Object-Based Adaptive Load Balancing for MPI Programs*

Milind Bhandarkar, L. V. Kalé, Eric de Sturler, and Jay Hoeflinger
Center for Simulation of Advanced Rockets
University of Illinois at Urbana-Champaign
{*milind,kale,sturler,hoefling*}*@cs.uiuc.edu*

October 6, 2000

Abstract

Parallel Computational Science and Engineering (CSE) applications often exhibit irregular structure and dynamic load patterns. Many such applications have been developed using procedural languages (e.g. Fortran) in message passing parallel programming paradigm (e.g. MPI) for distributed memory machines. Incorporating dynamic load balancing techniques at the application-level involves significant changes to the design and structure of applications. On the other hand, traditional run-time systems for MPI do not support dynamic load balancing. Object-based parallel programming languages, such as Charm++ support efficient dynamic load balancing using object migration for irregular and dynamic applications, as well as to deal with external factors that cause load imbalance. However, converting legacy MPI applications to such object-based paradigms is cumbersome. This paper describes an implementation of MPI, called Adaptive MPI (AMPI) that supports dynamic load balancing and multithreading for MPI applications. Our approach and implementation is based on the user-level migrating threads and load balancing capabilities provided by the Charm++ framework. Conversion from legacy codes to this platform is straightforward even for large legacy codes. We have converted the component codes ROCFLO and ROCSOLID of a Rocket Simulation application to AMPI. Our experience shows that with a minimal overhead and effort, one can incorporate dynamic load balancing capabilities in legacy Fortran-MPI codes.

1 Introduction

Many Computational Science and Engineering (CSE) applications under development today exhibit dynamic behavior. Computational domains are irregular to begin with, making it

*Research funded by the U.S. Department of Energy through the University of California under Subcontract number B341494.

difficult to subdivide the problem such that every partition has equal computational load, while optimizing communication. In addition to that, computational load requirements of each partition may vary as computation such as simulation of a complex system progresses. For example, applications that use Adaptive Mesh Refinement (AMR) techniques increase the resolution of spatial discretization in a few partitions, where interesting physical phenomena occur. This increases the computational load of those partitions drastically. As another example, in applications such as simulation of pressure-driven crack propagation using Finite Element Method (FEM), extra elements are inserted near the crack dynamically as it propagates through structures (such as the casing of solid fuel rockets), thus leading to severe load imbalance. Another type of dynamic load variance can be seen where heterogeneous computational platforms such as clusters of workstations are used to carry out even regular applications [BK99]. In such cases, the availability of individual workstations changes dynamically.

Such load imbalance can be reduced by decomposing the problem into several smaller partitions (much more than the available physical processors) and then mapping and remapping these partitions to physical processors in response to variation in load conditions. One cannot expect the application programmer to pay attention to dynamic variations in computational load and communication patterns, due to both internal and external factors described above, in addition to programming an already complex CSE application. Therefore, the parallel programming environment needs to provide for dynamic load balancing *under the hood*. Traditional runtime systems for message passing paradigms such as MPI do not allow efficient migration of tasks in order to provide dynamic load balancing capabilities. For the parallel programming environment to effectively load balance the application, it needs to know the precise load conditions at runtime. Thus, it needs to be supported by the runtime system of the parallel language. Also, it needs to predict the load patterns of the future based on current and past runtime conditions to provide an appropriate re-mapping of partitions.

Fortunately, an empirical observation of several such CSE applications suggests that such changes occur slowly over the life of a running application, thus leading to the *principle of persistent computation and communication structure* [KBB00]. Even when load changes are dramatic, such as in the case of adaptive refinement, they are infrequent. Therefore, by measuring variations of load and communication patterns, the runtime system can accurately forecast future load conditions, and can effectively load balance the application.

Charm++[KK96] is an object-oriented parallel programming language that provides dynamic load balancing capabilities using runtime measurements of computational loads and communication patterns, and employs object migration to achieve load balance. However, many CSE applications are written in languages such as Fortran, using MPI [GLS94] (Message Passing Interface) for communication. It can be very cumbersome to convert such legacy applications to newer paradigms such as Charm++ since the machine models of these paradigms are very different. Essentially, such attempts result in complete rewrite of applications.

Frameworks for computational steering and automatic resource management, such as AutoPilot [RVSR98], provide ways to instrument parallel programs for collecting load information at runtime, and a fuzzy-logic based decision engine that advises the parallel program

regarding resource management. But it is left to the parallel program to implement this advice. Thus, load balancing is not transparent to the parallel program, since the runtime system of the parallel language does not actively participate in carrying out the resource management decisions. Similarly, systems such as CARMI [PL95] simply inform the parallel program of load imbalance, and leave it to the application processes to explicitly move to a new processor. Multithreaded systems such as PM^2 [NM96] require every thread to store its state in specially allocated memory, so that the system can migrate the thread automatically. Load balancing strategies and instrumentation for performance monitoring are not integrated within PM^2, however. Other frameworks with automatic load balancing such as the FEM framework [BK00], and the framework for Adaptive Mesh Refinement codes [BN99] are specific to certain application domains, and do not apply to a general programming paradigm such as message-passing or to a general purpose messaging library such as MPI.

In this paper, we describe a path we have taken to solve the problem of load imbalance in existing Fortran90-MPI applications by using the dynamic load balancing capabilities of Charm++ with minimal effort. The next section describes the load-balancing framework of Charm++. Then we describe the multi-partitioning approach that is the basis of our work. In section 4, we describe the implementation of Adaptive MPI, which uses user-level migrating threads, along with message-driven objects. We show that it is indeed simple to convert existing MPI code to use AMPI. We discuss the methods used and efforts needed to convert actual application codes to use AMPI, and performance implications in section 5.

2 Charm++ Load Balancing Framework

Charm++ is a parallel object-oriented language. A Charm++ program consists of a set of medium-grained objects called *chares*. Chares are mapped to available processors by the message-driven runtime system of Charm++, called Converse [KBJ+96]. Communication between these chares is through asynchronous object-method invocations. These methods, called *entry* methods, when invoked, execute atomically, and cannot block waiting for messages. Powerful abstractions can be created by making chares members of arbitrarily indexed collections, such as multi-dimensional arrays of chares.

At the core of Charm++ is a message-driven scheduler that picks messages from a prioritized queue of pending messages, and schedules an object to execute the *entry method* of that object indicated in the message. Thus, the runtime system of Charm++ knows about the object it is executing methods on. It can track execution times (computational loads) of individual objects. Also, entry method execution is directed at objects, without having to know which processor it resides on. Therefore, the runtime system can gather communication patterns between objects.

Charm++ provides a dynamic load balancing (LB) framework, which gathers these load and communication data, and presents it as a distributed load database to load balancing strategies. Several such strategies are provided in Charm++ that can be plugged in at runtime into a Charm++ application. The load database can be viewed as a weighted communication graph, where the connected vertices represent communicating objects. Load balancing strategies produce a re-mapping of these objects in order to balance the load. This

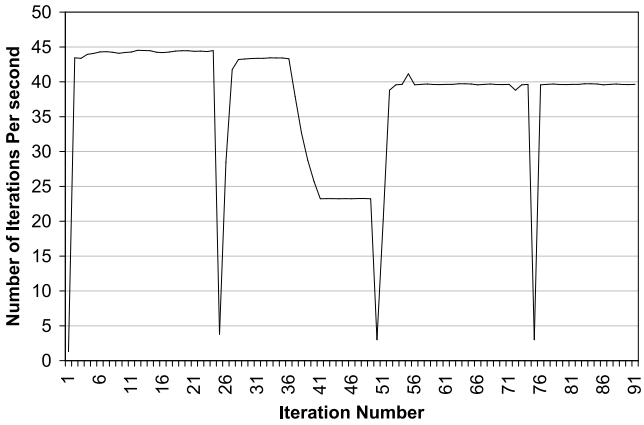

Figure 1: Performance of the Crack-Propagation application using Charm++ load-balancing framework. This experiment was performed on 8 processors of SGI Origin2000 at National Center for Supercomputing Applications (NCSA).

is an NP-hard multidimensional optimization problem, and producing optimal solution is not feasible. We have experimented with several heuristic strategies, and they have been shown to achieve good load balance [KBB00]. The new mapping produced by the LB strategy is communicated to the runtime system, which then invokes special serialization methods on objects to be migrated, and carries out the migration.

NAMD [KSB+99], a molecular dynamics application used routinely by biophysicists, is developed using Charm++. It combines spatial and functional decomposition. As simulation of a complex molecule progresses, atoms may move into neighboring partitions, and can lead to load imbalance. Charm++ LB framework is shown to be very effective in NAMD and has allowed NAMD to scale to thousands of processors achieving unprecedented speedups among molecular dynamics applications (1252 on 2000 processors). Another application that simulates pressure-driven crack propagation in structures has been implemented using the Charm++ LB framework [BK00], and has been shown to effectively deal with dynamically varying load conditions (Figure 1.) As a crack develops in a structure discretized as a finite element mesh, extra elements are added near the crack, resulting in severe load imbalance. Charm++ LB framework responds to this load imbalance by migrating objects, thus improving load balance, as can be seen from increased throughput measured in terms of number of iterations per second.

In order to use this LB framework, however, one needs to redesign the application to be *object-based*, and the entry methods of these objects must execute atomically. Converting parallel CSE applications written using MPI with blocking receives and/or blocking collective operations to use the load-balancing framework can be very cumbersome. In the next section

we describe the basic methodology used to adapt existing message-passing applications to use Charm++ LB framework.

3 Multi-partitioning Approach

The key to effectively using the Charm++ load-balancing framework is to split the computational domain into several small partitions, much more than the available physical processors. These smaller partitions, called virtual processors, (or *chunks*) are then mapped and re-mapped by the load-balancing framework in order to balance the load across physical processors. In message-driven object-based parallel programming paradigm, such as Charm++, chunks are implemented as objects. Communication between objects is accomplished with asynchronous remote method invocation. These methods execute atomically, i.e. they do not block waiting from messages. Since the Charm++ runtime system schedules objects to execute their entry methods, and routes remote method invocations to processors where the objects reside, chunks need not be aware of their physical location within the parallel system. Therefore, the runtime system is free to migrate these chunks to available physical processors.

Having more chunks to map and re-map results in better load balance. Large number of chunks can also result in better cache behavior because the resultant smaller partitions may utilize the cache better. Also, having more independent pieces of computation per processor results in better latency tolerance with computation/communication overlap. However, mapping several chunks on a single physical processor reduces the granularity of parallel tasks and the computation to communication ratio. Thus, chunks present a tradeoff in the overhead of virtualization and effective load balance. In order to study this tradeoff, we carried out an experiment using a Finite Element Method application that does structural simulation on an FEM mesh with 300000 elements. We ran this application on 8 processor Origin2000 (250 MHz MIPS R10000) with different number of partitions of the same mesh mapped to each processor. Results are presented in figure 2. It shows that increasing number of chunks is beneficial up to 16 chunks per physical processor. This increase in performance is caused by better cache behavior of smaller partitions, and overlap of computation and communication (latency tolerance). Further, the overhead introduced for 32 and 64 chunks per physical processor is very small. Though these numbers may vary depending on the application, we expect similar behavior for many applications that deal with large data sets and have near-neighbor communication.

3.1 Challenges for Existing MPI Applications

Converting the existing message-passing applications to the multi-partitioned approach is tricky. Such applications assume a distributed memory architecture, where each process contains data that can be accessed only by that process. Each process communicates with other process using point-to-point messages or with collective operations such as barriers, reductions, and broadcasts. When a process issues a receive request, it blocks the process until the requested message arrives from another process. If the process-to-processor mapping is one-to-one, blocking receives typically waste processor cycles on a dedicated machine.

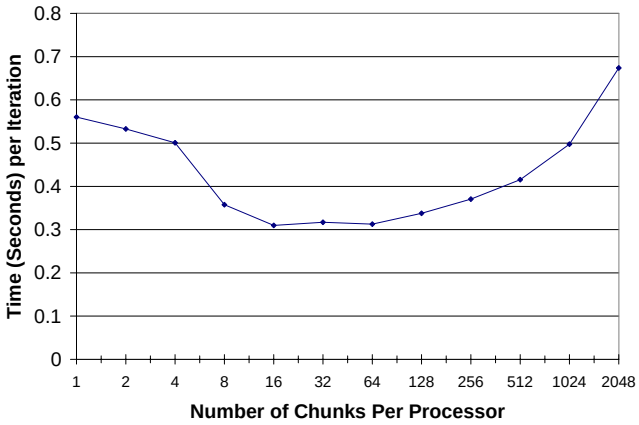

Figure 2: Effects of multi-partitioning in an FEM application.

When such mapping is many-to-one, the operating system can schedule another process to run instead of the blocked process, but such inter-process context switch is costly. Application programmers use non-blocking sends and receives to overlap communication with useful computation, but it requires careful tuning of applications to maximize such overlap. In any case, such techniques do not perform well, when the computational load dynamically varies, because the amount of overlap is typically determined by the useful work within a process, which may vary during the lifetime of the running process.

In order to convert the legacy MPI codes with blocking operations, we need to find locations in the program where it performs blocking operations (receives, barriers, and other collective operations.) At each of these locations, we need to break the flow of execution, so as to ensure atomicity of entry methods of chunks. We can achieve this by splitting the program or subroutine at the blocking call into two separate subroutines. In the first subroutine, we replace the blocking receive call by non-blocking receive, and specify the second subroutine as a continuation to the runtime system when the non-blocking receive is complete.

While this approach allows us to port our legacy MPI codes to take advantage of the Charm++ LB framework, it alters the structure of the program considerably. This is especially true if there are many blocking operations requiring us to split many subroutines. Also, if the blocking operation is performed in a subroutine, then that subroutine, as well as all its callers should be split. This has to be done recursively up to the outermost scope. This can lead to rise in the complexity of the resulting code (especially, when the communication is performed deep in the subroutine call hierarchy, or if such calls are made conditionally.)

Our Adaptive MPI implementation uses user-level threads, and allows blocking receives, so that with a slight overhead, MPI codes can use Charm++ LB framework with efforts less

than that of the original multi-partitioning approach.

4 Adaptive MPI

The main disadvantage of the message-driven multi-partitioning approach described earlier is that it significantly alters the original program structure, by forcing the use of non-blocking sends and receives. Also, since subroutines are split, the local variables in the original subroutines are not retained in the new program structure, and have to be made part of the dynamically allocated per-chunk data. This may disable some compiler optimizations. Adaptive MPI avoids such modification to the program structure, at the cost of potential overhead of context switching and migration.

AMPI implements chunks as user-level threads so as to enable them to issue blocking receives. Alternatives are to use processes or kernel-level threads. However, process context switching is costly, because it means switching the page table, and flushing out cache-lines etc. Process migration is also costlier than thread migration, because it will serialize entire core image of a process, rather than only the useful part. Kernel-threads are typically preemptive, thus accessing any shared variable would mean use of locks or mutexes, thus increasing the overhead. Also, one needs OS support for migrating kernel threads from one process to another. With user-level threads, one has complete control over scheduling, and also can maintain information on the communication pattern among chunks, and their computational and memory requirements. Entire collection of chunks is implemented as a Charm++ array of objects, where each object has a user-level thread associated with it. All communication primitives of MPI, both blocking and non-blocking, have been implemented in AMPI.

Since multiple chunks are mapped to one processor, data that were originally processor-private are now shared among the chunks. Thus, to convert an MPI program to use AMPI, we need to localize these data. One can do this by clubbing together the processor-private data into a user-defined type, and by making a variable of this type a local variable of the main subroutine, so that it resides on the thread's stack. If the processor-private data is too large to be accommodated on the thread's stack, one can dynamically allocate data for each chunk. These data are then registered with the runtime system, which makes them available to each chunk in each subroutine. Thus, references to global variables have to be changed to an indirect reference via the registered chunk data pointer. One also has to write serialization subroutines that will be called by the runtime system, when it decides to migrate a chunk to another processor for load balancing.

It is however tricky to migrate user-level threads by making sure that any references to the stack are valid after migration to different processors. Note that a chunk may migrate anytime when it is blocking for messages. At that time, if the thread's local variables refer to other local variables, these references may not be valid on another processor, because the stack may be located in a different location in memory. Thus, we need a mechanism for making sure that these references remain valid across processors. In the absence of any compiler support, this means that the thread-stacks should span the same range of virtual addresses on any processor where it may migrate.

Our preliminary implementation of migratable threads was based on a stack-copy mechanism, where contents of the thread-stack were copied at every context-switch between two threads. Thus, all threads execute with the same stack and refer to valid addresses even after migration (assuming that the main process stack on all processors begins at the same virtual address.) This has two drawbacks. First, it is inefficient because of the copy overhead on every context-switch (figure 3), and second, one thread cannot access another thread's local variables, thus making it mandatory to copy it to some shared location. Since the user program is written such that a process does not access other process's variables directly, the second restriction does not impact the application programmer. In order to ensure efficiency with this mechanism, it is recommended to keep the stack size as low as possible at the time of a context-switch.

Our current implementation of migratable threads uses a new scalable variant of the *isomalloc* functionality of PM^2 [ABN99]. In this implementation, each thread's stack is allocated such that it spans the same reserved virtual addresses across all the processors. This is achieved by splitting the unused virtual address space among physical processors. When a thread is created, its stack is allocated from a portion of the virtual address space assigned to the creating processor. This ensures that no thread encroaches upon addresses spanned by others' stacks on any processor. Allocation and deallocation within the assigned portion of virtual address space is done using the `mmap` and `munmap` functionality of Unix. Since we use isomalloc for fixed size thread stacks only, we can eliminate several overheads associated with PM^2 implementation of isomalloc. This results in context-switching overheads as low as non-migrating threads, irrespective of the stack-size, while allowing migration of threads. However, it is still more efficient to keep the stack size down at the time of migration to reduce the thread migration overhead.

4.1 Conversion to AMPI

In order to convert existing MPI programs to AMPI, one has to make sure that variables global in scope (such as common blocks, global variables etc) are not defined before and used after a blocking call, such as `MPI_Recv`. The reason for this restriction is straightforward. If a global variable is defined before a blocking call, it may be modified by another user-level thread when the defining thread blocks and another thread is scheduled. Thus, after returning from a blocked MPI call, the original thread will see a different value of that variable. Careful inspection of the program may reveal such variables. In order to use AMPI, such variables should be localized, i.e. copied to variables local to the subroutines, which are on stack.

However, sometimes such careful inspection may not be possible. In that case, we have devised a method to systematically put all the global variables in a private area allocated dynamically or on thread's stack. The idea is to make a user-defined type, and make all the global variables members of that type. In the main program, we allocate a variable of that type, and then pass a pointer to that variable to every subroutine that makes use of global variables. Access to the previously global variables in such subroutines should be made through this pointer. A simple source-to-source translator can recognize all global variables and automatically make such modifications to the program. We are currently working on

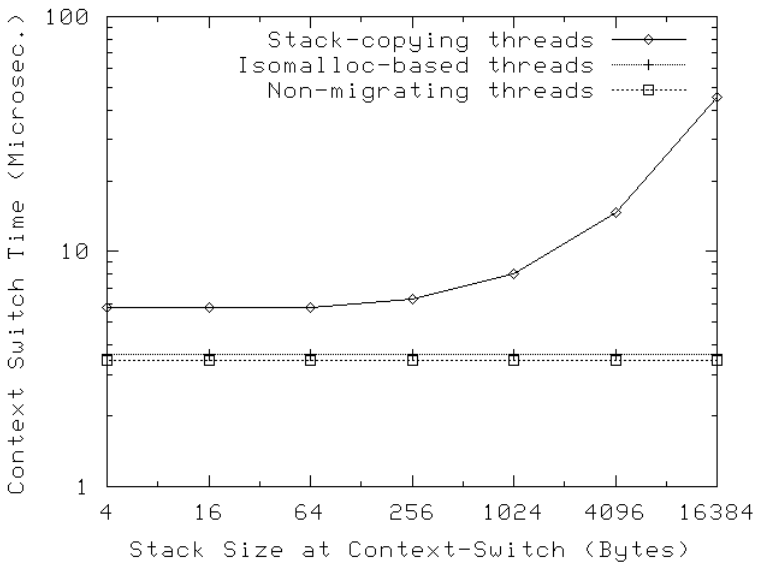

Figure 3: Comparison of context-switching times of stack-copying and isomalloc-based migrating threads with non-migrating threads. This experiment was performed on NCSA Origin2000, with 250 MHz MIPS R10000 processor.

modifying the front-end of a parallelizing compiler [BEF+94] to incorporate this translation. However, currently, this has to be done by hand.

We have converted some large MPI applications using this approach. The techniques used, efforts involved, and preliminary performance data are given in the next section.

5 Case Studies

We have compared AMPI with the original message-driven multi-partitioning approach to evaluate overheads associated with each of them using a typical Computational Fluid Dynamics (CFD) kernel that performs Jacobi relaxation on large grids (where each partition contains 1000 grid points.) We ran this application on a single 250 MHz MIPS R10000 processor, with different number of chunks, keeping the chunk-size constant. Two different decompositions, 1-D and 3-D, were used. These decompositions vary in number of context-switches (blocking receives) per chunk. While the 1-D chunks have 2 blocking receive calls per chunk per iteration, the 3-D chunks have 6 blocking receive calls per chunk per iteration. However, in both cases, only half of these calls actually block waiting for data, resulting in 1 and 3 context switches per chunk per iteration respectively. As can be seen from figure 4, the optimization due to availability of local variables across blocking calls, as well as larger subroutines in the AMPI version neutralizes thread context-switching overheads for a reasonable number of chunks per processor. Thus, the load balancing framework can be effectively used with user-level threads without incurring any significant overheads.

Encouraged by these results, we converted some large MPI applications using AMPI as

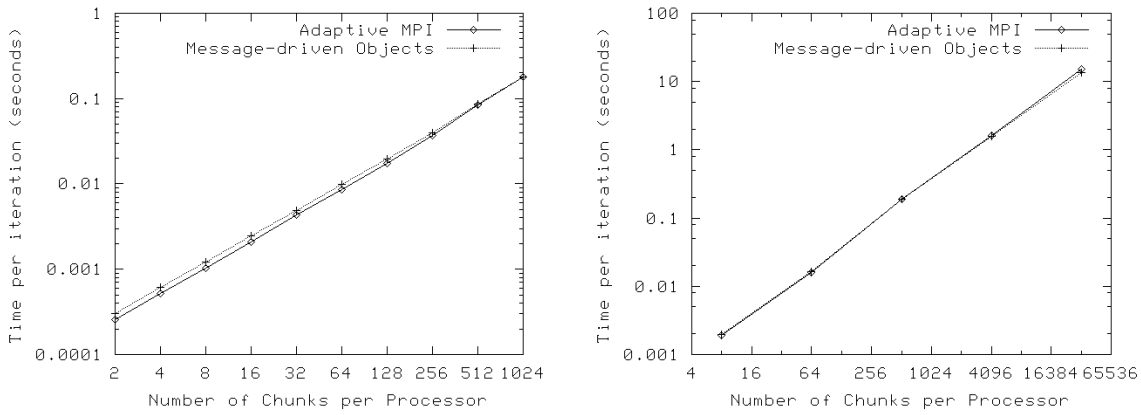

Figure 4: The throughput (number of iterations per second) for a Jacobi relaxation application. (Left) with 1-D decomposition. (Right) with 3-D decomposition.

No. of Processors	ROCFLO MPI(sec.)	ROCFLO AMPI(sec.)
1	1637.55	1679.91
2	957.94	916.73
4	450.13	437.64
8	234.90	278.93
16	142.49	126.59
32	61.21	63.82

Table 1: Comparison of MPI and AMPI versions of ROCFLO.

part of the Center for Simulation of Advanced Rockets (CSAR) at University of Illinois. The goal of CSAR is to produce a detailed multi-physics rocket simulation and virtual prototyping tool [HD98]. GEN1, a first generation integrated simulation code is composed of three coupled modules: ROCFLO (an explicit fluid dynamics code), ROCSOLID (an implicit structural simulation code), and ROCFACE (a parallel interface between ROCFLO and ROCSOLID) [PAN+99]. ROCFACE and ROCSOLID have been written using Fortran 90 (about 10000 and 12000 lines respectively), and use MPI as parallel environment. We converted each of these codes to AMPI. This conversion, using the techniques described in the last section, resulted in very few changes to original code (In fact, the changed codes can be made to run with original MPI, with about 10 preprocessor directives), and did not take much time for authors of this paper, who were unfamiliar with the codes (about a week for one person for each of these codes.) In addition, the overhead of using AMPI instead of MPI is shown (tables 1 and 2) to be minimal, even with the original decomposition of one partition per processor. We expect the performance of AMPI to be better when multiple partitions are mapped per processor, as depicted in figure 2. Also, the ability of AMPI to respond to dynamic load variations outweighs these overheads.

No. of Processors	ROCSOLID MPI(sec.)	ROCSOLID AMPI(sec.)
1	67.19	63.42
8	69.81	71.09
32	70.70	69.99
64	73.94	75.47

Table 2: Comparison of MPI and AMPI versions of ROCSOLID. Note that this is a *scaled* problem.

6 Conclusion

Efficient implementations of an increasing number of dynamic and irregular computational science and engineering applications require dynamic load balancing. Many such applications have been written in procedural languages such as Fortran with message-passing parallel programming paradigm. Traditional implementation of message-passing libraries such as MPI do not support dynamic load balancing. Charm++ parallel programming environment supports dynamic load balancing using object-migration. Applications developed using Charm++ have been shown to adaptively balance load in presence of dynamically changing load conditions caused even by factors external to the application, such as in time-shared clusters of workstations. However, converting existing procedural message-passing codes to use object-based Charm++ can be cumbersome. We have developed Adaptive MPI, an implementation of MPI based on message-driven object-based runtime system, and user-level threads, that run existing MPI applications with minimal change, and insignificant overhead. Conversion of legacy MPI programs to Adaptive MPI does not need significant changes to the original code structure; the changes that are needed are mechanical and can be fully automated. We have converted two large scientific applications to use Adaptive MPI and the dynamic load-balancing framework, and have shown that for these applications, the overhead of AMPI, if any, is very small. We are currently working on reducing the messaging overhead of AMPI, and also automating the code conversion methods.

References

[ABN99] Gabriel Antoniu, Luc Bouge, and Raymond Namyst. An efficient and transparent thread migration scheme in the pm2 runtime system. In *Proc. 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP) San Juan, Puerto Rico, Held in conjunction with the 13th Intl Parallel Processing Symp. (IPPS/SPDP 1999), IEEE/ACM. Lecture Notes in Computer Science 1586*, pages 496–510. Springer-Verlag, April 1999.

[BEF+94] William Blume, Rudolf Eigenmann, Keith Faigin, John Grout, Jay Hoeflinger, David Padua, Paul Petersen, Bill Pottenger, Lawrence Rauchwerger, Peng Tu, and Stephen Weatherford. Polaris: Improving the effectiveness of parallelizing compilers. In *Proceedings of 7th International Workshop on Languages and Com-*

pilers for Parallel Computing, number 892 in Lecture Notes in Computer Science, pages 141–154, Ithaca, NY, USA, August 1994. Springer-Verlag.

[BK99] 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.

[BK00] Milind Bhandarkar and L. V. Kalé. A parallel framework for explicit fem. Technical Report 00-01, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign (Submitted to HiPC 2000), May 2000.

[BN99] D. S. Balsara and C. D. Norton. Innovative language-based and object-oriented structured amr using fortran 90 and openmp. In Y. Deng, O. Yasar, and M. Leuze, editors, *New Trends in High Performance Computing Conference (HPCU'99)*, Stony Brook, NY, August 1999.

[GLS94] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 1994.

[HD98] M. T. Heath and W. A. Dick. Virtual rocketry: Rocket science meets computer science. *IEEE Comptational Science and Engineering*, 5(1):16–26, 1998.

[KBB00] L. V. Kale, Milind Bhandarkar, and Robert Brunner. Run-time Support for Adaptive Load Balancing. In *Proceedings of 4th Workshop on Runtime Systems for Parallel Programming (RTSPP) Cancun - Mexico*, March 2000.

[KBJ+96] 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.

[KK96] 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.

[KSB+99] 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 of Computational Physics*, 151:283–312, 1999.

[NM96] R. Namyst and J.-F. Méhaut. PM^2: Parallel multithreaded machine. A computing environment for distributed architectures. In E. H. D'Hollander, G. R. Joubert, F. J. Peters, and D. Trystram, editors, *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.

[PAN+99] I. D. Parsons, P. V. S. Alavilli, A. Namazifard, J. Hales, A. Acharya, F. Najjar, D. Tafti, and X. Jiao. Loosely coupled simulation of solid rocket moters. In *Fifth National Congress on Computational Mechanics*, Boulder, Colorado, August 1999.

[PL95] J. Pruyne and M. Livny. Parallel processing on dynamic resources with CARMI. *Lecture Notes in Computer Science*, 949, 1995.

[RVSR98] Randy L. Ribler, Jeffrey S. Vetter, Huseyin Simitci, and Daniel A. Reed. Autopilot: Adaptive Control of Distributed Applications. In *Proc. 7th IEEE Symp. on High Performance Distributed Computing*, Chicago, IL, July 1998.