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
{bhandark,l-kale1,sturler,hoefling}@uiuc.edu

Abstract. Parallel Computational Science and Engineering (CSE) applications often exhibit irregular structure and dynamic load patterns. Many such applications have been developed using MPI. 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. 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 for MPI applications. Conversion from MPI to this platform is straightforward even for large legacy codes. We describe our positive experience in converting the component codes ROCFLO and ROCSOLID of a Rocket Simulation application to AMPI.

1 Introduction

Many Computational Science and Engineering (CSE) applications under development today exhibit dynamic behavior. Computational domains are irregular to begin with, making it difficult to subdivide the problem such that every partition has equal computational load, while optimizing communication. In addition, computational load requirements of each partition may vary as computation 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. In applications such as the simulation of pressure-driven crack propagation using Finite Element Method (FEM), extra elements are inserted near the crack dynamically as it propagates through structures, thus leading to severe load imbalance. Another type of dynamic load variance can be seen where non-dedicated 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.

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

Such load imbalance can be reduced by decomposing the problem into several smaller partitions (many more than the available physical processors) and then mapping and re-mapping 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, in addition to programming an already complex CSE application. Therefore, the parallel programming environment needs to provide for dynamic load balancing *under the hood*. To do this effectively, 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 future load patterns 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 (Message Passing Interface) [GLS94] 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. 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. TOMPI [Dem97] and TMPI [TSY99] are thread-level implementations of MPI. The techniques they use to convert legacy MPI codes to run on their implementations are similar to our approach. However, they do not support FORTRAN. Also, they do not provide automated dynamic load balancing. TOMPI is a single processor simulation tool for MPI programs, while TMPI attempts to implement MPI efficiently on shared memory multiprocessors.

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++ and its multi-partitioning approach. In section 3, 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 4.

2 Charm++ Multi-partitioned Approach

Charm++ is a parallel object-oriented language. A Charm++ program consists of a set of communicating objects. These objects are mapped to available processors by the message-driven runtime system of Charm++. Communication between Charm++ objects is through asynchronous object-method invocations. Charm++ object-methods (called *entry* methods) are atomic. Once invoked, they complete without waiting for more data (such as by issuing a blocking receive). Charm++ tracks execution times (computational loads) and communication patterns of individual objects. Also, method execution is directed at objects, not processors. Therefore, the runtime system can migrate objects transparently.

Charm++ incorporates a dynamic load balancing (LB) framework, that acts as a gateway between the Charm++ runtime system and several "plug-in" load balancing strategies. The object-load and communication data be viewed as a weighted communication graph, where the connected vertices represent communicating objects. Load balancing strategies produce a new mapping of these objects in order to balance the load. This 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 carries out object migrations.

NAMD [KSB+99], a molecular dynamics application, is developed using Charm++. As simulation of a complex molecule progresses, atoms may move into neighboring partitions. This leads 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.

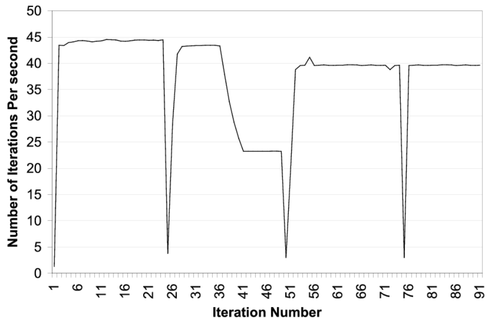

Fig. 1. Performance of the Crack-Propagation application using Charm++ load-balancing framework.

The key to effectively using the Charm++ LB framework is to split the computational domain into several small partitions. These smaller partitions, called virtual processors, (or *chunks*) are then mapped and re-mapped by the LB framework in order to balance the load across physical processors.

Having more chunks to map and re-map results in better load balance. Large number of chunks result in smaller partitions that 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, multi-partitioning 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 300K elements. We ran this application on 8 processor Origin2000 (250 MHz MIPS R10K) 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, as number of elements per chunk decreases from 300K to about 20 K. 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 (with 10K and 5K elements per chunk, respectively) 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.

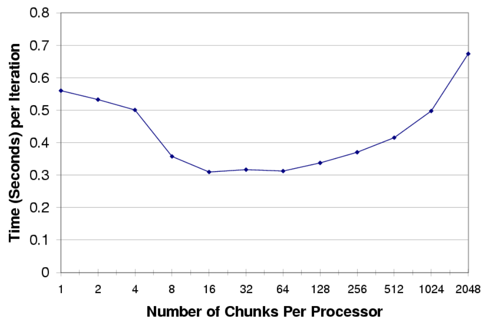

Fig. 2. Effects of multi-partitioning in an FEM application.

One way to convert existing MPI applications to the multi-partitioned approach is to represent an MPI process by a Charm++ object. However, this is not trivial, since MPI processes contain blocking receives and collective operations that do not satisfy the atomicity requirements of entry methods.

3 Adaptive MPI

AMPI implements MPI processes with 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. Kernel-threads are typically preemptive. 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 one can track the communication pattern among chunks, as well as their computational and memory requirements.

It is difficult to migrate threads because any references to the stack have to be valid after migration to a different address space. 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 on the stack, these references may not be valid upon migration, 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 first solution to this problem was based on a stack-copy mechanism, where all threads execute with the same system stack, and contents of the thread-stack were copied at every context-switch between two threads. If the system stack is located at the same virtual address on all processors, then the stack references will remain valid even after migration. This scheme's main drawback is the copy overhead on every context-switch (figure 3). In order to increase efficiency with this mechanism, one has to keep the stack size as low as possible at the time of a context-switch.

Our current implementation of migratable threads uses a variant of the *iso-malloc* 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 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 overheads.

3.1 Conversion to AMPI

Since multiple threads may be resident within a process, variables that were originally process-private will now be shared among the threads. Thus, to convert an MPI program to use AMPI, we need to privatize these global variables, which typically fall in three classes.

1. Global variables that are "read-only". These are either *parameters* that are set at compile-time, or other variables that are read as input or set at the beginning of the program and do not change during execution. It is not necessary to privatize such variables.
2. Global variables that are used as temporary buffers. These are variables that are used temporarily to store values to be accessible across subroutines. These variables have a characteristic that there is no blocking call such as `MPI_recv` between the time the variable is set and the time it is ever used. It is not necessary to privatize such variables either.
3. True global variables. These are used across subroutines that contain blocking receives and therefore there is a possibility of a context-switch between the definition and use of the variable. These variables need to be privatized.

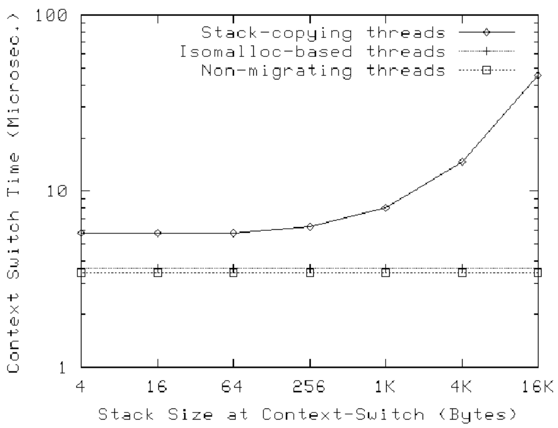

Fig. 3. Comparison of context-switching times of stack-copying and isomalloc-based migrating threads with non-migrating threads.

To systematically privatize all global variables, we create a FORTRAN 90 type, and make all the global variables members of that type. In the main program, we allocate an instance of that type, and then pass a pointer to that instance to every subroutine that makes use of global variables. Access to the members of this type have to be made through this pointer.

A source-to-source translator can recognize all global variables and automatically make such modifications to the program. We are currently working on modifying the front-end of a parallelizing compiler [BEF+94] to incorporate this translation. However, currently, this has to be done by hand. The privatization requirement is not unique to AMPI. Other thread-based implementations of MPI such as TMPI [TSY99] and TOMPI [Dem97] also need such privatization.

4 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 R10K 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.

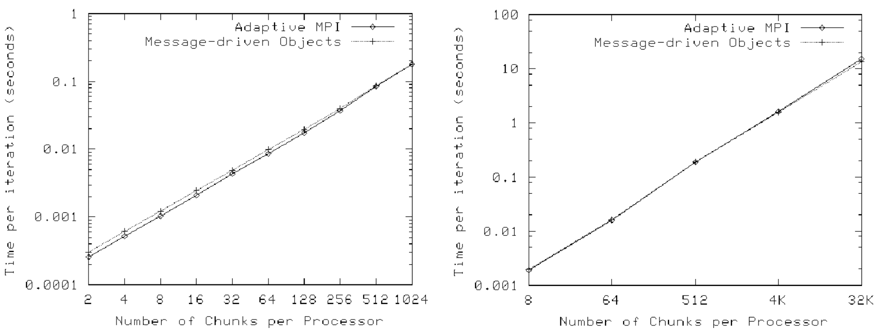

Fig. 4. Performance a Jacobi relaxation application. (Left) with 1-D decomposition. (Right) with 3-D decomposition.

Encouraged by these results, we converted some large MPI applications developed as 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 [HD98]. GEN1, the 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-MPI (about 10K and 12K lines respectively.) 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. The converted codes can still link and run with original MPI. In addition, the overhead of using AMPI instead of MPI is shown (table 1) 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, similar to the situation depicted in figure 2. Also, the ability of AMPI to respond to dynamic load variations outweighs these overheads.

5 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 FOR-

No. Of	ROCFLO		ROCSOLID	
Processors	MPI	AMPI	MPI	AMPI
1	1637.55	1679.91	67.19	63.42
2	957.94	916.73	–	–
4	450.13	437.64	–	–
8	234.90	278.93	69.81	71.09
16	142.49	126.59	–	–
32	61.21	63.82	70.70	69.99
64	–	–	73.94	75.47

Table 1. Comparison of MPI and AMPI versions of ROCFLO (Fixed problem size) and ROCSOLID (Scaled problem size). All timings are in seconds.

TRAN 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 timeshared 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 on a 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 PM^2 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 Compilers 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. In *Proceedings of the International Conference on High Performance Computing*, Bangalore, India, December 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.

[Dem97] Erik D. Demaine. A Threads-Only MPI Implementation for the Development of Parallel Programs. In *Proceedings of the 11th International Symposium on High Performance Computing Systems (HPCS'97), Winnipeg, Manitoba, Canada*, pages 153–163, July 1997.

[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.

[KK96] Laxmikant V. Kale and Sanjeev Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In Gregory V. Wilson and Paul Lu, editor, *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.

[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.

[TSY99] H. Tang, K. Shen, and T. Yang. Compile/Run-time Support for Threaded MPI Execution on Multiprogrammed Shared Memory Machines. In *Proceedings of 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99)*, 1999.