Adapting to Load on Workstation Clusters

Robert K. Brunner
Theoretical Biophysics Group
University of Illinois at Urbana-Champaign
rbrunner@uiuc.edu

Laxmikant V. Kalé
Department of Computer Science
University of Illinois at Urbana-Champaign
kale@cs.uiuc.edu

Abstract

Desktop workstations represent a largely untapped source of computational power for parallel computing. Two of the main problems in utilizing these workstations are developing strategies for migrating load so that partially loaded workstations can contribute CPU cycles to the computation, and making dynamically migratable application programs easy to write. This paper describes object arrays*, a construct which makes dynamically migratable applications easier to write, and a simple strategy for migrating load on a workstation cluster.*

1 Introduction

Workstation clusters are emerging as the most cost-effective platform for parallel computing. The low price of commodity hardware insures that such systems achieve excellent price to performance ratios. With advances in communication technology, and operating system level innovations to tune communication performance for parallel applications, a group of commodity computers with standard LAN hardware can serve as a high-throughput parallel computer. Several prominent projects have demonstrated the feasibility of putting together large collections of workstations into dedicated clusters. [18]

However, the most compelling argument in favor of using clusters for parallel computing is that such clusters already exist in individual departments. The workstations on individuals' desks are usually busy only a fraction of the time in a day; cluster computing offers the possibility of utilizing this latent computing power to perform valuable computation.

Even when a laboratory decides to acquire workstations as a dedicated computational cluster, the resultant system becomes a hybrid of compute-only and desktop machines. As an example, the biophysics application group we are associated with has gradually acquired more than twenty HP workstations to serve primarily as computational machines. Although the machines spend most of their time performing computations, several also spend some of there time as compile servers and program development machines. Several of these workstations are on users' desks, while others are in a pool in the computer room. The flexibility offered by such a cluster is tremendously useful. When new members join the group, they can be given one of the available workstations from the processor pool. When a researcher needs to run a job, but his own workstation is busy, he can run his job on one of the workstations in the pool.

For running parallel applications, however, such an environment poses several problems. One of the most severe problems is the performance impact of the shared usage on the parallel application. Specifically, when a parallel application is running on a set of workstations, and the performance of one of the workstations deteriorates due to shared usage, the performance of the entire parallel application degrades disproportionately. Mitigating the effects of background load is the focus of our research.

The next section describes and documents this problem with performance data from benchmark applications. Our solution strategy is based on the use of parallel object-oriented programming. We have added object-migration capabilities to the Charm++ system, through a new distributed object type, the object array, which is described in section 3. Section 4 describes a solution strategy for solving the performance deterioration problem, and describes the performance results obtained using the strategy. The last section concludes with some comparisons with other migration schemes, and thoughts for future work.

2 The Problem Documented

A large fraction of parallel applications today are loosely synchronous: computations on one processor continually depend on data produced by other processors. (Not all computations are globally synchronous. Divide and conquer computations, and the state-space search computations lead to computations trees, where the only dependencies are between parent and child nodes of the tree.) Consider a molecular dynamics application (such as NAMD [13]) which divides the simulation volume into cells distributed among various processors. Computation for each cell may depend only on data from adjacent cells during a single timestep. However, over a period of several steps, the data from a particular cell indirectly affects every other cell in the simulation. The dependency graph insures that no cell can proceed more than one step ahead of the slowest cell in the simulation. Some simulations require collective operations, such as reductions to compute global scaling factors, and in those cases the processors are even more tightly coupled.

In case of such globally synchronous computations, utilizing a shared workstation cluster presents special problems. Consider an application running on eight workstations of a cluster. When the computation begins, all eight workstations are otherwise idle. Let us further assume that our application can utilize all eight workstations with one hundred percent efficiency. Sometime during the parallel run, another user starts a single processor job (it may be as small as a compilation or a long-running single-processor computational job). Now our parallel job is receiving only about half of the processing power available on this workstation, but the other seven workstations are still fully committed to the parallel job. Yet, due to the dependencies among sub-computations, all the other processors will have to waste half of their time waiting for results from the busy processor. Thus, even though we have lost only 1/16th of the total computational power to the outside job, the performance of the program drops by fifty percent! We have observed this effect while running a parallel molecular dynamics application [13]. To study this phenomenon further, we implemented an artificial benchmark that allows us to study the phenomenon without the complexities of a full-featured code.

Our benchmark is a simple simulation. The computational domain is a two-dimensional square, which is divided into a uniform two-dimensional grid. The grid is divided into multiple cells, which are allocated among the available processors. The problem can be thought of as a classical heat conduction problem, with a time-dependent forcing function. One edge of the square is "heated" according to a predefined function. During each timestep, the temperatures at all the points in the grid are updated using a Jacobi relaxation scheme: each point on the grid is given the new value based on the average of all its neighboring values. The relaxation algorithm is repeated until a convergence criteria is met. To test the convergence criteria, a global reduction must be carried out to determine the maximum error. When the maximum error falls below a preset threshold, the iteration may stop, and simulation proceeds to the next timestep.

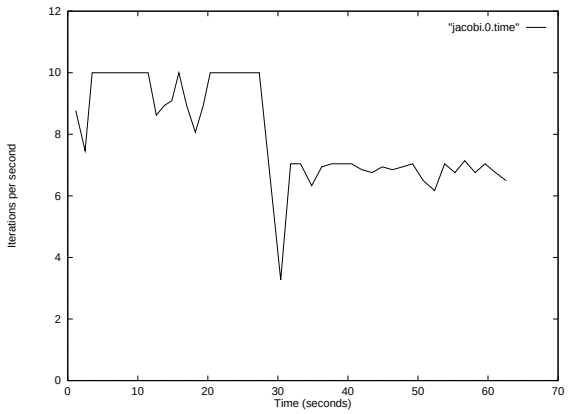

Figure 1. Number of iterations per second for the `jacobi` benchmark on eight processors. The cluster is composed of HP 735/125 workstations connected by an ATM network.

We ran the Jacobi benchmark on eight otherwise-idle HP 735/125 workstations communicating over a 100Mbps ATM network. About thirty seconds into the run, we started a computationally intensive job on one of the workstations. Figure 1 shows the impact on performance: prior to the onset of the new job, roughly ten iterations were executed per second. After the job starts, the number of iterations per second drops to about seven. Although the total compute power decreased only six percent, throughput decreased thirty percent. We did another experiment in which the reduction step was removed from the benchmark. Such a pattern of neighbor communication without global reduction is a common occurrence in parallel programs (such as in explicit methods). In that benchmark, because the global synchronization is absent, the processor utilization is much higher. As a result, the impact of the slowdown of one processor is even higher, leading to a fifty percent decrease in throughput after the intrusion of external load.

2.1 Related work

Our system attempts to schedule work to utilize the processing power of partially available workstations. A variation of the above problem in the cluster environment involves a situation where one of the workstations is available for the parallel computations only as long as the "owner" is not using it. As soon as interactive usage of the workstation begins, we wish to move the computation completely away from that workstation. Solutions to these problem must address a number of common sub-problems, including:

1. Where should the load be scheduled?
2. When should load be migrated?
3. How does the application programmer write migratable code.

A number of researchers have addressed the first two points. The approaches described in [2] and [14] concentrate on scheduling applications for heterogeneous collections of geographically separated processors. Slow communications links permit only occasional dynamic migration, so these systems must ascertain detailed information about each processor to find a good load distribution.

One approach used on homogeneous clusters is to migrate the entire process to another processor. Open files and sockets associated with the process present technical problems for this approach, which have been handled by several of the proposed systems. As the image of a Unix (or any operating system in use today) process is large, it requires significant communication and consequent delay before a process can be migrated. An interesting approach, explored by Rousch [16], is to migrate active memory pages (including the page containing the top of the stack) to the new site, and starting process execution at the new site before all of the pages are brought in. If the process tries to access one of these pages, a page fault is trapped and the process waits for the page. Although this approach reduces the latency somewhat, the high cost of communication still remains. Furthermore, another workstation must be available to accommodate the new process.

A number of researchers are developing libraries of C++ objects and templates to support distributed parallel objects [15]. These use the standard facilities of C++ to implement several styles of distributed objects, such as vectors and matrices of arbitrary data types. The application programmer is provided with data distribution directives and mathematical operations to perform common calculations on the data in parallel, similar to the capabilities provided by HPF. DOME [1] is one example of such a system, where an objects data distribution can be changed in response to load conditions, allowing migration of the work represented by that data.

Another approach [14] is to simply provide the application program with notification that it has to move, and forcing the user to explicitly move the process state. These systems work well with master-slave algorithms, where the slaves typically require little state information to be retained. The slaves simply terminate, and are recreated elsewhere where the computations can be repeated.

A few distributed multi-threaded systems allow threads representing tasks to migrate to different address spaces of a distributed-memory machine. UPVM [11] presents a lightweight process model with a PVM-like message-passing library, which supports thread migration independently of an object-oriented framework. PM2 [12] is another migratable-thread system, which treats threads as remote procedure calls, which return some data on completion.

A system, similar to ours in several respects, has been implemented by Ramkumar [6] in the context of the ELMO system (ELMO is based on Charm [9, 10]). As in our approach, ELMO adds object migration to Charm, but this system implements object migration without load balancing, mainly for fault-tolerant computing.

The focus of our work chiefly addresses subproblems two and three. We present an approach for obtaining better performance on smaller workstation networks, where migration can occur efficiently in time scales on the order of a second. The problem with programs implemented using PVM or MPI-style communication is that the programming style does not encourage decomposing the application into small, easily migratable pieces of work. Our object-oriented approach permits cheap, nearly automatic, migration of just a portion of a particular processor's load, and allows complex communication between objects without the application programmer having to worry about where a particular object resides. The system does not need to know the characteristics of the individual workstations since it works in a closed-loop fashion. Migrating too much load is sensed and corrected in the next migration period, so the program continually converges toward an optimal load balance. Furthermore, our approach allows partial utilization of processors running other tasks. Of course, object arrays would also simplify application development for several of the programming systems cited above.

3 Parallel Objects and Object Arrays

In this section we describe object arrays, a mechanism we developed that helps solve the problem identified above. Object arrays have been implemented as an extension to an existing parallel object system called Charm++. Charm++ [8] is a C++-based object-oriented message-driven language that supports encapsulation and multiple inheritance in parallel objects. Charm++ programs consist of potentially medium grained objects, *chares*, and object groups called branch-office chares or BOCs. Charm++ supports dynamic creation of chares, by providing dynamic (as well as static) load balancing of chare-creation (seed) messages. Chares interact by sending messages to each other and via specific information sharing abstraction. An instance of a BOC has a representative (branch) chare on every processor. All the branches of a single BOC instance share a global ID. One can send a message to a specific branch chare of a BOC on a particular processor, or broadcast it to all its branches. BOCs are useful whenever the programmer wants to explicitly control what each processor is doing, such as when implementing reduction operations, expressing static load balancing by explicitly distributing a problem across the processors, and writing SPMD-style programs. In addition to messages and BOCs, Charm++ provides information sharing abstractions such as read-only variables, monotonic variables, write-once variables, accumulators and distributed tables. Details about these features can be found in [5].

A Charm++ program consists of C++ code incorporating calls to the Charm++ runtime library (i.e. no language extension to C++). A small interface file lists which of the C++ object types are Charm++ message types, chares or BOCs. It also lists which methods of those classes can receive Charm++ messages. Since most of the program is standard C++, C and Fortran code can be called just as in a C++ program.

Charm++ is implemented on top of Converse [7]. The Converse runtime system is message-driven. Converse repeatedly selects one of the available messages from a pool of messages, switches to the context of the chare to which it is directed, and initiates execution of the code specified by the message.

3.1 Object Arrays

In addition to chares and BOCs, we have added a new construct to Charm++, object arrays. An object array consists of a multi-dimensional collection of data-driven objects. Messages may be directed to any individual element, multicast to a subset of them, or broadcast to all elements of the array. An individual element is specified using its array name and element indices only. The user does not need to know the processor on which the element is located, since the system maintains the mapping information for each array. Since the user does not know which processors host which elements, the run-time system is free to move elements to improve load balance or to accommodate external changes in machine load. An earlier implementation of object arrays in Charm++ [17] supported movable object arrays; however, the mapping had to be specified by a user-defined mapping function. Although this approach supported various regular mappings of objects to processors (for example, periodically realigning the rows or columns of the array for efficient communication during different stages of a computation), it was not under system control, so it did not allow automatic response to outside conditions. The current implementation of object arrays is a C++ class library implemented in Charm++, which supplies a migrate call to allow remapping by either the user or the runtime system.

3.2 Migration

Object migration requires two main mechanisms: a method of saving the object state on one processor and moving it to another, and a means of forwarding messages to the processor to which the object migrates. Our system depends on user assistance to transfer the object state. The user provides two methods in an array object, a pack-state method and an unpack-state constructor. A major programming advantage of object arrays is that it is much easier to write code to pack the states of particular objects than to automatically save the entire state of the program. At migration time, the pack/unpack routines are automatically called to transfer the object to the new processor. Message forwarding is handled automatically by the system. If a message arrives at a processor only to find that the destination object (element) is not there, the message is forwarded on to the new location. Each processor has a map of where objects reside, but this map may be out of date, so the fallback strategy is to forward messages to each element's original host processor. Each processor is kept informed of the location of each element it originally hosted, so it can always forward messages to the element with only one extra hop.

3.3 Object Array API

The object array API is an extension of the standard Charm++ syntax. The following is a description of the major routines. The programmer defines an object array by creating a class which inherits from the supplied `ArrayElement` chare type. The class may supply two constructors, one that is called at initialization time, and one that is called after migration and re-creation on the destination processor. This migrate constructor is responsible for unpacking the state packed before migration. The user may also supply a `pack` method, although the default behavior is to simply copy the memory space of the object.

Array instances are typically created at startup (although creation during a run is also possible). Here is the startup code to create a one-dimensional array of objects of type `Cell`:

```
arrayGroup = Array1D::CreateArray(array_dim,
  ChareIndex(MigrateMap),ChareIndex(Cell));
```

The first parameter is the number of elements in the array. The next parameter gives the type for the *map object*, in this case, the migratable-object map. The next parameter gives the array element type. The `Array1D` object is a Charm++ BOC, which takes care of message sending and coordinates object creation.

Messages are sent to array elements using the `send` method, which is part of the library-supplied `Array1D` class:

```
thisArray->send(msg,send_to,
  EntryIndex(Cell,neighbor_data,NeighborMsg));
```

This send call sends the message object `msg` of type `NeighborMsg` to entry method `Cell::neighbor_data`. The receiving element handles the message just as it would a normal Charm++ message, by executing an entry function on the destination processor, with the message as the parameter.

The `ArrayElement` base class provides a number of variables for programmer convenience. As used above, `thisArray` is a pointer to the local `Array1D` branch. For the one-dimensional case, `thisIndex` contains the element index. Two and three-dimension arrays also have `thisi`, `thisj`, and `thisk` to give the element coordinates.

4 Solution Strategy and Performance

Once the mechanisms of migration have been implemented, various strategies for migrating load become feasible. Our initial load balancer is a decentralized, application-independent system for detecting overloaded processors and migrating array elements to new locations. First, using the facilities available in the Converse run-time library [7], the load balancer obtains a measure of processor availability on each processor: this is essentially the CPU time obtained by a process during the time interval, divided by the elapsed wall-clock time. Converse also provides callbacks for when a processor becomes idle, and just before it resumes working (due to the receipt of a message). This data allows the load balancer to compute the CPU time consumed by other processes:

$$T_{other} = T_{wall} - T_{self} - T_{idle}$$

Based on this information, the system decides when there is a severe load imbalance due to outside load, and calculates what fraction of the load must migrate to other processors. During the run, the system automatically keeps track of the time consumed by each array element. At migration time, the system selects a set of elements for migration which most closely match the desired fraction of the total load. Currently, migrated elements are distributed evenly to the other processors, so no global load knowledge is necessary.

Load information is accumulated locally at regular intervals using a periodic timer callback supplied by the Converse system. For these experiments, load data is collected every two seconds, and migration may occur as soon as the load data indicates idle time.

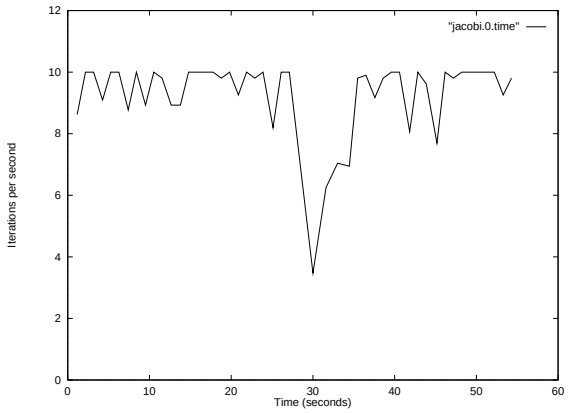

Figure 2. Number of iterations per second for the `jacobi` benchmark on eight processors, with automatic migration.

After implementing this migration strategy, the benchmark program described in section 2 was run again. The system automatically detected when a second job was started on one of the processors and moved about half of its load to other processors. The resultant

performance data is shown in figure 2. The data shows a brief but severe dip when the extra job starts. A few seconds later, the dip is detected and the system migrates array elements to new nodes. The performance quickly returns nearly to its initial level.

5 Conclusion

Migration of object arrays has several advantages over other task migration schemes. Process-migration methods do not permit fine control of processor load balance, allowing processors to be occupied or vacated, with no intermediate state. Data-parallel style C++ libraries offer superior ease of programming for many scientific applications, but do not permit the application programmer to create objects with complex communication patterns and control structures, which are necessary for irregular applications. Migratable thread systems do allow complex objects, but since these systems are usually based on a user-level thread package, the thread stack size must be selected when the thread is created, which can result in either running out of stack space or wasting memory if the stack estimate is poor. Furthermore, activating a thread, although much quicker than performing a process context switch, usually takes longer than making a function call to an object. Migratable threads also require more complex virtual memory management techniques to preserve pointers after migration. [11]

Future work with object arrays involves performance optimizations and improvements in programmer convenience. We are implementing automatic systems which observe communication patterns among elements (according to element index), so the system can autonomously discover communication structure among elements so that the migration strategy can account for communication when moving elements. Object arrays also enable several other capabilities which we will exploit, such as on-demand evacuation of all tasks from a processor, automatic adaptation to different processor speeds, and automatic load balancing of irregular applications which use object arrays. Programmer convenience will be served by creating libraries to implement common operations (e.g. reductions). Unlike normal Charm++ programs, migratable objects present problems for split-phase operations such as asynchronous reductions, where the data must be collected from a moving set of elements. More sophisticated bookkeeping that accounts for migrating data is a necessary component of such libraries. Further work will determine the most efficient rate of migration, and whether the system should try to balance the load perfectly each time, or just incrementally improve the load distribution to more slowly converge on the optimal distribution.

References

[1] J. N. C. Arabe, A. Beguelin, B. Lowekamp, E. Seligman, M. Starkey, and P. Stephan. Dome: Parallel programming in a heterogeneous multi-user environment. Technical Report CS-95-137, Carnegie Mellon University, School of Computer Science, Apr. 1995.

[2] F. Berman and R. Wolski. The AppLeS project: A status report. In *Proceedings of the 8th NEC Research Symposium*, May 1997.

[3] A. Chien, S. Pakin, M. Lauria, M. Buchanan, K. Hane, L. Giannini, and J. Prusakova. High Performance Virtual Machines (HPVM): Clusters with supercomputing apis and performance. In *Eighth SIAM Conference on Parallel Processing for Scientific Computing (PP97)*, March 1997.

[4] D. E. Culler, A. Arpaci-Dusseau, R. Arpaci-Dusseau, B. Chun, S. Lumetta, A. Mainwaring, R. Martin, C. Yoshikawa, and F. Wong. Parallel Computing on the Berkeley NOW. In *9th Joint Symposium on Parallel Processing, Kobe, Japan*, 1997.

[5] Department of Computer Science,University of Illinois at Urbana-Champaign, Urbana, IL. *The CHARM (4.5) programming language manual*, 1997.

[6] N. Doulas and B. Ramkumar. Task migration in message driven systems. In *First Annual Workshop on Message Driven Execution and Charm*, Urbana, Illinois, Oct 1994.

[7] L. V. Kalé, M. Bhandarkar, N. Jagathesan, S. Krishnan, and J. Yelon. Converse: An Interoperable Framework for Parallel Programming. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 212–217, Honolulu, Hawaii, April 1996.

[8] L. V. Kalé and S. Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In G. V. Wilson and P. Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, 1996.

[9] L. V. Kalé, B. Ramkumar, A. B. Sinha, and A. Gursoy. The CHARM Parallel Programming Language and System: Part I – Description of Language Features. *IEEE Transactions on Parallel and Distributed Systems*, 1994.

[10] L. V. Kalé, B. Ramkumar, A. B. Sinha, and V. A. Saletore. The CHARM Parallel Programming Language and System: Part II – The Runtime system. *IEEE Transactions on Parallel and Distributed Systems*, 1994.

[11] R. B. Konuru, S. W. Otto, and J. Walpole. A migratable user-level process package for PVM. *Journal of Parallel and Distributed Computing*, 40(1):81–102, Jan. 1997.

[12] R. Namyst and J.-F. Méhaut. PM2: parallel multithreaded machine. a computing environment for distributed architectures. In *Parallel Computing*

(ParCo'95), pages 279–285. Elsevier Science Publishers, September 1995.

[13] M. Nelson, W. Humphrey, A. Gursoy, A. Dalke, L. Kalé, R. D. Skeel, and K. Schulten. NAMD— A parallel, object-oriented molecular dynamics program. *J. Supercomputing App.*, 1996.

[14] J. Pruyne and M. Livny. Parallel processing on dynamic resources with CARMI. *Lecture Notes in Computer Science*, 949:259–??, 1995.

[15] J. V. W. Reynders, P. J. Hinker, J. C. Cummings, S. R. Atlas, S. Banerjee, W. F. Humphrey, S. R. Karmesin, K. Keahey, M. Srikan, and M. D. Tholburn. Pooma. In G. V. Wilson and P. Lu, editors, *Parallel Programming Using C++*, chapter 14, pages 547–587. The MIT Press, 1996.

[16] E. T. Rousch and R. H. Campbell. Fast dynamic process migration. In *ICDCS '96; Proceedings of the 16th International Conference on Distributed Computing Systems; May 27-30, 1996, Hong Kong*, pages 637–645, Washington - Brussels - Tokyo, May 1996. IEEE.

[17] Sanjeev Krishnan and L. V. Kalé. A parallel array abstraction for data-driven objects. In *Proc. Parallel Object-Oriented Methods and Applications Conference*, February 1996.

[18] T. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranawake, and C. V. Packer. BEOWULF : A parallel workstation for scientific computation. In *International Conference on Parallel Processing, Vol.1: Architecture*, pages 11–14, Boca Raton, USA, Aug. 1995. CRC Press.