion time is minimized. Before the problem of optimal allocations can be discussed, a method must be available for computing an expected completion time and deriving a probability distribution of the completion time for any given task graph, HCS, and allocation. A solution technique for series-parallel graphs relying on multiplication/convolution of parallel/series tasks is reported in [3]. Execution times of fork-join parallel programs in multiprocessor environments is discussed in [4]. The multiplication/convolution approach is applied to HCS at coarse and fine levels of granularity [5]. Also, in [6] performance prediction of fork-join task graphs is addressed, where the residence times of each task are estimated in terms of service demands and queuing delays; based on these estimations, the task graph is then systematically reduced. This approach is attractive because it avoids the state explosion encountered in Markov-based solutions. However, since only tightly coupled systems are addressed, there is no regard for preallocation schemes and communication costs are ignored.

0-8186-7879-8/97 $10.00 © 1997 IEEE

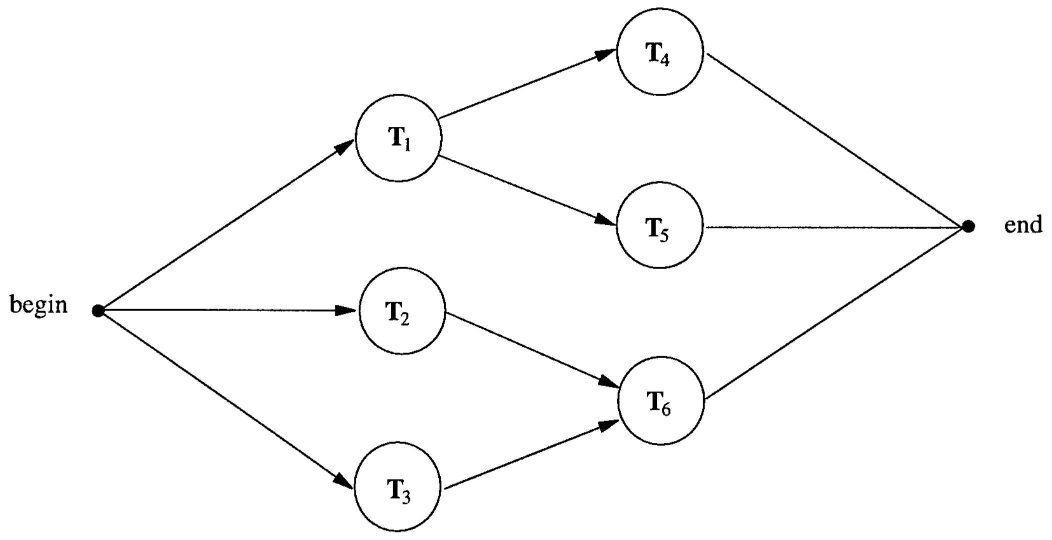


Figure 1. A simple task graph

A Markov-based solution technique of task graph systems has been reported in [7]; though limited to relatively small task graphs, this technique is used for the analysis of scheduling policies in [8]. SPN's directly incorporate the topological information of the input task graph and provide a systematic means for applying factors such as allocation schemes, processor heterogeneity, communication costs, and random execution times. Also, an analytical approach based on SPN's can be applied to arbitrary graphs which are acyclic, but not necessarily series-parallel. SPN tools can automatically generate Markov chains which are then solved to compute system performance characteristics such as a distribution of the overall completion time. Although the methodology described in this paper is not in itself an optimization technique, it can be used in conjunction with optimization techniques which attempt to search a space of completion time distributions [9]. Furthermore, the implementation of the proposed methodology can be easily adapted to become a testbed for various optimization heuristics.

2. HCS Model Parameters and Notation

Several formal definitions of the task scheduling problem in the context of HCS have been proposed [10, 1, 2]. The goal of such definitions is to express the precedence constraints and computational requirements of the application as well as the diverse processing capabilities of the HCS in a way such that performance evaluation and optimization can be mathematically formulated and resolved. For the analytical method of this paper, a HCS is assumed to be completely described by the following:

- a task graph $G(T,E)$ where the vertex set $T = \{T_1, T_2, \ldots, T_k\}$ consists of k tasks which compose some overall job and the edge set E consists of ordered pairs from T which correspond to data or control dependencies.

The topology of T is described in detail by the following:

– an in-degree vector $D = [d_1, d_2, \ldots, d_k]$ where d_i is the number of tasks which must complete before T_i may initiate execution.

– an out-degree vector $H = [h_1, h_2, \ldots, h_k]$ where h_i is the number of tasks which are spawned after the completion of T_i.

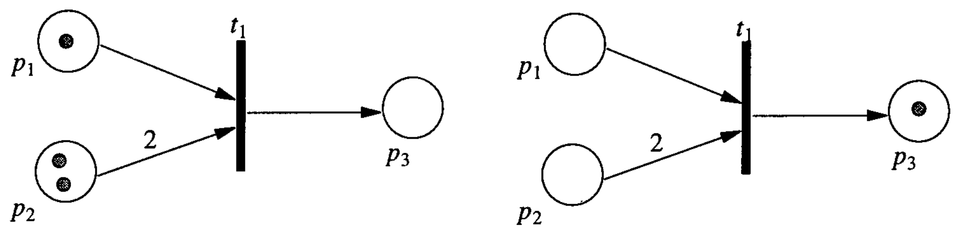
– a task graph structure $TG[i][j]$, $1 \le i \le k$, $1 \le j \le h_i$ where $TG[i]$ is an array specifying the h_i tasks which are spawned by the completion of T_i; thus, the ordered pair $(T_i, TG[i, j]) \varepsilon E$.

- a $k \times k$ matrix $pkt[i, j]$, $1 \le i, j \le k$ where $pkt[i, j]$ is the average number of data packets of standard size that is sent from T_i to T_j. Alternatively, these can be specified as edge weights for the elements of E.

- a priority vector $W = [w_1, w_2, \ldots, w_k]$ which induces a sequential ordering of any ready tasks assigned to the same processor; these priorities may be taken from the indices of the tasks, e.g. $w_i = k - i$, or they may be randomly or heuristically determined.

- a set $P = \{P_1, P_2, \ldots, P_n\}$ consisting of n processors composing a heterogeneous suite.

- a $k \times n$ execution time matrix $B[i, j]$, $1 \le i \le k, 1 \le j \le n$ where b_{ij} is the average execution



a) PN before transition fires b)PN after transition fires

Figure 2. Simple Petri net

time of T_i on P_j.

• an $n \times n$ communication time matrix $C[r, s]$, $1 \le r, s \le n$ where each entry c_{rs} is the average communication time to transfer a data packet of standard size from P_r to P_s.

• a $k \times n$ static allocation matrix $A[i, j]$, $1 \le i \le k, 1 \le j \le n$ where entry $a_{ij} = 1$ if T_i has been allocated to P_j, and 0 otherwise.

Task graphs are assumed to be series-parallel for several approaches to performance evaluation [11] and optimization [9]; however, this limitation is avoided in the SPN-based methodology of this work. Fig. 1 shows a simple task graph which will be used to illustrate the transformation of task graphs into SPNs.

3. Review of Basic Petri Net Concepts

A Petri net is a directed graph whose underlying graph N is directed, weighted, and bipartite [12]. Petri nets are bipartite in that nodes are of two types, *places* and *transitions*, with arcs occurring either from places to transitions or from transitions to places. When an arc is from a place p to a transition t, then p is an input place of t; a place p is an *output place* of t if an arc proceeds from t to p. Places and transitions are represented pictorially by circles and thin rectangles, respectively.

A third component of any PN are tokens which reside in places; pictorially, tokens are represented by dots within the perimeters of places. Tokens are transferred from one place to another by the firing of transitions. When a transition t fires, tokens are removed from all input places of t and placed in the output places of t. PN's enforce a logical flow of activity via the rule for enabling and firing of transitions. According to this rule, a transition can fire if it has been enabled, and it is enabled if all of its input places possess at least one token. An arc may be weighted where the weight specifies the number of tokens which must reside in an input place in order for a transition to be enabled, or the number of tokens placed in an output place by the appropriate transition; if weight is unspecified then it is assumed to be one. The PN in Fig. 2a depicts a system state in which both preconditions for an event have been fulfilled; Fig. 2b shows the resulting state after the occurrence of the event.

PN's and their dynamic behavior can be captured in mathematical notation via state vectors. Given a PN with k places, a marking q of the PN is denoted by M_q; a marking is described by a k – vector whose ith component denotes the number of tokens in place p_i; an initial marking of the PN is denoted by M_0. A particular PN with an underlying graph N is denoted (N, M_0). For the simple example in Fig. 2, the associated markings are $M_0 = [1\ 2\ 0]$ and $M_1 = [0\ 0\ 1]$. The reachability graph of a PN is a graph $G_R(M, \Delta)$ where the vertex set M is the set of all possible markings for the PN and the edge set Δ consists of all possible transition firings transforming one marking to another.

Stochastic Petri nets are PN's in which there is an exponentially distributed delay time between the enabling and firing of transitions. The reachability graph of a bounded SPN is isomorphic to a finite Markov chain (MC) [13]; in particular, the markings of the reachability graph comprise the state space of a MC, and the transition rate between any two states X_i and X_j is the sum of all firing delays for transitions transforming M_i into M_j. *Generalized stochastic Petri nets* (GSPN) have been proposed [14] in which transitions are of two types: *timed* transitions which have the exponentially determined firing rates and *immediate* transitions which have no firing delay and have priority over any timed transition. *Enabling functions* are marking dependent functions which can be defined on each transition as a switching mechanism. Transition priorities (timed vs. immediate), and enabling functions are logically equivalent extensions of SPN which endow them with the full computational power of Turing machines [15].

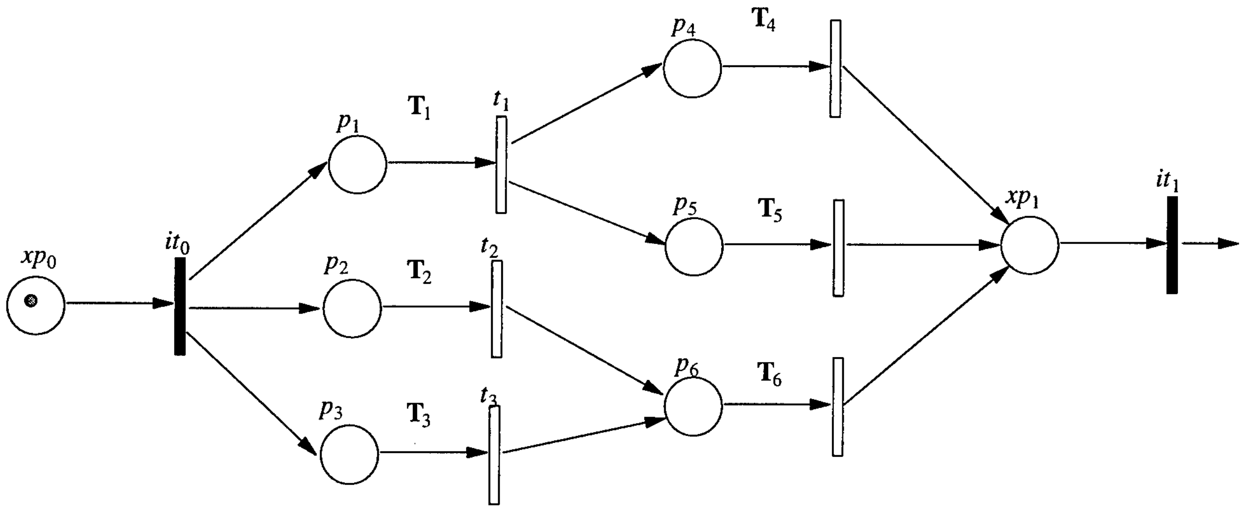


Figure 3. SPN model of the task graph from Fig. 1

4. Stochastic Petri Nets Applied to Task Graph Analysis

The transformation of a task graph into a SPN begins with the association of each task T_i with a place/timed transition pair, p_i and t_i. Fig. 3 shows the SPN corresponding to the task graph in Fig. 1. Auxiliary places xp_0 and xp_1 and immediate transitions it_0 and it_1 are used to enforce initiation and completion conditions, respectively, for the overall job. The presence of at least one token in a place may represent the fulfillment of all preconditions for the initiation of the task. The firing of a timed transition represents the completion of execution of the corresponding task. The delay time of each transition corresponds to the exponentially distributed execution time of the task. A place p_i can be associated with the in-degree d_i to enforce precedence constraints. Initially, the presence of a token in xp_0 enables it_0; the firing of it_0 represents the initiation of an execution cycle. The presence of three tokens in xp_1 and the firing of it_1 indicates that an execution cycle has been completed.

Timed transitions in the SPN model in Fig. 3 will fire once enabled according to an exponentially distributed delay. The markings generated correspond to the possible execution states of the system, where a system state is defined by the tasks which are executing concurrently. The reachability graph generated by the SPN reflects the space of potential execution paths for the task graph. Depending on the number of processing units, different reachability graphs can be generated with the same model if different enabling functions are associated with the timed transitions. Fig. 4 depicts a partial reachability graph for the SPN model shown in Fig. 3 under the assumption of an unlimited number of homogeneous processors.

Let $M_i = [x_i(p_j)]$, $1 \le j \le k$ denote a partial description of the ith marking, where $x_i(p_j)$ is the number of tokens in place j. Note that this description of a marking sets up a one-to-one correspondence between markings and system states. Consider the firing of the immediate transition it_1 from the initial marking M_0. A token is removed from xp_1 and one token is placed in places p_1, p_2, and p_3, respectively. The resulting marking is a tangible marking $M_1 = [1\ 1\ 1\ 0\ 0\ 0]$. Then, depending on restrictions due to the number of available processors, some or all of transitions t_1, t_2, and t_3 are enabled and can fire from M_1. Note that the initial marking indicates the presence of a token in xp_1, but it is shown in Fig. 4 as $M_0 = [0\ 0\ 0\ 0\ 0\ 0]$ because it is only described in terms of places corresponding to actual tasks of the original task graph.

Consider some marking M_i in which task T_6 should be ready to run. To make this possible, both T_2 and T_3 must have completed; this will be indicated by the presence of two tokens in p_6, i.e. $x_i(p_6) = 2$. To capture this precedence constraint it suffices to associate each input arc into a timed transition with a weight corresponding to the in- degree of each node in the task graph.

Given that the execution times of the tasks are exponentially distributed, then the firing rates of the transitions in the SPN are exponentially distributed. This makes the reachability graph of Fig. 4 equivalent to a continuous time MC (CTMC). Therefore, once the topological properties of the task graph have been used to build the SPN and the timed transition firing rates have been identified with the task execution rates, a CTMC can be generated.

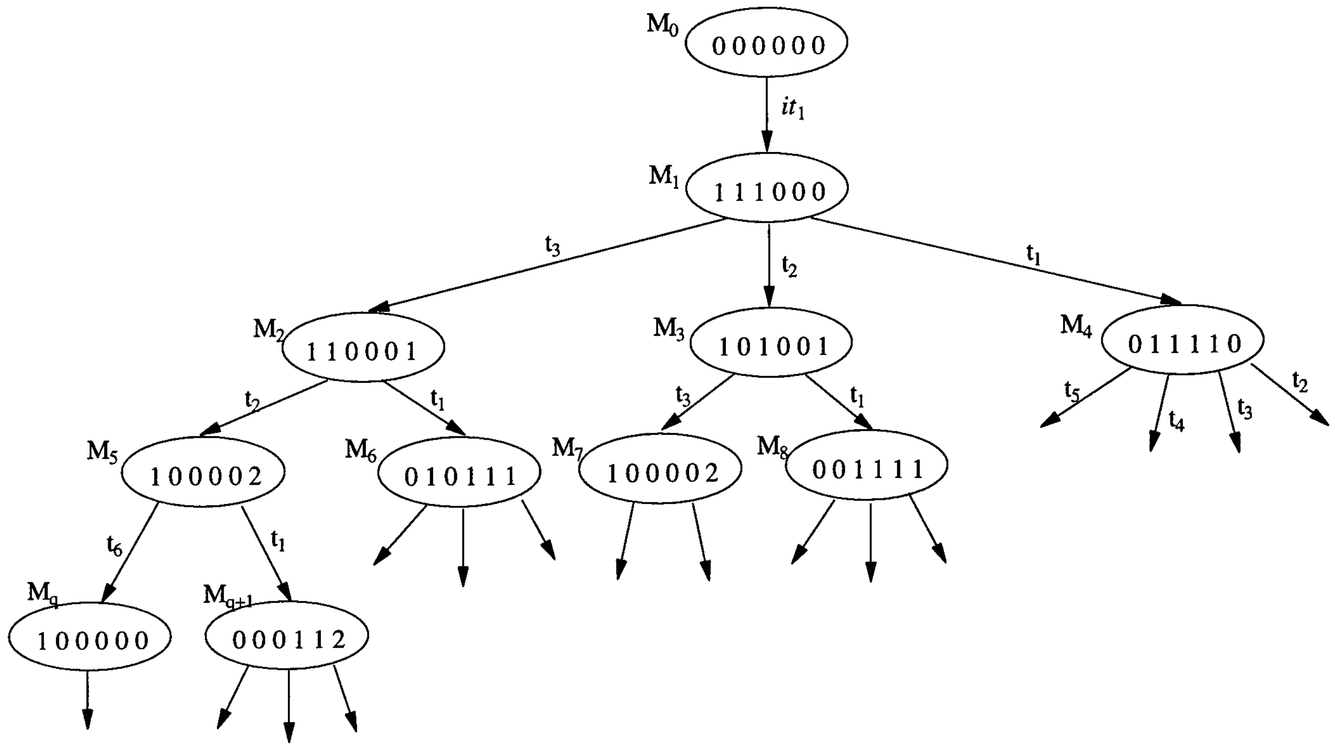


Figure 4. Partial reachability graph assuming unlimited number of processors

The CTMC model is then used to conduct a complete stochastic analysis of the underlying system. Since SPNs provide a natural representation of parallelism and synchronization, SPN models have been used to represent queuing networks, failure/repair models, and task graphs [16]. SPNs have been used to analyze specific multiprocessor systems and individual parallel or concurrent programs [17, 18, 19]. Dependability and performance analysis of systems characterized by distributed programs, distributed files and remote processing has also been modelled using SPN's [20]. The methodology of this paper was implemented with the SPNP software tool to systematically construct the SPN and conduct CTMC analysis [21]. In [22], SHARPE [3] is used to validate results obtained using SPN-based models similar to the models reported in this paper.

5. Incorporating HCS Parameters into SPN Models

Thus far, two systematic steps have been mentioned in the construction of a SPN model from the HCS parameters in Section 2: 1) the association of a place/timed-transition pair with each T_i in the vertex set of G and 2) weighting the input arc into each timed transition, t_i, with d_i. The result is that the SPN so constructed is topologically equivalent to G and fully captures the precedence relationships inherent in G. The dynamic behavior of the system resulting from factors not specified in the topology of G can be modeled by means of *enabling functions* and *rate functions*. Both types of functions manipulate the firing rate of transitions by incorporating dynamic information drawn from the evolution of the reachability graph as well as non-task-graph information concerning the processing system. Enabling functions act as switching mechanisms to turn timed-transitions on and off based on the availability or unavailability of processors to which tasks have been allocated. Rate functions can specify the appropriate rate of timed-transitions based on the task/processor combinations and processor-to-processor communication costs. Referring to the parameters proposed for a HCS in Section 2, let $F = [f_i]$, $1 \le i \le k$ define a vector of enabling functions such that for the jth column of A:

$$\sum_{i=1}^{k} f_i a_{ij} \le 1$$

This condition asserts that no more than one timed transition is ever enabled, i.e., no more than one task per processor is ever executing. The vector F can be defined in terms of another enabling vector $E = [e_i]$, $1 \le i \le k$ whose components are marking-dependent values for a marking M_q where I is the indicator function:

$$e_i = I(x_q(p_i) = d_i)$$

Thus, E captures the precedence relations in the original task graph. However, when two or more tasks could run concurrently but have been allocated to the same processor, they must be serialized and selected for execution according to some predefined priority scheme. To account for this necessity, let $\hat{A} = [\hat{a}_{ij}]$ define a weighted allocation where:

$$\hat{a}_{ij} = a_{ij} e_i w_i$$

Let $V = [v_{ij}]$ denote another $k \times n$ matrix which is determined by examining $\hat{A}$ such that for the jth column of V:

$$v_{ij} = I(w_i = \max\{\hat{a}_{qj}, i \le q \le k\})$$

Effectively, $v_{ij} = 1$ if T_i is the ready task with the highest priority on P_j and is 0 otherwise. Finally, using matrix V the enabling vector F can be obtained as follows:

$$f_i = I\left(\bigcup_{j=1}^{n} v_{ij} = 1\right)$$

As a result vector F accounts for the restriction on the number of processors and the allocation scheme.

Matrix V can also be utilized to determine a marking-dependent transition rate for the timed transitions corresponding to each T_i. Let $\hat{B}(t) = [\hat{b}_i(t)]$, $1 \le i \le k$ denote a column vector such that $\hat{b}_i(t) = b_{ij}$ if T_i is allocated to P_j and enabled at M_q, and is 0 otherwise. Then $\hat{B}(t)$ can be computed such that $\hat{B}(t) = [\mathbf{b}_i \mathbf{v}_i^T]$, $1 \le i \le k$ where $\mathbf{b}_i$ denotes the ith row of B and $\mathbf{v}_i^T$ denotes the transpose of the ith row of V. Let $\lambda = [\lambda_i]$, $1 \le i \le k$ denote a column vector specifying the effective firing rates for the k timed transitions at the marking M_q, where:

$$\lambda_i = \begin{cases} \dfrac{1}{\hat{b}_i(t)} & \text{if } \hat{b}_i(t) \ne 0 \\ \textit{undefined otherwise} \end{cases}$$

However, the above development of enabling and rate functions does not involve any consideration of network communication costs. Incorporation of communication costs into the SPN model of the HCS can be approached in different ways depending on the assumptions made about underlying network capabilities and the nature of task/network interaction. As with task execution times, the communication times are assumed to be exponentially distributed. Here two approaches are presented based on two types of interconnection networks: (a) a high-performance network characterized by high-connectivity and parallel communications and (b) a bus-oriented network with low-connectivity. In both cases, output data is assumed to be accumulated in a buffer during task execution and transmitted after task completion.

5.1. Case 1: Modeling High-performance Communication Networks

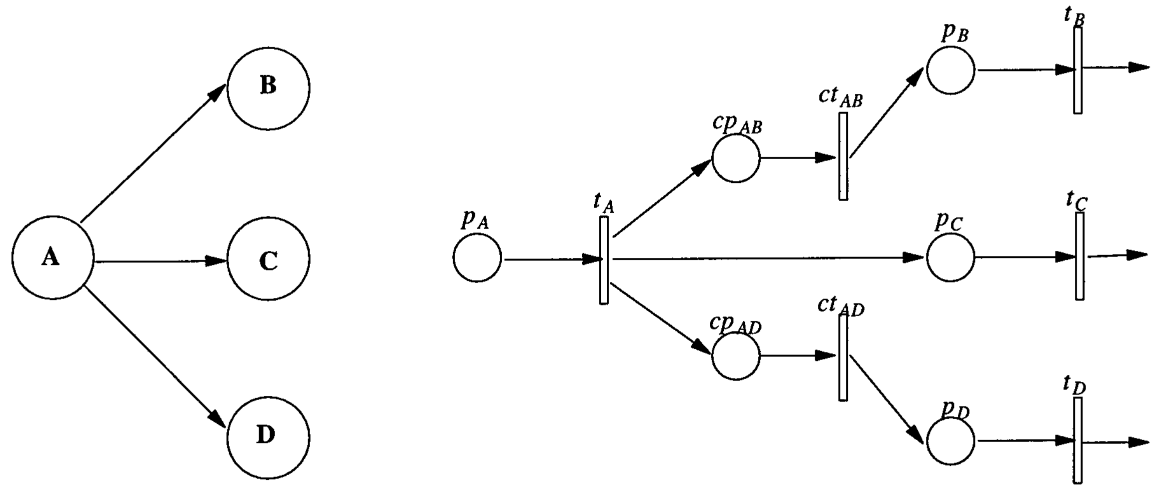
High-performance communication networks can be characterized as expensive systems in which inter-node communication takes place on dedicated, point-to-point links. Data intended for each successor is written to a separate buffer. Furthermore, each processor may be coupled with a front-end communication processor which enables parallel communication. In terms of a task graph, once a given task completes, successor tasks experience initiation delay equal to the data transfer time for all intended packets; ideally, any successor task allocated to the same processor as the parent task should be able to begin execution immediately after the completion of the parent. The properties of such a high-performance network can be modeled in an SPN by inserting additional place/timed-transitions to represent each individual communication; augmentation of the task graph with communication nodes has been proposed for CTMC- based analysis [23]. Each timed-transition inserted is associated with an exponentially distributed delay whose parameter is the average communication time between the host processors. Thus, given a completed task T_i allocated to processor P_r and a successor task T_j allocated to P_s, the average communication rate assigned to the transition modeling the transfer of data is given by:

$$\delta_{ij} = \frac{1}{c_{rs} pkt[i, j]}$$

Fig. 5a illustrates a segment of some task graph in which Task A spawns tasks B, C, and D. Suppose the four tasks are allocated to three processors such that A and C are allocated to one processor, and B and D are allocated to the other two processors, then the resulting SPN for Case 1 would be as shown in Fig. 5b. Note the insertion of place/transition pairs between A and B and A and D to represent the individual communications involved.

5.2. Case 2: Modeling Bus-Oriented Networks

In interconnection networks characterized by low-connectivity, groups of processors may have to share common communication links, as is the case with a bus-oriented architecture. Also, in lower cost systems processors may be forced to expend computation cycles on communication processing. If, additionally, output data packets for successor tasks are queued up in a single buffer in



a) Segment of a task graph b) SPN with communication nodes

Figure 5. SPN model assuming a high-performance network

some random ordering and transmitted on a FIFO basis, then it is highly unlikely that a successor task will receive all of its packets before any other successor task. In terms of the example in Fig. 5a, if the processor to which task A is allocated must broadcast packets in random order to the processors associated with tasks B, C, and D , then it is reasonable to assume that on average B, C, and D will experience uniform initiation delay. Such behavior can be reflected in the SPN simply by modifying the rate function governing the firing of the transitions associated with each task. In this case, no extra nodes are inserted in the SPN model. Rather, the firing delay of each transition is increased by the sum of communication costs associated with each successor task. Let T_i be allocated to P_j where completion of T_i spawns $m = h_i$ tasks $T_{q_1}, T_{q_2}, \ldots, T_{q_m}$ which are allocated to processors $P_{y_1}, P_{y_2}, \ldots, P_{y_m}$. Then a modified firing rate for transition t_i is given by:

$$\tilde{\lambda}_i = \frac{1}{b_{ij} + \sum_{k=1}^{m} c_{jy_k} pkt[i, q_k]}$$

It should be noted that in reality a given network may be heterogeneous with respect to interconnection capabilities. In this case the SPN model can be systematically constructed to appropriately model each segment of the network, reflecting the different sets of assumptions mentioned above.

The net result is that an SPN with dynamically determined transition rates and enabling functions can represent the full interplay of task precedence relationships, allocations specifications, availability of idle processors, diverse execution rates across a heterogeneous suite, and communication costs. Assuming exponentially distributed execution and communication times, an overall completion time distribution can be generated which is itself exponentially distributed. In addition, the *Mean Time To Completion* (MTTC) for the overall graph is computed.

6. Numerical Example

To reinforce the concepts and notation involved in the methodology proposed, a straightforward numerical example follows which is based on the 13-node non-series-parallel task graph of Fig. 6. Although the example is simple, it illustrates the versatility of the method and the direct manner in which the space of task allocations and prioritizations can be traversed. As mentioned above, this method is not an optimization scheme but an analytical tool which can be readily harnessed to implement the objective function of optimization approaches.

The edge weights indicated in Fig. 6 are the number *pkt* of standard size packets transmitted from one task to its successor. The following matrix specifies an arbitrary allocation of the tasks of this graph onto a network of 6 processors:

$$A^T = \begin{bmatrix} 1\,0\,0\,0\,0\,0\,0\,0\,0\,0\,1\,0\,0 \\ 0\,1\,0\,0\,0\,0\,0\,0\,0\,1\,0\,1\,0 \\ 0\,0\,1\,0\,0\,0\,0\,0\,1\,0\,0\,0\,1 \\ 0\,0\,0\,1\,0\,0\,0\,1\,0\,0\,0\,0\,0 \\ 0\,0\,0\,0\,1\,0\,1\,0\,0\,0\,0\,0\,0 \\ 0\,0\,0\,0\,0\,1\,0\,0\,0\,0\,0\,0\,0 \end{bmatrix}$$

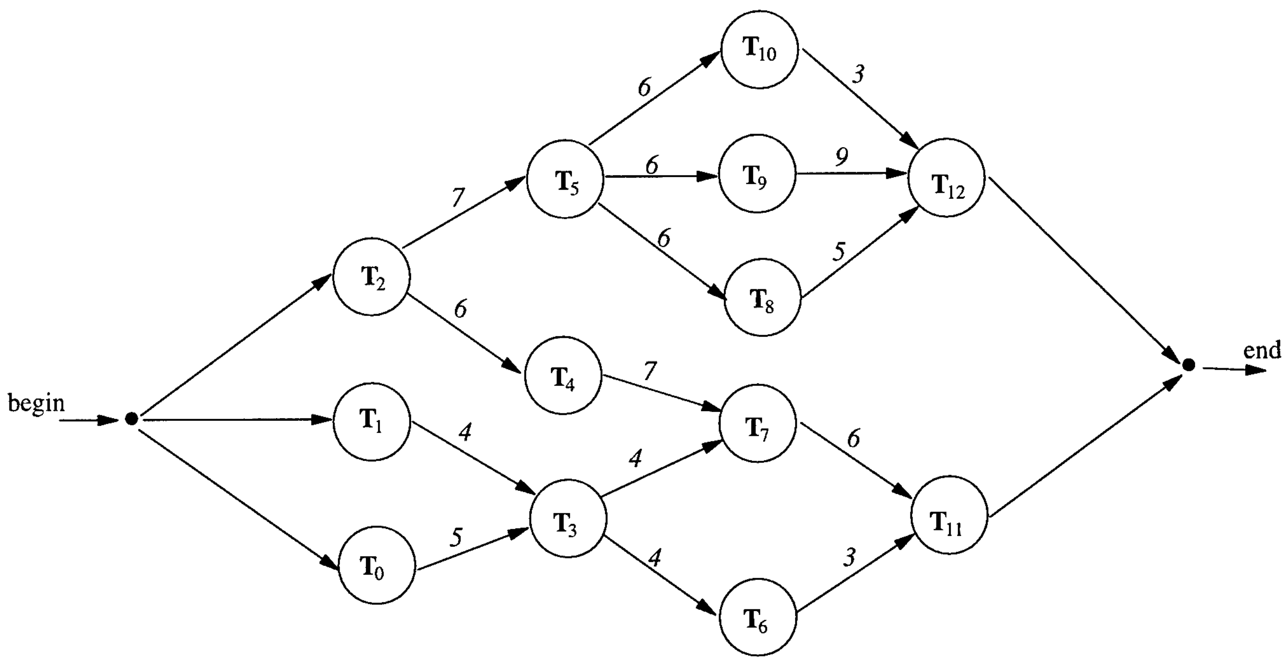


Figure 6. A 13-node complex task graph

The matrix B specifies the spectrum of execution times for each task across all processors of the system in standard time units per execution:

$$B^T = \begin{bmatrix} .9 & 2 & .3 & 1 & .3 & 4 & 2 & 1 & 3 & 2 & .3 & .2 & .1 \\ .3 & 4 & .3 & 1 & .3 & 5 & 2 & 1 & 3 & 4 & .5 & .5 & .1 \\ .5 & 1 & .5 & 1 & .3 & 5 & 2 & 1 & 4 & 2 & 5 & .2 & .3 \\ .5 & 2 & .2 & 2 & .3 & 5 & 2 & 2 & 2 & 2 & .5 & .2 & .1 \\ .5 & 2 & .3 & 1 & .6 & 5 & 3 & 1 & 3 & 2 & .5 & .2 & .1 \\ .5 & 2 & .3 & 1 & .3 & 7 & 1 & 1 & 3 & 2 & .3 & .2 & .1 \end{bmatrix}$$

The communication delays per data packet in the interconnection network between the six processors are characterized by the matrix C in terms of standard time units per packet:

$$C = \begin{bmatrix} 0 & .1 & .1 & .2 & .2 & .1 \\ .1 & 0 & .4 & .3 & .2 & .1 \\ .1 & .4 & 0 & .2 & .3 & .3 \\ .2 & .3 & .2 & 0 & .3 & .2 \\ .2 & .2 & .3 & .3 & 0 & .1 \\ .1 & .1 & .3 & .2 & .1 & 0 \end{bmatrix}$$

Relative priorities among the 13 tasks are specified thus:

$$W = [13\ 12\ 11\ 8\ 9\ 10\ 7\ 6\ 5\ 4\ 3\ 1\ 2]$$

It should be noted that this priority scheme is entirely arbitrary as is the initial allocation.

The plots in Fig. 7 correspond to the probability of completion at time t, $P(X \leq t)$ of the overall job based on two possible allocations; also, three communication scenarios are considered: a) there are no communication costs, b) communication occurs over a high- performance network (Case 1 outlined above) , and c) communication takes place over a low-performance network (Case 2 above). The *MTTC* in each case is 16.2272, 21.6014, and 26.3359, respectively.

Obviously, it is easy to find a better allocation than the one specified by A. A better matching is derived from the first allocation by moving each T_i from the current processor j to $(j+1)$ *mod* 6. With the new allocation the MTTCs are reduced to 7.7928, 11.7537, and 15.0375.

One further important numerical result relates to the size of the reachability graph generated by each approach to modeling communication costs. When communication costs are modeled by modified transition rates without the insertion of new nodes into the SPN, the reachability graph consisted of 122 markings and 305 marking-to-marking transitions; if new place/transition pairs are inserted into the SPN to model communication effects, then the reachabilty graph grows to 2576 markings and 9922 marking-to-marking transitions, indicating the state-space limitations of this approach.

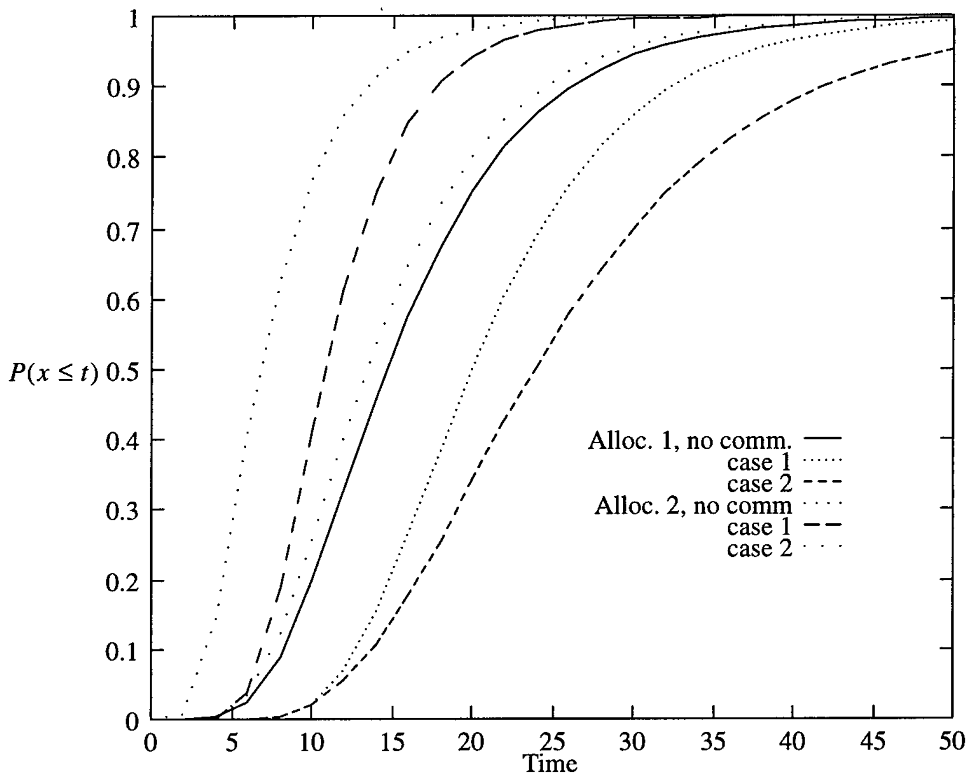


Figure 7. CDF of completion time given static allocation and network type

7. Conclusions

A direct method for analyzing static allocation schemes of task graphs in the context of HC environments has been presented. Task graphs can be systematically translated into unique SPN models which are then modified to account for the parameters of a HCS. A direct evaluation of the SPN model estimates the average execution time of the job represented and generates an exponential distribution for completion time. Further areas of investigation include incorporation of multiple data copies and task replication into the model. Also, the number of system states increases with the complexity of the task system, indicating the need for approximate solutions using state reduction techniques. The manner in which HCS parameters are incorporated into the SPN suggests the potential of joining this methodology with heuristics for optimization by means of perturbing or otherwise exploring allocations and priority schemes.

8. Acknowledgements

The authors wish to aknowledge the comments of several anonymous reviewers which greatly improved this paper. This work was partially supported by the National Science Foundation under Grant No. CCR-9520226.

References

[1] H. J. Siegel, J. K. Antonio, R. Metzger, M. Tan, and Y. Li, "Heterogeneous Computing," in *Parallel and Distributed Computing Handbook,* A. Zomaya, ed., McGraw-Hill, New York, 1996.

[2] H. J. Siegel, H. Dietz, and J. K. Antonio, "Software Support for Heterogeneous Computing," in *CRC Handbook of Computer Science and Engineering,* Tucker Jr. A. B., ed., CRC Press, 1997.

[3] R. A. Sahner and K. S. Trivedi, "Reliability modeling using SHARPE," *IEEE Trans. Reliability,* Vol. R-36 No. 5, June 1987, pp. 186-193.

[4] D. Towsley, C. G. Rommel, and J. A. Stankovic, "Analysis of Fork-Join Program Response Times on Multiprocessors," *IEEE Trans. Parallel and Distributed Systems,* Vol. 1, No. 3, July 1990, pp. 586-303.

[5] Y. A. Li, *A Probabilistic Framework for the Estimation of Execution Times in Heterogeneous Computing Systems,* Ph.D. dissertation, Purdue University, School of Electrical Engineering, 1996.

[6] V. W. Mak and S. F. Lundstrom, "Predicting Performance of Parallel Computations," *IEEE Trans. Parallel and Dis-*

tributed Systems, Vol. 1, No. 3, July 1990, pp. 557-270.

[7] A. Thomasian and P. F. Bay, "Analytic Queueing Network Models for Parallel Processing of Task Systems," *IEEE Trans. Comp.,* Vol. C-35, No. 12, Dec. 1986, pp. 1045-1054.

[8] D. A. Menasce, D. Saha, S. C. Da Silva Porto, V. A. F. Almeida, and S. K. Tripathi, "Static and Dynamic Processor Scheduling Disciplines in Heterogeneous Parallel Architectures," *Parallel and Distributed Computing,* Vol. 58, 1995, pp. 1-18.

[9] P. Shroff, D. Watson, N. Flann, and R. Freund, "Genetic Simulated Annealing for Scheduling Data-Dependent Tasks in Heterogeneous Environments," *Proceedings Heterogeneous Computing Workshop 96,* 1996, pp. 98-103.

[10] B. Narahari, A. Youssef, and H. Choi, "Matching and Scheduling in a Generalized Optimal Selection Theory," *Proc. Heterogeneous Comp. Workshop ,* 1994, pp. 3-8.

[11] R. A. Sahner and K. S. Trivedi, "Performance and Reliability Analysis Using Directed Acyclic Graphs," *IEEE Trans. Software Engineering,* Vol. SE-13 No. 10, Oct. 1987, pp. 1105-1114.

[12] T. Murata, "Petri Nets: Properties, Analysis and Applications," *Proc. IEEE,* Vol. 77, No. 4, Apr. 1989, pp. 541-580.

[13] M.K. Molloy, "Performance Analysis Using Stochastic Petri Nets," *IEEE Trans. Comp.,* Vol. C-39 No. 9, Sept. 1982, pp. 913-917.

[14] M. A. Marsan, G. Conte, and G. Balbo, "A Class of Generalized Stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems," *ACM Trans. Computer Systems,* Vol. 5 No.2, May 1984, pp. 93-122.

[15] G. Ciardo, "Toward a Definition of Modeling Power of Stochastic Petri Net Models," *Proceeding Intn'l Workshop on Petri Nets and Performance Models,*

[16] C. G. Cassandras, *Discrete Event Systems, Modeling and Performance Analysis,* Irwin Inc. and Aksen Associates, Inc., 1993.

[17] Balbo G., S. Donatelli, and G. Franceshinis, "Understanding Parallel Program Behavior through Petri Net Models," *J. Parallel and Distributed Comp.,* Vol. 15, 1992, pp. 171-187.

[18] Balbo G., S. Donatelli, G. Franceshinis, A. Mazzeo, N. Mazzocca, and M. Ribaudo, "On the COmputation of Performance Characteristics of Concurrent Programs using GSPNs," *Performance Evaluation,* Vol. 19, 1994, pp. 195-222.

[19] Q. Jin and Y. Sugasawa, "Representation and Analysis of Behavior for Multiprocess Systems using Stochastic Petri Nets," *Math and Comp. Modelling,* Vol. 52, 1995, pp. 109-118.

[20] N. Lopez-Benitez, "Dependability Modeling and Analysis of Distributed Programs," *IEEE Trans. Software Engineering,* Vol. 50 No. 5, May 1994, pp. 345-352.

[21] G. Ciardo, Fricks R. M., J. K. Muppala, and K. S. Trivedi, *SPNP User's Manual ,* Version 3.1, Duke University. Dept. of Electrical Engineering, 1993.

[22] J. F. Decker, *Systematic Generation and Evaluation of Stochastic Petri Net Models for the Performance Analysis of Task Graphs,* Master Thesis, Dept. of Comp. Sci., Texas Tech University, 1995.

[23] K. C. -Y. Kung, *Concurrency in Parallel Processing Systems,* Ph.D. Dissertation, Dept. of Comp. Sci., University of California, 1984.

Author Biographies

Albert R. McSpadden is currently pursuing graduate study in Applied Mathematics at Texas Tech University. He completed his M.S. in Computer Science from Texas Tech in December 1996. During his graduate study he conducted research in the areas of performance evaluation, stochastic processes, and applications of Petri Net theory. His research interests in addition include distributed computing, applied linear algebra and computational methods.

Noé Lopez-Benitez received the BS degree in Communications and Electronics from the University of Guadalajara, Guadalajara, Mexico. The MS degree in Electrical Engineering from the University of Kentucky, and the PhD in Electrical Engineering from Purdue University in 1989. From 1980 to 1983, he was with the IIE (Electrical Research Institute) in Cuernavaca, Mexico. From 1989 to 1993, he served in the Dept. of Electrical Engineering at Louisiana Tech University. He is now a Faculty member in the Dept. of Computer Science at Texas Tech University. His research interests include fault-tolerant computing systems, reliability and performance modeling and distributed processing. He is a member of the IEEE, ACM and The Society for Computer Simulation.

Supporting Fault-Tolerance in Heterogeneous Distributed Applications

Piyush Maheshwari and Jinsong Ouyang
School of Computer Science and Engineering
The University of New South Wales, Sydney, NSW, Australia 2052
{piyush, jinsong}@cse.unsw.edu.au

Abstract

Heterogeneous computing opens up new challenges and opportunities in fields such as parallel and distributed processing, design of algorithms for applications, scheduling of parallel tasks, interconnection network technology and support for reliable distributed heterogeneous computing. A trend of supporting fault-tolerance in distributed computing systems is to incorporate fault-tolerance into applications at low cost, in terms of both run time performance and programming effort required to construct reliable application software. We present an approach for developing efficient reliable distributed applications for heterogeneous computing systems. In this paper we propose a library prototype, called H-Libra, *to support fault-tolerance in heterogeneous systems with low run-time cost. Fault-tolerance is based on distributed consistent checkpointing and rollback-recovery integrated with a user-level network communication protocol. By employing novel mechanisms, minimum communication overhead is involved for taking a consistent distributed checkpoint and catching messages in transit during a checkpoint. By providing fault-tolerance transparency and a simple, easy to use high-level message-passing interface,* H-Libra *simplifies the development of reliable heterogeneous distributed applications.*

1: Introduction

Hardware and software heterogeneity arises in many computing environments, for example, in an academic department with different experimental research machines and software systems. A distributed heterogeneous computing system (DHCS) consists of a connected set of traditional computer systems. Open architectures, workstations and multicomputers are a natural environment for heterogeneity. A simple heterogeneous computing environment is a departmental network with some SUN workstations, some DEC workstations and a high-performance graphics workstation. Heterogeneous computing (HC) is also a promising cost-effective approach to the design of high-performance parallel computers, which generally incorporates proven technology and existing designs and reduces new design risks from scratch [5, 8].

In recent years the abundance of variety of workstations and networked computers has established distributed computing as a mainstream paradigm suitable to achieve high utilisation of available computing resources. In a setting consisting of a potentially large number of heterogeneous computers connected by an unreliable network, fault-tolerance becomes a major issue. Naturally the new challenge is to incorporate fault-tolerance into applications at low cost in terms of both run-time performance and programming effort required to construct the application software. The combined complexity of dealing with network communications and fault-tolerance makes the development of efficient reliable distributed software on heterogeneous systems difficult.

There are basically three types of approaches that can be used to support fault-tolerance in distributed applications. 1. Coding within applications to explicitly deal with the potential failures during program execution. For distributed heterogeneous applications, it is tremendously complex to do so and software development costs are simply too high. 2. Replication. By running n instances of an application on different processing resources, the computation can still proceed even if $n-1$ instances fail. This is a very useful approach especially in real-time systems while, for general-purpose distributed applications, the cost of replication is too expensive. In fact, for a DHCS it may not be feasible at all, if individual processes are meant to be run on specific machines. 3. Checkpointing and rollback-recovery. It has been widely considered as a

0-8186-7879-8/97 $10.00 © 1997 IEEE

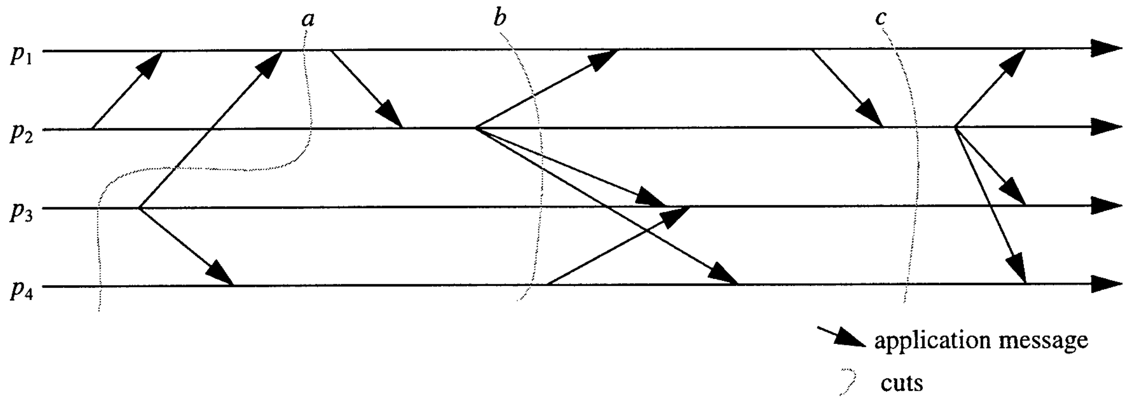


Figure 1: Consistent and inconsistent system states

general way to provide fault-tolerance in distributed systems. Our approach integrates distributed checkpointing and rollback-recovery protocols with a network communication protocol, called the user-level reliable transmission protocol (URTP).

The rest of the paper is organised as follows. Section 2 provides some background to distributed checkpointing and rollback-recovery mechanisms. Section 3 describes our checkpointing and rollback-recovery protocols including the URTP protocol. Section 4 describes the architecture and implementation of our library prototype, called H-Libra, which supports reliable heterogeneous distributed computing at low run-time cost. Section 5 concludes the paper.

2: Background

Though a distributed heterogeneous computing system provides a cost-effective approach to parallel and distributed processing, its reliable use depends on careful planning and design. The key issue of supporting fault-tolerance in distributed systems using checkpointing and rollback-recovery is how to obtain a consistent state of a distributed system. Chandy and Lamport [2] formally define the concept of a consistent distributed system state, and introduce an algorithm by which a process in a distributed system determines a global state of the system during a computation.

Briefly, a set of process states forms a consistent distributed system state if it satisfies the following condition: *For each message among the processes, if it is recorded in the state of the receiving process, it must also be recorded in the state of the sending process.* Informally, we can use a time diagram to describe a system's execution, where horizontal lines are time axes of executing processes, and messages are represented by arrows. For example, in Figure 1, p_1, p_2, p_3, and p_4 are four processes, and *a*, *b*, and *c* are cuts (sets of process states) each of which forms a distributed system state. According to the definition, cuts *b* and *c* are consistent cuts, while cut *a* is an inconsistent cut, as process p_1 recorded its state after it received the message while process p_3 recorded its state before it sent the message. If the system restarts from system state *a*, process p_1 restarts from a point where it already received the message from p_3, but p_3 restarts from a point where it has not sent the message to p_1, so process p_1 will actually receive the message from p_3 twice. This incorrect execution results from the inconsistency of cut *a*. Another important fact is that although cut *b* is a consistent distributed system state, the messages to processes p_1, p_3 and p_4 must be recorded in some way, otherwise message losses will occur if the system restarts from state *b*.

A variety of approaches to checkpointing and rollback-recovery have been proposed in the literature. Some are based on *independent checkpointing* [7, 18, 19, 21], while others use *consistent* or *coordinated checkpointing* [1, 2, 4, 9-11, 17]. Processes, using an independent checkpointing protocol, perform their message logging and checkpointing independently. With message logging, every process can detect its dependency on the states of other processes with which it communicates, and the dependency control information enables a reconstruction of a consistent distributed system state following a failure, using process rollback and message replay. By using consistent checkpointing, checkpointing of processes is synchronised in such a way that the resulting distributed checkpoint forms a consistent system state.

Our approach of checkpointing is basically a consistent checkpointing scheme which is different from other schemes in many sense. A message is *in transit* if it was sent within the previous checkpoint interval and is received within the current checkpoint interval. Our checkpointing protocol involves minimum communication overhead for constructing a consistent distributed checkpoint in a distributed system and catching messages in transit. It provides tolerance to message losses due to site failures or unreliable non-FIFO networks. The protocol reduces the run-time overhead, thus enhances the efficiency of reliable distributed applications [13].

Incorporating efficient checkpointing and rollback-recovery, we propose a library prototype, called H-Libra which transparently supports fault-tolerance in distributed heterogeneous applications. H-Libra is an extension of our previous library (called Libra) for homogeneous distributed systems [13, 14] which implements our distributed consistent checkpointing and rollback-recovery protocols, including a user-level network communication protocol [12]. The library exports high-level message-passing primitives which hide the complexity of fault-tolerant network communications from the application. This approach, besides significantly simplifying the application programmer's task, allows us to interweave message-passing tightly with distributed checkpointing and rollback-recovery, and thus implement them efficiently.

The same motivations drove the work of other researchers who developed reusable components for reliable systems. Some [15, 16] do not deal with distributed fault-tolerance while others [6] address fault-tolerant network communications by providing low-level primitives, by which it is still difficult to construct reliable distributed applications. H-Libra differs from these not only by the underlying mechanisms, but also by offering fault-tolerance transparency together with a simple, high-level message-passing interface. H-Libra also differs from other message-passing systems, such as PVM [20], which do not support fault-tolerance at the application level.

3: Distributed checkpointing and rollback-recovery

3.1: The system model

Our system model consists of heterogeneous computing nodes connected by a high-speed communication network. Without going into the implemenation issues of supporting communication, we assume that each node is able to communicate with any other node in the system though all nodes may not be fully connected. Nodes can fail by stopping. When a recovery is performed, the process states can be restored from the checkpoint stored on stable storage of the respective node. We assume that each node shares a reliable network file server. Processes communicate by passing the messages over the communication network. The network channels are unreliable non-FIFO channels which may loose or reorder messages, and may temporarily be broken. For simplicity, we also assume that all processes involved in a consistent checkpoint or a rollback-recovery belong to a single distributed heterogeneous application, checkpointing or recovery of different applications does not interfere with each other.

In the following section we describe our checkpointing and rollback-recovery protocols which basically works for any generalized distributed system. Our library prototype H-Libra incorporates these protocols to help the user to develop reliable applications.

3.2: The protocols

In our distributed consistent checkpointing protocol, each distributed checkpoint is uniquely identified by an increasing *checkpoint sequence number* (CSN) and a *status bit.* CSNs are also used by other researchers [4, 10-12, 17]. The status bit on a node is set when the local checkpoint is part of the latest committed distributed checkpoint. Synchronised by the coordinator, a variant of a two-phase commit protocol is employed, where the second phase proceeds lazily and therefore does not require extra messages. The protocol tags each normal (i.e., application-level) message with the current CSN and status bit of the sender. If any message is received with a CSN greater than the local one, a local checkpoint is taken. If the message's CSN is less than the local one, the message was in transit during the checkpoint and must be logged. If the CSNs agree but

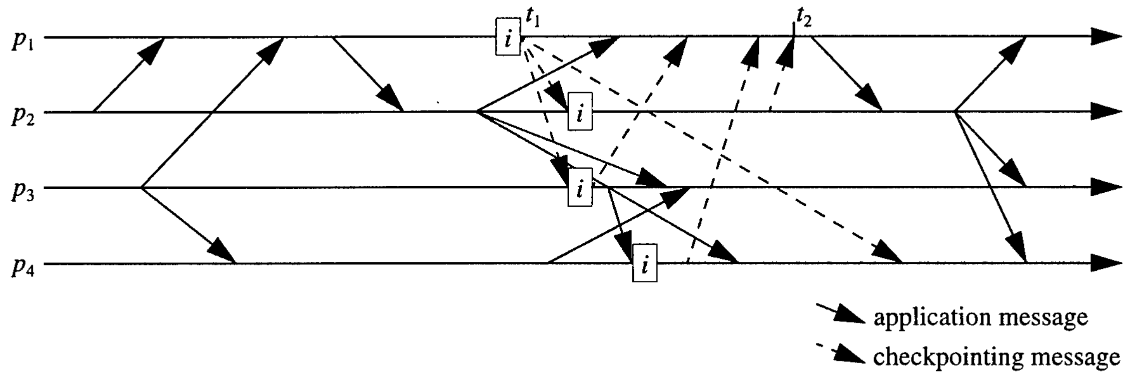


Figure 2: Consistent checkpointing with URTP and ACs

the message's status bit is set while the local one is not, the local checkpoint is committed. Communication overhead for a distributed checkpoint is thus reduced to that of systems using a one-phase commit, while stable storage is utilised more efficiently, as previous checkpoints can be discarded once the present checkpoint is committed.

To prevent message loss following a rollback, messages in transit during a distributed checkpoint need to be discovered and logged as part of the current checkpoint. While other approaches, for example, due to Chandy and Lamport [2] and Mattern [11], require additional messages to catch such messages in transit, we avoid this overhead by integrating the checkpointing algorithms with the network communication protocol. We employ a novel *user-level reliable transmission protocol* (URTP) having the following features. The details of URTP will be discussed in Section 4.2.

- It provides user-level reliable message-passing. A reliable message delivery is realized by retransmitting a message a number of times until an acknowledgement is received from the destination process. If no acknowledgement is received after a certain number of retransmissions, URTP assumes an error due to a node failure or a temporarily partitioned channel, and informs the rollback-recovery coordinator of the failure.
- Threads are used to provide non-blocking asynchronous communications amongst heterogeneous nodes.
- The protocol cooperates with the checkpointing and rollback-recovery algorithms (i.e., the logging of messages in transit) to transparently handle distributed checkpointing and rollback-recovery.

A second novelty of our approach is the use of an *acknowledgement counter* (AC) to record the number of message packets originating from the local node between two checkpoints that have not been acknowledged. Each node in the system maintains two ACs: *previous* AC (PAC) and *current* AC (CAC). An AC is incremented by the number of packets used when sending a message, and is decremented by the same amount once the last packet of that message has been acknowledged. The PAC is updated while there exists no uncommitted local checkpoint, otherwise the CAC is updated when sending or receiving a message. On commit, the PAC is set equal to the value of the CAC, and then the CAC is initialized to zero. The local node does not inform the coordinator of the local checkpoint having been taken until its PAC becomes zero (indicating that all messages originating at that node between the last two checkpoints have arrived at their destinations and have been logged if necessary). This guarantees that all messages in transit have been logged and no message losses due to site failures or unreliable non-FIFO networks have occurred once the coordinator commits. The coordinator assumes a failure and initiates a rollback-recovery if some nodes fail to respond within a timeout interval. Other failures, such as unacknowledged messages, are detected by the URTP protocol (Section 4.2).

Figure 2 shows how a consistent distributed checkpoint is taken within an application of four processes on different nodes. Coordinator p_1 initiates at point t_1 the ith distributed checkpoint. As informed either by a checkpointing request or by an application message, other processes take their local checkpoints,

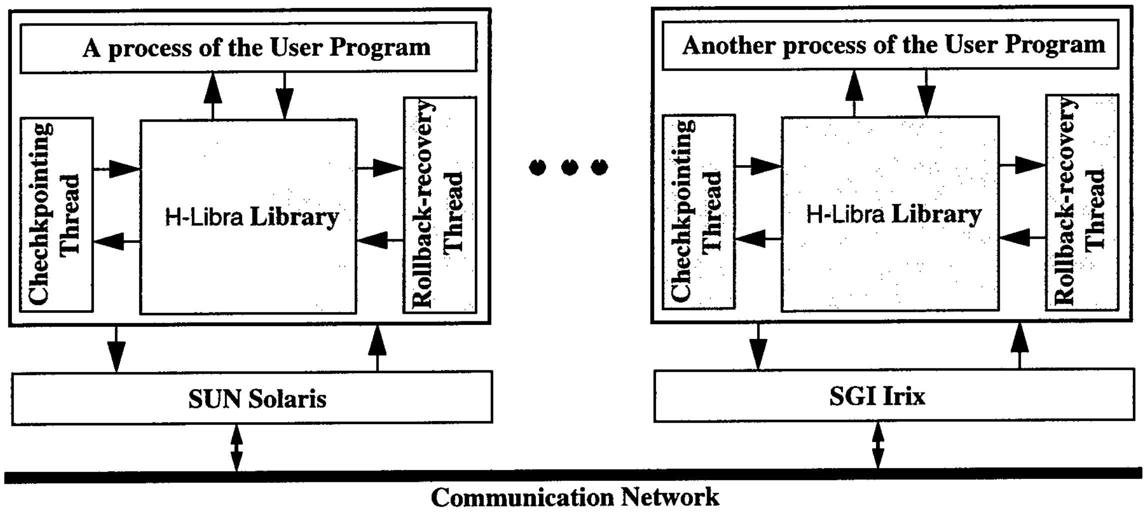


Figure 3: H-Libra runtime configuration on a Distributed Heterogeneous Computing System

and will not send the acknowledgements to p_1 until their local PACs become zero. p_1 knows at point t_2, after receiving all the acknowledgements, that not only all the processes within the application have been checkpointed, but also no messages originating in the last checkpoint interval are in transit or lost. p_1 can set its local status bit and commit the ith checkpoint. As described above, the commit decision is delivered to other processes lazily by tagging the status bit on each application message.

Rollbacks are also uniquely identified, by a *recovery sequence number* (RSN), to avoid *livelocks* and maximise parallelism during recovery. The RSN is also tagged on every message. A one-phase commit protocol is used for the distributed rollback. If a message (either a specific rollback request or a normal message) is received with a RSN greater than the local RSN, a local rollback-recovery is performed and an acknowledgement is sent to the coordinator. If a RSN is received which is less than the local one, the message was sent before the sender performed its rollback and is therefore discarded.

4: Library prototype

4.1: Library architecture

Based on the checkpointing and rollback-recovery protocols described in the previous section, we have developed a library prototype called Libra which transparently provides fault-tolerance to distributed applications on homogeneous systems [14]. The library prototype has been built on an Ethernet network of Sun workstations running SunOS 4.1 and Solaris 2.5. In the following discussion we propose how it can be implemented on a heterogeneous systems.

Figure 3 shows the overall H-Libra run-time configuration on a DHCS where each participating node uses the local instance of H-Libra on its operating system. Distributed applications use threads and H-Libra's message-passing and memory allocation primitives; the checkpointing thread and rollback-recovery thread are created by H-Libra when the user program starts; fault-tolerance is then automatically provided by the library. Table 1 shows the library interface (functions for configuring parameters, such as the number of participating nodes, checkpoint frequency and timeout intervals, have been omitted from the table for simplicity).

The functions `ft_send` and `ft_recv` provide basic message passing. Threads are created by the library to perform the actual send operation without blocking the application. The tasks of initiating and committing checkpoints and rollbacks, and handling the message logs, are performed transparently by H-Libra (through background threads and the application's calls to `ft_send` and `ft_recv`). The functions `ft_malloc` and `ft_free`, exported by H-Libra, are used for memory management at the user-level. Their use by the application ensures that `ft_malloc` arena is checkpointed.

Table 1: Important functions provided by H-Libra

Message-passing primitives

```
int ft_send(char *msg, int size,
            int dest)
int ft_recv(char *msg, int size,
            int *sender)
```

Memory allocation primitives

```
char *ft_malloc(int size)
int ft_free(char *addr)
```

Initialization

```
int ft_init(int my_ide, ...)
```

The application needs to call ft_init so H-Libra can initialise its internal data structures. This call, when executed on the coordinator node (node 0) will create a coordinator thread, cp_coor, which initiates distributed checkpoints, and commits or aborts them. On other nodes a cp_node thread is set up. This thread performs local checkpoints, as requested by the coordinator. On the coordinator node, a separate thread rr_coor is responsible for rollback-recoveries; this thread initiates, coordinates and commits or aborts the recovery as appropriate. Local recovery action is performed by thread rr_node running on non-coordinator nodes.

Note that the functions presented in Table 1 are exported for constructing application software while the other internal functions such as checkpoint, msg_log, restart, msend and mrecv are transparent to the user code. They are used by thread cp_coor, cp_node, rr_coor and rr_node to transparently handle checkpointing, message logging and rollback-recovery. As a simple, easy to use high-level message-passing interface is provided and fault-tolerance is completely transparent to the user, H-Libra can significantly simplifies the development of reliable distributed applications. The following example is part of the *client* code of *Quicksort* program. It demonstrates that, using H-Libra, little programming effort is required to construct a reliable version of a distributed heterogeneous program.

```
void Client()
{
  QSmsg mesg;
  int   bytes, recvid;

  mesg.right = mesg.left = DONE;

  for (;;) {
    mesg.sender = locid;
    mesg.type   = WORKER_REQ;
    if (mesg.left != DONE) {
      bytes = (mesg.right - mesg.left + 1)
              * sizeof(int);
      bcopy((char *)data, mesg.buf, bytes);
    }
    /* send a message to the server */
    ft_send((char *)&mesg, sizeof(mesg),
            toid);
    /* wait for a message from the server */
    ft_recv((char *)&mesg, sizeof(mesg),
            (int *)&recvid);

    if (mesg.type == MASTER_DON) {
      printf("Node %d done!", locid);
      exit(0);
    }
    if (mesg.type != WORKER_ACK) {
      printf("Some error from master %d.",
             mesg.type);
      exit(1);
    }
    bytes = (mesg.right - mesg.left + 1) *
            sizeof(int);
    bcopy((char *)mesg.buf, (char *)data,
          bytes);
    printf("Received [%d, %d] from server.
        sorting.", mesg.left, mesg.right);
    Bubblesort(0, mesg.right - mesg.left);
  }
}

main(argc, argv)
unsigned argc;
char **argv;
{
  /* initialisation operations */
  ...
  /* create the client thread */
  ...
  ft_init(my_id, ...);
}
```

Figure 4 presents an example which demonstrates how a reliable distributed computation proceeds by using H-Libra. The picture depicts the interactions between the user code and the library functions as well as those between the library functions themselves. Suppose that two threads exchange messages across networks: thread T_i on node i sends message M_{ij} to thread T_j on node j, meanwhile T_j sends message M_{ji} to T_i (as indicated by the heavy dashed arrow). The shaded

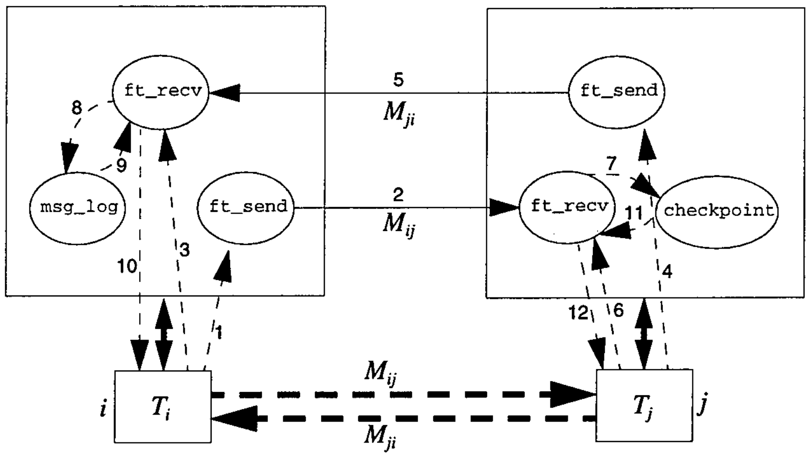


Figure 4: An example of reliable distributed computation using H-Libra

functions are used by user threads for message passing. Each light dashed arrow is an interaction between two library functions or between a library function and the user code. Each light solid arrow is a message delivered across networks. The heavy solid arrows indicate the user-code/library relationship. The sequence number associated with each light arrow represents the logical time when the corresponding interaction or event occurs.

According to the logical time when each interaction or event occurs, we go through a scenario as follows. (1) T_i sends T_j message M_{ij} by calling ft_send. Suppose that, at this point, the state of T_i has been saved as part of the latest consistent distributed checkpoint (its local CSN is n and local status bit is 0) while T_j has not been checkpointed (its local CSN is $n-1$ and local status bit is 1). (2) ft_send starts a separate thread and then returns. The created thread sends M_{ij} to T_j by using the URTP protocol. (3) After returning from ft_send, T_i waits for message M_{ji} from T_j by calling ft_recv. (4) At this point T_j calls ft_send to send message M_{ji}. (5) ft_send creates a thread which sends M_{ji} by using URTP protocol. (6) After returning from ft_send, T_j calls ft_recv to receive message M_{ij} which has arrived at node j. (7) Suppose that the CSN on node j is still $n-1$ when ft_recv is called. Because the CSN (n) tagged on M_{ij} is greater than the local one ($n-1$), ft_recv performs a call to checkpoint to save the local state as part of the latest distributed checkpoint. (8) M_{ji} arrives at node i and is received by ft_recv. Since the CSN ($n-1$) tagged on M_{ji} is less than the local one (n), M_{ji} is a message in transit and ft_recv performs a call to msg_log to log this message. (9) After logging M_{ji}, msg_log returns to ft_recv. (10) ft_recv returns to the user code after receiving M_{ji}. (11) checkpoint takes the local checkpoint and returns to ft_recv. (12) After receiving the last packet of M_{ij} ft_recv reassembles the message and then returns to the user code.

4.2: Library implementation

This section describes the main features of the implementation of the library prototype, which is built up on the mechanisms described in Section 2. In particular we emphasise on the following issues: how to implement the high-level message-passing interface by using URTP which interweaves tightly with the distributed checkpointing and rollback-recovery protocols, and how to reduce the latency and disk usage resulted from checkpointing. Note that we do not go into any implementation details specific to the underlying operating systems.

4.2.1: The message-passing interface and the underlying URTP

We model a network of heterogeneous workstations, on which H-Libra is running, as an array of nodes — Node_Array[n] where n is the number of nodes in the network. Each node associated with an element of the array is represented by the following structure, comprising of its network address and port numbers. Different port numbers are used for message passing, distributed checkpointing, and rollback-recovery. A copy of the array is made available on each node.

```
typedef struct node {
    char        *hostaddr;
    unsigned    ap_port;
    unsigned    cp_port;
    unsigned    rr_port;
} node_t;
node_t Node_Array[n];
```

Functions ft_send and ft_recv provide basic message-passing according to the URTP protocol. When sending a message, it is essential to efficiently address the following issues: 1. detecting missing, duplicate and other unexpected message packets for reliable message passing; 2. synchronisation between threads when more than one thread on a node simultaneously send messages to the same remote node; 3. providing control information required by distributed checkpointing and rollback-recovery; and 4. implementing non-blocking communications.

H-Libra maintains on each node a *sending sequence number vector* (SSV) of length n, the number of nodes in the distributed system. When sending a packet to node j, SSV[j] on sender i is incremented and tagged on the packet. The receiver will use SSV[j] tagged on the packet for detecting missing, duplicate and other unexpected packets.

When more than one thread on a node simultaneously send messages to the same remote node, the threads must be synchronised properly to ensure that messages to the same destination are delivered sequentially. To achieve this, the library maintains on each node a sending lock vector (SLV) as well as a *next sequence number vector* (NSV) with the same length as a SSV. A thread on node i can send a message to node j *only* when it has acquired the lock — SLV[j], and SSV[j] tagged on the first packet of the message is equal to NSV[j]. When the last packet of the message is acknowledged, NSV[j] is incremented by the number of packets used for the message, and SLV[j] is released.

According to the mechanisms described in Section 2, ft_send tags each outgoing packet with the local RSN, CSN and status bit. When a packet is sent or acknowledged, the corresponding AC must be updated accordingly. In order to achieve efficient message passing, non-blocking communications is supported such that ft_send starts a separate thread which sends the message by using the URTP protocol, and it can return without waiting for the acknowledgements.

When ft_send is called, the following self-explanatory operations are performed.

1) tags the message with the identifier of the local node and the sending sequence number, then increments the sending sequence number by the number of packets needed for this message;
2) tags the message with RSN, CSN, and status bit;
3) increments the value of the corresponding AC by the number of packets needed for this message;
4) creates a thread to send the message, and then returns.

The created thread then performs the following operations based on the URTP protocol.

1) acquires the lock associated with the destination;
2) if SSV[dest] is equal to NSV[dest], goes to the next step; otherwise, releases the lock and goes to step 1;
3) opens and binds a UDP socket;
4) loads the predefined number of bytes into a packet tagged with the local id, number of packets used for the message, sending sequence number, RSN, CSN, and status bit;
5) sends the packet through the socket and waits for the acknowledgment;
6) if the acknowledgment arrives within the timeout interval, goes to the next step, if the acknowledgment has not arrived and the packet has been retransmitted for a certain number of times, goes to step 9, otherwise, goes to step 5 for retransmission;
7) decrements the PAC while there exists no uncommitted local checkpoint, otherwise the CAC;
8) if the last packet of the message is acknowledged, the sending thread increments NSV[dest] by the number of packets used for the message, releases the lock, and exits normally, otherwise goes to step 4 for the next packet;
9) at this point a failure is assumed to have occurred, and a rollback request is sent to thread rr_coor, the rollback-recovery coordinator.

The implementation of ft_recv is more complex. When receiving a message the following issues must be addressed: 1. detecting missing, duplicate and other unexpected message packets at the receiving end; 2. synchronisation when message packets from different source nodes arrive simultaneously; 3. checkpointing (when needed) at a correct point within ft_recv; 4. retrieving messages from the message log if any; 5. dealing with an incoming message from a different checkpoint interval whose CSN is greater or less than the local one; 6. dealing with an incoming message from a different rollback-recovery interval whose RSN is greater or less than the local one.

In order to detect missing, duplicate and other unexpected packets at the receiving end, H-Libra maintains on each node a *receiving sequence number vector* (RSV) with the same length as a SSV. When a packet from sender i is received by a call to `ft_recv` on node j, the SSV[j] tagged on the packet is compared with the local value of RSV[i]. If these agree, the packet is valid and RSV[i] is incremented. If $\text{SSV}[j] = \text{RSV}[i] - 1$, a duplicate packet has been received and is ignored, as are unexpected packets recognised by other cases of non-matching sequence numbers.

When message packets from different source nodes arrive simultaneously, it must be ensured that messages are received sequentially. For instance, after `ft_recv` first receives the packets of message M_1 from node i, a packet of message M_2 from node j arrives before the last packet of M_1 is received, and the packets of message M_2 cannot be received until M_1 has been reassembled. This is done by that, when the first packet is received, `ft_recv` records the source id tagged on the packet and then only receives the packets from the same source node until it returns and another call to `ft_recv` is performed.

When a call to `ft_recv` is performed, it is impossible to know in advance the CSN of an incoming message packet. In order to construct a consistent checkpoint, `ft_recv`, before receiving an incoming message, pre-saves the machine-dependent context of the receiving thread to a buffer. When a local checkpoint needs to be taken before `ft_recv` returns, the saved context indicates a correct point where the checkpoint is constructed, and the local state including the saved context is written to stable storage; otherwise the context is discarded right before `ft_recv` returns.

When a normal message packet is received with a RSN greater than the local one, a local rollback-recovery must be taken, and the packet is saved to the *suspended packet buffer* and will be replayed by a call to `ft_recv` after the rollback. Messages in the message log are retrieved to the *logged message buffer* when a rollback is done. Before receiving a message from networks, `ft_recv` needs to check whether the logged message buffer and suspended packet buffer are empty. If the logged message buffer is not empty, `ft_recv` removes a message from the buffer and returns it to the caller. If the suspended packet buffer is not empty, `ft_recv` removes the suspended packet from the buffer, if the packet is the last packet of a message, `ft_recv` reassembles the message and returns it to the caller; otherwise waits for receiving the next packet of the same message.

When receiving a message packet either from a different checkpoint interval or from a different rollback-recovery interval, the corresponding operations, according to the distributed checkpointing and rollback-recovery protocols (see Section 2), are performed.

With the techniques as described above, the following self-explanatory operations are performed when `ft_recv` is called.

1) pre-saves the machine-dependent context of the receiving thread;
2) if the logged message buffer is not empty, removes a message from the buffer and returns it to the caller, otherwise goes to the next step;
3) if the suspended packet buffer is not empty, removes the packet from the buffer and goes to step 5, otherwise goes to the next step;
4) waits for an incoming packet from networks;
5) when receiving the first packet, records the source node id tagged on the packet, if, when receiving the next packet, it is from the same source node, goes to the next step, otherwise goes to step 4;
6) if the packet is a duplicate, acknowledges it and goes to step 4;
7) if the packet is an unexpected one, ignores it and goes to step 4;
8) if the sending sequence number tagged the packet is equal to the local receiving sequence number, compares the incoming RSN, CSN and status bit with the local ones, and the corresponding operations are performed as appropriate;
9) acknowledges the packet;
10) if the packet is the last of a message, reassembles the message and returns it to the caller, otherwise goes to step 4 for the next packet.

4.2.2: Efficient checkpointing

In this section we describe what is necessary to be included in a checkpoint and how to reduce the latency and disk usage due to checkpointing. When checkpointing, H-Libra saves the state of the local process within the distributed application which contains, the states of user threads (not H-Libra threads)

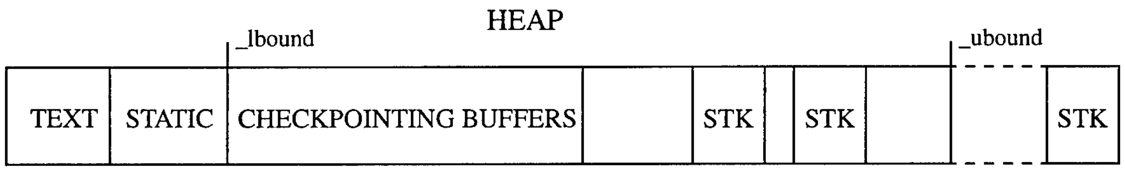


Figure 5: An address space of a multithreaded process

including the machine-dependent contexts and stacks, global/static and heap data.

H-Libra maintains on each node `thr_list`, as defined below, each element of which records the state of a local thread such as thread identifier, machine-dependent context, stack top, stack pointer, the location and size of the stack buffer, and names of its message logs (a thread has no message logs if merely doing local processing).

```
struct thr_stat {
    thread_t      tid;
    machstat_t    machstat
    caddr_t       stacktop;
    caddr_t       stackptr;
    caddr_t       stkbuf_addr;
    int           stacksz;
    unsigned      pre_ctx;
    char          prev_log[nmlen];
    char          curr_log[nmlen];
} thr_list[thrnum];
```

`gdb_list` records the control information for checkpointing global/static data, comprising of their addresses, sizes, and the locations of the associated buffers. The buffers are used to buffer global/static data before written to stable storage.

```
struct gdb_stat {
     caddr_t   addr;
     int       len;
     caddr_t   gdbbuf_addr;
} gdb_list[gdbnum];
```

It is common for applications to use heap space so that heap state should be saved as part of a checkpoint. Figure 5 shows an address space of a multithreaded process. The heap is divided into several segments: checkpointing buffers where checkpointed data are saved before written to stable storage, red-zone protected stacks of threads, and normal `malloc` arena (unshaded) which need to be checkpointed. Although it is feasible to checkpoint the separated unshaded heap areas, there are two serious problems: 1. the locations, sizes, and number of unshaded areas may change dynamically when red-zone protected stacks are allocated or freed due to the creation and termination of threads, and it is complicated to keep the trace of the unshaded areas; 2. checkpointing all of the unshaded heap areas unnecessarily increases the size of a checkpoint and therefore increases the checkpointing overhead because the thread library and H-Libra use a large part of heap space which is irrelevant to a checkpoint.

H-Libra uses another approach which guarantees that only relevant heap data are checkpointed. As indicated by an application, H-Libra, when program starts, allocates a dedicated memory segment which is large enough to contain the heap data requested by the user code. As described in Section 4.1, `ft_malloc` and `ft_free` are provided to manage the dedicated memory segment. The user code allocates or frees memory by calling `ft_malloc` and `ft_free`. It is the user's responsibility to guarantee that `ft_malloc` arena contains all of the heap data which need to be checkpointed. In order to manage the memory segment and correctly checkpoint its state, H-Libra maintains the following control data structure used by `ft_malloc`, `ft_free`, and other checkpointing related library functions. `memseg` recodes, the base and size of the segment, the highest segment location currently used by the program, and the location of the associated buffer which is used to buffer heap data before written to stable storage. It also records the pointers to the allocation list and free list which are used by `ft_malloc` and `ft_free` for memory management. When checkpointing, `ft_malloc` arena from `memseg_base` to `memseg_ptr` as well as `memseg` are saved.

```
struct memseg_stat {
     caddr_t    memseg_base;
     int        memseg_size;
     caddr_t    memseg_ptr;
     caddr_t    segbuf_addr;
     memblk_t   *mlist_bptr;
     memblk_t   *mlist_eptr;
     memblk_t   *flist_bptr;
```

```
        memblk_t   *flist_eptr;
    } memseg;
```

The latency and disk usage due to checkpointing can be significantly reduced by using *copy on write* and *incremental* checkpointing [4]. The techniques are implemented in H-Libra as follows: First, H-Libra freezes user threads, saves the machine-dependent contexts, and changes the access protections, to be "read-only", on the pages within the address space which contain what should be checkpointed. Next it unfreezes user threads and starts a separate *writer* thread that copies, to the checkpointing buffers, the pages which have been modified since the last checkpoint. If a user thread generates a page access violation, the page fault handler writes that page to the buffer *only* if the writer thread has not done this, then it sets the page's protection to "read-write", sets the page as modified since the last checkpoint, and restarts the user thread. After copying the local state to the checkpointing buffers the writer thread writes, to stable storage, the pages which have been modified since the last checkpoint.

4.2.3: Implementation issues related to heterogeneous environment

The implementation of H-Libra in a heterogeneous environment is different from the implementation of Libra in a homogeneous system. Note that we assume *sockets* are supported by all nodes as sockets are used to handle message-passing in heterogeneous environments. We think that a heterogeneous environment can affect the implementation and efficiency of H-Libra due to the following reasons:

1. Different types of threads are supported by operating systems. For instance, SunOS 4.1 and the OSF's Distributed Computing Environment (DCE) support user-level threads, where threads management is done in user time and the operating system has no control of the threaded evironment except to make resources available to the entire process. However, Solaris and Ultrix support kernel-level threads, which are visible to the operating system. The type of thread supported largely decide whether H-Libra can take full advantage of the underlying mechanisms (i.e., copy on write checkpointing, URTP) which maximise the concurrency and parallelism, and reduce the overhead and latency of checkpointing.
2. Different operating systems provide different policies for scheduling and resource allocation which may also affect the efficency of our checkpointing and rollback-recovery algorithms.
3. The implementation techiques for checkpointing and recovering threads on different operating systems may be diferent from one to another. For instance, checkpointing threads are straightforward in SunOS 4.1 by simply calling `lwp_getregs`, and executions of threads can be resumed by calling `lwp_setregs` after restoring the states of threads. However, checkpointing and recovering threads in Solaris are mainly based on signal handling. The target threads are interrupted by signals, and checkpointed and recovered by the corresponding signal handlers. Again, different degrees of invasiveness of H-Libra result.

5: Performance evaluation

We present in this section the performance of Libra with respect to communication, and running time overheads. We expect that the communication overheads for H-Libra should be comparable to that of Libra as both use the same underlying protocols. We also compare the run-time performance of Libra implemented on SunOS 4.1 and Solaris 2.5. For this purpose, we choose three message-passing applications with quite different communication patterns: CST, a program for maintaining a balanced concurrent search tree $2^{B-2} - 2^{B}$ search tree [3]; QSORT, a distributed quicksort implementation; and FFT, the Fast Fourier Transform of 64k to 2M data points. CST exchanges many small messages, while FFT and QSORT is somewhere in between these two extremes. Each application is distributed by using a number of *client* and one or more *server* processes.

5.1: Communication overhead

We classified the existing approaches based on consistent checkpointing into five categories according to how consistent checkpoints are taken and how messages in transit are caught. We chose a typical representative from each category to compare communication overhead in terms of the number and size of messages for fault-tolerance. The choices are: 1. a variant of Chandy and Lamport's one-phase commit snapshot algorithm [2] for non-FIFO systems; 2.

Table 2: Communication overheads due to checkpointing

Programs	Benchmark Statistics			Communication overheads				
	Number of Processes	Number of Checpoints	Number of Messages	Chandy Lamport	DCTD (Mattern)	VCP (Mattern)	Elnozahy	URTP & ACs
CST	110	9	114	435	332	288	436	218
QSORT	101	9	100	400	300	321	400	200
FFT	65	5	45	256	173	192	256	128

Mattern's one-phase commit snapshot algorithm with the *deficiency counting termination detection method* (DCTD) for catching messages in transit [11]; 3. Mattern's one-phase commit algorithm with the *vector counter principle* (VCP) for catching messages in transit [11]; 4. Elnozahy *et al.*'s two-phase commit algorithm [4] (which does not catch messages in transit); 5. our algorithm based on URTP and ACs.

We simulated the three programs on a single machine, using the light-weight process library provided in Sun OS 4.1. Each program was implemented in five versions, one for each of the checkpointing algorithms examined. The message overheads for the five algorithms for the three benchmarks are shown in Table 2. There are two types of overhead messages: overhead for checkpointing and overhead for catching messags in transit. The major benefit of our algorithm (URTP & ACs) is that it does not cause any further message passing for catching messages in transit, and hence it exhibits the lowest communication overhead.

5.2: Time overhead

All the benchmark programs were run on a network of four Sun workstations, running SunOS 4.1 and Solaris 2.5. Figure 6 presents a comparison between the running overheads on the two systems. All times are averages of three runs on an otherwise essentially empty system. The overhead is generally quite low, below 10% even with the shortest checkpointing interval on SunOS 4.1 where over 30 checkpoints were written. This version of library has no kernel support for threads. As a result, in the implementation user threads are blocked when a write system call is performed during checkpoints. This even occurs when the so-called "non-blocking I/O library" is used. The implementation of Libra on Solaris 2.5 is more efficient than that on SunOS 4.1 because Solaris 2.5 supports threads at kernel-level.

6: Conclusions

We have described in this paper an approach for supporting the development of reliable heterogeneous

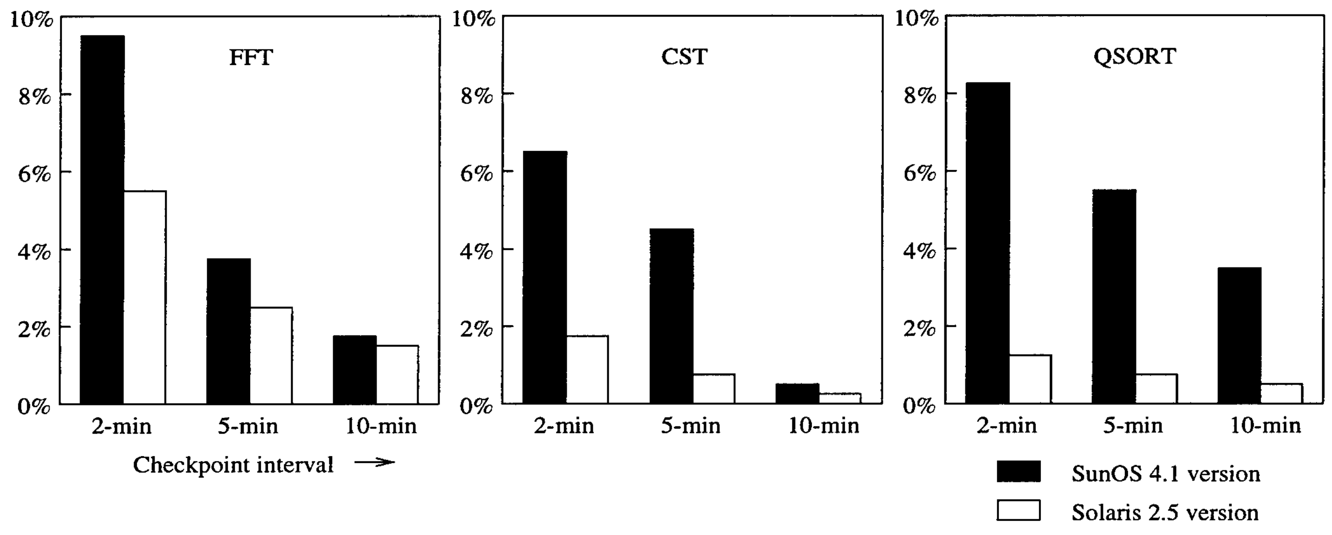


Figure 6: Checkpointing overhead comparison between two versions of Libra

distributed applications. The approach meets two objectives: to simplify the development of reliable distributed applications, and to achieve fault-tolerance at low run-time cost. The first objective is met by the provision of fault-tolerance transparency and a simple, easy to use high-level message-passing interface. Fault-tolerance is provided to applications transparently and is based on the distributed consistent checkpointing and rollback-recovery protocols integrated with a user-level network communication protocol. The second objective is met by the use of protocols which minimise communication overhead for taking a consistent distributed checkpoint and catching messages in transit, and impose low overhead in terms of running times.

Our benchmarks have shown that it can achieve high efficiency and be used as a practical tool to construct reliable distributed applications. We are now implementing H-Libra on a real heterogeneous environment comprising of other operating systems such as SGI Irix and Digital Ultrix. The performance and overheads due to both checkpointing and rollback-recovery will be analyzed on such a heterogeneous system.

References

1. G. Cabillic, G. Muller, and I. Puaut. The performance of consistent checkpointing in distributed shared memory system. *Proc. 14th Symp. on Reliable Distributed Systems*, pages 96-105, October 1995.
2. K. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. on Computer Systems*, 3(1):63-75, February 1985.
3. A. Colbrook, E. Brewer, C. Dellarocas, and W. Weihl. An algorithm for concurrent search trees. *Proc. 1991 Int. Conf. on Parallel Processing*, volume 3, pages 138-141, August 1991.
4. E.N. Elnozahy, D.B. Johnson, and W. Zwaenepoel. The performance of consistent checkpointing. *Proc. 11th Symp. on Reliable Distributed Systems*, pages 39-47, October 1992.
5. R.F. Freund and H.J. Siegel, "Heterogeneous Processing," *IEEE Computer*, Vol. 26, No. 6, June 1993, pp. 13-17.
6. Y. Huang, C. Kintala, and Y..M. Wang. Software tools and libraries for fault tolerance. *IEEE Bulletin of the Technical Committee on Operating Systems and Application Environments*, 7(4):5-9, 1995.
7. D.B. Johnson and W. Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. *Journal of Algorithms*, 11:462-491, 1990.
8. A.A. Khokhar, V.K. Prasanna, M.E. Shaaban, and C.-L. Wang, "Heterogeneous computing: Challenges and opportunities," *IEEE Computer*, Vol. 26, No. 6, June 1993, pp. 18-27.
9. R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Trans. on Software Engineering*, 13(1):23-31, January 1987.
10. K. Li, J.F. Naughton, and J.S. Plank. Checkpointing multicomputer applications. *Proc. 10th Symp. on Reliable Distributed Systems*, pages 66-75, September 1991.
11. F. Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing*, 18:423-434, August 1993.
12. J. Ouyang and G. Heiser. Checkpointing and recovery for distributed shared memory applications. *Proc. 4th Int. Workshop on Object Orientation in Operating Systems*, IEEE Computer Society Press, pages 191-199, Lund, Sweden, August 1995.
13. J. Ouyang and G. Heiser. Libra: A library for reliable distributed applications. *Proc. 1996 Int. Conf. on Parallel and Distributed Processing Techniques and Applications*, pages 801-810, Sunnyvale, California, August 1996.
14. J. Ouyang and P. Maheshwari. Architecture and implementation of Libra — a library for reliable distributed applications. *Proc. IEEE 2nd Int. Conf. on Algorithms and Architectures for Parallel Processing*, pages 263-270, Singapore, June 1996.
15. J.S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under UNIX. *Proc. 1995 Winter USENIX Conference*, pages 213-223, January 1995.
16. J.S. Plank, M. Beck, and G. Kingsley. Compiler-assisted memory exclusion for fast checkpointing. *IEEE Bulletin of the Technical Committee on Operating Systems and Application Environments*, 7(4):10-14, 1995.
17. J.S. Plank and K. Li. Ickp — a consistent checkpointer for multicomputers. *IEEE Parallel and Distributed Technology*, 2(2):62-67, 1994.
18. S.W. Smith, D.B. Johnson, and J.D. Tygar. Completely asynchronous optimistic recovery with minimal rollbacks. *Proc. 25th Int. Symp. on Fault-Tolerant Computing*, pages 361-370, June 1995.
19. R. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Trans. on Computer Systems*, 3(3):204-226, 1985.
20. V.S. Sunderam. PVM: A framework for parallel and distributed computing. *Concurrency: Practice and Experience*, 2(4):315-339, December 1990.
21. Y.M. Wang and W.K. Fuchs. Lazy checkpoint coordination for bounding rollback propagation. *Proc. 12th Symposium on Reliable Distributed Systems*, pages 78-85, October 1993.

The Hopping Ruse

Marina Chen James Cowie

Cooperating Systems Corporation
Chestnut Hill, MA 02167

Abstract

We describe a novel framework for early detection and isolation of security violations in heterogeneous environments, based on realtime service hopping. *In distributed client-server systems, service hopping fills a role analogous to frequency-hopping spread-spectrum techniques for secure wireless communication.*

The framework incorporates design principles for secure hopping, as well as engineering principles for improving throughput in the presence of a statistically noisy interconnection network. We describe potential large-scale applications of the hopping techniques, and present some initial experimental results with a hopping client-server system.

1 Motivation

For many widely deployed applications with broad commercial appeal, security from attack and confidentiality of content are pre-requisites. These systems include electronic stock exchanges, electronic commercial banking, telemedicine, medical informatics, manufacturing design, and defense applications. The heterogeneous computing base upon which these widely distributed applications rely poses special challenges from the perspective of security. Waging or defending against information warfare in such a system requires detection of intrusion in any component, and insulation of the system from those components' effects until countermeasures can be brought to bear.

Perils of Homogenization Traditional strategies for managing and exploiting heterogeneity have focused on portable implementations of standard transport- and application-level protocols. These protocols allow heterogeneous hardware to support a more homogeneous software base, so that diverse systems can cooperate to carry out large-scale coordinated computation.

With this homogenization of software tools, however, have come some serious potential security problems. When every system honors a common mechanism for access, any flaw in those mechanisms can be universally exploited. When systems are configured to support transparent migration of code or live processes from one heterogeneous host to another, one compromised host can "infect" all the hosts that trust it. Worst of all are common implementations of common access mechanisms (using portable source or object code). In such cases, designers of heterogeneous software must warrantee as bug-free not only the access mechanisms, but their universally accepted implementations as well, to feel certain that there are no exploitable vulnerabilities.

Transport-level Security To date, the security of the distributed heterogeneous computing base has largely relied on basic access control and authentication of communication partners, coupled with transport-level protocols for packet encryption and data integrity [3]. Encryption-based approaches are both effective at protecting content and access, and predictable in their effect on the performance of network applications.

However, by eliminating cleartext network traffic, transport-level encryption protocols make it significantly harder for network managers to tell successful intruders from legitimate users at a glance, and to spot the emergent traffic patterns of probing and spoofing that signal impending trouble. Worse, encryption-based approaches by themselves fail to give administrators a complete picture of the security of a heterogeneous distributed computing environment. To scale beyond a handful of nodes, network security must rely on data collection, recognition of patterns, and classification of security threats as they emerge from the noise of legitimate network use.

1.1 Application-level Security

We are interested in developing higher-level *structural security techniques* for heterogeneous computing systems, to complement simple access control and protocol-level encryption. These techniques exploit

0-8186-7879-8/97 $10.00 © 1997 IEEE

or modify the structure of a heterogeneous computation — the mapping of tasks to processors throughout the system — to help quantify security risks, identify emergent threats, and contain intrusions.

By re-injecting an element of software heterogeneity into an increasingly homogeneous software base, and being creative (if not unpredictable) in our mapping of tasks to heterogeneous resources, we hope to improve the overall resistance of large distributed systems to attack. In this paper, we propose *service hopping*, a realtime application-level scheme that allows early detection and monitoring of unauthorized access to sensitive resources within a heterogeneous computational environment.

Frequency hopping. The basic concept of service hopping uses an analogue of the frequency-hopping strategies used in secure radio transmission. Frequency hopping is one of several spread-spectrum techniques originally used to secure military radio traffic against interception and jamming. A frequency-hopping radio transmitter changes the broadcast frequency synchronously many times each second; the pattern of "hopping" is controlled by bits sampled from a periodic pseudorandom signal. Authorized client radio receivers can follow the hop sequence, but potential eavesdroppers and jammers are frustrated.

Service hopping. In a service hopping system, several real and false instances of each sensitive network service are active at any given moment, spread out across the nodes and ports of a heterogeneous computing system. Rather than migrate the services themselves, the "token of legitimacy" changes hands periodically, following a random walk over the individual service instances within the heterogeneous workspace. True instances periodically cease to function, becoming false instances, and vice versa.

Using one of a number of alternative schemes for distribution of secret data, legitimate network clients are provided with secure access to the same hopping sequence data. In an environment where only legitimate clients (those who know the dance) can follow the dynamic host/port binding of legitimate network services, unauthorized accesses to false services are instantly identifiable. False instances can assume that any requests for service are intrusive probes, and report them to the security monitoring layer. Network administrators can take security countermeasures, while the intruder wastes precious minutes attempting to compromise a compartmentalized dummy resource.

Organization In the sections that follow, we discuss some design strategies for constructing a hopping service system that will betray intruders while preserving throughput within the heterogeneous environment. We then address some of the practical engineering issues — how to construct and distribute hopping information to clients, minimize throughput overhead due to hopping, and maintain hopping connections in the face of network noise. Finally, we sketch applications for hopping techniques, and describe our prototype implementation of a hopping web server.

2 Design Issues for Secure Hopping

Designing network services that implement secure realtime service hopping requires attention to several areas.

- First, standard network services and clients should be extended with **false versions** to frustrate a traffic analysis attack on the hopping scheme.
- Second, the problem requires mechanisms for secure generation and propagation of **hop sequence data** to clients and servers, and provisions for invalidating and reissuing that sequence in case of attack.
- Finally, the true versions of clients and servers require extra logic for implementing the **"hopping cycle"** under less-than-ideal network conditions.

Hopping clients and network services must act, in effect, as coupled, phase-locked oscillators. Mechanisms for carrying out clock synchronization at the hardware or ICMP network levels are well known [6]. Tracking a server's hop phase over a statistically noisy Internet, at user level, creates some additional complications.

2.1 The Bait

The first component of the hopping ruse is to derive a *false service* that offers some simulated behavior which is subjectively similar to the actual behavior of the base service. That is, it offers the same interface to network clients, simulating both the overt content and side effects of the legitimate network service, including contributions to load average and network traffic generation. In general, the higher the fidelity of the false data, the longer an unauthorized browser can be held on the hook while countermeasures are put in motion.

For example, a network service that provides a realtime interface to a stream of video frame data might have a false version that returns frames of static, simulating a temporarily broken connection, or even frames of video captured from an older session. A false database application may return innocuous dummy data in response to queries. We supply an ersatz Web server which points to a distinct document tree of interesting, but sanitized content. Figure 1 presents a snapshot of a hopping collection of real and false service instances.

As an optional extension, we also implement a false client to exercise the false service interface. Without false clients, an intruder capable of monitoring network traffic patterns would readily discern the *bona fide* services from the false versions. These clients make periodic requests for service which mimic those of real users. A video server client might simply request playback of a few seconds of video; a database client might make periodic queries and discard the results, and so forth. One prototype we built, a hopping version of a Web server, has "false browsers" which replay random sequences of requests directly from the server log.

2.2 The Swap

When a hop takes place, all legitimate clients pause in their interaction with the legitimate services. True services become false services, and vice versa. Legitimate clients are privy to the secret host and port of the next "true" service instance in the hop sequence. Throughout the heterogeneous computing space, computation between hops takes place in phases, separated by short "hop windows."

Some of the performance overhead of hopping techniques can be minimized by careful engineering: the time spent in the global synchronization/migration phase of the hop window, and the overhead required to avoid false security violations due to network noise and server load. In the following sections, we consider techniques to address these two factors, as well as the additional (administrator-configurable) execution overhead of multiple false clients and servers.

3 Engineering Issues for Effective Hopping

How long it takes to synchronize the various clients and service instances in order to effect a hop (the transfer of legitimacy from one point in the heterogeneous workspace to another) depends in large part on the strategy used for derivation and distribution of hop sequence data. All legitimate entities in the system must know the answer to two questions: where and when the network service should hop.

3.1 Where to Hop?

Figure 2 shows a schematic snapshot of events within the next hop window of the system shown in figure 1. Two true service instances have stopped offering true service; their jobs are taken over by two previously false service instances. The authorized client must use its knowledge of the hopping sequence to synchronously close connections to the newly invalidated instance, and direct future connections to a newly validated instance, whereever that may be.

To carry out the migration of legitimacy from one service to another, all network clients and services in the "hop group" are effectively globally synchronized. During this synchronization period (called the "hop window"), secure network services may be briefly unavailable while the transfer of legitimacy takes place. Our goal is to make those services maximally available by minimzing that window.

Encrypted, embedded hop information. In one design, every burst of content produced by the true network service might include an embedded trigger for a global hop, plus the encrypted target of the hop. Since we already contemplate significant modifications to the service and client code to support service hopping, extending the service interface to include this field would not be unreasonable. This would eliminate the need to distribute pregenerated blocks of hop sequence information to clients, the least trusted links in a heterogeneous client-server system. It also has some significant drawbacks, however. Primarily, it adds significant overhead by forcing clients to issue extra periodic traffic (at least one transaction per hop cycle, if only to receive the target of the next hop).

Out-of-band sequence generation. A better strategy for hop sequence propagation is to generate a large personalized cache of hop sequence information. A *hop sequence generator* presamples a sequence of n hops from a random distribution, or chooses a seed for a deterministic random number sequence generator. This sequence will represent the global "hop map" for some number n of computational phases, and consists of a list of valid service host/port addresses following each transition $H_1..H_n$.

This dataset, valid for N hop cycles, may then be transmitted to clients through existing secure channels; for example, using public-key cryptography to verify the client to the hop sequence generator, and

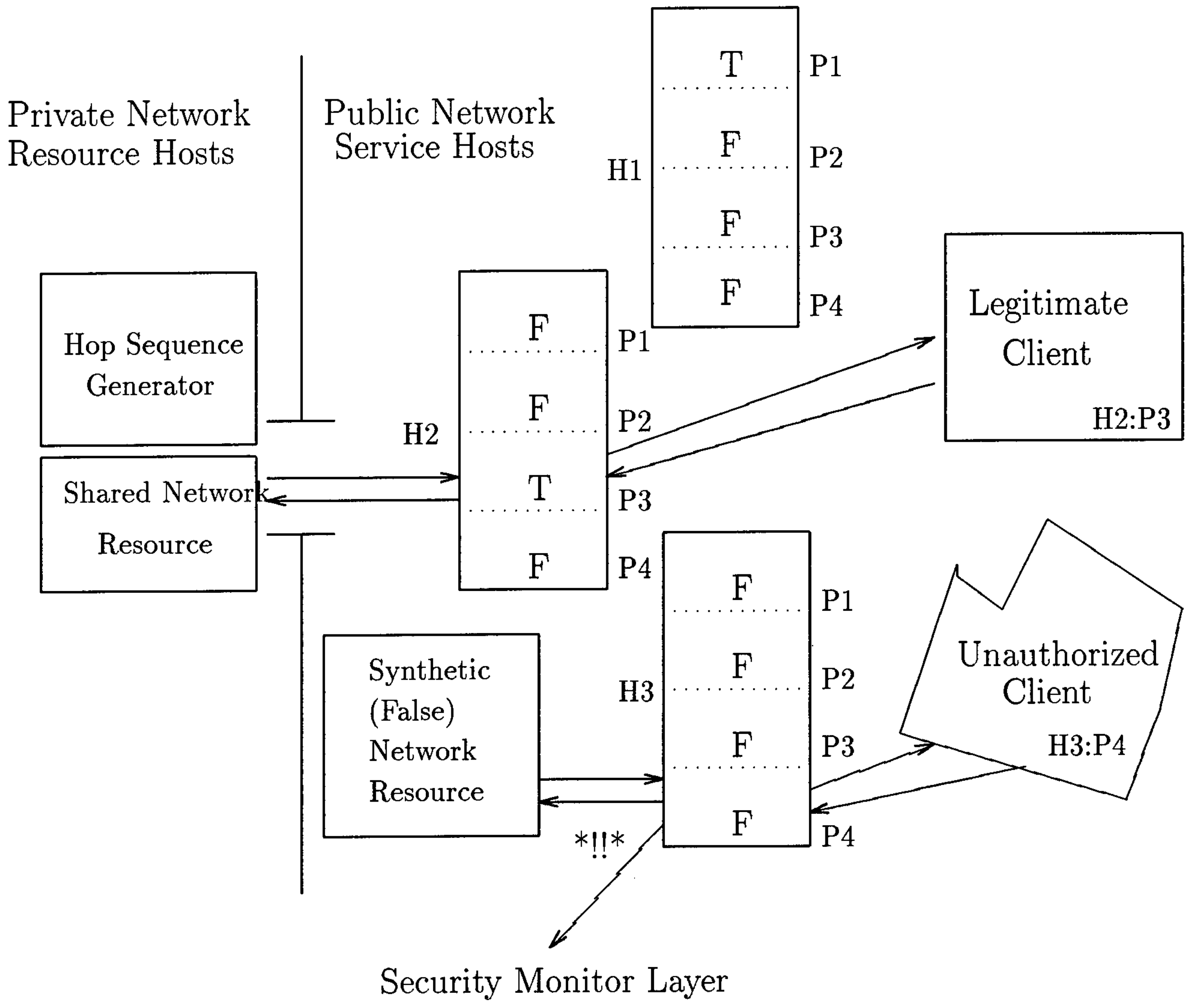


Figure 1: Service-hopping system snapshot between hop windows. Service instances are identified by **H**ost and **P**ort. One client has found a legitimate service instance at H2:P3, designated by a **T**, relaying sensitive data from a shared resource in a secure subnet. The other client has stumbled across a false service instance at H3:P4; this service gives out dummy data while alerting the security layer to the presence of an intruder.

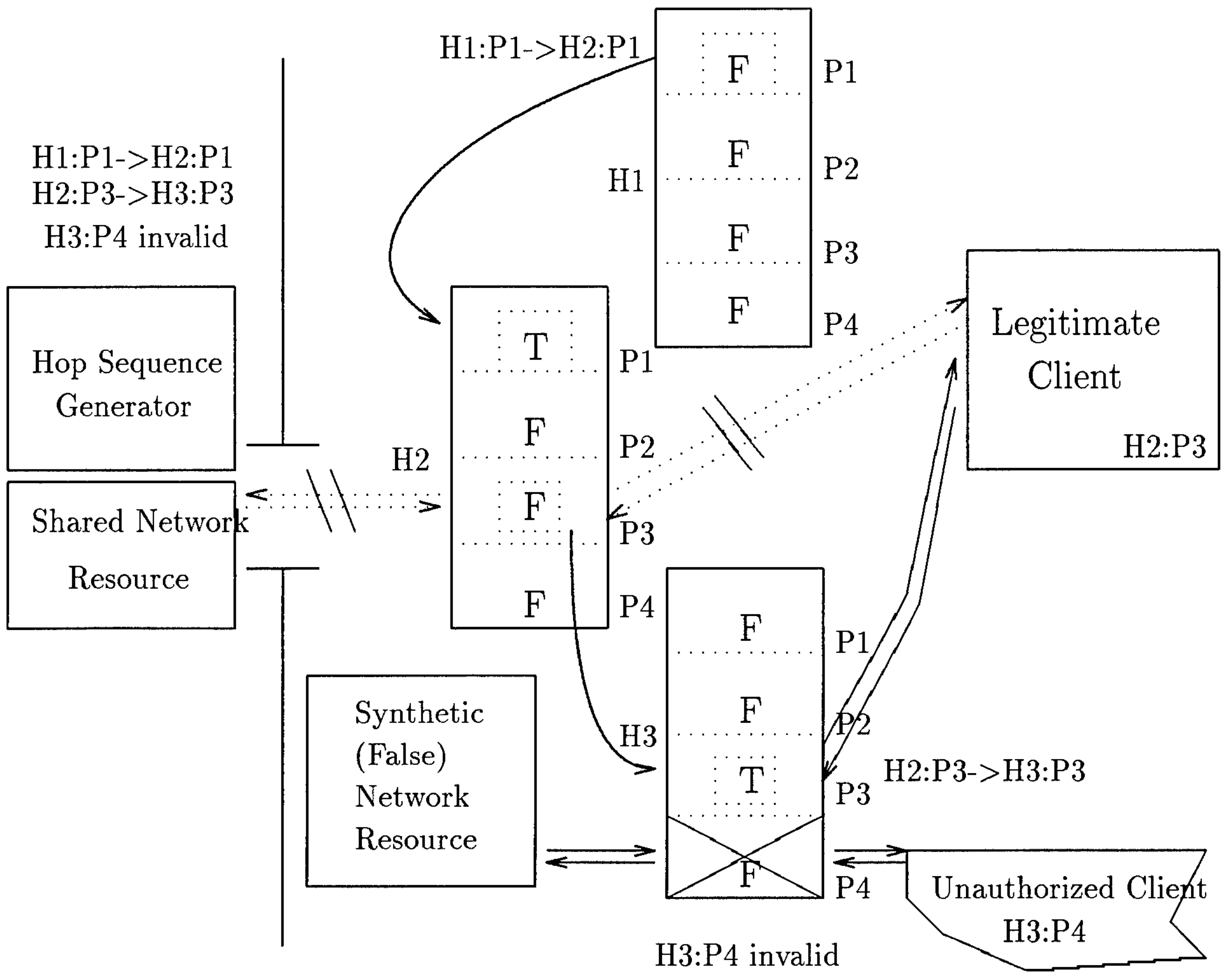


Figure 2: Service-hopping system snapshot of events within one hop window. Two true service instances have stopped offering true service; their jobs are taken over by two previously false service instances. The false service instance which registered a possible intrusion in the previous phase has been marked invalid and removed from the hopping sequence. The authorized client uses its knowledge of the hopping sequence to synchronously close its connections to the newly invalidated instance, and direct future connections to a newly validated instance.

vice versa. The hop sequence itself may be transmitted, or just the random sequence seed (if the generator is known to both servers and clients).

Servers get their hop sequence information through a slightly different mechanism. The hop sequence generator sends each service instance an encrypted, personalized projection of the global hop sequence. The primary component of this projection, the *veracity sequence*, is simply a bitsequence in which a 1 represents a scheduled phase of local real service, and a 0 represents a phase of local false service.

In a system that used hop signals embedded in server content, the hop sequence projection would include a second component: the *target sequence*. This bytestream represents the host and port of the next hop target following each phase, including apparently reasonable, but meaningless target data during phases when a false service is called for.

Yet another variant might distribute a synchronized token stream generator (like those found in authentication smart cards) to each client and server. In the last case, the token generator could provide the global timing information for each hop, as well as the dynamic host/port target information for the legitimate service in the current hop cycle. Out-of-band sequence propagation has the benefit of reducing or eliminating the synchronization time required in each hop window, and strengthens the system against eavesdropping attacks.

Invalidating and Reissuing the Hop Sequence. At some point before the expiration of N hop cycles, all clients and servers must receive a refresher sequence. This refresh may be forced early, if a false service detects a security violation. For example, the false service instance in figure 2 which registered a possible intrusion in the previous phase has been marked invalid and removed from the hopping sequence. The remaining hops in that sequence are therefore discarded and globally replaced by new data — an expensive, but hopefully infrequent operation.

The security of the hopping ruse derives from two sources: the encryption of the hop sequence data in transit, and the requirement that an eavesdropper must intercept, decrypt, and merge many independent subsequences in order to reliably reconstruct even a partial consecutive hop sequence. Presampling the random hop information allows us to improve the performance of service-hopping computations by eliminating the time spent determining a destination in the synchronized hop window.

3.2 When to Hop?

Having answered the question of where legitimate services can be found, there are at least two ways of defining the intervals at which a hop is to take place in a client-server system.

An *asynchronous hopping scheme* keys the hop times to the underlying structure of the data being served. Many important network services, including servers for audio and video data, are "fronts" for an underlying *marked-stream* resource. Marked streams allow the hop points to be keyed to marks within the data itself. For example, a video stream server may be keyed to "hop" every 30 frames.

For other network applications, including transactional services such as databases and web servers, there may be no appropriate structural clue to guide the placement of a hop. A *synchronous hop scheme* uses a generalized coupled-oscillator method to bring client clocks into synchrony with service clocks, and then hop on the rising edge of each common clock phase.

Making this work across a less-than-ideal network can be tricky; figure 3 illustrates the problem schematically. If T_h is the server's hop window delay, and T_t is the total hop cycle time (S.), then only the time $T_c = T_t - T_h$ is available for accepting transactions. From the client's perspective, this available window is phase-shifted by the minimum transit time of service requests across the network, and then muddied by the introduction of network timing noise. As a result, there are three regions of network conditions clients must consider when timing their transactions with a hopping server.

Region A. Constant Latencies and Infinite Server Capacity. In an ideal client/server network, the time for a transaction to travel from the client to the server and enter into service will be a well-behaved constant. We discuss some design elements of such a system in [1]. In this idealized region of the problem space, a remote client synchronizes once, and subsequently observes a fixed phase correction T_d to the server to determine the next hop time.

Region B. Variable Latencies and Server Loads. When server load, clock skew, and network propagation delay variance are taken into account, the effective window T_e within which a client can confidently issue a transaction without risking an inadvertant security false alarm begins to shrink. Under these circumstances, the client will be forced to delay some

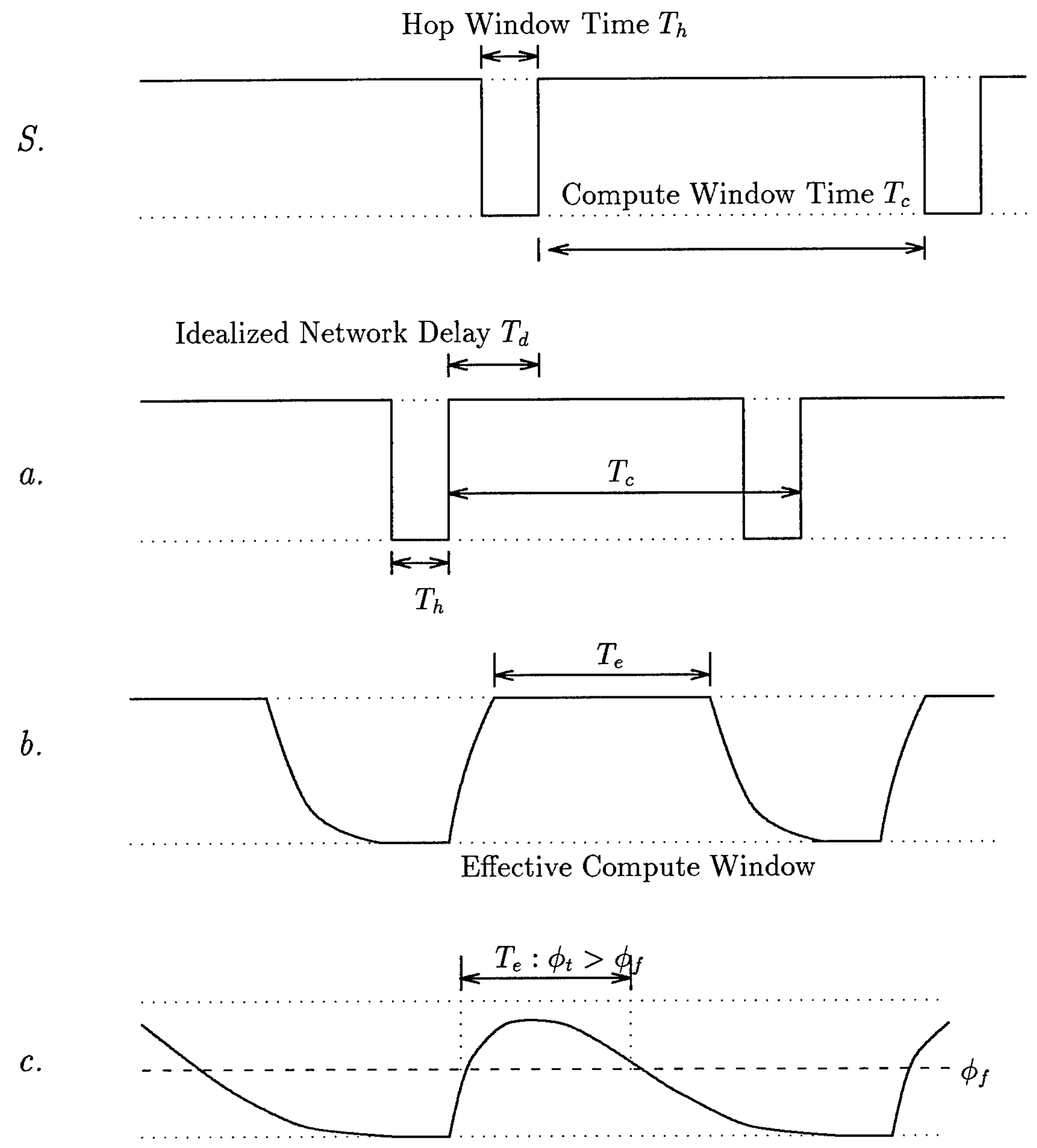


Figure 3: Schematic illustration of the problem of synchronizing the effective computation windows of a hopping client and hopping server separated by a less-than-ideal network. The collection of hopping services (S) stops accepting transactions during the hop window T_h. Remote clients (A, B) try to get their requests into service with the correct service instance within this window, subject to transmission delays, server load, clock skew, and unreliable links. In the worst case (C), clients have no guaranteed window for issuing transactions.

transactions until the rise of the next hop window. This is a problem shared with all coupled-oscillator methods across a network containing multiple gateways, routers, and unreliable links; the best that can be hoped for is a graceful degradation of throughput as the noise between clients and servers increases.

Region C. Unstable Network and Service Times. In this region, the probability of a successful (non-security-violating) transaction never reaches one at any point in the hop cycle. In such an extreme network environment, hopping clients have no guaranteed effective window T_e within which to issue transactions. Under these circumstances, the user can apply a probabilistic filter which allows tentative progress during all times t within T_c when the probability ϕ_t of a successful transaction exceeds a trigger ϕ_f.

In our prototype, for example, the client tracks the average and variance of its historical service times, and waits for the next window whenever a service time outside a specified probability tolerance would result in a false security violation. This event (a "hop stutter") adds to document retrieval time by as much as the length of the hop window. in practice, the overhead is much less, precisely because the stuttering client believes that the time to be spent sleeping until the rising edge of the next hop window is small.

Safety Margin Considerations. The safety margin can be computed via a number of strategies, from simple conservative calculations (1/2 or 3/4 of the total hop window) to elaborate statistical estimates that factor in the variance in the round trip time between document request and header receipt. In the most conservative strategies, clients may experience as much as a 95% stutter rate per document request. This can add nearly one hop window of worst-case latency to each document fetch (1 to 3 seconds), which is a significant subjective delay.

To minimize this delay, hopping service clients can use alternative strategies, with less conservative safety margins. Rather than simply issuing a stutter within a fixed percentage of the end of the window, more intelligent strategies factor in information about the noise of the link and server load, by maintaining a history of document fetch time variances. These improved safety margins cause stutter rates to drop, and improve document fetch throughput, at the expense of increasing the number of false security alarms. In the next section, we describe the role of the security manager in managing real and false alarms in a heterogeneous hopping system.

4 Security vs Performance

Realistic networks of heterogeneous processors are noisy places. Sometimes, despite a client's best efforts to remain synchronized with a hopping server at a reasonable stutter rate, a legitimate client will access a stale server (one which is no longer legitimate). The stale server must assume that the request for service is an intrusive probe, and report it to the security monitoring layer.

This network management layer, in turn, must implement a policy for filtering incoming alarms. By themselves, single alarms may indicate a transient problem resulting from congestion or server overload; beyond some sequence threshold, however, false server accesses from a given site or domain should be abstracted into a single intrusion warning that draws attention from an administrator. Our experience with a hopping Web server prototype indicates that a few observed fault behaviors are typical, and should be factored into the design of this filtering system.

Single-step violations. In order to practically eliminate false accesses, the hop window length must be extremely long (on the order of tens of seconds for an intranet, minutes across a noisy internet) and the client must agree to suffer some hop stutter due to conservative safety margins. It's not necessary to actually eliminate all errors, however; with proper alarm filtering and context information, more generous strategies can still provide adequate security guarantees.

We observed that a very high percentage of inadvertant false accesses (90% for 500ms hop windows, 99% for 2500ms hop windows) are only out-of-step by one hop. That is, the failed access attempt would have been valid in the immediately previous hop window.

This confirms that the security layer has an important role to play in maintaining a good tradeoff of security for performance. Rather than expand safety margins on the client side, or expand hop windows on the server side, the security layer can choose to selectively classify one-hop errors as stale accesses, and ignore them. This enables the use of short (1-2 second) hop windows, and liberal safety margin calculations that reduce stutter rates by 70% or more.

Malignant patterns. The probability of falsely categorizing a true illegal access as a benign stale hop is $1/N$, where there are N hop servers. For a reasonable number of hop service instances (in our trials, 4-8), this still leaves a reasonable chance for a single il-

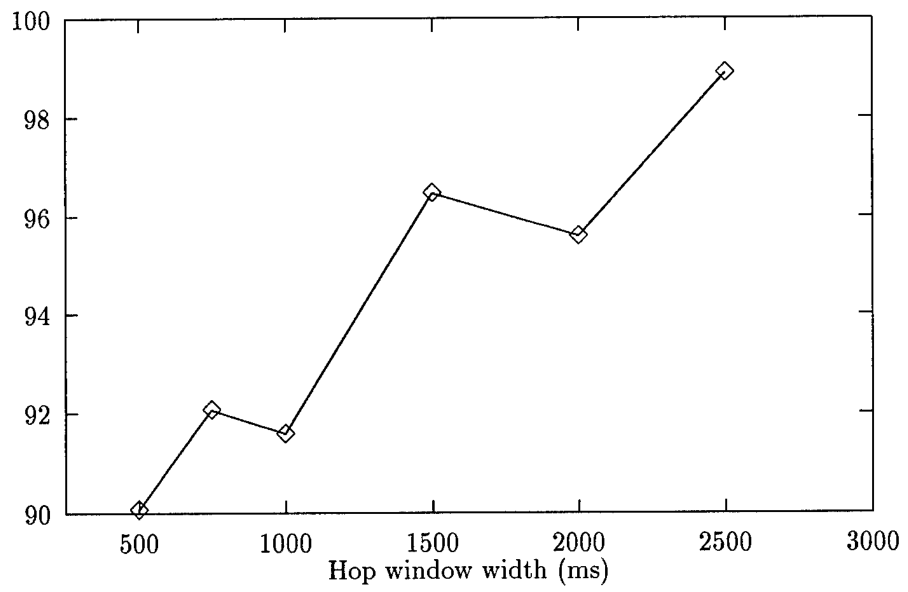


Figure 4: Percentage of false alarms due to a single-hop error in a prototype hopping Web server implementation. The vast majority of the inadvertant security violations, regardless of safety margin strategy or traffic load, are due to single-hop errors and can be distinguished from sequences of intrusive probes.

legal access to be accidentally silenced by the security layer.

In general, however, intrusive probes can be expected to repeat, obeying spatial and temporal patterns that can be detected. The security layer might reasonably maintain a history of all hopping violations from a given host or domain, and flag those alarm sequences whose length exceeds some small expected threshold value for the hopping server parameters that are in force.

Intrusion Containment. Once such an intrusion is detected, the administrators of heterogeneous network environments must regard all system services which may have been compromised as suspect. The security layer therefore has the power to invalidate outstanding hopping codes and force them to be reissued. This new hop sequence may exclude service hosts that may have been the target of an intrusion, or which are consistently overloaded, resulting in false alarms due to missed hop windows. (The security layer may also log and filter future requests from the domain that originated the offending request, to frustrate denial-of-service attacks based on a flood of indiscriminate accesses to known false services.)

Beyond first contact, the false services' bogus content may help administrators keep an intruder online long enough to log and trace their actions. This gives time to identify the mode of attack and assess the extent of damage to system components, a key requirement for closing holes and warding off future intrusions [2].

4.1 Mitigating Hopping Overhead

Minimizing the additional server load and network bandwidth required to support a secure hopping scheme is a key concern. Like any other security measure, the implementation of service hopping trades away some performance; how much and what kind depend on several design factors. As we have already described, two of these design factors can be minimized by careful engineering: the time spent in the global synchronization/migration phase of the hop window, and the overhead required to avoid false security violations due to network noise and server load. The server administrator must make locally appropriate decisions for the other two factors: the fidelity of the content provided by false services, and the relative population of true and false instances throughout the heterogeneous computer.

False server overhead. The number of false hopping services alone does not affect network bandwidth consumption; legitimate service requests are simply "smeared" over the available services, following the hopping sequence. Instead, server load may become an issue, especially if there are not enough spare nodes willing to host false server instances, and multiple hopping services must be mapped to the same processor.

If server load, and not local network bandwidth, is the limiting factor, a system of lightweight "hopping proxies" can pass through content from two hidden, centralized services, one providing real content and the other false content. This strategy restricts the number of full servers to 2 (or even 1 if the difference is precomputed; e.g., a different document root for true and false HTML content). Of course, there's a price for protecting the anonymity of the underlying server from its clients: up to twice the local bandwidth requirement, to support the hopping server proxies that relay content.

False client overhead. Similarly, maintaining false clients can have a considerable effect on resource consumption. Incoming network traffic and server load scales linearly with the false client count, a fundamental restriction if network bandwidth is scarce, or servers are inadequately sized.

Recall, however, that hopping service schemes work entirely well without the presence of false clients, which only serve to confuse traffic analysis. If secure out-of-band hop sequence generation is available, administrators can combine extremely long, nonrepeating hop sequences with relatively short hop windows. That effectively eliminates an eavesdropper's advantage, since snooped hop sequence data goes stale before it can be used, and never becomes useful again to derive the identity of the true servers. Under such circumstances, administrators may elect to live without the smokescreen of false clients, and eliminate their additional bandwidth requirements and server load.

5 Applications

Consider a heterogeneous environment consisting of three corporate campuses, each protected by its own firewall from casual attack from the Internet. (This network configuration will become increasingly common with the advent of global-scale design and manufacturing operations, as industrial espionage motivates concerted attacks on proprietary data.) Each of the three campuses needs to publish certain information services for the benefit of the others, but wishes to exercise control over the distribution of sensitive content. In figure 5, site A offers realtime video and audio content, site B offers access to the corporate database, and site C stores the corporation's master Web document tree. A high-speed interconnect links individual hosts within each campus, and there are two alternatives for links between campuses. Internet routes that share bandwidth with strangers' traffic offer an inexpensive solution for connectivity, but dedicated private lines offer physical security.

One solution is to use a combination of structure-based (hopping) and content-based (encryption) techniques to secure client-server resources against attack over insecure networks. Consider site A, the campus that needs to offer secure streaming video service. Without hopping, site A would probably dedicate one port to video, configure the firewall to pass traffic on that port, authenticate clients using a public or private key scheme, and encrypt each video transmission. Adding hopping to the picture, site A could dedicate one port for hop sequence data, plus 4 ports for video service. They might run four video servers on each of four hosts within the campus network, giving a false-true server ratio of 15:1, while continuing to authenticate clients and encrypt content as before.

The true services and clients in such a hopping system use roughly the same amount of server time and network bandwidth as if only a single non-hopping server were involved. The performance tradeoff actually involves an orthogonal issue: paying for deception. False servers can be configured to produce a common load average across campus C's four hosts (true and false) to prevent a casual observer from narrowing the odds. False clients multiply bandwidth requirements a fewfold, but only enough to mask the relatively small number of legitimate clients who are likely to connect to a secure network service. These parameters are user-specified, according to the sensitive nature of the application.

If we can rewrite the video client code to implement hopping directly, then users from campuses B and C will use a modified multimedia client to authenticate themselves to A, receiving authenticated hop sequence data in return. An alternative that avoids rewriting client code involves the use of a "hopping proxy" video server installed at B and C, which handles the authentication steps and hopping logic. In their spare time, these proxy servers also launch false database clients to confuse eavesdroppers. In figure 5, campus B is offering hopping database service, and proxy database servers have been installed at campuses A and C.

5.1 Prototype System

As a proof of concept, we constructed a web client-server system implementing a service-hopping proto-

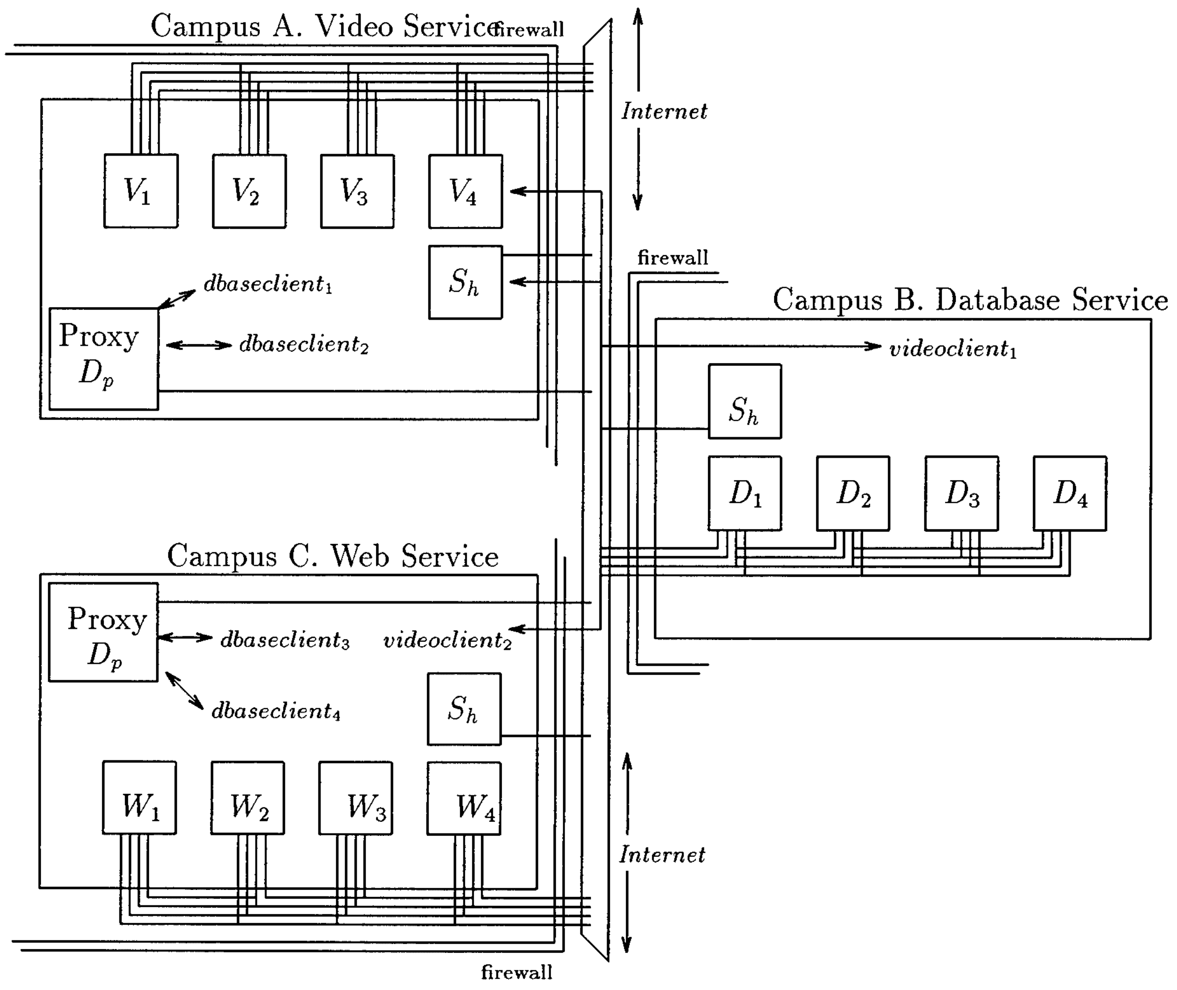


Figure 5: Large, heterogeneous service-hopping system consisting of three campuses, four hosts per campus, using four ports to implement a total of 45 false and 3 true services. Campus A publishes video data, campus B offers database access, and campus C hosts the corporate Web document tree. Clients of the video server interact directly with the remote service instances; campuses A and C offer their users a simpler proxy interface to B's database, handling authentication and hopping mechanics transparently.

col, like that of campus C in figure 5. Web server security rang[illegible]om simple username/password authentication for individual subtrees within the document space, to complex schemes that encrypt and authenticate client-server traffic [4, 5, 7]. A hopping scheme can add an extra measure of security to such a system, even in cases where the passwords for a page have been compromised.

For the "bait" portion of the ruse, we built Java application classes that serve HTML content to Web browsers using the HTTP/1.0 protocol. The distinction between a real service and a false service, in this simplified case, lies in the document tree. We provided a hierarchy of convincing fake documents for the false servers to provide to clients; true servers reference the standard document tree. We also constructed Java application classes that, given a fragment of representative access log, stage periodic accesses to HTTP server documents, replicating the activity of a human browser to defeat traffic-based identification of the real web server of the moment.

We then constructed a proxy Web server, also in Java, which redirected all document requests to the currently legitimate Web server on a remote host. In our scenario, the proxy server runs at a selected remote site, giving users within that site access to a sensitive document tree through the hopping mechanism. An intruder who sniffed the page passwords from the net would nonetheless be locked out without the proxy server's private hopping information.

We had no trouble keeping the proxy locked in sync with the hopping servers, using hop windows of 1 to 2 seconds, as long as the interval between submission of a document request by the proxy and the approval of the request within the current window on the remote server was relatively stable. If the proxy issues a request which takes too long to enter service on the remote host, it may miss the current hop window, resulting in a false security alarm. The alternative is to delay service requests until the rise of the next window, reducing throughput. The proxy server must start with a conservative estimate of the width of its effective compute window, and then work to expand that window to improve performance.

Initial Synchronization First, during initial authentication with the remote server, the proxy downloads its initial hop sequence. Each line in that sequence is timestamped as it is generated by the hop sequence server and downloaded (byte-at-a-time) by the proxy. This gives the proxy a rough estimate of the average and variance of the phase offset T_d between its clock and the server clock, including the transit time for short messages. If the hop window exceeds this average transit time by less than several standard deviations, the proxy server attempts resynchronization periodically until the network has quieted down.

This process only attempts to synchronize the proxy, or client, with the distant collection of hopping servers. Because the service hosts and the sequence server are all assumed to be within the same administrative domain, we assume that their system clocks could be trivially presynchronized using, for example, NTP[6].

Once the servers have synchronized successfully, the proxy then makes document requests within T_e of the rising edge of the remote hop window. T_e starts off as $T_c - T_d - 2\sigma_{T_d}$ — the compute window time less the clock offset and two standard deviations.

Periodic Adjustment Because some variable delay arises from load on the remote server (to begin to locate and transmit the document), and because message transit times may vary, each document served by the hopping server includes a header line for clock adjustment. This line contains the server clock state at the moment the document began transmission, and allows the proxy to compute the time which remained in the current hop cycle. The proxy uses a sliding window of these times to recompute the variance and adjust its estimate of the phase correction to arrive at a new effective window T_e. If the network becomes quieter, T_e expands; if the network becomes noisier, T_e shrinks.

Conclusion. All these measures are necessary because above some moderate noise level, hopping strategies break down due to sporadic missed hop windows, resulting in nuisance security alarms. For this reason, hopping techniques will be of most value in contexts where the network links between clients and hopping servers are relatively low-latency, when servers are adequately sized to avoid service delays due to load, and when the time required to initiate each service is relatively constant.

Application-level structural techniques such as service hopping help balance security, performance, and convenience: three primary design criteria for heterogeneous network management tools. The hopping ruse provides a new way to map tasks to heterogeneous resources, in order to augment the security provided by more traditional content-based encryption techniques.

Service hopping has the dual advantages of complicating the intruder's job, while giving administrators early warning of a potential breakin attempt and isolation of the affected component. We expect that the general techniques presented will be applicable to many heterogeneous distributed application contexts of commercial and military significance, including databases, multimedia, signal processing, and distributed control.

References

[1] A. Bestavros, M. Chen, M. Crovella, A. Heddaya, S. Sclaroff, and J. Cowie. TR96-008. Responsive Web Computing: resource management, protocol techniques, and applications. Technical report, Boston University and Cooperating Systems Corporation, 1996.

[2] W. Cheswick and S. Bellovin. *Firewalls and Internet Security.* Professional Computing Series. Addison-Wesley, 1994.

[3] The Open Group. DCE security. Technical report, http://www.opengroup.org/tech/dce/security, 1997.

[4] S. Kent. RFC-1421. privacy enhancement for internet electronic mail. Technical report, Network Working Group, 1993.

[5] J. Kohl and C. Neuman. RFC-1510. the kerberos authentication service (V5). Technical report, Network Working Group, 1993.

[6] D. Mills. RFC-1305. network time protocol (version 3). Technical report, Network Working Group, 1992.

[7] E. Rescorla and A. Schiffman. Secure hypertext transfer protocol (working draft). Technical report, Enterprise Integration Technologies, 1995.

Marina Chen received the B.S. degree in Electrical Engineering from National Taiwan University in 1978, and the Ph.D. degree in Computer Science from California Institute of Technology in 1983. She is currently Professor and Chair of the Computer Science Department at Boston University, where she serves on the Council of the Center for Computational Science at Boston University. She is President of Cooperating Systems Corporation, and Secretary of the Association of Computing Machinery.

James Cowie received the B.S. degree in Computer Science from Yale University in 1991. He is currently the Director of Technology Development at Cooperating Systems Corporation. His FAFNER package for Web-based distributed computing brought the RSA130 team the "Most Heterogeneous/Most Geographically Distributed" award in the HPC Challenge at Supercomputing '95. His current research interests include software support for secure wide-area collaboration, distributed simulation, and computer-assisted financial engineering.

Case Study

A Performance and Portability Study of Parallel Applications Using a Distributed Computing Testbed

Viorel Morariu, Matt Cunningham, Mark Letterman
Concurrent Technologies Corporation,
Johnstown, PA, USA

A Performance and Portability Study of Parallel Applications Using a Distributed Computing Testbed

Viorel Morariu, Matthew Cunningham, and Mark Letterman
National Applied Software Engineering Center (NASEC)[1]
Concurrent Technologies Corporation (*CTC*)
1450 Scalp Avenue
Johnstown, Pennsylvania, U.S.A.

ABSTRACT

A case study was conducted to examine the performance and portability of parallel applications, with an emphasis on data transfer among the processors in heterogeneous environments. Several parallel test programs using MPICH, a Message Passing Interface (MPI) library, and the Linda parallel environment were developed to analyze communication performance and portability. These programs implement loosely and tightly synchronized communication models in which each processor exchanges data with two other processors. This data-exchange pattern mimics communication in certain parallel applications using striped partitioning of the computational domain. Tests were performed on an isolated, distributed computing testbed, a live development network, and a symmetrical multi-processing computer system. All network configurations used asynchronous transfer mode (ATM) network technologies. The testbed used in the study was a heterogeneous network consisting of various workstations and networking equipment. This paper presents an analysis of the results and recommendations for designing and implementing course-grained, parallel, scientific applications.

1.0 Introduction

Solving today's complex, scientific and technological problems requires powerful computer platforms. Currently, many high-end computational systems feature parallel architectures. Consequently, parallel implementations of scientific applications are becoming more important. In addition to using high-end computer platforms, a growing number of organizations are using distributed, networked resources to run parallel applications. The use of networked computers presents additional challenges to the application developer. Among these challenges are the relatively limited communication performance of the existing networking technologies and the difficulty in using the available bandwidth and processing power consistently.

Parallel implementations enable us to solve more complex problems and to reduce execution time; however, the increase in performance depends on the type of application, algorithms, and hardware characteristics. Powerful processors found in today's parallel systems and limitations in networking technology favor coarse-grained parallel implementations, in which a high number of operations per byte transferred minimize communication requirements. Coarse-grained parallelism [1] allows more computation to be performed locally between two synchronization events. It reduces the number of messages and the volume of data that processors exchange. Due to its suitability for several parallel architectures, coarse-grained parallelism also enables the development of portable applications. One can develop a parallel application in any parallel environment, such as a network of workstations (NOW), and subsequently port it to shared-memory multiprocessors or distributed-memory parallel machines.

Test programs were developed to evaluate communication performance using several communication patterns. The results of these tests can be used in designing and implementing parallel algorithms and communication techniques. Two scientific applications

[1] NASEC is operated by Concurrent Technologies Corporation (*CTC*) and is sponsored by the Defense Applied Research Projects Agency (DARPA). The content of this information does not necessarily reflect the position or the policy of the government, and no official endorsement should be inferred.

0-8186-7879-8/97 $10.00 © 1997 IEEE

were used as references for this study: a finite-differences time-domain (FDTD) electromagnetic simulation and a viscous fluid flow simulation. The electromagnetic simulation is based on solving a reduced set of the Maxwell equations for the electromagnetic field. The implementation is an explicit, multi-domain model, suitable for parallel execution, that requires data to be exchanged at each time step. For the viscous fluid-flow simulation, the single-domain numerical solution requires large systems of equations to be solved (implicit implementation). In the multi-domain model, an iteration by subdomains method that requires local solvers and interface conditions is considered. The computational domain is partitioned into subdomains, and each subdomain is allocated to a processor. Multiple subdomains can coexist on the same processor. For both applications, each processor communicates with at least two other processors. The partitioning of the domain into subdomains and the allocation of subdomains to participating processors induce certain communication patterns that were captured in the test programs.

In most implementations, the programs running on multiple processors are synchronized periodically, and messages are sent within each cycle (timestep or iteration on the complete domain); however, the developer has the latitude to implement either tightly or loosely synchronized models. In this paper, we have analyzed the network response to both tightly and loosely synchronized models. Low computation for each byte transferred implies that computation and communication must be overlapped to provide good performance for parallel execution of the application.

The test programs implement data-exchange patterns that mimic communication in parallel applications similar to the electromagnetic and viscous fluid flow simulations presented above, using striped partitioning of the computational domain. The main hardware target for the tests is a heterogeneous network of workstations. Test results on the heterogeneous network are compared to those on homogeneous, isolated and active networks and on a symmetrical, multiprocessing platform. MPICH [2,3], a public domain Message Passing Interface (MPI) implementation, was used in the parallel test programs. In addition, comparable implementations using Linda [4], a Scientific Computing Associates, Inc. (SCA)[2] product, were used in some tests to complement or reinforce the conclusions.[3]

2.0 Test environment

MPICH (version 1.0.12) and Linda (version 3.1) were used in the tests. Several implementations of MPI are available in the public domain or from commercial vendors. Linda is a commercially available parallel programming environment. MPI and Linda are accepted in the high-performance computing industry and are reported to have good performance and portability. The two environments are very different in their level of abstraction and programming complexity. MPI is based on the message passing programming paradigm, while Linda is based on the distributed data paradigm (virtual shared memory). When using MPI, the programmer can perform fine tuning of communication [5]. Explicit communication among processors and a wide range of message passing capabilities make this possible. Fine tuning, however, requires extensive knowledge of the MPI environment and parallel architectures, making the programmer's job more challenging.

Performance testing was conducted on heterogeneous and homogeneous NOWs and on a Silicon Graphics, Inc. (SGI) Onyx symmetric multiprocessor (SMP). Tests were performed on an isolated local area network (LAN) removed from user activity, as well as on an active LAN with varied traffic load. Both networks employed ATM technology. The Andrew File System (AFS V3.4) was used as the distributed shared file system in all configurations employed by the study.

The isolated NOW testing used an established, heterogeneous, distributed computing testbed environment. System resources within the testbed included:

- SGI Indy workstations, each equipped with one MIPS R4000 100 MHz microprocessor and 48 Mbytes of RAM
- SGI Indigo II workstations equipped with a MIPS R4400 200 MHz microprocessor and 96 Mbytes of RAM
- Sun Ultra 1 workstations, each equipped with one UltraSPARC 167 MHz microprocessor and 96-128 Mbytes of RAM
- IRIX 5.3 operating system on all SGI workstations
- Sun OS 5.5.1 (Solaris 2.5) operating system on all Sun platforms.

[2] Linda is a registered trademark of Scientific Computing Associates, Inc.

[3] This paper does not choose between Linda or MPICH for the development of parallel applications.

The testbed also provided several network technology and protocol options such as Ethernet, Classical IP (Internet Protocol) over ATM, and FORE IP over ATM. Based on network communication benchmarking [6], it was determined that the standard Classical IP protocol over ATM results are very similar to FORE IP over ATM. We chose to use the FORE IP over ATM for this study. Figure 1 shows the heterogeneous testbed environment that was used for this study.

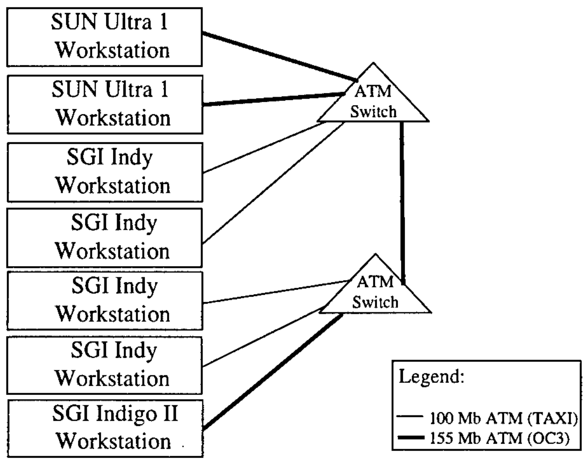


Figure 1. Heterogeneous Testbed Environment

The testbed network configuration consisted of a two-switch ATM work group with an OC3 (155 Mbits per second) connection between the switches. The ATM switches used were the FORE Systems ASX200BX models with a combination of 100 Mbits per second Taxi interfaces for SGI Indys, and 155 Mbits per second OC3 interfaces for SGI Indigo II and Sun Ultra 1. All ATM equipment used FORE Systems software version 4.0. Because the testbed was completely isolated during the tests, the results represent ideal or best case situations.

The NOW on the active LAN featured similar characteristics to that of the isolated NOW, including similar workstations and network configurations. The active LAN supported approximately 50 users. Workstations with low computational load were selected for the test. Therefore, we assumed that the pre-existing computational load would not affect the test results significantly. The goal was to reveal the impact of the pre-existing network traffic on the communication-intensive, parallel applications.

The SGI Onyx featured eight (200 MHz) microprocessors with a main memory of 1 Gbyte. Except for choosing a time period when the Onyx had low computation load, no special measures were taken to control the SMP.

3.0 Test programs

The performance tests target communication aspects of parallel applications, such as FDTD electromagnetic simulations and fluid flow simulations, using an iteration by subdomains method, described in section 1.0. Two synchronization models are considered—a loosely synchronized model and a tightly synchronized model. Communication using the loosely synchronized model has the advantage of smoother, non-burst transfers. When all senders use the network or memory bus at one time (tight synchronization), arbitration delays may limit the performance. Within the loosely synchronized model, requests for network or bus access occur at a more uniform rate. In general, better execution times are expected for the loosely synchronized model.

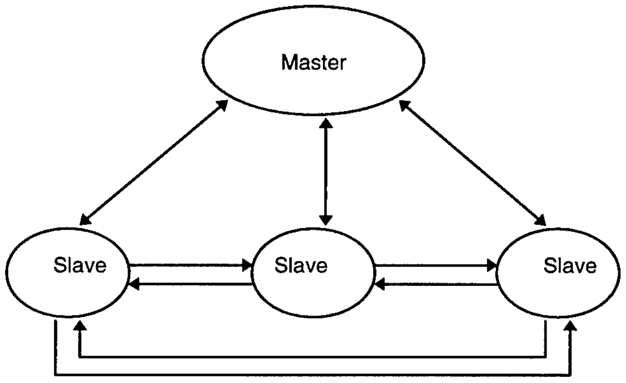


Figure 2. Generic Performance Testing Model

To determine practical and obtainable performance expectations, it was necessary to devise tests that closely simulate parallel application behavior with regard to data exchange. The test software configuration consists of a master process and multiple slave processes, as shown in Figure 2. The master process manages the slave processes and reports the results. The slave processes perform data transfers without processing the data received. Each participating processor executes only one process (master or slave).

The test software begins execution by examining input parameters and performing various overhead operations. Once the network topology and test environment parameters are known and distributed throughout the configuration, the test programs measure communication performance, specifically, data transfer times. Transfers are performed repeatedly to generate valid statistics.

The loosely synchronized model relies on slave-slave communication for minimal synchronization, and the tightly synchronized model uses slave-master communication for more rigid synchronization. Table 1 shows the main design characteristics of the two models. The total execution time is used to compare the models.

Each test was executed three times. The execution time average was then used for analysis. Each test consisted of 100 iterations. Within one iteration, each of the four slave processes sends the specified size message to two partners. Using execution time and message size, one can approximate the application bandwidth (total amount of data exchanged for 100 iterations divided by execution time). In configurations with four processors running slave programs, the total amount of data exchanged over the network for 100 iterations is *8*100*(message size).* The bandwidth defined above depends on the characteristics of both the test program and the parallel environment (hardware and software).

Test Model	Loosely Synchronized	Tightly Synchronized
Specific Characteristics	• Master only communicates with slaves on initialization and termination • Slaves can only send successive messages after receiving a message from each partner	• Master communicates with all slaves in each iteration to start slave-slave transfer • Each slave-slave transfer is acknowledged
Generic Characteristics	• Number of slaves in configuration is variable • Each slave sends and receives from two slave partners • Message transfer size is variable • Number of iterations for testing is variable • Master determines total execution time (all iterations)	

Table 1. Performance Testing Software Model Characteristics

4.0 Performance tests on NOWs

For the MPICH implementations, loosely and tightly synchronized models were developed using the blocking standard communication mode. (MPI may buffer outgoing messages.) Other variations, such as user buffering and nonblocking, were considered but not studied due to time constraints. Test results are presented for heterogeneous and quasi-homogeneous, isolated NOWs and for an active, homogeneous NOW. Several data types were used (byte, char, and double) for tests on the isolated NOW, but no significant differences in performance were found. Transfers using MPI_BYTE data type are discussed in this paper.

Additionally, the models were implemented using Linda, and the test results for a quasi-homogeneous, isolated NOW are discussed in this paper.

The quasi-homogeneous configuration employed four Indy workstations running the slave programs and an Indigo II running the master program. For the performance tests described in the previous section, this configuration can be considered homogeneous because the amount of data communicated to and from the master processor is very small compared to the amount of data exchanged among the slaves. The heterogeneous configuration employed two INDY workstations, two Ultra 1 workstations running the slave programs, and an Indigo II running the master program. The Ultra 1 workstations were equipped with a faster microprocessor, more memory, and a faster ATM card; however, the allocated workload and the data transfer volume were the same.

4.1 Tests using MPICH on isolated NOWs

This section compares the performance results of the MPI test programs for loosely and tightly synchronized communication on isolated NOWs in quasi-homogeneous and heterogeneous configurations.

In heterogeneous environments, the loosely synchronized model performed significantly better than the tightly synchronized model for messages up to 1 Mbyte. For messages larger than 1 Mbyte, the tightly synchronized model yielded better performance results. Note that for some tests in the heterogeneous configuration, the results varied significantly among the three executions. (As mentioned in section 3.0, each test was executed three times.) This variation was not exhibited in the homogeneous configuration.

On the homogeneous NOW (Figure 3a), the difference in the execution times for loosely and tightly synchronized models was even greater for very large messages. For the tightly synchronized tests, the execution times were up to three and one-half times longer.

A sharp decrease in the transfer rate for messages between 200 Kbytes and 500 Kbytes was common for both the homogeneous and the heterogeneous environments when the tightly synchronized model was used. In the tightly synchronized model, the network bottlenecks became critical due to data bursts. Performance of the tightly synchronized model improves relative to the performance of the loosely synchronized model when large messages are passed on a heterogeneous NOW. This result was not anticipated and is opposite to the result on the homogenous NOW; however, the absolute values of execution times were shorter in the homogeneous environment. In the loosely synchronized model, the execution time on the homogeneous NOW was more than five times smaller than the execution time on the heterogeneous NOW.

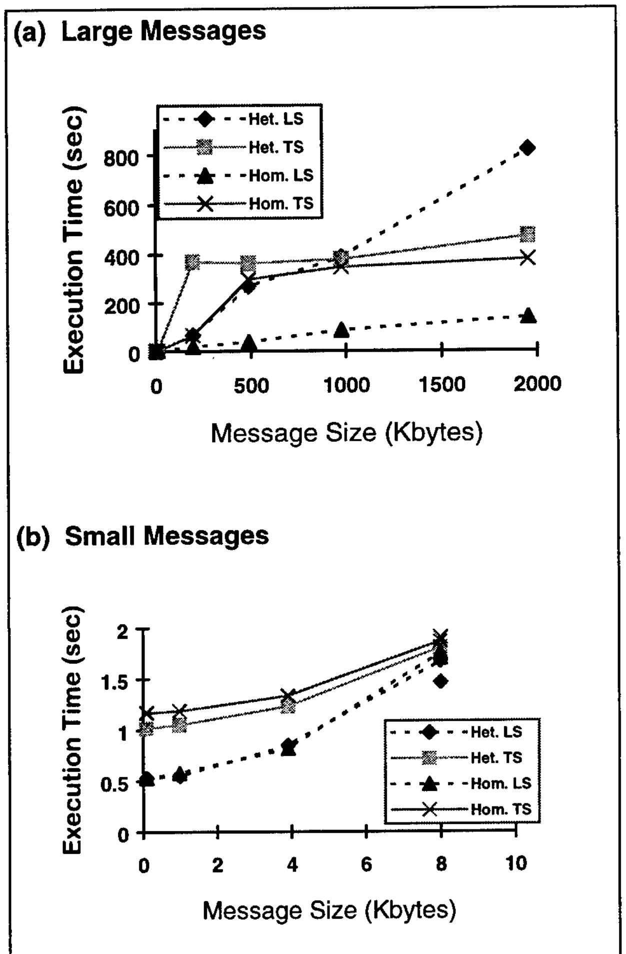


Figure 3. Tests using MPICH in heterogeneous (Het.) and homogeneous (Hom.), isolated NOW configurations. Loosely Synchronized (LS) implementations are compared to Tightly Synchronized (TS) implementations.

For small messages (to 8 Kbytes), the tests showed similar performance in both homogeneous and heterogeneous environments (Figure 3b). Loosely synchronized model implementations executed up to two times faster than the tightly synchronized model implementations for very small messages (100 bytes). Synchronization overhead and communication with the master could have caused this behavior.

Even though the cumulative computational power and network bandwidth were superior for the heterogeneous NOW, tests on the homogeneous NOW yielded better performance results. As expected, additional tests with two and three slaves on homogeneous NOWs showed better results on Ultra 1 NOWs than on Indy NOWs (30 percent shorter execution time). In general, for equal load, one would expect that the performance in a heterogeneous NOW should be at least the same as the performance in a homogeneous NOW using the least powerful workstation type in the heterogeneous NOW; however, additional measurement without MPI showed smaller throughput on a communication line connecting two different machines (Ultra 1 and Indy) than between two Ultra 1 or two Indy workstations. Also, a significant difference was noticed between the two directions of communication (from SUN workstations to SGI workstations and from SGI workstations to SUN workstations). We conclude that each platform's communication layers have been optimized for best communication in a homogeneous configuration. This behavior is consistent with the results of the MPI tests.

4.2 Tests using MPICH on an active NOW

Figure 4 and Figure 5 compare results obtained on the active, homogeneous NOW to results measured on the isolated, homogeneous and heterogeneous NOWs. The workstations and network hardware on the active network were similar to the workstations and network hardware on the isolated network.

As expected, the execution times for the active network were longer than they were for the isolated network (homogeneous NOW), as shown in Figure 4, due to varied, active network load. The execution times of the loosely synchronized tests were three to four times faster on the homogeneous, isolated network. In contrast, the execution times of the tightly synchronized implementations on the isolated network were comparable to the execution times of the test programs on the active network (Figure 5). Overall, tightly synchronized implementations for large messages yielded poor performance results on both active and isolated, homogeneous networks.

For large messages, the execution times on the isolated, heterogeneous network were longer than the execution times on the active network in a homogeneous configuration for both loose and tight synchronization. Relatively low, pre-existing communication load on the active network and the behavior of communication in heterogeneous versus homogeneous environments (section 4.1) contributed to these results.

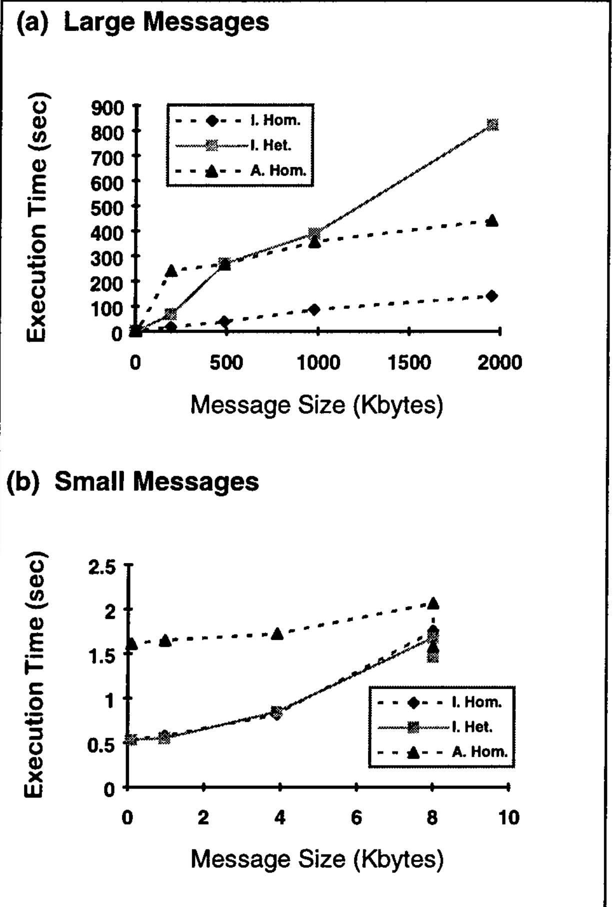


Figure 4. Loosely Synchronized (LS) tests using MPICH on an active, homogeneous (A. Hom.) NOW. The results of tests on the A. Hom. NOW are compared to the results on isolated, homogeneous (I. Hom.) and isolated, heterogeneous (I. Het.) NOWs.

The application bandwidth computed from the graphs shows values of up to 11.4 Mbytes per second for loosely synchronized tests on the isolated, homogeneous NOW. The maximum application bandwidth for tests on the active network is approximately 3.6 Mbytes per second (large messages).

For small messages, the execution times for the tightly synchronized tests (Figure 5b) on the active network were significantly higher (20 times) than the execution times for the loosely synchronized tests (Figure 4b). Given that on the isolated NOW the difference between the tightly and loosely synchronized implementations was significantly smaller, it can be concluded that the behavior was caused by network loading on the active network. The behavior of the loosely synchronized implementations (Figure 4b) seems to confirm this hypothesis. In this case, the execution times

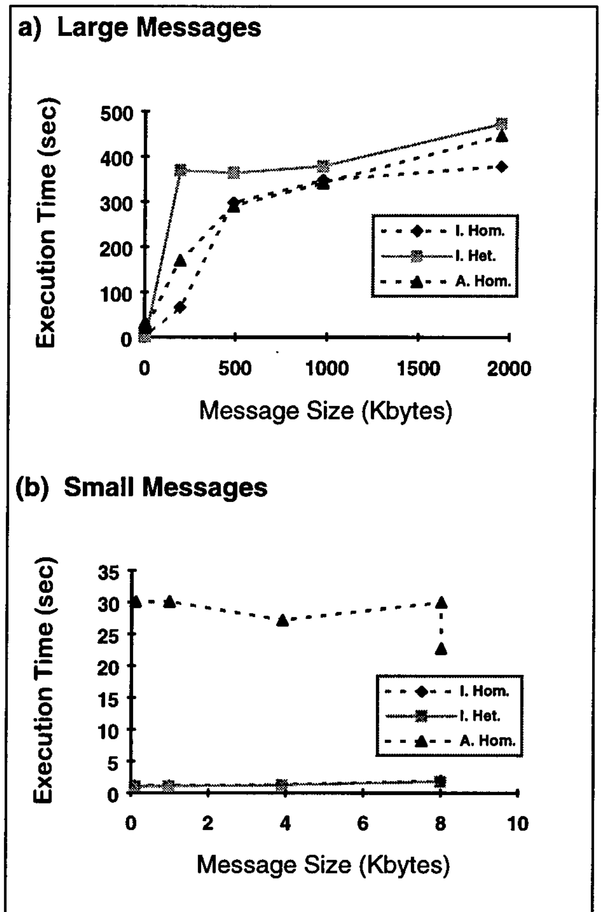


Figure 5. Tightly Synchronized (TS) tests using MPICH on an active, homogeneous (A. Hom.) NOW. The results of tests on the A. Hom. NOW are compared to the results on isolated, homogeneous (I. Hom.) and isolated, heterogeneous (I. Het.) NOWs.

on the active network were only slightly higher than the execution times for similar tests on the isolated network.

A drop in the execution time can be noticed at 8 Kbytes for both the isolated and the active NOWs (Figure 4b and 5b), possibly due to the packet size used at one of the multiple layers of communication.

As noted in section 3.0, four of the less utilized workstations on a LAN of 50 were used for the tests. The network equipment provided communication support to other applications, also. The results on the active NOW show that the un-utilized computing power can be harnessed to solve large problems; however, a proper jobs distribution tool based on the actual load and usage profile of the workstations should be used. Reliable, application-driven load balancing and check-pointing are also desirable.

4.3 Tests using Linda on an isolated NOW

This section presents the performance results of the Linda (version 3.1) test programs for loosely and tightly synchronized implementations on an isolated, homogeneous NOW. Results for the heterogeneous NOW are not available. Tests were also performed on an active network; however, only the results gathered on the isolated NOW are discussed here. (Results on the active NOW depended on the network load; the conclusions drawn from the tests using MPICH apply here, also.)

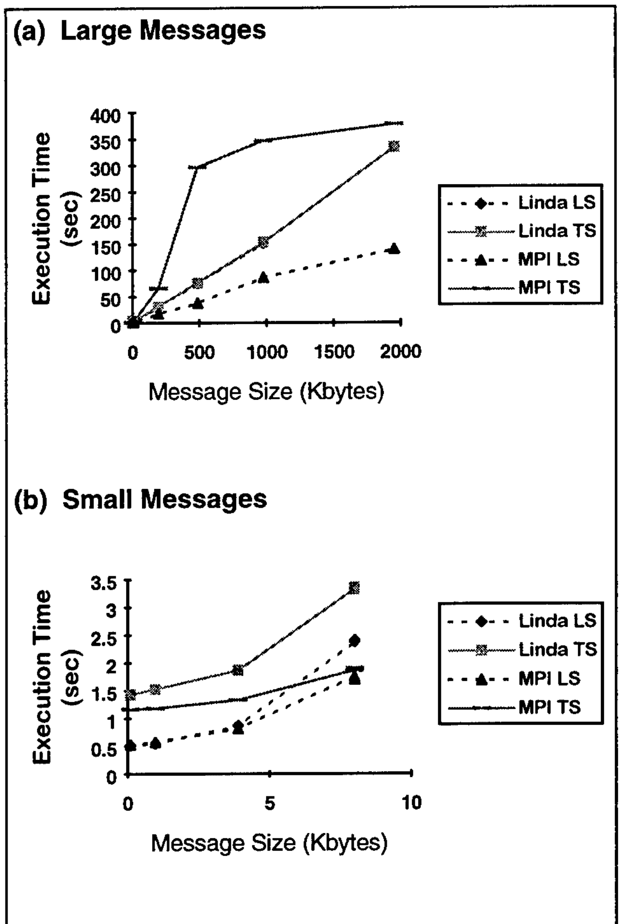


Figure 6. Tests using MPICH and Linda on an isolated, homogeneous NOW. Execution times of Loosely Synchronized (LS) and Tightly Synchronized (TS) tests using MPICH are compared to the execution times of LS and TS tests using Linda.

For small messages (from 100 bytes to 8 Kbytes), as shown in Figure 6b, the execution times for the tightly synchronized tests were longer than the execution times for the loosely synchronized tests. This can be attributed to the communication overhead in the tightly synchronized tests and is consistent with the general behavior of loosely and tightly synchronized implementations. As in the tests using MPICH, a change in bandwidth is noticed around 8 Kbytes for tests using Linda. This could have resulted from a change in Linda's internal mechanism, such as different handling procedures for small and large messages or the discrete behavior of a lower-level communication layer (or a combination of both).

Figure 6 shows that the execution times for the loosely and tightly synchronized test programs using Linda were nearly the same for larger messages (from 500 Kbytes to 2 Mbytes). The differences can barely be noticed from the graph. These results are situated between the performance results of the loosely and tightly synchronized test programs using MPICH. It can be concluded that Linda smoothes out data bursts in such a manner that execution times for tightly and loosely synchronized tests are comparable, especially for larger messages. Also, note that test programs using Linda and test programs using MPICH are comparable, but not identical. This could have impacted the behavior of the loosely and tightly synchronized models, as well as the results.

5.0 Portability

As mentioned in section 1.0, applications can be developed and fine tuned on NOW and subsequently ported to other parallel environments. MPI defines the user interface and functionality for a wide range of message-passing capabilities. Although the MPI environment management may vary, the functionality, semantics, and syntax are the same for all implementations. MPI supplies a portable and efficient method for process communication, providing flexibility and control to the application developer. By using a standard method of message passing, the burden of low-level communication is lifted from the application developer. Programs based on both the Single Program Multiple Data (SPMD) paradigm and Multiple Program Multiple Data (MPMD) paradigm are supported, although the startup procedure and configuration may vary for different MPI implementations.

MPICH supports a wide range of systems, from NOWs to shared-memory and distributed-memory systems (such as IBM, SGI, HP, and CRAY products). Other MPI libraries are available in the public domain (LAM, CHIMP). Platform-specific MPI implementations are also available and provide the same functionality with better performance levels, since they are written and tuned for a particular platform.

Linda is available on a large number of parallel computing systems, including NOW, shared-memory computers, and distributed-memory computers, such as IBM, CRAY, HP, and SGI products. Most applications run on different machines without any source code changes.

Very little investigation was conducted on porting MPICH programs to other MPI environments. (Some programs were executed in a LAM environment.) The study's emphasis was on porting test programs from NOW to shared-memory multiprocessors, using MPICH in both cases. The test programs using Linda were also ported to the shared-memory multiprocessor platform. The following section discusses a few portability issues, including changes in data transfer performance.

5.1 Porting test programs using MPICH to an SMP

The test programs were executed in both NOW and SMP environments. To obtain optimum results on the SMP, only the SPMD paradigm was used. Currently, MPICH does not have a shared memory-based implementation for MPMD. MPMD parallel programs can be executed on an SMP, but the performance is poor.

Figure 7a shows that there is little difference in the execution times for the loosely and tightly synchronized test results on the SMP, whereas there is significant difference between loosely and tightly synchronized tests on the NOWs. It appears that SMPs handle data bursts more effectively. The reasons for this behavior include a high bus bandwidth per processor and the implementation of the message passing through a shared-memory library. In a network, data bursts can cause packets to be dropped and, thus, trigger delays.

For message sizes above 500 Kbytes, the tightly synchronized tests had execution times that were four to six times longer than those of the loosely synchronized tests. For the loosely synchronized model, the total application bandwidth was approximately 11.4 Mbytes per second (isolated homogeneous NOW) versus 28 to 40 Mbytes per second (SMP) for large messages. When comparing the results on NOWs to those on the SMP, one should also consider that the SMP had processors with a clock rate two times greater than that of the workstations (200 MHz compared to 100 MHz on workstations). In addition, the memory available to each processor was more than three times greater on the SMP.

Figure 7b shows that on the SMP, for small messages, test programs had an execution time proportional to the message size, while the isolated network did not. When message size exceeded 4 Kbytes, the isolated network's loosely and tightly synchronized slope doubled on the graphs. This could be due to the discrete behavior of low-level communication software (dependent on the message size). For small messages, the loosely synchronized model yielded better performance results than the tightly synchronized model, similar to the behavior on the NOWs.

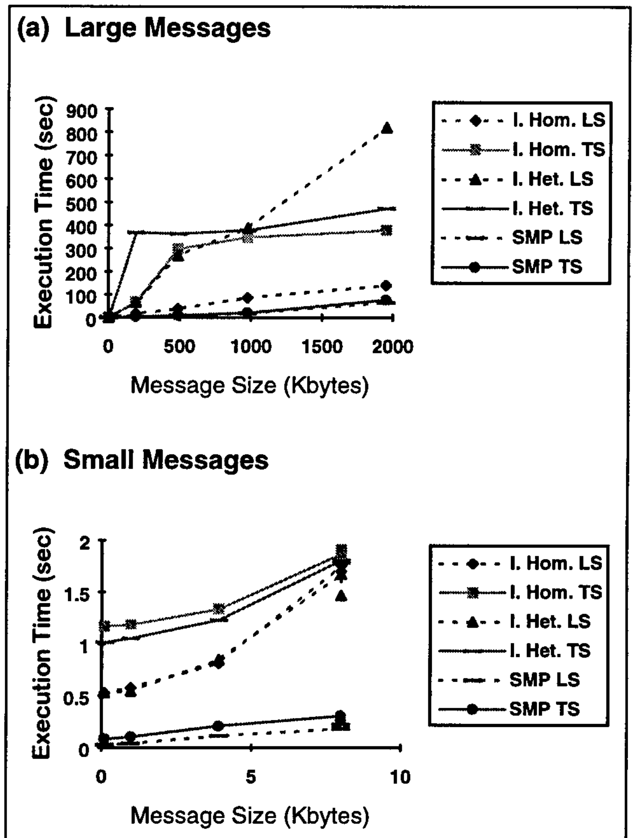


Figure 7. Tests using MPICH on the SMP and NOWs. Execution times of Loosely Synchronized (LS) and Tightly Synchronized (TS) tests using MPICH on an SMP are compared to the execution times of LS and TS tests on an isolated NOW in homogeneous (I. Hom.) and heterogeneous (I. Het.) environments.

It is evident that testing conducted on the SMP resulted in better data exchange performance compared to testing on the NOWs (Figure 7). Because data exchange on an SMP does not require network access, SMP transfers should be faster, as long as there is sufficient processing power to handle both data processing and communication. The data exchange performance on the SMP and NOW may change if processing or simulation is included in the experiment.

5.2 Porting test programs using Linda to an SMP environment

The loosely and tightly synchronized test programs using Linda were ported easily from a NOW to an SMP environment. It was only necessary to recompile the source code. A comparison of results on the isolated, homogeneous NOW and SMP follows.

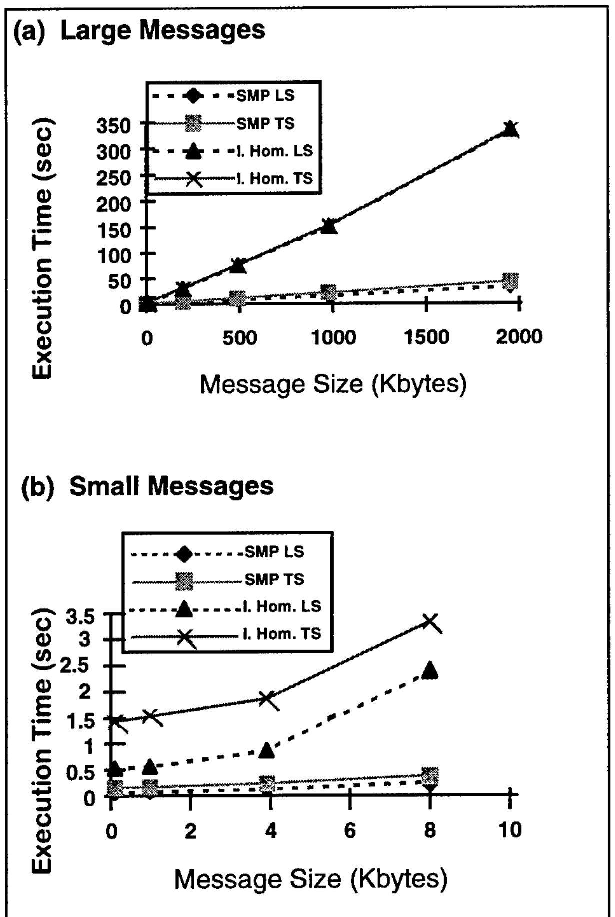


Figure 8. Tests using Linda on the SMP and isolated, homogeneous NOW. Execution times of Loosely Synchronized (LS) and Tightly Synchronized (TS) tests using Linda on the SMP are compared to the execution times on an isolated, homogeneous NOW (I. Hom. LS and I. Hom. TS).

Figure 8a shows very little difference between loosely and tightly synchronized tests on the SMP. Most comments on tests using MPICH with regard to NOW versus SMP execution times apply to tests using Linda for both large and small messages (Figure 8b); however, for large messages, the results of tests using Linda show an almost linear increase in execution time with the message size for both environments discussed (isolated NOW and SMP). This is not the case for tests using MPICH, especially on the NOW (Figure 6a).

6.0 Conclusions

The tests presented in this paper targeted performance and portability in three environments—the isolated NOW, active NOW, and SMP. Most of the tests used four slave processors and one master processor. Various message sizes were tested, from 100 bytes to 2 Mbytes. Fine tuning of the environments to improve performance was not part of the scope of this study. Instead, design aspects, mainly related to inter-processor communication, were investigated. These aspects need to be considered when developing parallel applications. Overall results are reviewed in this section.

Environment. As expected, when a limited number of processors (four in these tests) were used, the tests yielded better performance results on the SMP than on the NOWs. On the NOWs, environment control (workstations and network) was more important than the tools and methods employed in the applications. Adapting to the environment (i.e., load balancing) would be beneficial; however, the additional communication generated by dynamic load balancing needs to be monitored.

Synchronization. In tests using MPICH on NOWs, the loosely synchronized model performed significantly better than the tightly synchronized model. In the tightly synchronized model, data bursts occurred, and certain components of the communication chain were overloaded. Thus, the preferred model for applications exhibiting an inter-processor data exchange pattern similar to that described in this paper is the loosely synchronized model. Use of the loosely synchronized model in MPICH implementations, especially on homogeneous NOWs, is essential for good performance. The execution time was three to five times longer for tightly synchronized implementations with large messages and could be up to two times longer for very small messages. For small messages, the overhead caused the difference in performance, while for large messages, the difference was due to burst transfers that were not handled efficiently by the network. Implementations using Linda did not show important differences between the two models, except in the case of small messages. On the SMP, the loosely synchronized tests (using MPICH or Linda) yielded better results, also; however, the relative gap between the results of the two models was smaller, especially for large messages.

Portability and Programming Paradigm. In general, coarse-grained MPI and Linda parallel programs can be ported among parallel environments. The SMP implementation of MPICH, however, did not fully support the MPMD programming paradigm. MPICH could be configured to accept MPMD, but it did not use the shared-memory library. For a portable implementation that preserves good performance, an SPMD approach should be used.

Bandwidth. The isolated network in a homogeneous configuration supported a maximum application bandwidth of 11.4 Mbytes per second when test programs used the loosely synchronized model. For the tightly synchronized model, the bandwidth was less than 4.3 Mbytes per second. Bandwidth measurements in tests using MPICH, when only one sender and one receiver were active, showed an application performance of 6 Mbytes per second. These measurements confirm the conclusion recommending loose synchronization for optimum performance. The number of processors involved, network configuration, message size, and communication load of each processor and network switch are also important points to be considered. Tests on the SMP, using either MPICH or Linda, yielded an application bandwidth up to 40 Mbytes per second.

Message Size. All the tests yielded low performance results for small messages. This behavior is attributed to synchronization overhead, communication latency, and non-homogeneity. For larger messages, the bandwidth increased considerably. Tests using MPICH on the SMP, however, displayed a decrease in performance for messages larger than 500 Kbytes (from 40 Mbytes per second to 28 Mbytes per second).

Heterogeneity. The performance in a heterogeneous environment was notably lower than the performance in a homogeneous one. Additional measurement, without MPICH, exhibited smaller throughput on a connection between workstations of a different type, than on a connection between two workstations of the same type. In addition, a significant difference was noticed between the two directions of communication. The conclusion is that each platform's communication layers are optimized for best communication in a homogeneous configuration. This behavior is consistent with the results of tests using MPICH.

Scalability. The application bandwidth supported by the network increased with the number of processors involved in the parallel execution (from two to four processors). It is expected that, with the appropriate configuration, an ATM network can support additional processors efficiently.

References

[1] V. Morariu, "Manufacturing Process Simulation on a Cluster of Workstations," NASEC Technical Report, pp.3, 6-10, Concurrent Technologies Corporation, June 15, 1994.

[2] W. Gropp and E. Lusk, "Installation Guide to MPICH, a Portable Implementation of MPI," Argonne National Laboratory, ANL/MCS-TM-ANL-96/5

[3] W. Gropp and E. Lusk, "User's Guide for MPICH, a Portable Implementation of MPI," Argonne National Laboratory, ANL/MCS-TM-000

[4] Linda User's Guidė and Reference Manual: v 3.0, Scientific Computing Associates Inc., 1995.

[5] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, J. Dongarra, "MPI: The Complete Reference," The MIT Press, Cambridge, Massachusetts, 1996

[6] M. Cunningham and B. Wechtenhiser, "Achievable System and Network Performance," NASEC Technical Report, Concurrent Technologies Corporation, 1996

Biographies

Viorel Morariu is a Senior Software Engineer at Concurrent Technologies Corporation, where he develops high-performance computing applications and tools. Previously, Mr. Morariu held Research Staff and Principal Research Staff positions at the Institute of Automation, Bucharest. His interests include applied parallel and distributed computing, parallel algorithms and performance, domain decomposition methods, and real-time multiprocessor applications. Mr. Morariu holds an M.S. degree in Electrical Engineering from Bucharest Polytechnic Institute. He is a member of the IEEE and IEEE Computer Society.
(E-mail address: morariu@ctc.com)

Matthew Cunningham is a Software Engineer at Concurrent Technologies Corporation, where he develops network communication tools and network-intensive software applications. His interests include high-speed networking and communication analysis. He is a member of the IEEE and IEEE Computer Society.
(E-mail address: cunning@ctc.com)

Mark Letterman is a Computer Engineer at Concurrent Technologies Corporation where he works on the High-Performance Computing project. He received a B.S. degree in Electrical Engineering Technology from Rochester Institute of Technology, Rochester New York.
(E-mail address: letterma@ctc.com)

Open Discussion

How Do We Know How Well We Are Doing?

Chair:
Andrew Grimshaw
University of Virginia, Charlottesville, VA, USA

An open discussion of how we can best evaluate and compare heterogeneous computing techniques and tools, covering topics such as benchmark applications, benchmark systems, performance metrics, and quality of service issues. The community needs to establish guidelines for analyzing the goodness of our own work - what should these guidelines be?

Author Index

Notes

IEEE COMPUTER SOCIETY

http://computer.org

Press Activities Board

Vice President:
I. Mark Haas
Managing Partner
Haas Associates
P.O. Box 451177
Garland, TX 75045-1177
m.haas@computer.org

Jon T. Butler, Naval Postgraduate School
James J. Farrell III, Motorola
Mohamed E. Fayad, University of Nevada
I. Mark Haas, Haas Associates
Ronald G. Hoelzeman, University of Pittsburgh
Gene F. Hoffnagle, IBM Corporation
John R. Nicol, GTE Laboratories
Yale N. Patt, University of Michigan
Benjamin W. Wah, University of Illinois
Ronald D. Williams, University of Virginia

Editor-in-Chief
Advances in Computer Science and Engineering Board
Pradip Srimani
Colorado State University
Dept. of Computer Science
601 South Hows Lane
Fort Collins, CO 80525
Phone: 970-491-5862 FAX: 970-491-2466
srimani@cs.colostate.edu

Editor-in-Chief
Practices for Computer Science and Engineering Board
Mohamed E. Fayad
Computer Science, MS/171
Bldg. LME, Room 308
University of Nevada
Reno, NV 89557
Phone: 702-784-4356 FAX: 702-784-1833
fayad@cs.unr.edu

IEEE Computer Society Executive Staff
T. Michael Elliott, Executive Director
Matthew S. Loeb, Publisher

IEEE Computer Society Press Publications

The world-renowned Computer Society Press publishes, promotes, and distributes a wide variety of authoritative computer science and engineering texts. These books are available in two formats: 100 percent original material by authors preeminent in their field who focus on relevant topics and cutting-edge research, and reprint collections consisting of carefully selected groups of previously published papers with accompanying original introductory and explanatory text.

Submission of proposals: For guidelines and information on CS Press books, send e-mail to cs.books@computer.org or write to the Acquisitions Editor, IEEE Computer Society Press, P.O. Box 3014, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720-1314. Telephone +1 714-821-8380. FAX +1 714-761-1784.

IEEE Computer Society Press Proceedings

The Computer Society Press also produces and actively promotes the proceedings of more than 130 acclaimed international conferences each year in multimedia formats that include hard and softcover books, CD-ROMs, videos, and on-line publications.

For information on CS Press proceedings, send e-mail to cs.books@computer.org or write to Proceedings, IEEE Computer Society Press, P.O. Box 3014, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720-1314. Telephone +1 714-821-8380. FAX +1 714-761-1784.

Additional information regarding the Computer Society, conferences and proceedings, CD-ROMs, videos, and books can also be accessed from our web site at www.computer.org.

12/12/96