# Run-time Support for Adaptive Load Balancing

Milind A. Bhandarkar, Robert K. Brunner, and Laxmikant V. Kalé

Parallel Programming Laboratory,
Department of Computer Science,
University of Illinois at Urbana-Champaign, USA
{milind,rbrunner,kale}@cs.uiuc.edu,
WWW home page: http://charm.cs.uiuc.edu/

**Abstract.** Many parallel scientific applications have dynamic and irregular computational structure. However, most such applications exhibit persistence of computational load and communication structure. This allows us to embed measurement-based automatic load balancing framework in run-time systems of parallel languages that are used to build such applications. In this paper, we describe such a framework built for the Converse [4] interoperable runtime system. This framework is composed of mechanisms for recording application performance data, a mechanism for object migration, and interfaces for plug-in load balancing strategy objects. Interfaces for strategy objects allow easy implementation of novel load balancing strategies that could use application characteristics on the entire machine, or only a local neighborhood. We present the performance of a few strategies on a synthetic benchmark and also the impact of automatic load balancing on an actual application.

## 1 Motivation and Related Work

An increasing number of emerging parallel applications exhibit dynamic and irregular computational structure. Irregularities may arise from modeling of complex geometries, and use of unstructured meshes, for example, while the dynamic behavior may result from adaptive refinements, and evolution of a physical simulation. Such behavior presents serious performance challenges. Load may be imbalanced to begin with due to irregularities, and imbalances may grow substantially with dynamic changes. We are participating in physical simulation projects at the Computational Science and Engineering centers of University of Illinois (Rocket simulation, and Simulation of Metal Solidification), where such behaviors are commonly encountered.

Load balancing is a fundamental problem in parallel computing, and a great deal of research has been done in this subject. However, a lot of this research is focussed on improving load balance of particular algorithms or applications. General purpose load balancing research deals mainly with process migration in operating systems and more recently in application frameworks. C++ libraries such as DOME [1] implement the data-parallel programming paradigm as distributed objects and allow migration of work in response to varying load conditions. Systems such as CARMI [10] simply notify the user program of the load

imbalance, and leave it to the application process to explicitly move its state to a new processor. Multithreaded systems such as $PM^2$ [9] require every thread to store its state in the specially allocated memory, so that the system can migrate the thread automatically. An object migration system called ELMO [3], built on top of Charm [6, 7], implements object migration mainly for fault-tolerance. Applications in areas such as VLSI, and Computational Fluid Dynamics (CFD) use graph partitioning programs such as METIS [8] to provide initial load balance. However, every such application has to specifically provide code for monitoring load imbalance and to invoke the load balancer periodically to deal with dynamic behavior.

We have developed an automatic measurement-based load balancing framework to facilitate high-performance implementations of such applications. The framework requires that a computation be partitioned into more pieces (typically implemented as objects) than there are processors, and letting the framework handle the placement of pieces. The framework relies on a "principle of persistence" that holds for most physical simulations: computational load and communication structure of (even dynamic) applications tends to persist over time. For example, even though the load of some object instance changes at adaptive refinement drastically, such events are infrequent, and the load remains relatively stable between such events.

The framework can be used to handle application-induced imbalances as well as external imbalances (such as those generated on a timeshared cluster). It cleanly separates runtime data-collection and object migration mechanisms into a distributed database, which allows optional *strategies* to plug in modularly to decide which objects to migrate where. This paper presents results obtained using our load balancing framework. We briefly describe the framework, then the strategies currently implemented and how they compare on a synthetic benchmark, and finally results on a crack-propagation application implemented using it.

## 2 Load Balancing Framework

Our framework [2] views a parallel application as a collection of computing objects which communicate with each other. Furthermore, these objects are assumed to exhibit temporal correlation in their computation and communication patterns, allowing effective measurement-based load balancing without application-specific knowledge.

The central component of the framework 1 is the load balancer distributed database, which coordinates load balancing activities. Whenever a method of a particular object runs, the time consumed by that object is recorded. Furthermore, whenever objects communicate, the database records information about the communication. This allows the database to form an object-communication graph, in which each node represents an object, with the computation time of that object as a weight, and each arc is a communication pathway representing

communication from one object to another object, recording number of messages and total volume of communication for each arc.

The design of Charm++ [5] offers several advantages for this kind of load balancing. First, parallel programs are composed of many coarse-grained objects, which represent convenient units of work for migration. Also, messages are directed to particular objects, not processors, so an object may be moved to a new location without informing other objects about the change; the run-time system handles the message delivery with forwarding. Furthermore, the message-driven design of Charm++ means that work is triggered by messages, which are dispatched by the run-time system. Therefore, the run-time knows which object is running at any particular time, so the CPU time and message traffic for each object can be deposited with the framework. Finally, the encapsulation of data within objects simplifies object migration.

However, the load balancing framework is not limited to Charm++ only. Any language implemented on top of Converse can utilize this framework. For this purpose, the framework does not interact with object instances directly. Instead, interaction between objects and the load balancing framework occurs through object managers. Object managers are parallel objects (with one instance on each processor) that are supplied by the language runtime system. Object managers are responsible for creation, destruction, and migration of language-specific objects. They also supply the load database coordinator with computational loads and communication information of the objects they manage. Object managers register the managed objects with the framework, and are responsible for mapping the framework-assigned system-wide unique object identifier to the language-specific identifier (such as thread-id in multithreaded systems, chare-id in Charm++, processor number in MPI etc.)

We have ported a CFD application written using Fortran 90 and MPI with minimal changes to use our framework using MPI library called ArrayMPI on top of the Converse runtime system. The ArrayMPI library allows an MPI program to create a number of virtual processors, implemented as Converse threads, which are mapped by the runtime system to available physical processors. The application program built using this MPI library then executes as if there are as many physical processors in the system as these virtual processors. The LB framework keeps track of computational load and communication graph of these virtual processors. Periodically, the MPI application transfers control to the load balancer using a special call `MPI_Migrate`, which allows the framework to invoke a load balancing strategy and to re-map these virtual processors to physical processors thus maintaining load balance.

## 3 Load Balancing Strategies

Load balancing strategies are a separate component of the framework. By separating the data collection code common to all strategies, we have simplified the development of novel strategies. For efficiency, each processor collects only a portion of the object-communication graph, that is, only the parts concerning

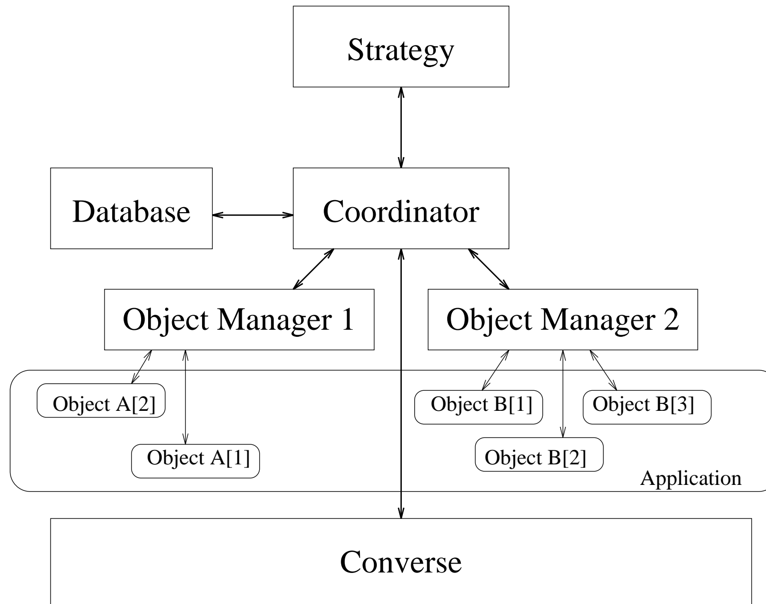


**Fig. 1.** Components of the load balancing framework on a processor.

local objects. This gives the strategy the freedom to ignore or locally analyze part of the graph (to minimize load-balancing overhead), or to collect the graph all in one place for a more thorough, centralized analysis. The strategy chooses a number of objects to migrate to improve program efficiency, and those decisions are handed back to the framework, which packs and migrates the objects to their new locations.

Once the run-time instrumentation has captured running times and communication graph, it is necessary to have a re-mapping strategy in place, which will attempt to produce an improved mapping. This is a multi-dimensional optimization problem, as it involves minimizing both the communication times and load-imbalances. Producing an optimal solution is not feasible, as it is an NP-hard problem. We have developed and experimented with several preliminary heuristic strategies, which we describe next.

**Greedy Strategy:** The simplest strategy is a greedy strategy. It organizes all objects in decreasing order of their computation times. All the processors are organized in a min-heap based on their assigned loads. The algorithm repeatedly selects the heaviest un-assigned object, and assigns it to the least loaded processor, updating the loads, and re-adjusting the heap. Although this strategy is capable of taking the communication costs into account while computing processor loads, it does not explicitly aim at minimizing communication. For $N$ objects, this strategy has the re-mapping complexity of $O(N \log N)$. Also, since

this strategy does not take into account the current assignments of objects, it may result in a large number of migration requests.

**Refinement Strategy:** The refinement strategy aims at minimizing the number of objects that need to be migrated, while improving load balance. It only considers the objects on overloaded processors. For each overloaded processor, the algorithm repeatedly moves one of its objects to an underloaded processor, until its load is below acceptable overload limit. Acceptable overload limit is a parameter specified to this strategy and may vary based on the overhead of migration. Typically this overload limit is between 1.02 and 1.05 which governs by what factor any processor may exceed the average load.

**Metis-based Strategy:** Metis [8] is a graph partitioning program and a library developed at University of Minnesota. It is mainly used for partitioning large structured or unstructured meshes. It provides several algorithms for graph-partitioning. The object communication graph that is obtained from the load balancing framework is presented to Metis in order to be partitioned onto the available number of processors. The objective of Metis is to find a reasonable load balance, while minimizing the edgecut, where edgecut is defined as the total weight of edges that cross the partitions, which in our case denotes number of messages across processors.

Figure 2 shows time taken per iteration of a synthetic benchmark when run with load balancing strategies described above. This benchmark consists of 32 objects with different loads and relatively low communication, initially mapped in a round-robin fashion to 8 processors. Load balancing is performed after every 500 iterations. All strategies improve performance, with Metis-based strategy leading to the best performance.

A load balancing strategy may improve performance of a parallel application, but if the load balancing step consumes more time than is gained by load redistribution, it may not be worthwhile. Today's parallel scientific applications run for hours. Thus it may be possible for the load balancers to spend more time in finding a better load distribution. All the three load balancing strategies described above take less than 0.5 seconds for load balancing 1024 objects on 8 processors. Thus a moderate decrease in time per iteration justifies use of any of these strategies. Also, owing to the principle of persistence, load balance deteriorates very slowly with drastic changes occurring very infrequently. Thus it may be possible to employ multiple strategies in such situations: One thorough load re-distribution in case of drastic changes, and a refinement strategy for slower load variations. We are currently experimenting with such combined strategies.

Also, note that all the strategies presented above take into consideration the application performance characteristics across all the processors. For ease of implementation, we used a global synchronizing barrier. Thus, all objects are made to temporarily stop computation while the load balancer re-maps them. However, this is usually not necessary. One can use a local barrier (barrier synchronization among objects on a single processor) for load database update, and another local barrier for performing load re-distribution, thus reducing the overheads associated with global synchronization. We are also implementing load

balancing strategies that take only a partial object communication graph (based on a few neighboring processors) into account.

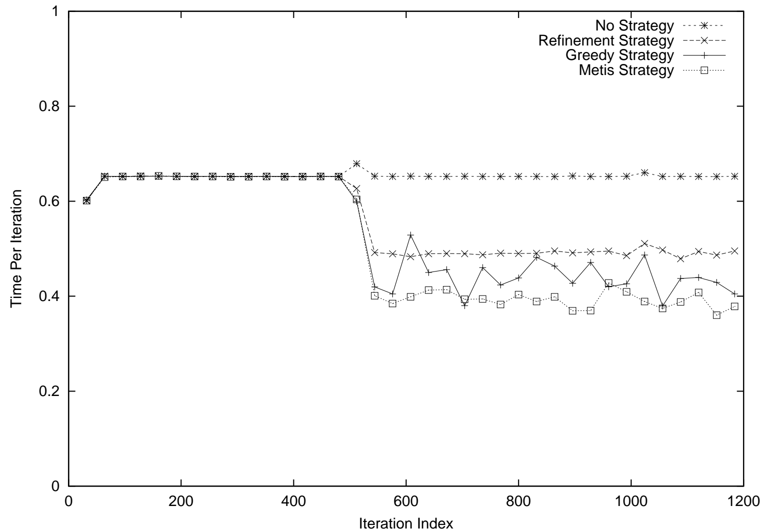


**Fig. 2.** Comparison of various load balancing strategies

## 4 Application Performance

In order to evaluate the framework, we implemented a Finite Element application that simulates pressure-driven crack propagation in structures. The physical domain is discretized into a finite set of triangular *elements*. Corners of these elements are called *nodes*. In each iteration, displacements are calculated at the nodes from forces contributed by surrounding elements. Typically, the number of elements is very large, and they are split into a number of *chunks* distributed across processors. In each iteration of simulation, forces on boundary nodes are communicated across chunks, where they are combined in, and new displacements are calculated. To detect a crack in the domain, more elements are inserted between some elements depending upon the forces exerted on the nodes. These added elements, which have zero volume, are called *cohesive* elements. At each iteration of the simulation, pressure exerted upon the solid structure may propagate cracks, and therefore more cohesive elements may have to be inserted. Thus, the amount of computation for some chunks may increase during the simulation. This results in severe load imbalance.

This application, originally written in sequential Fortran90, was converted to a C++-based FEM framework being developed by authors. This framework

presents a template library, which takes care of all the aspects of parallelization including communication and load balancing. The application developer simply provides the data members of the individual nodes and elements, and a function to calculate the values of local nodes, and a way to combine them.

Figure 3 presents results of automatic load balancing of the crack propagation simulation on 8 processors of SGI Origin2000. Immediately after the crack develops (between 10 and 15 seconds) in one of the chunks, the computational load of that chunk increases. Since the other chunks are dependent on node values from that chunk, they cannot proceed with computation until an iteration of the *heavy* chunk is finished. Thus, the number of iterations per second drops considerably. After this, the Metis-based load balancer is invoked twice (at 28 and 38 seconds). It uses the runtime load and communication information collected by the load database manager to migrate chunks from the overloaded processor to other processors, leading to improved performance. (In figure 3, this is apparent from increased number of iterations per second.)

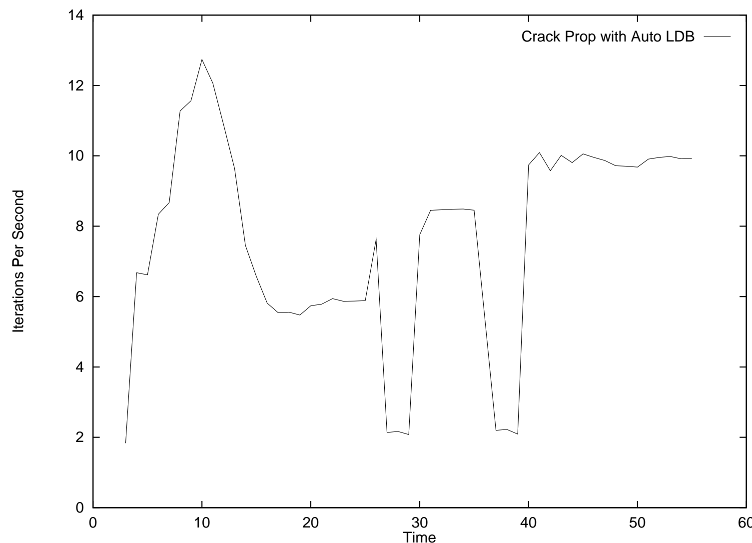


**Fig. 3.** Crack Propagation with Automatic Load Balancing. Finite Element Mesh consists of 183K nodes.

## 5 Conclusion

In this paper, we described a measurement-based automatic load balancing framework implemented in the Converse interoperable runtime system. This framework allows for easy implementation of novel load balancing strategies,

while automating the tasks of recording application performance characteristics as well as load redistribution. A few strategies have been implemented and their performance on a synthetic benchmark have been compared. A real finite element method application was ported to use our load balancing framework, and its performance improvement has been demonstrated. Based on the encouraging results with such real applications, we are currently engaged in developing a more comprehensive suite of load balancing strategies, and in determining suitability of different strategies for different kinds of applications.

## References


1. Jose Nagib Cotrim Arabe, Adam Beguelin, Bruce Lowekamp, Erik Seligman, Mike Starkey, and Peter Stephan. Dome: Parallel programming in a heterogeneous multi-user environment. Technical Report CS-95-137, Carnegie Mellon University, School of Computer Science, April 1995.
2. 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.
3. 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.
4. L. V. Kale, Milind Bhandarkar, Narain Jagathesan, Sanjeev Krishnan, and Joshua Yelon. Converse: An Interoperable Framework for Parallel Programming. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 212–217, April 1996.
5. L. V. Kale 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.
6. 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.
7. 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.
8. George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. TR 95-035, Computer Science Department, University of Minnesota, Minneapolis, MN 55414, May 1995.
9. R. Namyst and J.-F. Méhaut. $PM^2$: Parallel multithreaded machine. A computing environment for distributed architectures. In *Parallel Computing: State-of-the-Art and Perspectives, Proceedings of the Conference ParCo'95, 19-22 September 1995, Ghent, Belgium*, volume 11 of *Advances in Parallel Computing*, pages 279–285, Amsterdam, February 1996. Elsevier, North-Holland.
10. J. Pruyne and M. Livny. Parallel processing on dynamic resources with CARMI. *Lecture Notes in Computer Science*, 949:259–??, 1995.