Parallel Programming with CHARM: An Overview

L. V. Kale
Department of Computer Science,
University of Illinois, Urbana-Champaign.
email : kale@cs.uiuc.edu
Phone : (217) 244-0094, Email: kale@cs.uiuc.edu

Abstract

This Paper describes the research centered on the Charm Parallel Programming System. Charm is a portable parallel programming system under development at the University of Illinois for the past six years. The system enhances latency tolerance via message driven execution, supports dynamic load balancing for medium grain tasks, and provides innovative language features such as specifically shared variables and "branch office" processes. The paper describes the philosophy behind the system, and elaborates on its features and their motivation. Completed and ongoing related projects are summarized, including: *Dagger*, a textual notation and a corresponding visual language for simplifying expression of split-phase communication that is necessary in message driven execution; A subset of the High Performance Fortran that demonstrates superior latency tolerance; Performance feedback and debugging tools that exploit the specificity of information available to the runtime system to give the user a highly refined and usable feedback; strategies for parallel execution of speculatively parallel computations; and a summary of preliminary applications.

1 Introduction

A very high amount of raw computational power is now becoming available commercially in the form of high performance parallel computers. Fueled by advances in microprocessor technology and interconnection networks over the last decade, this trend is likely to continue toward realizing the milestone of "teraFlop speed" by mid decade.

At the same time, developments in computational science and engineering, as well as business information processing have resulted in increased demand for computational power. Significant advances in our understanding of nature, and in our ability to design better products are possible with substantial computational power, which is either available or just around the corner. Computational and molecular biology is likely to have a very significant impact on the society in the next few decades, comparable to the impact of physics on this century.

However, before this perfect marriage can be consummated, a significant hurdle in the form of the difficulty of software development must be overcome. Despite claims to the contrary from some vendors, and a few computer scientists, parallel programming remains a relatively difficult task. Some of the reasons for this difficulty are:

1. *Lack of Portability:* Programs written for one machine cannot be run directly on another machine. This holds true even for machines with similar architectures. There are classes among MIMD machines such as shared memory machines, and non shared memory machines with different programming models. Even two distributed memory MIMD machines (of different vendors) do not allow portability of user programs. [1] This makes it very difficult for a commercial software developer to justify the investment in a parallel program - as its use is limited to only the installed base of a particular machine, which is quite fickle. [2]

2. *Inherent complexity:* Parallel programming is more difficult than sequential programming. This is evidently so, because in addition to the usual difficulties of sequential programming, parallel programming involves dealing with issues of asynchrony, communication latency, load balance, and scheduling.

3. *Dealing with communication latency :* Designing efficient parallel programs is made complicated by the fact that remote data is much slower to access than local data. The architectural advances make this ratio even worse, because the processors are speeded up at a faster rate than the communication latencies. Thus, this problem can be expected to get worse over time. In addition, the computation of the response by the remote processor may get unpredictably delayed due to a variety of factors, including other parts of the computation running on that processor.

4. *Irregular problems:* Many of the early successes of parallel processing were on relatively regular applications and kernels. In contrast, many applications involve irregular computations. The irregularities may arise from

   (a) the structure of the data itself, as in sparse matrix computations arising in finite element formulations applied to structures such as airplane wings;

   (b) time varying nature of the computation - as in adaptive grids, particles-in-a-cell simulations, tree-codes for the n-body problem, discrete event simulations, etc.

   (c) The need for dynamic load balancing and prioritization arising in heuristic search and optimization problems, such as those in operations research, VLSI CAD, AI, protein folding etc.

5. *Difficulty of Reuse:* Parallel software is notoriously difficult to reuse, due to various sources of interactions among modules that are difficult to foresee. Some of these interactions, such as those arising out of interleaving of messages can be eliminated or reduced at the cost of efficiency by resorting to strict transfer-of-control protocols across modules - one module must finish its work on all processors before transferring control to another. However, even such a drastic measure does not eliminate all sources of unwanted interactions. A more thorough and clean approach to compositionality and modularity is needed.

The Charm parallel programming system [11, 24, 13] has been developed as a response to these concerns, at University of Illinois at Urbana Champaign over the past several years (1986-1993). Charm is a C based portable parallel programming system - programs written with Charm

[1] The emerging MPI standard may alleviate some of these concerns.

[2] Portability is now recognized as a desirable objective; However when this project was begun in 1986, it was generally scoffed at because of the belief that significant performance loss will be incurred to achieve portability. The necessity of portability for software development has led to the appreciation of portability as a desirable feature, yet efficiency is still an important factor to keep in mind while aiming at portability.

run efficiently without change on shared and distributed memory machines. The set of machine currently supported includes iPSC/860, Paragon, CM-5, NCUBE, networks of workstations, and many shared memory machines. It helps control complexity of parallel programming by providing a set of useful abstractions, and by freeing the users from the details of parallel processing. Its message driven execution style allows one to tolerate communication latencies, and unpredictabilities in other processor's response times. The system allows dynamic creation of work, and supports it with a suite of user-selectable dynamic load balancing strategies. It allows associating priorities with messages and tasks, supported by prioritized scheduling and load balancing strategies [25]. Finally, it supports a sophisticated module system that allows one to reuse modules (even separately compiled ones) effectively without loss of efficiency.

This paper provides an overview of Charm, and related research projects. We begin by summarizing the core language supported by Charm. Two novel features in the system - specifically shared variables and branch office chares - are described in section 3, while the support for reuse is described in Section 4. The performance benefits of message driven execution are explicitly demonstrated in Section 5. The split phase transactions necessitated by the message driven execution sometimes obscures the flow of control within a single multi-threaded process. Dagger is a small language on top of CHARM that deals with this problem. Dagger is developed both as a textual and visual language and is described in Section 6. The programming concepts embodied in CHARM are independent of the base language. Implementation of CHARM with C++ as a base language is described in section 7. Section 8 describes performance feed-back and debugging tools being developed for CHARM and explains what advantages they offer over contemporary tools. Other higher level languages can be built on top of CHARM. Our current efforts in this direction include implementation of High Performance Fortran and Parallel Prolog, as described in section 9. One of the computer science application areas where we have a significant body of results is speculatively parallel computations, which are discussed in Section 9. These include search and combinatorial optimization problems. Section 10 describes some additional applications developed using CHARM which is followed by a summary.

2 Charm Core Language: Chares and Messages

A Charm programmer specifies a parallel computation in terms of processes (called chares) and messages. A message is a structure consisting of various data fields and is defined similarly to the structure definition in C. A chare is a potentially medium grained, message driven, process. On each processor, at any given time there may exist thousands of chares, or just one chare, depending on the application.

A chare definition consists of local variables and a set of entry point functions. It may also include private functions. The entry point functions are just like C functions except that they take only one parameter which must be a pointer to a message. The code inside these functions may access the local variables of the chare that they are part of. Every program must include a *main* chare which must have an entry point named *CharmInit*. The execution of the program begins by creation of one instance of the main chare and the execution of its *CharmInit* entry point. Figure 1 shows an example of a simple Charm program. The *main* chare here creates n instances of the *compute* chare, which calculates a value based on the seed given to it and sends a message to the *result* entry point of the main chare. The *CreateChare* call simply deposits a seed for a new chare with the system; the dynamic load balancing strategy provided in the system will

```
message { int seed; ChareIdType parent;} DownMsg;
message { int value;} UpMsg;

chare main {
int n, total;
    entry CharmInit: {
        DownMsg *m;
        CkScanf("%d",&n);
        for(i=0; i<n; i++) {
            m = (DownMsg *)CkAllocMsg(DownMsg);
            m->seed = i;
            MyChareId(&(m->parentId));
            CreateChare(compute, compute@start, m);
        }
    }
    entry Result: (message UpMsg *result) {
        total += result->value;
        CkFreeMsg(result);
        if (--n ==0) { CkPrintf("The final Total is: %d\n", total); CkExit();}
    }
}

chare compute {
    entry start: (message DownMsg *m) {
        up = (UpMsg *) CkAllocMsg(UpMsg);
        up->value = calculate(m-¿seed);
        SendMsg(main@Result, up, m->parent);
        CkFreeMsg(m);
        ChareExit();
    }
}
```

Figure 1: A Simple Charm Program

eventually decide the processor on which the chare instance corresponding to this seed is to be created. The *ChareExit()* call signals termination of the chare making the call whereas *CkExit()* signifies termination of the entire program.

From the programmers point of view the system operates with a pool of messages. These include messages specifying creation of new chares as well as messages for existing chares. Each processor picks up a message from this pool, executes the entry point function indicated by it, possibly modifying the local variables of the associated chare instance and depositing new messages in the systems pool, and then returns to pick another message from the pool. Two messages directed to the same chare are not concurrently executed. Also, the execution on an entry function on a processor is not interrupted to execute another entry function on that processor. The *SendMsg* call as well as all other system calls are non-blocking (i.e. they do not wait for a remote event). It is worth emphasising that although Charm provides *send* calls it does not provide any *receive* calls. Instead, the execution is driven by the existence of arrived messages.

The arrived messages are scheduled according to a scheduling strategy. The scheduling strategy and dynamic load balancing strategy are modularly separable components of the Charm runtime system – called the Chare Kernel – which can be substituted at will by the user. The system provides FIFO, LIFO, and priority-based scheduling strategies, with unbounded levels of priorities. Similarly, it provides a variety of dynamic load balancing strategies developed over the years.

3 Enhancing Expressiveness

The language defined so far might not appear novel. The reader might compare the chares to actors[1], processes, monitors, concurrent objects[4, 27, 3], etc. with each of which it shares some similarities and differences. Even at this level, Charm probably was one of the first such systems implemented in portable manner [11] with support for dynamic load balancing. However, our significant point of departure comes with the features we describe next: Information sharing abstractions and branch office chares.

3.1 Specifically Shared Variables

The language described so far allows dynamically created processes to send messages to each other. However, message passing as the sole information exchange mechanism is not adequately expressive or efficient, and nor is any other single generic mechanism. Instead, we believe that processes should be allowed to exchange information via multiple alternative information sharing modes. The task then is to identify commonly used modes, to define abstractions for them, and then to implement them, possibly differently, on different parallel machines [20]. The conceptual model of a Charm computation, with chares that send messages to each other, create new instances, and share information via specifically shared variables is shown in Figure 2.

Charm supports the following abstractions in addition to messages.

1. *Read only variables.* These are the variables whose values are fixed at the beginning of

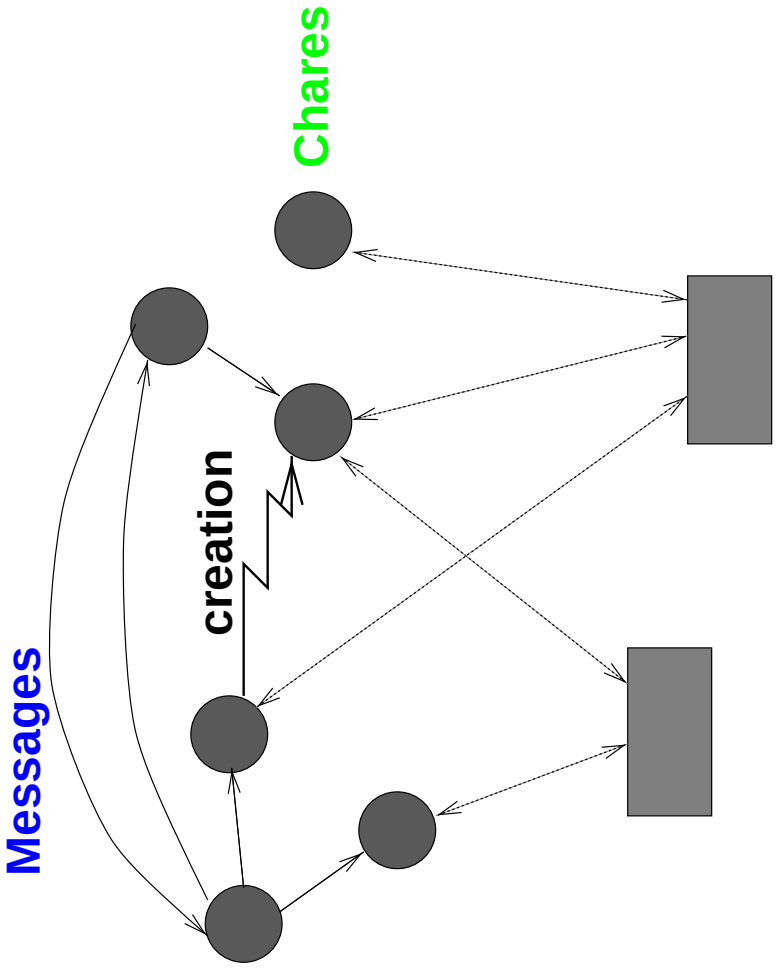

Figure 2: Chares and Information Sharing Abstractions

the computation. The chares on any processor can access this value in-line—at the cost of a local variable access. The system may implement these with a single shared copy or a private copy on individual processors. Read only variables are the only global variables permitted in Charm. There also exists a dynamically creatable version of read only variables for information that is created in the middle of a parallel computation but remains frozen after its creation.

2. *Distributed tables.* A distributed table is a collection of [key, data] pairs. It supports *insert*, *find*, and *delete* operations. As the name suggests, the pairs may be distributed across processors. In keeping with the message-driven nature of Charm, the call to find the data corresponding to a given key, for example, deposits a request to send this data to a named chare chare_id at a named entry point entry_point, as shown below.

   find (table_name, key, chare_id, entry_point);

   Distributed tables subsume the functionality of tuple spaces [20], but are nonblocking. They are also related to the *I*-structures[2] of the data flow languages.

3. *The accumulator.* The accumulator is a shared data structure with a commutative and associative operator. The users are free to define their own data structures and operators as long as they satisfy this requirement. In addition users must provide a function for combining multiple copies of the accumulators. The only functions allowed are *add* and a final, destructive, *collectValue*. These variables are useful for maintaining global totals, histograms and accumulated sets. The system may implement the accumulators with a single shared copy or with multiple copies that are combined at the end.

4. *The monotonic variables.* Values of these variables changes monotonically in only one direction. An access function allows any chare to find the local estimate of this variables value while another function changes this value monotonically and in an idempotent manner (that is applying the same function repeatedly yields the same result as applying it once). These variables have a narrow, specific, but important use for maintaining the cost of the best solution in branch and bound computations and in distributed simulations for maintaining the global maximum time.

Over the past years we have used these mechanisms in numerous applications and found them to be adequate, particularly when combined with the branch office chares, which also act as a catch-all information sharing mechanism.

3.2 Branch Office Chares

A branch office chare (BOC) is a replicated chare with a branch on every processor. All the branches answer to the same name. In addition to private functions and entry functions for handling messages, the BOCs also include *public* functions. The entry functions are required to accept only a pointer to a message as its single paramenter, whereas public and private functions are not similarly restricted. Thus, a dynamically load balanced chare which lands on some processor can make a function call to a BOC and this call will always be handled by the local branch of the BOC without the calling chare having to know which processor it is on. One can send a message to a particular branch (on a particular processor) of an instance of a BOC or also broadcast a

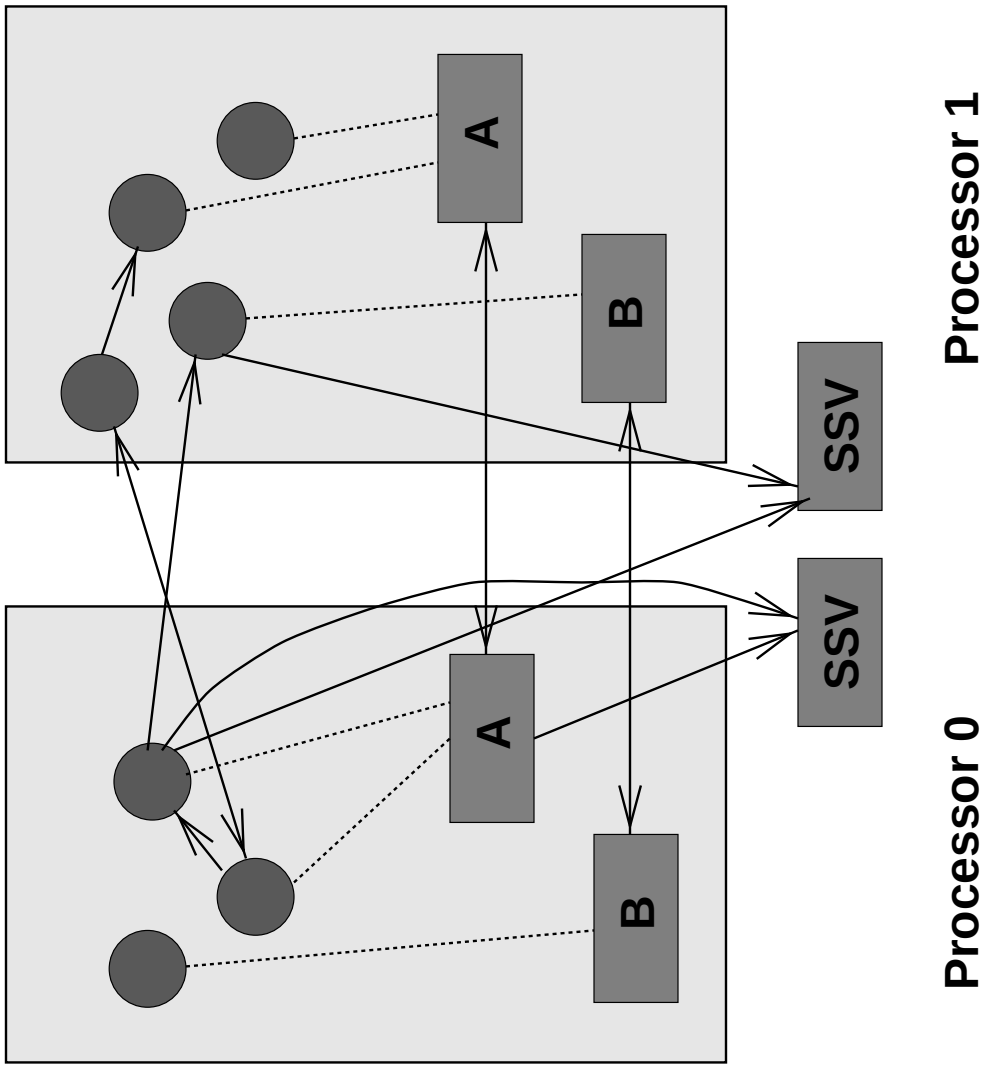

Figure 3: Chares and Branch Office Chares. A and B are branch office chare instances

message to all its branches. The expanded conceptual model for the Charm programmer is shown in Figure 3.

As they are replicated chares, branch office chares can be used to implement static load balancing. More importantly, they are useful for implementing distributed data structures, implementing local services (e.g. memory management, or local "caches"), providing mechanisms for intermodule interfaces, and simply for improving the structure of the program. This last advantage is similar to that of coroutines and threads in that it allows one to separate independent threads of control in the program without sacrificing the interleaved execution of such threads. A more thorough description of branch office chares can be found elsewhere [15]. Here we will give a simple example.

Consider the program of Figure 1. Suppose we now also want to record each solution obtained by the calculate function. In addition, we desire to record these solutions in separate files such that solutions in the range $0..M/P$ are stored in the first file and so on, where M is the maximum value

```
BranchOffice store {
        FILE *fp;
        entry init : (message FileName *msg) {
                char name[256] name;
                sprintf(name,"%s%d",msg->name,McMyPeNum());
                fp = fopen(name,"w");
        }
        entry writeToFile : (message UpMsg *msg) {
                fprintf(fp,"%d\n",msg->value);
        }
        public deposit(value)
        int value;
        { int penum;
                UpMsg *msg;
                penum = value*McMaxPeNum()/range;
                msg = (UpMsg *)CkAllocMsg(UpMsg);
                msg->value = value;
                SendMsgBranch(store@writeToFile,msg,penum,MyBocNum());
        }
}
```

Figure 4: BOC Example

that any solution may have and P is the number of processors. We choose to have each processor be responsible for storing one file. This strategy can be implemented with a branch office chare (figure 4). An instance of the BOC store can be created by a CreateBoc() call in the CharmInit() entry point of the main chare in figure 1. The store BOC provides a public function `deposit` for accepting a solution. This function computes the processor responsible for storing this solution and sends a message to the branch on that processor. The SendMsgBranch() call identifies the destination branch by specifying the BOC instance id (obtained here by the MyBocNum() call) and the processor number in addition to specifying the entry point and message as in the SendMsg() call. The receiving entry point for this message simply stores the message in the file. The compute chare can call the `deposit` function of the BOC (which is handled by the local branch of the BOC instance that was created by CreateBoc()) to record a solution right after sending a message to the main chare.

4 Improving Productivity in Parallel Programming

To be able to support reuse of parallel software components, the following objectives must be met:

- *Distributed intermodule interfaces :* Modules should be able to exchange data in a distributed fashion.

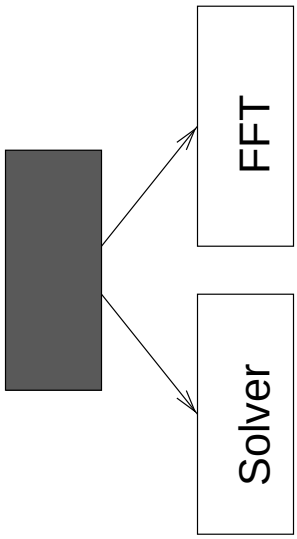

Figure 5: Concurrent Invocation of Modules

- *Flexible interfaces :* A module should be re-usable in different contexts, even when its client/s may employ different data distributions for the data that is input to entities in these modules.
- *Ability to reuse object-code modules :* For practical promotion of re-use, it essential to protect the proprietary nature of modules developed by different software developers. It is therefore desirable to be able to use modules which are distributed in object-code format only.

All of the above must be accomplished without sacrificing efficiency, and in particularly without giving up the advantages of message driven execution.

A Charm program is organized as a set of modules, one module per file. Associated with each module is an interface statement that defines the objects that are being exported from that module. Message types, regular C type definitions, function definitions, as well as names of chares, branch office chares and entry points can be exported. A module wishing to import another module simply includes this interface statement. It then refers to entities in the imported module via the scope resolution operator "::". The call `SendMsg(cid, m::c@ep, msg)` sends a message to the named chare instance which is defined in module `m`.

This static interface is not always adequate. When module M1 is written prior to a client module M2, M2 may import M1, but M1 (which might be a library module meant to interact with many possible clients) cannot interact statically with entities in M2. Say M2 wishes to invoke an entity in M1 and have M1 invoke an entity in M2 possibly at a different processor. This is possible in Charm because names of such entities (function names, entry point names and chare names) can be passed as variables and fields of messages in such a manner that they are valid across processors. Note that this is not as trivial as passing a pointer in sequential modules because the pointers are invalid across processors.

As branches of a boc in one module can interact with branches of another boc in a second module, distributed exchange of data across modules is possible. In addition, distributed tables provide another mechanism for distributed exchange; entities in a module may scatter the data they compute into a distributed table while entities in the second module may fetch the data from this table. The bi-directional data exchange across modules permits flexible intermodule interfaces that are independent of data distributions. For example, a matrix multiply module may be invoked by a client. This module, during its parallel computation, may request the client on any arbitrary processor, via a branch call, to provide specific rows of a matrix. The client module which knows about the distribution of the matrix, may then collect this set of rows and send them back to the requestor. Thus the multiplication module can be reused even when the client matrices are distributed differently.

The modularity features combined with message driven execution gives Charm a much cleaner ability to compose modules into larger programs without sacrificing performance. For example, consider the situation depicted in figure 5 , involving a replicated (SPMD-style) computation. On every processor the program is ready to invoke a parallel FFT operation which requires communication. It is also ready to start a parallel solver operation. Assume that these two operations are independent. One would like to resume work on the solver while FFT is waiting for messages and vice-versa. This is not possible in the standard SPMD model, because when one is blocked in the FFT module one is waiting for the FFT message only. One can use a "wild card" receive function to accomplish this. Now, however, when the FFT module receives a message meant for the solver, it must invoke the appropriate function in the solver. This destroys modularity, as the FFT module must know about the existence and the structure of the solver module and vice-versa. With Charm both the modules are invoked concurrently and any message that arrives is delivered automatically to the appropriate module by the system.

The module system is a major strength of Charm. It is being leveraged with development of a large set of library modules. The modules being developed include simple functions such as global reductions to complex functionalities such as linear systems solvers, etc.

5 Performance Advantages of Message-Driven Execution

The performance benefits of message driven execution can be illustrated with a simple computational experiment. We programmed a concurrent reduction program in Charm, and in C with blocking calls, to run on the NCUBE/2. Each processor has an array which is partitioned into blocks. A reduction (maximum) operation is performed for each block after the block is computed locally. The reductions are independent of each other. The blocking-receive version calls the NCUBE/2 system library function *nrmaxn* to perform reduction. In Charm, the reduction is performed through a spanning tree of processors, asynchronously.

Figure 6-a shows the execution times for the reduction program for the C and Charm versions. The amount of local computation per block is fixed for this example. The cost of the blocking version is of the order of $\log p$, where p is the number of processors.

Figure 6-b shows the execution times for the reduction programs, when the amount of local computation has a significant variance across processors. Again, the message driven Charm program lets each processor execute a computation as soon as it has asynchronously started the previous reduction, and so wins by a substantial margin.

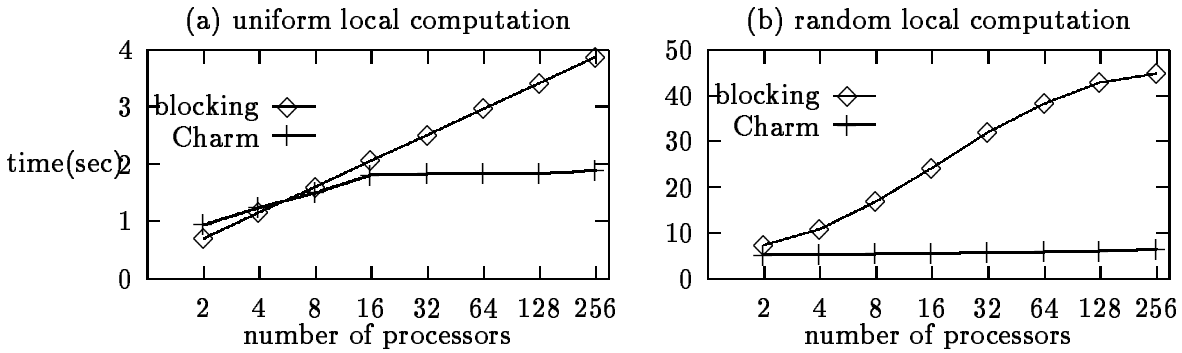

Figure 6: Concurrent Reductions

6 Dagger: Controlling the complexity of message driven execution

Although message driven execution improves performance and modularity, its expression in a program may sometimes get cumbersome. Such programming has some similarities to event based programming used for X-windows or Microsoft windows programs, but is more complex due to dependencies among messages and the nature of this asynchrony. This sometimes leads to bugs due to a message processing sequence that is different than those anticipated by the user program. We have developed a notation called *Dagger* [7], and an associated graphical program editor that helps programmers deal with this complexity.

Consider the graph shown in Figure 7. The rectangles in this graph represent subcomputations and circles represent messages. Note that all the subcomputations are part of a single process executing on a single processor. Arrows from messages to subcomputations represent dependencies while arrows to messages represent readiness to receive messages. Double circles represent entry points which may receive multiple messages. The arrow from inner circle point to computations that are performed when any one message to that entry point arrives. Arrows from the outer circle point to computations that are performed when all expected messages for this entry point have been received. Thus the process specified by this graph carries out the initial subcomputation then expects the `E1` message. When this message is received it carries out the `C1` subcomputation, and is ready to receive messages to either the `REDN` or `FFT` entry point. When all the messages to both these entry points have been received it initiates the `proceed` computation. By programming in this notation the need to specify counters, flags, and buffers to store messages is eliminated. The runtime system automatically keeps track of the dependencies and buffers messages as necessary. The Dagger programs can be expressed textually or visually. The visual editor allows user to create the dependence graphs and to fill in text for the computations and declarations of messages interactively. This program is then automatically translated to an equivalent Charm program. We find the Dagger notation to be extremely effective in expressing complex message driven programs.

The fact that Dagger codes the dependencies among computations and messages can be reused to facilitate trace driven simulation of message driven programs. Predicting performance of such programs on simulated architectures runs into following difficulty. As architecture parameters

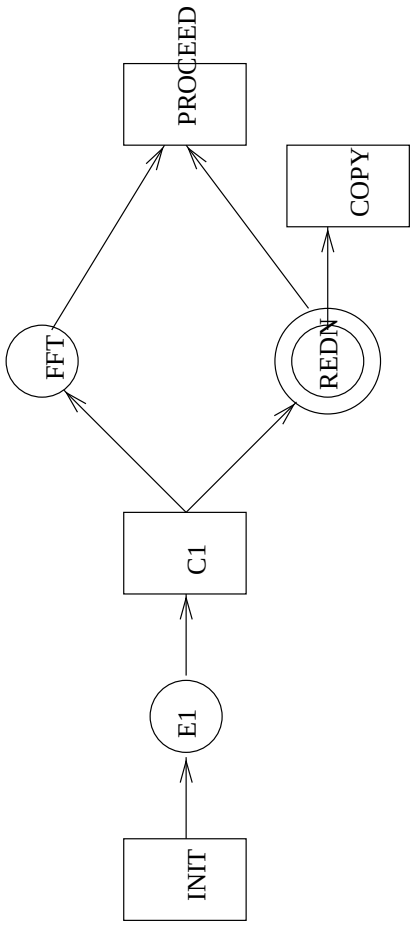

Figure 7: A Dagger Example

vary, message arrival sequence may change. As the traces were obtained with respect to a specific sequence, faithful simulation becomes difficult. By tracing individual computation blocks in a Dagger program and using the dependence information in the dag graph solves this problem. A simulator embodying these ideas has been implemented [8]. The architecture models supported by the simulator appears to subsume the LogP model proposed recently in [5].

7 Supporting Parallel Programming

In contrast to SPMD programs running on vendor supplied software, the Charm runtime system has much more specific, application level, information about the events in the parallel computation. This fact can be exploited by developing performance feedback and debugging tools that provide high-level application-specific feedback and analysis to the user. Whereas the traditional performance feedback system will only be able to say "processor X was overloaded", a tool for Charm would be able to provide reasons. It may say that processor X is overloaded due to a large number of dynamically load balanced processes on it, or that this was due to a large number of messages requesting data for a particular key from the distributed table. It may be able to go further and suggest improvements, as in "as this data in the table is not updated and is accessed from many other processors, it may be better to store it as a read only variable." One may be able to carry out more sophisticated critical path analysis and develop intelligent trace driven debugging tools.

We have only begun to explore these possibilities. A preliminary performance feedback tool called Projections [19] has been developed. The only specificity exploited in this tool is the *category* of a message. It distinguishes messages for creation of new chares, for existing chares, and for branch office chares. Even with this simple model, it was still very helpful for carrying out performance improvements in many application domains, including a prioritized dynamic load balancing for branch-and-bound computations. An even simplier tool that only provides one screen-full of information but is much less intrusive (as it involves no file I/O till the end of the program) has also been developed.

8 Other Base Languages

The basic ideas and techniques used in CHARM are independent of the base language used. We have developed a C++ based concurrent object oriented language called *CHARM++*[17], which reuses the runtime system of CHARM. This language provides a clear separation between sequential objects and parallel objects, simplifying the cost model of a computation (local as opposed to remote) for the programmer, and making parallel algorithm design easier. CHARM++ provides all the standard object oriented features for parallel objects, including multiple inheritance and reuse, dynamic binding, polymorphism and overloading. Inheritance and dynamic binding are particularly useful in the parallel context, because they can be used to interface to parallel library objects providing a service (e.g. a reduction operation). CHARM++ is unique among concurrent object oriented languages because of its support for regular, data-parallel as well as highly irregular applications, its specific information sharing abstractions, message driven execution model, and dynamic load balancing. Moreover, since it uses the CHARM runtime system, it does not depend on the operating system to provide a threads package, and also has been implemented on

many large commercial shared as well as distributed memory machines.

We also plan to develop versions of CHARM based on other sequential languages like Fortran and Scheme in the near future.

9 Higher Level Languages

Charm can be seen as a "universal back end" for parallel programming. One can write direct applications using Charm, but one can also develop other higher level languages and specialized packages on top of Charm. Two such projects, one ongoing and one older, carried out at the Parallel Programming Laboratory are described below.

DP-Charm[21] is a *data-parallel language* being developed on top of Charm. The language implements a subset of the official HPF (High Performance Fortran) language. It is an array oriented language, in which the programmer specifies the distribution of each array explicitly, as in Fortran-D [9]. It supports data parallel loops, and primitives for shifts, reductions, and other array operations. The compiler for this language takes a user program and translates it into a intermediate language of smaller primitives. It then carries out dependence analysis to generate a dependence graph between messages and primitive computations, we preprocesses this graph for optimization, and directly emits code in the Dagger notation described above. It also links in a Charm library that supports many operations such as shifts and reductions for various permitted ray distributions.

The technical objective of this project is to demonstrate the performance advantages of message driven execution for data parallel programs. The schedules for execution generated by the compiler are dynamic and adaptive in that they adapt to the runtime conditions during execution for maximal overlap of communication and computation. We expect that the code produced by this compiler will yield performance better than contemporary HPF compilers which generate SPMD style blocking received based code. Dagger also turns out to be an appropriate back-end language for this compiler, allowing us to concentrate on the compilation issues while leaving the scheduling of messages to Dagger. As DP-Charm generates Charm modules, they can be easily linked with other Charm modules. Thus one can compose programs with highly regular, data parallel, components with irregular components requiring dynamic load balancing, etc.

A parallel implementation of *Prolog*[14, 10, 22] was one of the earliest projects carried out on top of Charm. In fact, it is a progenitor of Charm—Charm originated when the runtime system for the Parallel Prolog was separated and independently developed in 1986. The computation strategy for sequential Prolog is different than that for Fortan or C. The source program is translated to an intermediate language called the byte code. A byte code interpreter is written, typically in C, to execute the generated code. The parallel Prolog compiler, thus, also generated byte code while the byte code was interpreted by a parallel Charm program. The system called ROLOG, was based on the REDUCE - OR process model for Prolog. This was one of the first models to combine AND and OR parallelism effectively. The implementation was also one the first implementations of parallel Prolog on shared and distributed memory machines and to demonstrate the performance of Prolog on machines with hundreds of processors.

Table 1: The figure shows the execution times and the number of nodes generated for executions of a 60 city asymmetric TSP on the NCUBE/2 with upto 512 processors using the tokens strategy to balance load. In this case the cluster size is 16 processors.

Processors	1 (estimated)	64	128	256	512
Time	19,366	302.6	151.1	86.2	42.1
Estimated Speedup	1	64	128	225	460
Number of Nodes expanded	-	85,165	84,030	93,816	85,420

10 Speculatively Parallel Computations

One of the major application areas within computer science, where we have obtained significant results, involves speculatively parallel computations. In these computations, some of the actions that can be initiated in parallel might be wasted or not needed for the final result. Examples of such computations include state space search, branch-and-bound, planning, problem solving, and game tree search. As was pointed out early on, a major problem in parallelizing such applications is that of anomalies. Choices made in picking the next piece of work to compute might effect the total amount of work done significantly. This manifests itself in the performance of the same program on the same data being highly inconsistent from run to run even with a fixed number of processors. When the number of processors is increased, the performance may decrease or increase even beyond the increasing number of processors. In addition the memory usage of such programs is a concern as it may vary between linear and exponential function of the depth of the search trees involved. The objective of our research here was therefore to achieve consistent (i.e. from run to run) speedups that increase monotonically with the number of processors. We developed a series of prioritization strategies[12, 16] for different application domains that help achieve this objective. We also developed prioritized load balancing schemes[23, 26] to support this. The priorities required for the these applications need to be unbounded. For example, the state space search is prioritized by attaching a bit vector priority to every node of the search tree. The length of the bit vector expands as one goes deeper into the tree. So prioritization strategies were included in Charm that allow for arbitrary length bit-vectors and hence unbounded levels of priorities. These were found to be useful in many other domains as well. The problems in these different application domains, although they share the common difficulty of speculatively parallel computations, are very distinct and need different solution strategies. However a common theme in all the solutions the use of priorities. The full extent of such efforts is described in [16]. The applications carried out using these approaches include a traveling salesperson problem, various puzzle searches, an iterative deepening algorithm, bi-directional search which can be used when both the starting state and the goal state are known, game tree search as applied to the game of Appolo, graph isomorphism, graph coloring, etc. These strategies were also used by Banerjee, Ramkumar, et al for test pattern generation for VLSI CAD, using Charm. A sample result from the traveling salesperson problem is shown in Table 1. Note that the number of nodes expanded do not increase substantially with inclusion number of processors. This is a sign of effectiveness of the prioritization strategy. The strategy used also needed to balance memory requirements across processors. Thus it achieves a triple objective of (a) balancing the load (b) adhering to the priority (c) balancing the memory requirements across all processors.

11 Applications

In addition to the AI applications mentioned above, several applications have been developed using Charm within the parallel programming laboratory and outside. Several CFD algorithms are being implemented to study the effectiveness of message driven execution for them [6]. A molecular dynamics program named EGO, originally written in OCCAM, was recently ported to Charm. A novel comparison based sorting algorithm—which can be used for floating-point numbers, strings, as well as integers has been implemented. Following successful demonstration of a branch and bound algorithm a toy problems, several applications in operations research are planned. A research group led by Prof. P. Banerjee at University of Illinois used Charm to implement several VLSI-CAD algorithms with excellent results.

As an example, the performance results obtained using Charm for a parallel sorting algorithm are shown in Tables 2, 3, 4 and 5. [18, 25]. The key set for all tables consists of integers obtained by averaging 4 sets of random integers.

Table 2: Parallel Sort Basic Timings on the CM5.

Processors	64	128
Keys	$64 * 10^6$	$128 * 10^6$
CM5 (s)	22.24	26.20

Table 3: Parallel Sort Basic Timings on the nCUBE/2.

Processors	64	128	256	512	1024	1024
Keys	2^{23}	2^{23}	2^{23}	2^{23}	2^{23}	2^{26}
nCUBE/2 (s)	12.30	6.87	3.93	2.46	2.00	9.14

Table 4: Parallel Sort Basic Timings on the iPSC/860.

Processors	64	128	128
Keys	2^{23}	2^{23}	2^{24}
iPSC/860 (s)	3.87	2.66	5.04

12 Summary

The Charm Parallel Programming system was originally designed with portability across MIMD machines—with shared memory or without—as its main objective. This objective has clearly been met as Charm programs run unchanged and efficiently on shared memory machines including sequent's symmetry, Encore Multimax, Alliant, and nonshared memory machines including N-cube, Intel, Paragon and iPSC/860, CM-5 and networks of workstations. The system supports medium-grained processes and dynamic load balancing. In fact, the system acted as a useful test-bed for a series of dynamic load-balancing strategies. Charm was one of the first systems to employ message driven execution on stock multicomputers. Message driven execution is a promising

Table 5: Parallel Sort Basic Timings on the Sequent Symmetry (a shared memory multiprocessor).

Processors	1	4	8	16
Keys	2^{21}	2^{21}	2^{21}	2^{21}
Sequent Symmetry (s)	202.1	58.6	30.4	21.8

technique for enhancing the performance of parallel programs by allowing them to tolerate communication latencies and delays in remote response time. Its advantages are especially important for applications involving multiple global operations (such as reductions) with significant variance in the amount of computation or speed across processors. We expect message driven execution to pass into common use in the near future—bolstered by theoretical work on actors and supported by OS level mechanisms such as active messages. The Dagger notation and its associated visual language will play an important part in simplifying the task of writing message driven programs. The idea that data is shared in a few distinct modes among parallel processes, and the consequent development of data abstractions for information sharing is another important contribution for the Charm project. Branch office chares of Charm — the replicated processes with sequential and parallel interfaces — constitute an extremely useful program structuring device. The BOC's along with distributed tables—one of the information sharing abstractions—provide support for modules that can exchange data in a distributed fashion. The Charm modules can be distributed in the object code form to different users and integrated in different applications with flexibility. Intelligent, "perceptive" performance analysis tools and debugging systems will be feasible with Charm based on the specificity of information available to the system. Versions of Charm with other based languages (e.g. C++ and Fortran) are being developed such that modules written in different languages can coexist in a single application. In addition to supporting direct application development, we expect Charm to be useful as a "universal back end" for higher level languages.

After demonstrating the usefulness of Charm for several key benchmark applications, we plan to use it more routinely to develop grand challenge applications, and to explore new parallel algorithms. In addition, we will continue our research on techniques and tools for supporting parallel programming in and around the Charm system. We are confident that application developers as well as computer scientists will find Charm to be a useful and usable system for building applications that fully utilize the parallel processing technology, and and developing additional higher-level tools.

Acknowledgemens:

I am grateful to Sandia National Laboratory for providing access to their parallel machines. The work described in this paper involves many current and past graduate students, including Wayne Fenton, Manish Gupta, Attila Gursoy, Ed Kornkven, Sanjeev Krishnan, B. Ramkumar, V. Saletore, Wennie Shu, Amitabh Sinha, and Nimish Shah.

References

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[2] Arvind, R. S. Nikhil, and K. K. Pingali. I-structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632, October 1989.

[3] K. Mani Chandy and Carl Kesselman. Compositional C++: Compositional parallel programming. Technical Report Caltech-CS-TR-92-13, Department of Computer Science, California Institute of Tech nology, 1992.

[4] A. Chien. *Concurrent Aggregates*. MIT Press, 1993.

[5] David Culler, Richard Karpa, David Patterson, Abhijit Sahay, Abhijit Sahat, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. Logp: Towards a realistic model of parallel computation. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming PPOPP*, San Diego, California, May 1993.

[6] A. Gursoy, L.V. Kale, and S.P. Vanka. Unsteady fluid flow calculations using a machine independent parallel programming environment. In R. B. Pelz, A. Ecer, and J. Häuser, editors, *Parallel Computational Fluid Dynamics '92*, pages 175–185. North-Holland, 1993.

[7] Attila Gursoy and L.V. Kale. Dagger: Combining the benefits of synchronous and asynchronous communication styles. Technical Report 93-3, Parallel Programming Laboratory, Department of Computer Science, University of Illinois, March 1993.

[8] Attila Gursoy and L.V.Kale. Simulating message driven programs. Technical Report 93-9, Parallel Programming Laboratory, Department of Computer Science, University of Illinois, Jul 1993.

[9] S. Hiranandani, K. Kennedy, and C. Tseng. *Compiler support for machine independent parallel programming in Fortran-D*. Elsevier Science Publishers B.V., 1992.

[10] L. V. Kale. A tree representation for parallel problem solving. In *Proceedings of AAAI*, pages 677–681, August 1988.

[11] L. V. Kale. The Chare kernel parallel programming language and system. In *Proceedings of the 1990 International Conference on Parallel Processing*, volume II, pages 17–25, St. Charles, IL, August 1990.

[12] L. V. Kale and V. Saletore. Parallel state-space search for a first solution. *International Journal of Parallel Programming*, 19:251–293, 1990.

[13] L. V. Kale and W. Shu. The Chare Kernel language for parallel programming: A perspective. Technical Report UIUCDCS-R-88-1451, Department of Computer Science, University of Illinois, August 1988.

[14] L.V. Kale. The REDUCE OR process model for parallel execution of logic programs. *Journal of Logic Programming*, 11(1):55–84, July 1991.

[15] L.V. Kale. A tutorial introduction to Charm. Technical Report 92-6, Parallel Programming Laboratory, Department of Computer Science, University of Illinois, 1992.

[16] L.V. Kale. Prioritization in parallel symbolic computing. In T. Ito and R. Halstead, editors, *To Appear in Lecture Notes in Comp. Science*, pages 146–181. Springer-Verlag, 1993.

[17] L.V. Kale and Sanjeev Krishnan. Charm++ : A portable concurrent object oriented system based on C++. In *To appear in the Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications, September 1993*, March 1993. (Also: Technical Report UIUCDCS-R-93-1796, March 1993, University of Illinois, Urbana, IL.

[18] L.V. Kale and Sanjeev Krishnan. A comparison based parallel sorting algorithm. In *Proceedings of the 1993 International Conference on Parallel Processing*, St. Charles, IL, August 1993.

[19] L.V. Kale and A.B. Sinha. Projections: A scalable performance tool, April 1993. Parallel Systems Fair, International Parallel Processing Symposium.

[20] L.V. Kale and Amitabh Sinha. Information sharing mechanisms in parallel programs. Technical Report 93-4, Parallel Programming Laboratory, Department of Computer Science, University of Illinois, March 1993.

[21] Edward Kornkven and Laxmikant Kale. Dynamic adaptive scheduling in an implementation of a data parallel language. Technical Report 92-10, Parallel Programming Laboratory, Department of Computer Science, University of Illinois, October 1992.

[22] B. Ramkumar and L.V. Kale. Machine independent AND and OR parallel execution of logic programs: Part I and II. *To appear in IEEE Transactions on Parallel and Distributed Systems*, 1991.

[23] V. Saletore. *Machine Independent Parallel Execution of Speculative Computations.* PhD thesis, Dept. of Computer Science, University of Illinois, 1990.

[24] W. W. Shu and L. V. Kale. Chare Kernel - a runtime support system for parallel computations. *Journal of Parallel and Distributed Computing*, 11:198–211, 1990.

[25] Amitabh Sinha and L.V. Kale. A load balancing strategy for prioritized execution of tasks. In *Workshop on Dynamic Object Placement and Load Balancing, in co-operation with ECOOP's 92*, Utrecht, The Netherlands, April 1992.

[26] Amitabh Sinha and L.V. Kale. A load balancing strategy for prioritized execution of tasks. In *International Parallel Processing Symposium*, New Port Beach, CA., April 1993.

[27] A. Yonezawa. *ABCL: An Object Oriented Concurrent System.* MIT Press, 1990.