Handling Application-Induced Load Imbalance using Parallel Objects

Robert K. Brunner and Laxmikant V. Kalé

Abstract

One of the problems in applying parallel computing for a broad class of applications arises from the dynamic nature of computations being parallelized. Even if the computation is carefully load-balanced at the beginning, the balance deteriorates over time, either due to adaptive refinements, or gradual change in the load of different components. Such imbalances can have a dramatic effect on performance, especially when a large number of processors are used. We present a methodology based on data-driven objects that can automatically handle such application-induced load imbalances and rebalance the load as needed. The methodology relies on automatic instrumentation to record load and communication patterns, and a flexible load-balancing framework that facilitates development of different strategies that may be appropriate for different classes of applications.

1 Introduction

A major factor slowing deployment of parallel programs is that efficient parallel programs are difficult to write. Parallel programming adds a second dimension to programming: not just when will a particular operation be executed, but where, i.e. what processor will perform it. A vast number of parallelizable applications do not have a regular structure for efficient parallelization. Such applications require load balancing to perform efficiently in parallel. The load in these applications may also change over time, requiring rebalancing. The programmer is left with the choice of either distributing computation in an ad-hoc manner, producing poorly-performing programs, or spending more development time including load-balancing code in the application.

Work migration [5, 2, 7, 15, 4, 1] is a unified scheme for handling both application-induced and externally-arising (such as that on timeshared clusters) load imbalance. The difficulty with migrating work is that either work is repartitioned in an application-specific way, placing the burden on the application programmer, or that automatic migration is supported, but with poor accuracy, due to the lack of application-specific knowledge.

Object migration [6, 14] provides a way of performing accurate, fine-grained automatic load balancing. Objects usually have small, well-defined regions of memory on which they operate, reducing the cost of migration. Using the Charm++ object model, the run-time system can measure the work represented by particular objects, rather than deriving execution time from application-specific heuristics. Furthermore, the run-time system can record object-to-object communication patterns, so the load balancer can asses the communication impact of migrating particular objects.

In this paper, we present a framework for adaptive load balancing based on measurement-based object migration, and demonstrate its utility for application-induced load imbalance. Application induced load imbalance may arise abruptly, as with adaptive mesh refinement, which can increase the computational load of a partition assigned to a processor by many orders of magnitudes, or continuously, as in molecular dynamics applications (e.g. NAMD [13, 11, 12]) where some atoms move slowly to newer partitions, over a number of timesteps. To be successful in these and other application-specific conditions, the load balancing framework must be versatile. Further, it is necessary to have a programming model that permits this framework to do measurement-based balancing (which is more accurate than prediction-based methods) via automatic instrumentation. These objectives are achieved by using the Charm++ object model, which is described in section 2. Section 3 describes our framework, which provides the mechanisms necessary for load balancing, including object migration and instrumentation. Section 4 describes application characteristic and corresponding distinct categories of load balancers that can be implemented in the framework. This framework has been implemented and available for use on most parallel machines and clusters. Performance results obtained using two benchmarks, a synthetic random computation/commmunication benchmark and a simple adaptive refinement program are described in section 5.

2 The Charm++ Object Model

Charm++ [10] implements an object-based message-driven execution model [9], which is well suited to object migration. Charm++ programs are composed entirely of objects, which are instances of C++ classes in which certain *entry methods* are invoked by the receipt of messages directed at that entry method. Sending a message to an entry method may be viewed as invoking a method on a remote object. Charm++ assumes a distributed-memory model, although a particular application may use shared memory when provided by the machine, with potential loss of portability. When a method is invoked, it runs until completion. Then the next message is retrieved from a scheduler queue and the indicated method of a particular object executed.

There are three types of objects in Charm++. *Chares* are single instances of objects. Chare creation results in the creation of a *seed message*, which is enqueued in the message queue on some arbitrary processor. The seed message may move among the processor message queues to provide load balancing in some application. Eventually the seed message is dequeued, the chare is created and its constructor called. Once the chare is created, it may send its chare ID to other objects, who may then use that ID to invoke methods on that chare. The second type of Charm++ objects are *object groups* (also called *branch-office chares*). Creation of an object group results in the creation of one object instance on each processor, identified by a group ID. Individual objects in an object group are accessed using the pair of group ID and processor number. Other objects may invoke entry methods on any particular processor of an object group, or may broadcast method invocations to every member of the group. Furthermore, any object may use the group ID to obtain a pointer to the branch object on that processor, which may then be used to invoke non-entry methods of the object, or access public data members of the object. The third type of objects, *object arrays* [16, 3], are arbitrarily-sized sets of objects, where the size is defined when the array is created. Array elements are accessed by remote method invocation, using their *array ID* and element index.

Several characteristics of the Charm++ object model make it particularly well-suited to object migration. First, messages to chares and array elements refer to particular objects, not to processors. Therefore, the run-time system has the freedom to relocate those objects without informing any object that may have an object ID. The run-time system forwards messages to objects as necessary. Also, object methods are executed non-premptively. This dramatically simplifies object

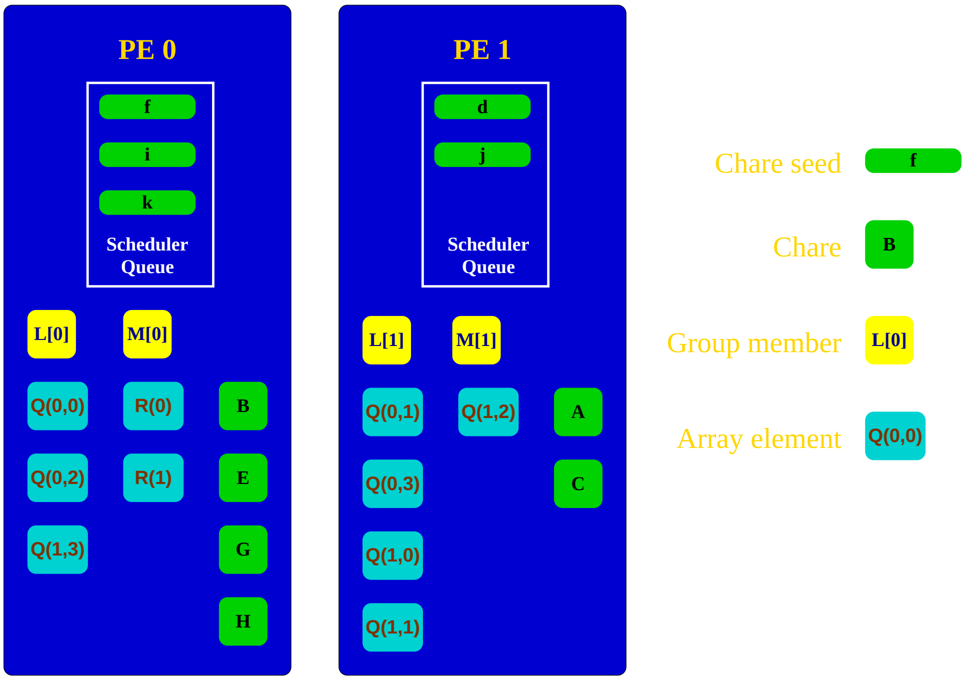

Figure 1: A typical object distribution on two processors in a Charm++ program.

migration. Since migration can only occur between method invocations, no stack data needs to be moved to re-create the state of the object. Object state is limited to the data members of the object, resulting in more efficient migration. Because of the way object groups are accessed, they are more difficult to migrate transparently. An analysis of how object groups are used in applications shows that it is not necessary to migrate object group members. Object groups are employed for one of two purposes. First, object groups may be used to break a problem into parts for parallelization. In this case, replacing the object group with an object array of a desired size and dimension is more convenient for application design as well as migration. The second use of object groups is as a local coordination object for collective operations. For example, a reduction library would be implemented as an object group. Client objects using the reduction library make a function call to the local representative of the object group. If the client object migrates, it may easily obtain the local reduction representative in its new location. For this use, it is important that there is exactly one branch of the object group on each processor.

The way objects are triggered by messages allows the Charm++ run-time to know which object is executing, and for how long it executes. The run-time system can also record which objects send messages, to which objects the messages are delivered, and the message size. This information can be logged and used to guide object migration.

Fig. 1 shows various objects in a Charm++ program. Several chares have been created on the two processors, and several chare seeds are awaiting creationg in the message queue. Two object groups, `L` and `M` have been created, each with one object on each processor. Finally, two object arrays, `Q` and `R`, have elements distributed across the processors.

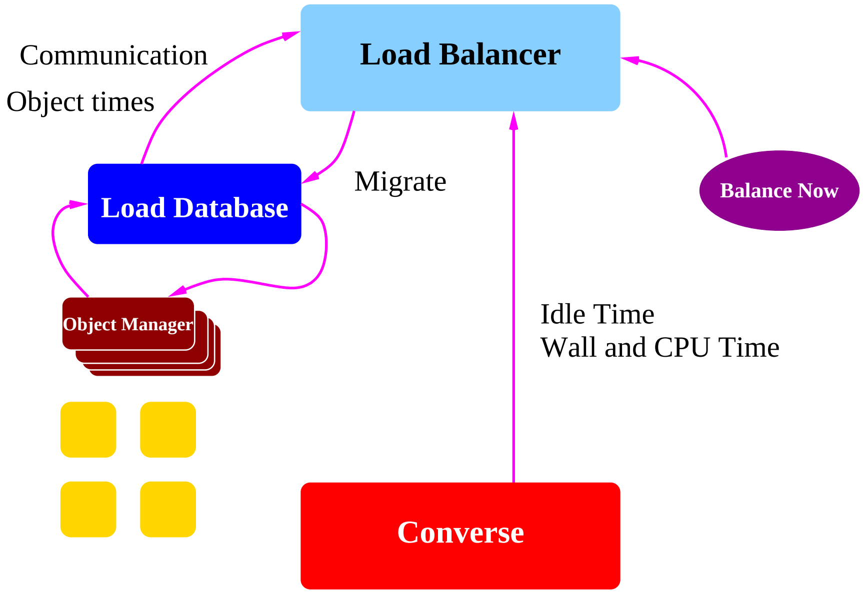

Figure 2: The components and interactions in the load balancing framework.

3 An Automatic Load Balance Framework

We have designed a framework for creating automatically load balanced Charm++ applications. Furthermore, we have carefully designed the framework to allow other programming models, (such as thread migration) to use load balancers written for the framework, permitting load balancing for multilingual parallel programs [8].

Fig. 2 shows the main components of the load balancing framework. Note that this is a view of the components on a single processor. Each processor will have a single load balancer object, but most load balance algorithms will require the load balancer object on each processor to communicate with its partners on other processors.

At the top level of the framework is the load balancer object itself. It responds to external triggers by executing the load balancing algorithm. The *Balance Now* trigger may come from a number of sources. The application may send the trigger when it has performed some action which will result in a load change. An external timer routine may request periodic load balancing, to account for changes in either the application or the background load on the machine. A user-generated trigger may occur when the workstation owner requests the application to vacate his workstation.

The load balancer gets information about the state of the processor from two sources. The *Load Database* records information about all the objects on the processor. The information includes how much processor time each object has consumed, and the destinations and sizes of messages those objects have sent. The load balancer may choose to clear the accumulated load in the database, or turn object measurement on and off, to eliminate the (minimal) time consumed by timer calls during periods when timing is not required. When the load balancer requires objects to migrate, the migration is passed to the load database, which passes the requests to the next level. The other source of information for the load balancer is the Charm++/Converse runtime system. Converse provides callbacks to provide processor idle time, as well as total wall-clock and CPU times spent executing code on the processor. This information allows the load balancer to account for load not reported to the load database, and to take corrective action when a particular processor is not busy.

The load database interacts with objects through a number of *object managers*. For object arrays, each array manager is an object manager. Other Charm++ objects (or other program entities such as threads) will have their own object managers. Object managers are responsible for reporting the arrival and departure of objects to the load database. They also report when a particular object is being executed, so that the database may accumulate time for the method, and when objects send messages to other objects. Object managers may register non-migratable objects, so that communication with these objects may also be taken into account. When the load database receives a request to migrate an object, it forwards the request to the object manager, which coordinates the migration with the user object.

4 Load Balancing Strategies

Different applications exhibit diverse patterns of load imbalance. Load balancers must be specifically tuned to work most effectively for the nature of the application. We divide these load imbalance patterns into three categories.

Initial Imbalance: All except the most regular parallel computations require a non-trivial initial balancing. What processor performs what particular part of the complete computation depends on factors such processor architecture, computation to communication ratio, and problem size. Initial load distribution is usually determined in an application specific manner, using some heuristic assumptions about computation and communication costs to compute an initial work distribution. Determining a good heuristic model is a time-consuming task, which may have to be repeated for different problem sizes or parallel machines. Our alternative solution is to use the same measurement-based methods used for dynamic load balancing. Objects are initially distributed using some unsophisticated heuristic method. After a short period of normal computation, actual object times and communications patterns are used to determine an accurate load balancing solution.

Slow Variation: Many problems exhibit slow variation in program load. One particular example is molecular dynamics simulation. [12] Simulations typically run for hundreds of hours of real time. During that time, the load distribution may change dramatically, as pieces of the molecule move around the simulation space, changing the number of particles for which each processor is responsible. Therefore, some load balancing is required for efficient execution. However, these changes occur slowly. Typically the load remains nearly constant for hours at a time, so rebalancing the load every few hours is sufficient to maintain parallel efficiency. Since load balancing is done so infrequently, more time consuming balancer algorithms, possibly requiring considerable communication and several minutes of computation, are preferred, since the extra cost is more than offset by the increased efficiency that results.

Abrupt Variation: More often, irregular applications exhibit fast or abrupt variation in load. Adaptive mesh refinement methods are typical examples of this. As the mesh changes, particular regions in the simulation can become much more expensive. Then, as the perturbation moves on, the mesh may become less dense, resulting in less load. Since the computation is still tightly coupled, even one overloaded processor dramatically decreases the parallel efficiency of the program. In such cases, only very frequent load balancing can maintain good load balance. Since load balancing must occur often, there is not much time for communication. In these cases, local strategies which only consider the loads on a few neighboring processors, requiring only limited communication, are likely to result in the best overall efficiency.

In addition, applications may differ in their computation-to-communication ratio. More sophisticated strategies are used when communication performance dominates execution time.

The load balancing framework described in section 3 supports a wide variety of possible load balancing strategies. The load database provides all the mechanisms necessary for implementing any load balancing strategy (the "Load Balancer" in Fig. 2). The database provides information about all the objects on each processor to the representative of the load balancer on that processor. Thus, it is up to the load balancer to decide whether this information needs to be collected in a central place. A purely local strategy may utilize locally collected information to decide whether this processor is overloaded (or underloaded), and consequently migrate some of the objects out to a heuristically chosen remote processor, or request work from a remote processor. Another strategy may send load information to immediate neighbors in some virtual topology, and utilize information received from them to migrate its objects out to neighbors, when necessary.

We're currently identifying the space of possible load balancers and implementing them within the framework. The strategies we have implemented so far include:

1. A simple greedy strategy that ignores communication costs;

2. Some experimental variations of greedy strategies that take the communication costs into account

3. A "refinement" strategy that tries to retain objects on the current processors to the extent possible, A global reduction gets the load information to all processors, and then each overloaded processor finds a different set of underloaded processors, and migrates some ofits objects to them to bring their loads to the desired level.

4. A parallel branch and bound strategy that seeks an optimal solution, but can be set to run for a specified duration or until a specified improvement is attained. This strategy tends to supply a continuous stream of better solutions, and often reaches a close-to-optimal solution relatively early in the search.

5 Results

In this section, we describe two of the benchmarks used to evaluate the load balancing framework, and discuss the performance results obtained with the refinement load balancer.

The first program is a synthetic benchmark to emulate an iterative program with arbitrary work distribution and communication patterns. The program creates a number of objects, each of which sends messages to several other objects, waits for messages from other objects, and then performs a computation of some randomly-determined length. Next, the objects all go to a barrier and then repeat.

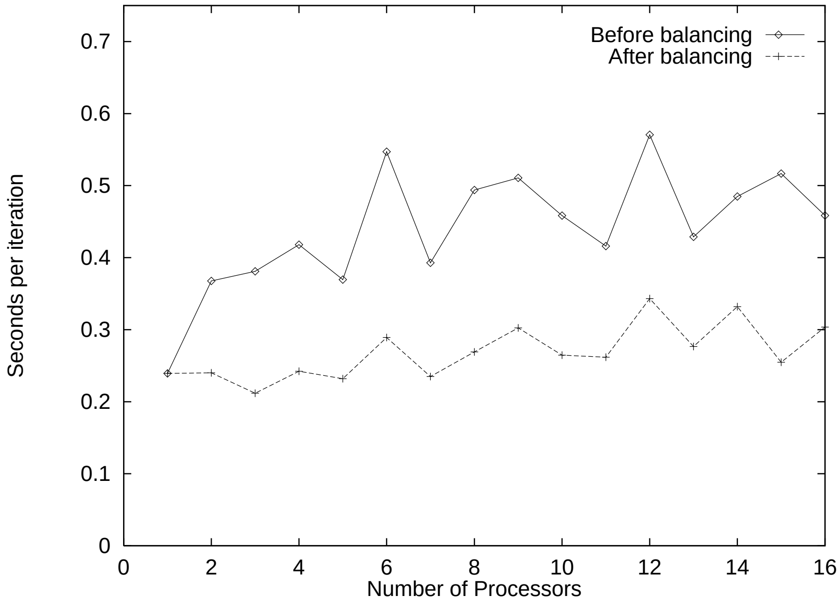

Figure 3: Seconds per iteration for the synthetic benchmark, where computation is scaled proportionally to the number of processors.

Fig. 3 shows the performance of the benchmark in which the problem is scaled so that each processor initially has eight objects, each with randomly-determined work loads. Since the total work increases in proportion to the number of processors, we expect the time (seconds per iteration) to increase with the number of processors, due to extra communication and load imbalance. After load balancing, the figure shows that the time per iteration is nearly constant, indicating that we have found a good load balance without significantly increasing communication overhead.

The speedup for a fixed problem size is shown in Fig. 4. Due to the synchronization in the benchmark, each iteration takes as long as the slowest processor to complete. As a result, the unbalanced speedup is limited. Since load balancing guarantees that no processor will be grossly overloaded, additional processors continue to be used effectively.

The second benchmark is intended to mimic the adaptive refinement technique in a simplified matter. Adaptive refinement is used in a variety of applications, including finite element and fluid flow studies. A portion of the simulation space, which has become either geometrically more complex or numerically more error prone, is discretized at a finer level of granularity in the middle of a parallel computation. The impact of such refinement on performance can be dramatic. The processors that hold the region being refined see their share of work increase considerably. Due to inherent data dependencies, the rest of the processors must wait for these processors to finish their work in each iteration, and the average processor utilization drops dramatically.

Our benchmark simulates heat conduction on a two-dimensional plate. Initially, the plate is discretized as a 1024×1024 array. In our parallel implementation, this array is decomposed into a 16×16 array of cells, each holding a 64×64 swath of the original array (a 66×66 array including space for the boundary ghost elements). The program performs the simulation using a simple Gauss-Jacobi iteration: the new value for each element is a simple function of the four neighboring

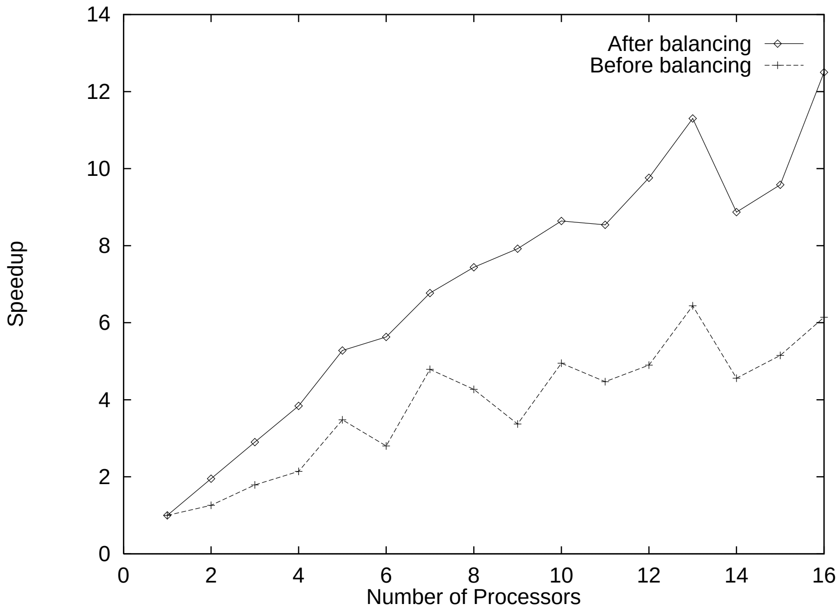

Figure 4: The speed-up of the synthetic benchmark for a fixed amount of work.

values. As a boundary condition, certain places on the plate are held at a higher temperature. Each cell is capable of refining itself on command (in a real application, any arbitrary region may be refined).

To experiment with this benchmark, we utilized an interactive application monitoring tool called Conspector. With Conspector, one can start a parallel application via the Web and monitor its performance. Moreover, one can interact with a running parallel application by injecting messages into it, and receiving data from it. Using Conspector, we injected messages to refine specific cells interactively, watched the performance impact on the utilization monitors, and then injected a rebalance message into the system. The results of this experiment are shown in Fig. 5.

As can be seen, after refinement the utilization drops to 75 percent from the initial value of close to one hundred percent. When the load balance is initiated, the system monitors and times individual objects for a few iterations, collecting data in the load database. After load balancing, the computation resumes, with the object array manager ensuring that messages are delivered to the array elements at their new destinations. The average processor utilization can be seen to return to over 95 percent.

The buttons for interactive refinement and interactive load balancing are used only for the purpose of demonstrating the utility of the framework. In real applications, refinements are triggered by changing conditions within the application, while the load monitoring (and the consequent triggering of the load balancing step) is conducted by the load balancer itself.

6 Summary

The object-based load balancing framework we have presented allows programmers to write applications without worrying about load balance, and permits the same load balancing algorithm to be

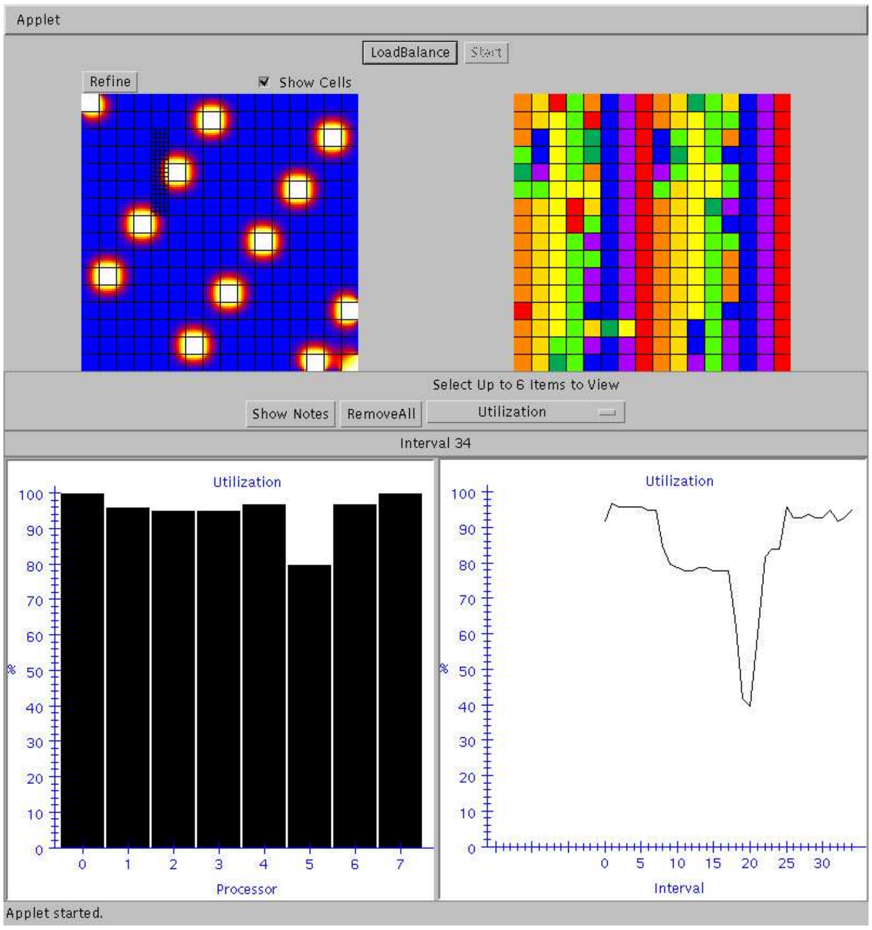

Figure 5: The Conspector interface to the Jacobi application.

used with a variety of applications. By providing a fixed framework based on object measurement, load balancers can treat many specifics of particular applications in a general manner, encouraging the creation of reusable load balancer libraries. We have taken advantage of the framework to create several applications and load balancers, and we present performance results for two of these applications.

References

[1] Gabriel Antoniu, Luc Bouge, and Raymond Namyst. An efficient and transparent thread migration scheme in the PM2 runtime system. Technical Report RR-3610, Inria, Institut National de Recherche en Informatique et en Automatique.

[2] A. Beguelin, E. Seligman, and M. Starkey. Dome: Distributed Object Migration Environment. Technical Report CMU-CS-94-153, School of Computer Science, Carnegie-Mellon University, May 1994.

[3] Robert K. Brunner and Laxmikant V. Kalé. Adapting to load on workstation clusters. In *The Seventh Symposium on the Frontiers of Massively Parallel Computation*, pages 106–112. IEEE Computer Society Press, February 1999.

[4] J. Casas, R. Konuru, S. W. Otto, R. Prouty, and J. Walpole. Adaptive load migration systems for PVM. In IEEE, editor, *Proceedings, Supercomputing '94: Washington, DC, November 14–18, 1994*, Supercomputing, pages 390–399, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1994. IEEE Computer Society Press.

[5] F. Douglis and J. Ousterhout. Transparent process migration: Design alternatives and the spr ite implementation. *Software - Practice And Experience*, 21(8):757–786, August 1991.

[6] N. Doulas and B. Ramkumar. Efficient Task Migration for Message-Driven Parallel Execution on Nonshared Memory Architectures. In *Proceedings of the International Conference on Parallel Processing*, August 1994.

[7] Hinkyung Hwang and Myong Soon Park. Adaptive load migration feasibility in a non-dedicated distributed system. In *Proceedings of the 10th International Conference on Information Networking, ICOIN-10*, pages 119 – 20, Jan 1996.

[8] L. V. Kale, Milind Bhandarkar, Robert Brunner, and Joshua Yelon. Multiparadigm, Multilingual Interoperability: Experience with Converse. In *Proceedings of 2nd Workshop on Runtime Systems for Parallel Programming (RTSPP) Orlando, Florida - USA*, Lecture Notes in Computer Science, March 1998.

[9] L. V. Kalé and Attila Gursoy. Modularity, reuse and efficiency with message-driven libraries. In *Proc. 27th Conference on Parallel Processing for Scientific Computing*, pages 738–743, February 1995.

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

[11] Laxmikant Kalé, Robert Skeel, Milind Bhandarkar, Robert Brunner, Attila Gursoy, Neal Krawetz, James Phillips, Aritomo Shinozaki, Krishnan Varadarajan, and Klaus Schulten. NAMD2: Greater scalability for parallel molecular dynamics. *Journal Computational Physics*, 1998. In press.

[12] Milind Bhandarkar L. V. Kale and Robert Brunner. Load balancing in parallel molecular dynamics. In *Fifth International Symposium on Solving Irregularly Structured Problems in Parallel*, August 1998.

[13] Mark Nelson, William Humphrey, Attila Gursoy, Andrew Dalke, Laxmikant Kalé, Robert D. Skeel, and Klaus Schulten. NAMD— A parallel, object-oriented molecular dynamics program. *Intl. J. Supercomput. Applics. High Performance Computing*, 10(4):251–268, Winter 1996.

[14] Balkrishna Ramkumar and Gopal Chillariga. Performance analysis of task migration in a portable parallel environment. In *Proceedings of the 1996 International Conference on Parallel Processing*, volume III, pages 191–198. IEEE Computer Society Press, August 1996.

[15] 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.

[16] 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.