A portable software support system for irregular computations

(extended abstract)

Laxmikant V. Kalé, Amitabh B. Sinha, Attila Gursoy
Department of Computer Science,
University of Illinois,
Urbana, IL 61801.
email: {kale,sinha,gursoy}@cs.uiuc.edu

1 Introduction

In the last decade, parallel programming has emerged as a powerful new technology. Many large commercial parallel machines are available today, such as Intel iPSC/860 (and soon, the Paragon), NCUBE's 1024 processor machines, CM-5, etc. A large class of algorithms in science, engineering, operations research, and artificial intelligence can potentially benefit from parallel processing. However most of the applications that have been successfully solved using parallel machines have a very regular structure. Irregular computations, such as branch&bound, adaptive grids, are an important class of algorithms. The effective parallel solution of irregular computations poses greater problems because of the need for facilities for dynamic creation of work, and dynamic load balancing.

The features required of any software support for irregular computations are:

- Portability: The application program should be portable across the wide variety of parallel machines that are currently available.
- Dynamic load balancing: The system should provide support for automatically balancing load, in addition to the capabilities of user-controlled load balancing.
- Dynamic creation of work: A large number of irregular computations could be more easily solved using parallel computers if tasks could be inexpensively created dynamically. The support system should allow inexpensive, dynamic creation of tasks.
- Modularity and reuse: An application program could consist of different modules that may be written by different programmers; therefore the system should allow programs to be written in a modular fashion and the reuse of modules.
- Prioritization: In irregular computations, there may exist many different tasks in the system at the same time. Additionally, the order of processing of these tasks may be important. In such cases, the system should provide support for the prioritized execution of tasks.

Over the last six years, we have developed a machine independent parallel programming system called Charm. The system provides all the support needed by a software support system for irregular computation. Charm has been extensively used for sucessfully solving large irregular computations arising in branch&bound algorithms, CAD problems etc.

In this abstract, we describe the basic features of Charm and an example of a branch&bound algorithm that was successfully solved using parallel computers.

2 Charm

Charm is a machine independent parallel programming system. Programs written using this system will run unchanged on MIMD machines with or without a shared memory. The system currently runs on Intel's iPSC/860, iPSC/2, NCUBE, Encore Multimax, Sequent Symmetry, ALLIANT FX/8, single-processor UNIX machines, and networks of workstations. It is being ported to a CM-5, Parsytec GC-el, and Alliant FX/2800.

Programs consist of potentially small-grained processes (called chares), and a special type of replicated processes, called branch-office chares. Charm supports dynamic creation of chares, by providing dynamic (as well as static) load balancing strategies. There may be thousands of small-grained chares on each processor, or just a few, depending on the application. Chares interact by sending messages to each other and via specific information sharing modes described below.

A charm program consists of chare (small processes or concurrent objects) definitions, message definitions, and declarations of specifically shared objects in addition to regular C language constructs. A chare definition consists of local variable declarations, entry-point functions that handle incoming messages and private function definitions. Some of the important Charm system calls are: *CreateChare(chareName,entryPoint,msg)* and *SendMsg(chareID,entryPoint,msg)*. CreateChare call is used to create an instance of a chare named as *chareName*. As all other Charm system calls, CreateChare is a non-blocking call, i.e., it immediately returns. Eventually the system creates an instance of chare *chareName* on some processor, starts its execution the *entryPoint* with the message *msg*. The SendMsg call deposits the message *msg* to be sent to the *entryPoint* of chare instance *chareID*.

The execution model of Charm is message-driven. The runtime system on each processor selects a message from the pool of available messages. If the message is directed to an existing chare, it invokes this chare with the message. If the message specifies creation of a new chare, the system creates it, and invokes it immediately with the message.

Messages represent just one mode of information sharing. Charm provides six information sharing modes, each of which may be implemented differently and efficiently on different machines. These include abstractions such as read-only variables, accumulators, distributed tables, and monotonic variables [1, 2].

Charm also provides a sophisticated module system that facilitates reuse, and large-scale programming for parallel software. From the point of this paper, the following properties of Charm are important:

1. Charm provides efficient portability across a range of parallel machines,
2. dynamic creation of processes and *dynamic load balancing* and features such as priorities, and distributed tables, Charm is uniquely suited for solving irregular problems.
3. Message driven execution in Charm enables creation of parallel programs that are highly *latency tolerant.* Processors are never blocked waiting for a particular message. When one processor waiting for data from a remote process, another ready process may be scheduled for execution. Also, even a single process may wait for multiple data items simultaneously, and continue execution whenever any of the expected items arrive.
4. The *modularity* features supported in Charm permit flexible reuse of parallel software. Separately compiled library modules can be put together in a varying of combinations and contexts.

3 Application: Traveling Salesman Problem

The Traveling Salesman Problem (TSP) [3] is a typical example of an optimization problem solved using branch&bound techniques. In this problem a salesman must visit n cities, returning to the starting point, and is required to minimize the total cost of the trip. Every pair of cities i and j has a cost C_{ij} associated with them (if $i = j$, then C_{ij} is assumed to be of infinite cost).

We have implemented the branch&bound scheme proposed by Little, et. al. [4]. In Little's approach one starts with an initial partial solution, a cost function (C) and an infinite upper bound. A partial solution comprises a set of edges (pairs of cities) that have been included in the circuit, and a set of edges that have been excluded from the circuit. The cost function provides for each partial solution a lower bound on the cost of any solution found by extending the partial solution. The cost function is monotonic, i.e., if S_1 and S_2 are partial solutions and S_2 is obtained by extending S_1, then $C(S_1) <= C(S_2)$. Two new partial solutions are obtained from the current partial solution by including and excluding the "best" edge (determined using some selection criterion) not in the partial solution. A partial solution is discarded (pruned) if its lower bound is larger than the current upper bound. The upper bound is updated whenever a solution is reached.

In the Charm implementation of the branch&bound solution of TSP, each partial solution is represented by a chare and the cost of the partial solution is the priority of the new chare message. A monotonic variable is used to maintain the upper bound.

Figure 1 shows the execution times and the number of nodes generated for runs of a 60-city asymmetric TSP on the NCUBE/2 with the token load balancing strategy. Notice that the number of nodes generated in this case are fairly constant for up to 512 processors and the speedups are good.

We are working on developing a generalized branch&bound package that provides the parallel component of the branch&bound algorithm. Different branch&bound applications can be written on top of the parallel component by providing functions, such as *branching* and *lower-bounding*, for that particular application.

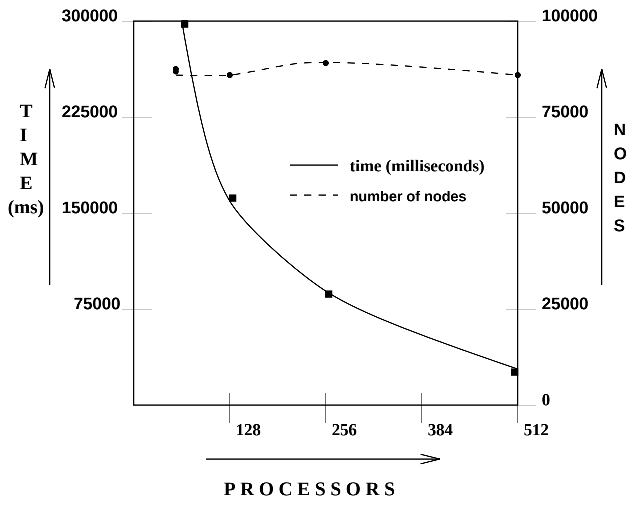

Figure 1: **The figure shows the execution times and the number of nodes generated for executions of a 60 city asymmetric TSP on the NCUBE/2 with upto 512 processors using the tokens strategy to balance load. In this case the cluster size is 16 processors.**

4 Discussion

We plan to use Charm to solve other irregular computations. These obviously include tree-structured computations, such as state space search, bi-directional search game tree search, and planning. More importantly, we plan to focus on numerical computations with dynamic or irregular structure, such as finite element computations and adaptive grid refinements. Several other seemingly regular numerical computations also exhibit dynamic behavior due to indeterminacy in message arrival times and critical path considerations.

In summary, Charm provides a great deal of support that is required in solving irregular computations on parallel machines. We have illustrated its utility in the solution of a parallel branch&bound algorithm. Using Charm, researchers developing algorithms to solve irregular computations will be able to leverage their efforts and concentrate on the algorithmic issues by leaving the task of machine dependent implementations, load balancing and prioritized scheduling to the system.

References

[1] L. V. Kale. The Chare Kernel Parallel Programming System. In *International Conference on Parallel Processing*, August 1990.

[2] L. V. Kale *et. al.* The Chare Kernel Programming Language Manual (Internal Report).

[3] Edward W. Reingold, Jurg Nievergelt, and Narsingh Deo. *Combinatorial Algorithms: Theory and Practice.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1977.

[4] J. D. C. Little, K. G. Murty, D. W. Sweeney, and C. Karel. An algorithm for the traveling salesman problem. *Operations Research*, 11:972–989, 1963.