Performance Evaluation of Adaptive MPI *

Chao Huang Gengbin Zheng
Laxmikant Kalé
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
{chuang10,gzheng,kale}@cs.uiuc.edu

Sameer Kumar
IBM T. J. Watson Research Center
Yorktown Heights, NY 10598, USA
sameerk@us.ibm.com

Abstract

Processor virtualization via migratable objects is a powerful technique that enables the runtime system to carry out intelligent adaptive optimizations like dynamic resource management. CHARM++ is an early language/system that supports migratable objects. This paper describes *Adaptive MPI* (or *AMPI*), an MPI implementation and extension, that supports processor virtualization. AMPI implements virtual MPI processes (VPs), several of which may be mapped to a single physical processor. AMPI includes a powerful runtime support system that takes advantage of the degree of freedom afforded by allowing it to assign VPs onto processors. With this runtime system, AMPI supports such features as automatic adaptive overlapping of communication and computation, automatic load balancing, flexibility of running on arbitrary number of processors, and checkpoint/restart support. It also inherits communication optimization from CHARM++ framework. This paper describes AMPI, illustrates its performance benefits through a series of benchmarks, and shows that AMPI is a portable and mature MPI implementation that offers various performance benefits to dynamic applications.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: Parallel programming

General Terms Performance, Experimentation, Languages

Keywords MPI, Adaptivity, Processor Virtualization, Load Balancing, Communication Optimization

* This work was supported in part by DOE Grant B341494 and B505214, and by the National Science Foundation through Grant ITR 0205611 and TeraGrid resources at NCSA and Terascale Computing System at the Pittsburgh Supercomputing Center.

PPoPP'06 March 29–31, 2006, New York, New York, USA.

1. Introduction

The new generation of parallel applications are complex, involve simulation of dynamically varying systems, and use adaptive techniques such as multiple timestepping and adaptive refinements. Typical implementations of the MPI do not support the dynamic nature of these applications well. As a result, programming productivity and parallel efficiency suffer. In this paper we present performance evaluation of Adaptive MPI (AMPI), an adaptive implementation of MPI. Through analysis of the results from a series of benchmarks, we illustrate that AMPI, while still retaining the familiar programming model of MPI, is better suited for such new generation applications, and does not penalize performance of those applications without the dynamic nature.

The key concept behind AMPI is processor virtualization. Standard MPI programs divide the computation onto P processes, and typical MPI implementations simply execute each process on one of the P processors. In contrast, an AMPI programmer divides the computation into a number V of virtual processors (VPs), and AMPI runtime system maps these VPs onto P physical processors. In other words, AMPI provides an effective division of labor between the programmer and the system. The programmer still programs each process with the same syntax as specified in the MPI Standard. Further, not being restricted by the physical processors, he/she is able to design more flexible partitioning that best fits the nature of the parallel problem. The runtime system, on the other hand, has the opportunity of adaptively mapping and re-mapping the programmer's virtual processors onto the physical machine.

In AMPI, the MPI processes are implemented by user level threads embedded in migratable parallel objects, many of which can be mapped onto one physical processor. The number of virtual processors V and the number of physical processors P are independent, allowing the programmer to design more natural expression of the algorithm. For example, algorithmic considerations often restrict the number of processors to a power of 2, or a cube, and with AMPI, V can still be a cube even though P is prime. When $V = P$, the program executes the same way it would with other MPI im-

plementation, and it enjoys only part of the benefit of AMPI, such as collective communication optimization. To take full advantage of the AMPI runtime system, typically we have V significantly larger than P. Before describing the details for design and implementation of AMPI and the underlying CHARM++ Framework, we first motivate AMPI by explaining the benefits of using multiple virtual processors per physical processor.

1.1 Benefits of Virtualization

In [1], the authors have discussed in detail the benefits of processor virtualization in parallel programming. The CHARM++ system has indeed taken full advantage of these productivity benefits. AMPI inherits most of the merits from CHARM++, while furnishing the common MPI programming environment. The following is a list of the benefits that we will demonstrate in this paper. We will show that AMPI, with these benefits, effectively improves the performance of complex and dynamic parallel programs with virtualization, and incurs very little overhead for applications without the dynamic nature.

Adaptive overlapping of communication and computation: If one of the virtual processors is blocked on a receive, another virtual processor on the same physical processor can run. This largely eliminates the need for the programmer to manually specify some static computation/communication overlapping, as is often required in MPI.

Automatic load balancing: If some of the physical processors become overloaded, the runtime system can migrate a few of their virtual processors to relatively underloaded physical processors. Our runtime system can make such load balancing decision based on automatic instrumentation.

Flexibility to run on arbitrary number of processors: Since more than one VPs can be executed on one physical processor, AMPI is capable of running MPI programs on any arbitrary number of processors. This feature proves to be useful in application development and debugging phases.

Optimized communication library support: Beside the communication optimization inherited from CHARM++, AMPI supports asynchronous, or non-blocking, interfaces to collective communication operations. This allows the overlapping between time-consuming collective operations with other useful computation.

Better cache performance: A virtual processor handles a smaller set of data than a physical processor, so a virtual processor will have better memory locality. This blocking effect is the same method many *serial* cache optimizations employ, and AMPI programs get this benefit automatically.

Other features and benefits have been explained in a previous workshop paper [2].

1.2 Related Work

The virtualization concept embodied by AMPI is very old, and Fox et al. [3] made a convincing case for virtualizing parallel programs. Unlike Fox's work, AMPI virtualizes at the runtime layer rather than manually at the user level, and AMPI can use adaptive load balancers. Virtualization is also supported in DRMS [4] for data-parallel array based applications. Charm++ is an early processor-virtualization system implemented on parallel machines[5]. AMPI builds on top of Charm++, and shares the run-time system with it.

There are several excellent, complete, publicly available non-virtualized implementations of MPI, such as MPICH [6], MPI/LAM [7], and MVAPICH [8]. A more recent joint effort for open source high performance computing, Open MPI [9], has also attracted much attention in HPC circle. Many machine vendors also provide their own native implementation of MPI. Many MPI implementations support multi-thread programming within one processor to allow one more degree of concurrency and exploit more parallelism in the program. CHARM++ differs from these efforts in that it provides full object-level virtualization. With the migratable parallel objects, CHARM++/AMPI lets the runtime system change the assignment of VPs to physical processors at runtime, thereby enabling a broad set of optimizations.

In this paper we aim at using a series of benchmarks to illustrate the series of benefits of AMPI and to show that they do not come with undue overheads. Many technical details about what these benefits are and how they are achieved can be found in earlier papers, and thus will not be the focus of this paper.

Next section describes the design and implementation of AMPI. In Section 3 we go through the features of AMPI and analyze its performance with various benchmarks. Through the performance evaluation, we illustrate the performance benefits listed in Section 1.1, as well as the low overhead of virtualization. We will summarize our experience in using AMPI in several large applications before concluding the paper.

2. Design and Implementation

AMPI is built on CHARM++, shares its runtime system, and inherits its features. We begin with a brief introduction to CHARM++.

2.1 CHARM++

CHARM++ is an object-based, message-driven parallel programming framework that embodies the concept of processor virtualization. In CHARM++, virtual processors are implemented with migratable parallel objects. The execution of the object is invoked by message sent from other objects through the recipient's "entry point", or a specially registered function for remote invocation. Note that the remote invocation is asynchronous: it returns immediately after sending out the message, without blocking or waiting for the response. This mechanism allows for adaptive overlapping between computation and communication, a major feature that

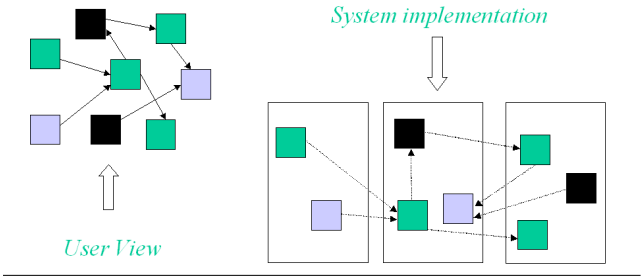

Figure 1. Virtualization in Charm++

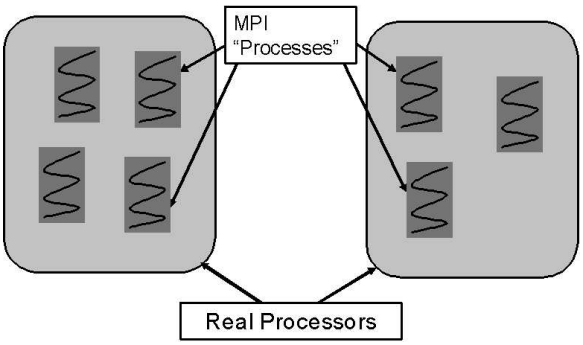

Figure 2. Implementation of AMPI virtual processors

helps CHARM++ and AMPI to utilize both CPU and network resources efficiently.

A group of VPs performing one parallel task can be organized into an indexed group called a *chare array*. A CHARM++ program typically consists of one or more chare arrays. In the programmer's point of view, one or more groups of virtual processors execute the tasks in parallel, and the runtime system maps these VPs onto physical processors adaptively and migrates them as load balancing requires (See Figure 2.1)[1]). Section 3.2 has detailed description of the load balancing module.

2.2 AMPI Design and Implementation

AMPI implements its MPI processes as CHARM++ user-level threads bound to CHARM++ communicating objects (See Figure 2). The threads used by AMPI are light-weight user-level threads; they are created and scheduled by user-level code rather than by the operating system kernel. The advantages of user-level threads are fast context switching[2], control over scheduling, and control over stack allocation. Thus, it is feasible to run thousands of such threads on one physical processor. CHARM++'s user-level threads are scheduled non-preemptively.

Message passing between AMPI virtual processors is implemented as communication among these CHARM++ objects, and the underlying messages are handled by the CHARM++ runtime system. Even with object migration, CHARM++ supports efficient routing and forwarding of the messages.

CHARM++ supports the migration of objects via efficient data migration and any necessary message forwarding. Migration presents interesting problems for basic and collective communication which are effectively solved by the CHARM++ runtime system[10]. Migration can be used by the built-in measurement-based load balancing [11, 12], adapting to changing load on workstation clusters, and even shrinking/expanding jobs for timeshared machines.

Naturally inherited from CHARM++ are more features including *communication optimization* and *fault tolerance*. *Comlib* is a module of CHARM++ framework that boosts performance of collective calls as well as point-to-point communications. It furnishes optimized strategies for communications in a parallel system and leverages the power of communication co-processors in modern networks. The effectiveness of *Comlib* has been illustrated in [13, 14]. Fault tolerance is a component that is being actively investigated. We started from a synchronous on-disk checkpoint/restart mechanism[15] and scalable in-memory checkpoint scheme[16], and further designed automatic fault-tolerant protocol for massively parallel systems like CHARM++ and AMPI[17]. AMPI has integrated CHARM++'s *Comlib* and fault-tolerance modules. A proactive fault avoidance scheme that responds to hardware generated warnings of impending fault [18] is also under development.

3. Performance Evaluation

In this section we present the results from a collection of benchmarks, analyze the performance of AMPI, and demonstrate its advantages on various aspects. We cover four major aspects in the following subsections. Our earlier workshop paper [2] describes several additional features. Our main benchmarking platforms are the Turing Cluster with 640 dual Apple G5 nodes connected with Myrinet network at University of Illinois of Urbana-Champaign, NCSA's IA-64 TeraGrid Cluster with 888 dual Intel Itanium 2 nodes and Myrinet network, NCSA's Tungsten Cluster with 1280 dual Intel Xeon nodes and Myrinet network, and the Lemieux Cluster with 750 quad Alpha nodes and Quadrics network at Pittsburgh Supercomputer Center.

3.1 Adaptive Overlapping

In MPI programming, having to block the CPU and wait for communication to complete can result in inefficiency. This is especially true in the context of modern supercomputing platforms equipped with powerful communication co-processors. The co-processors have the capability of offloading the CPUs by taking over the communication related processing. thus the CPU is not invovled in a large fraction of

[1] Figure taken from [1]

[2] On a 1.8 GHz AMD AthlonXP, overhead for a suspend/schedule/resume operation is 0.45 microsecond.

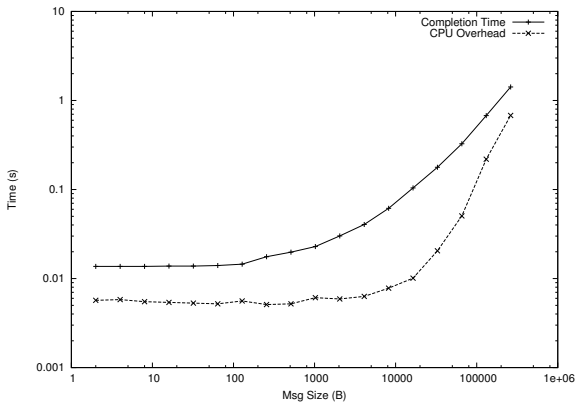

Figure 3. Completion time and CPU overhead (s) of ping-pong program on Turing (Apple G5) Cluster

the overall point-to-point communication time, and therefore it is important for the CPUs not to be blocked.

Figures 3 illustrates the relation between the total completion time and the actual CPU time spent on typical MPI function call with a simple two-way ping-pong benchmark. While the completion time increases with the message size, the true CPU overhead remains low and rises very slowly at message size up to about 200KB. (Beyond that point, the CPU is possibly involved in memory allocation, pinning pages and copying data due to limited system buffer size, depending on machine architecture and run-time system.) This benchmark captures the typical compute-communicate pattern of many MPI applications and suggests that much of the time usually attributed to communication can be utilized for useful computation.

Non-blocking point-to-point calls like `MPI_Irecv` may help in this situation by allowing some other computation to be done while CPU is waiting for communication to finish, but the programs do not always have "other computation" to do within the very same process. Also, the increase in efficiency comes at the price of additional programming complexity.

In AMPI, several virtual processors (VPs) can be mapped onto one physical processor, and the message passing among VPs is done through communication between these objects. This object-based message-driven paradigm naturally allows adaptive overlapping of computation and communication without any additional programming complexity. When one VP is blocked at a communication call, it yields the CPU so that another VP residing on the same processor can take over and utilize it.

A more realistic benchmark we have is a 3D stencil-type calculation. It is a multiple timestepping calculation involving a group of objects in a mesh. At each timestep, every object exchanges part of its data with its 6 immediate neighbors in 3D space and does some computation based on the neighbors' data. This is a simplified model of many applications, like fluid dynamics or heat dispersion simulation, so it serves well the purpose of demonstration.

K	AMPI-1	AMPI-2	AMPI-8
32	0.21	0.31	0.42
64	0.63	0.61	0.54
128	2.66	2.18	2.01
256	30.82	32.03	25.98

Table 1. Execution time [ms] of one iteration of a K^3 3D 7-point stencil calculation with AMPI of different virtualization ratios on 8 PEs of NCSA IA-64 Cluster

Table 1 shows the execution time of 3D stencil calculations on 8 physical processors on the TeraGrid IA64 Cluster. The calculations are of different sizes K^3, and with AMPI of different degrees of virtualization (the number of AMPI threads on each processor). It can be observed that the overall performance increases with the degree of virtualization. The underlying reason is illustrated in Figure 4, the output from our visualization tool *Projections*. The solid blocks represent computation and the gaps are idle time when CPU is waiting for communication to complete. As the degree of virtualization increases, there are more opportunities for the smaller blocks (smaller pieces of computation on multiple VPs) to fill in the gaps and consequently the CPU utilization increases.

Besides adaptive overlapping, the caching effect is also a favorable influence. VPs residing on the same processor can increase the spatial locality and turn some inter-processor communication into intra-processor communication. This effect is expectedly felt on unstructured grid computations.

It can also be observed that virtualization does not always result in performance improvement. For example, when K=32, the amount of adaptive overlap is limited due to the small problem scale, therefore as we introduce more VPs on one processor, virtualization overhead appears to be the dominant factor. In Section3.4, we will discuss in detail when the virtualization overhead will be offset and how.

3.2 Automatic Load Balancing

Load balancing is one of the key factors for achieving high performance on large parallel machines when solving highly irregular problems. Built with CHARM++ load balancing framework, AMPI supports automatic measurement-based dynamic load balancing and thread migration.

During the execution of an AMPI program, the load balancing framework collects workload information and object-communication pattern on each physical processor in the background, and at load balancing time, load balancer uses this information to redistribute the workload, migrating the AMPI threads from overloaded processors to underloaded ones. Thread migration in AMPI can be done either automatically or with user's help in transferring thread's stack and

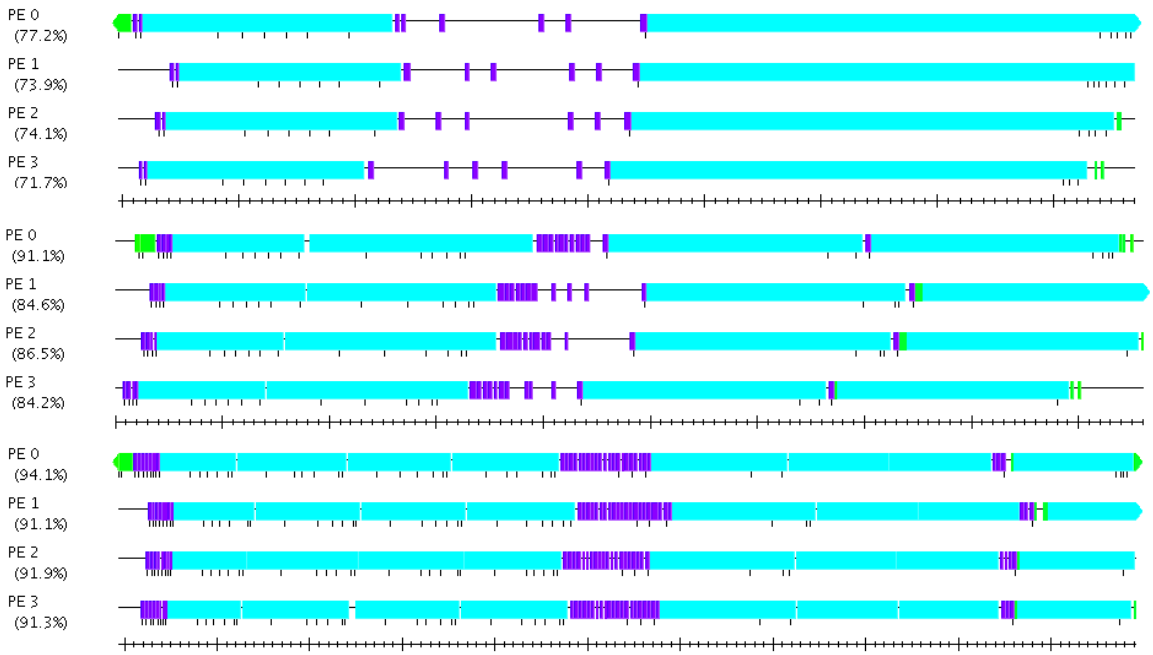

Figure 4. 7-point stencil timeline with 1, 2 and 4 VPs per processor

heap allocated data. *Isomalloc stacks and heaps* [2] (when supported on a platform) provide a clean way of moving thread's stack and heap data to a new machine by preserving the same address of data across processors. For isomalloc heaps, user's heap data is given globally unique virtual address, so that it can be moved to a new machine without changing its address. Isomalloc stacks that AMPI threads run on are allocated from isomalloc heap. Thus migration is transparent to the user code. Alternatively, users can write their own helper functions to pack and unpack heap data on both processors of a migration. This is useful when application developers wish to reduce the data volume by using application-specific knowledge and/or by packing only variables that are live at the time of migration.

In this section, we present the case studies of load balancing several MPI benchmarks and a real-world application.

3.2.1 NAS Benchmark BT-MZ

NAS Parallel Benchmark is a well known parallel benchmark suite. Its Multi-Zone version, LU-MZ, SP-MZ and BT-MZ, "solve discretized versions of the unsteady, compressible Navier-Stokes equations in three spatial dimensions"[19]. The multi-zone version is characterized with partitioning of the problems on a coarse-grain level to expose more parallelism and to stress the need for balancing the computation load. Especially, in BT-MZ, the partitioning of the mesh is done such that the sizes of the zones span a significant range, therefore creating imbalance in workload across processors. For such a benchmark or the category of parallel applications represented by this benchmark, the load balancing requires two considerations, as suggested in [20]: careful zone grouping to minimize inter-processor communication and a multi-threading scheme to balance the computation workload across processors.

AMPI is naturally equipped with automatic load balancing module to take into consideration these two aspects of a parallel program: communication load and computation load. The following results illustrate AMPI's effectiveness on load balancing BT-MZ.

In this benchmark, we run the BT-MZ benchmark on both native MPI and AMPI. Through *Projections* we confirmed there is a load imbalance across processors. Then we insert the function call to trigger the automatic load balancing in AMPI runtime system. After 3 timesteps, when the runtime has collected sufficient information to advise the load balancer, the AMPI VPs are migrated from heavier-loaded processors onto lighter-loaded ones. The execution time is visualized in Figure 5.

When the number of processor increases for the same problem scale, we can make two observations. Firstly, the execution time without load balancing increases. BT-MZ creates workload imbalance by allocating different amounts of work among the processors, and with larger number of processors, the degree of imbalance increases to a broader range. Consequently, the overall utilization drops. Secondly, with load balancing, higher degree of virtualization allows the load balancing module to work more effectively, simply because there are more objects to move around if necessary. That is the reason that having number of VP much larger

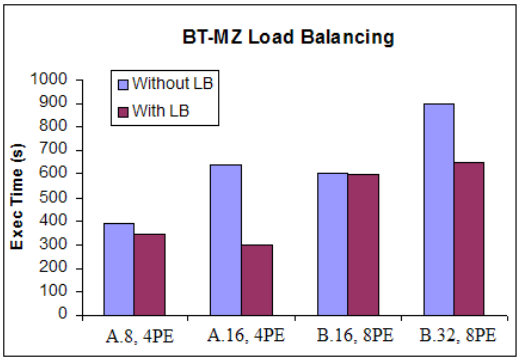

Figure 5. Load Balancing on NAS BT-MZ

than number of P is recommended for the load balancer to be effective.

This benchmark demonstrates the effect of load balancing on applications with static load imbalance. This scenario is not uncommon. Handling uneven initial workload distribution and migrating the job away from faulty nodes are two examples that we have observed where such load balancing is useful. For the other type of applications, with dynamically varying workload, load balancer can be triggered periodically. Next section exemplifies this use.

3.2.2 Fractography3D

Fractography3d is a dynamic 3D crack propagation simulation program to simulate pressure-driven crack propagation in structures. It was developed by Prof. Philippe Guebelle [3] and his students and collaboration with our group. Fractography3d code is implemented on Charm++ FEM framework [21] and AMPI.

The test problem we run simulates a crack propagation with a force and the process of elastic turning into plastic zone along the crack. The crack propagation simulation was run with 1000 AMPI virtual processors on 100 processors of the Turing Apple cluster at UIUC.

There are two factors that may contribute to the load imbalance in this simulation problem. When external force applies to the material in study, the initial elastic state of the material may change into plastic along the wave propagation, which results in much heavier computation. Second, 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* [22]. At each iteration of the simulation, pressure exerted upon the plastic 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.

[3] Prof. Philippe Guebelle is a professor of Dept. of Aerospace Engineering at University of Illinois at Urbana-Champaign

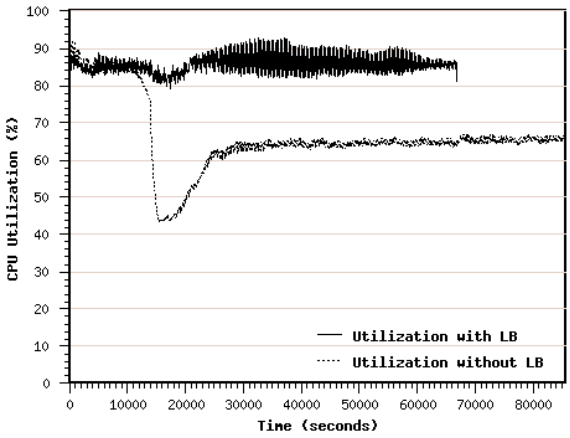

Figure 6. CPU utilization *Projections* graph of Fractography3D over time without vs. with load balancing

The simulation without load balancing runs for 24 hours. The *Projections* view on CPU utilization over time is shown in the bottom curve of Figure 6. It can be seen that at time around 1000 seconds, the application CPU utilization dropped from around 85% to only about 44%. This is due to the start of the process of elastic turning into plastic zone along the crack, leading to load imbalance. As more elastic part turns into plastic, the CPU utilization slowly increases until all turn into plastic. The load imbalance can be easily seen in the CPU utilization graph over processor as shown in the upper part of Figure 7. While some of the processors have the CPU utilization as high as about 90%, some processors only have about 50% of the CPU utilization during the whole execution time.

The top curve of Figure 6 illustrates results of automatic load balancing of the same crack propagation simulation in the view of overall CPU utilization over the time. the load balancing is invoked every 500 time-step of the simulation. with a greedy-based algorithm. The automatic load balancer uses the runtime load and communication information instrumented by the Charm++ runtime to migrate chunks from the overloaded processor to underloaded processors, leading to improved performance. As figure 6 shows, the overall CPU utilization on all processors throughout the entire simulation stays around 80-90%. The lower half of Figure 7 further illustrates that load balance has been improved from the upper part in the view of the CPU utilization over processors. It can be seen that CPU utilization of at least 80% is achieved on all processors with little load variance. The simulation with load balancing now takes about 18.5 hours to complete, yielding about 23% of performance improvement with load balancing. It is important to note that this improvement is attained without any additional programming effort beyond the original MPI program.

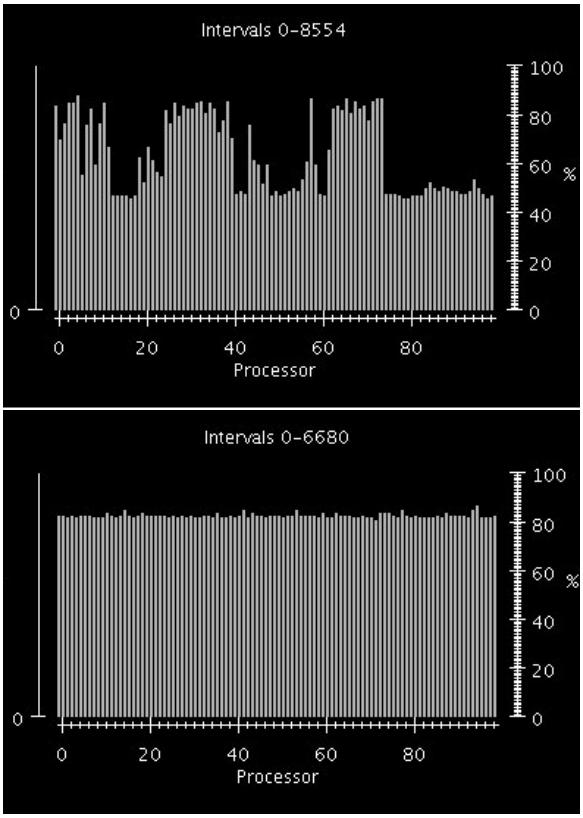

Figure 7. CPU utilization graphs of Fractography3D across processor without vs. with load balancing

3.3 Communication Optimization

AMPI has a run-time system capable of observing the communication patterns, which gives us the ability to optimize communication performance by substituting different communication algorithms automatically. For this purpose, AMPI inherits the communication optimization library (*Comlib*) from CHARM++ framework. *Comlib* works by delegating communications in CHARM++/AMPI and applying suitable optimization strategies dynamically and intelligently. This sections includes two examples: streaming point-to-point communication and multiple all-to-all communication optimization strategies.

3.3.1 Streaming

Many short messages in a parallel system can be combined and sent in fewer batches to reduce the per-message overhead. This is the basic idea of Streaming strategy for point-to-point communication. The following benchmark is based on a multi-ping program. In AMPI Multi-Ping, processor A sends several messages to processor B, which responds with a short message after it has received all. This communication pattern is able to fill the pipeline on a message's path from sender to receiver: sender CPU, sender NIC, interconnect, receiver NIC, and receiver CPU. The time obtain through multi-ping represents the limiting factor in this

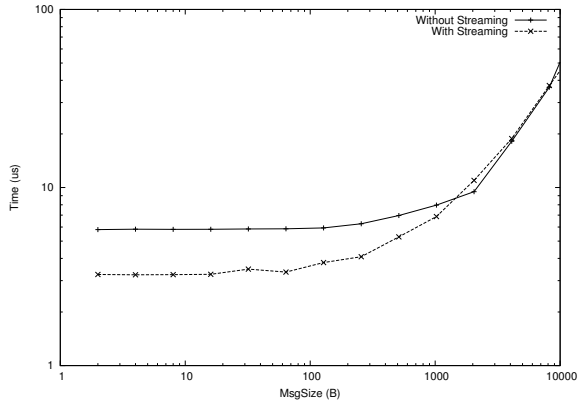

Figure 8. Streaming strategy for point-to-point communication on NCSA IA-64 Cluster

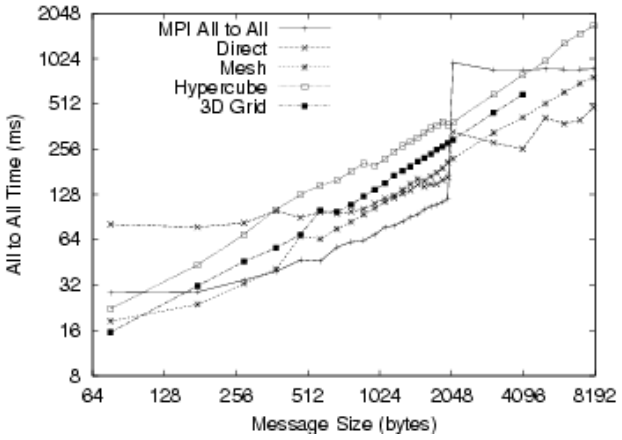

Figure 9. All-to-all completion time on 1024 processors on Lemieux

pipeline, namely the due price for a point-to-point communication. Figure 8 shows how Streaming strategy can bring the multi-ping point-to-point overhead even lower for messages shorter than 1000 bytes. This benefit is enhanced by processor virtualization, since messages from multiple VPs on the same processor can all be grouped in the same batch.

3.3.2 Asynchronous Collectives

Collective calls involves many or all processors and are time-consuming. *Comlib* implements a variety strategies to tackle this issue. In early work, we described in [13, 14] the series of efforts to improve performance of collective operations including broadcast, multicast, and all-to-all. We take all-to-all as an example. In Figure 9, 10(Figure taken from [13]), completion time of All-to-all operation is compared between native MPI and AMPI with Mesh strategy. According to the figure, Mesh strategy helps AMPI to do better than MPI for messages smaller than 400 bytes. For larger messages and different network topology, probably other strategy is

#PE	19	27	33	64	80	105	125	140	175	216	250	512
Native MPI	-	29.44	-	14.16	-	-	9.12	-	-	8.07	-	5.52
AMPI	42.41	30.53	24.65	15.64	12.62	10.94	10.78	10.62	9.39	8.63	7.55	5.46

Table 2. Timestep time [ms] of 240^3 3D 7-point stencil calculation with AMPI vs. native MPI on Lemieux

#PE	19	27	33	64	80	105	125	140	175	216	250	512
Native MPI	-	127.7	-	31.9	-	-	8.87	-	-	3.62	-	1.34
AMPI	128.15	125.60	36.03	33.42	10.18	10.14	8.34	4.57	4.19	3.67	3.16	1.17

Table 3. Timestep time [ms] of 240^3 3D 7-point stencil calculation with AMPI v.s. native MPI on Turing (Apple G5) Cluster

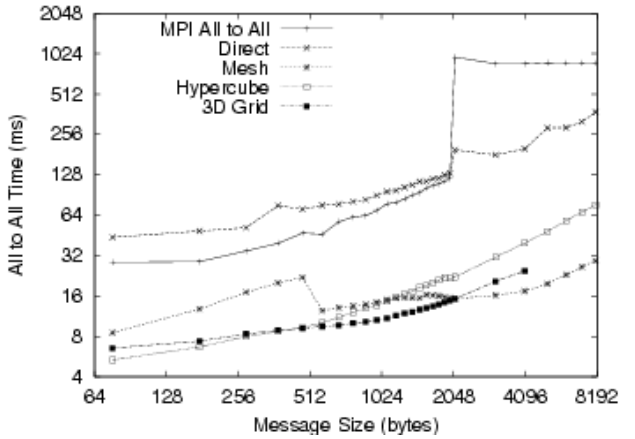

Figure 10. All-to-all CPU time on 1024 processors on Lemieux

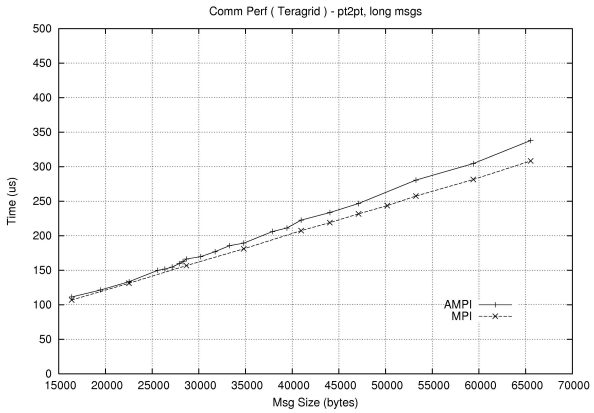

Figure 12. Performance for point-to-point communication (long messages) on NCSA IA-64 Cluster

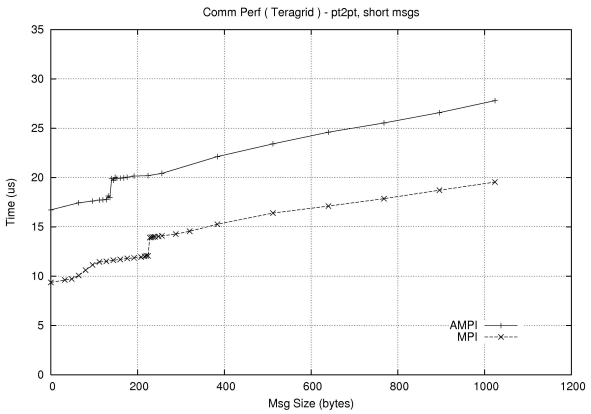

Figure 11. Performance for point-to-point communication (short messages) on NCSA IA-64 Cluster

a better choice. Therefore *Comlib* developers are trying to make it "smart", able to dynamically learn and switch to the most suitable strategy. Moreover, AMPI exploits the large gap between elapsed time and CPU time of these collective operations by providing asynchronous versions of them.

3.4 Flexibility and Overhead

In this section we demonstrate the flexibility virtualization provides, as well as the overhead virtualization incurs. Because AMPI is implemented on top of Charm++, which is typically implemented on top of native MPI (or the lowest level communication layer accessible to us), we do not expect to have better performance than native MPI on a ping-pong style microbenchmark. Figure 11 illustrates that AMPI has a left-shift due to the 70+ byte AMPI message header, and a 2-4 microsecond increase in time for the short message latency due to thread context switch overhead as well as scheduling overhead. For longer messages, we pay the overhead of extra message copying in order to support migratable objects. Active research work is being carried out to reduce overhead for both situations, and more importantly, AMPI is expected to outperform native MPI when its features such as

automatic adaptive overlap and dynamic load balancing are utilized in the real application.

In the next benchmark of 240^3 3D 7-point stencil calculation, we show the flexibility of AMPI. The algorithm of this particular benchmark divides a 240^3 block of data into k-cube partitions, each of which is a smaller cube assigned to an MPI process. Natural expression of this algorithm requires k-cube number of processors to run on. This benchmark represents the type of applications that require specific number of processors.

We first run the benchmark with native MPI. As described above, this program runs only on k-cube processors: 27, 64, 125, 216, 512, etc. Then, with AMPI powered with virtualization, the program runs transparently on any given number of processors, exhibiting the flexibility that virtualization offers. The comparison between these two runs are listed in Tables 2 and 3. Note that on some arbitrary number of processors such as 19 and 80, the native MPI program cannot be launched, whereas AMPI runs the job with no difficulty. This flexibility has been proven to be very useful in real application development experiences, including that of the CSAR described in Section 4.

For the data points where native MPI is able to run too (K-cube processors), the AMPI version is run with 1 VP per processor, without compounding with benefit of overlap, so there is some AMPI overhead entailed by virtualization. From our experiences, however, the few microseconds of virtualization overhead is negligible when the average work driven by one message is at hundreds of microseconds level, which is achieved by appropriate choice of virtualization degree. Moreover, the cost of supporting virtualization and coordinating the VPs is further offset by other benefits of virtualization. Therefore, it is safe to conclude that the flexibility and load balancing advantages of AMPI do not come at an undue price in basic performance.

4. Conclusions

AMPI, which began as a proof-of-principle project to demonstrate message-passing can be effectively supported in a migratable-objects system such as CHARM++, is now a full-fledged MPI implementation. We expect to continue to improve AMPI further, as outlined below; but in its current state it is already a mature system that can be used in applications, especially if they can benefit from its adaptive features.

AMPI is being used in multiple parallel applications and frameworks, such as the FEM framework described in Section 3.2. Another important application of AMPI is in the collaboration with the Center for Simulation of Advanced Rockets (CSAR), an academic research organization funded by the Department of Energy and affiliated with the University of Illinois. The focus of CSAR is the accurate physical simulation of solid-propellant rockets, such as the space shuttle's solid rocket boosters. The main CSAR simulation code consists of four major components, and each one of these components - fluids, burning, interface, and solids - began as an independently developed parallel MPI program. One of the most important benefits CSAR found in using AMPI is the ability to run on a different number of virtual processors than physical processors. For example, a CSAR developer was faced with an error in mesh motion that only appeared when a particular problem was partitioned for 480 processors. Finding and fixing the error was difficult, because a job for 480 physical processors can only be run after a long wait in the batch queue at a supercomputer center. Using AMPI, the developer was able to debug the problem interactively, using 480 virtual processors distributed over any arbitrary number of physical processors on a local cluster, which made resolving the error much faster and easier.

We presented AMPI, an adaptive implementation of MPI on top of CHARM++. AMPI implements migratable virtual and light-weight MPI processes. It assigns several virtual processors on each physical processor. This efficient virtualization provides a number of benefits, such as the ability to automatically load balance arbitrary computations, automatically overlap computation and communication, emulate large machines on small ones, and respond to a changing physical machine. AMPI has been ported to a variety of modern supercomputing platforms, including Apple G5 Cluster, NCSA's IA-64 Cluster, Xeon Cluster, IBM SP System, PSC's Alpha Cluster and IBM Blue Gene. We demonstrated, via simple performance studies, that the overhead of AMPI is small and tolerable for most application. Thus, the benefits of adaptivity it provides come at only a small cost.

As illustrated in this paper, AMPI is a mature and portable system which fully supports the dynamic nature of new generation parallel applications with its performance and features. Moreover, AMPI is also an active research project; much future work is planned. AMPI already supports most MPI-2 features, and we expect to achieve full MPI-2 standards conformance soon. We are rapidly improving the performance of AMPI, bringing out more benefits for dynamic applications and further reducing the already small overhead for non-dynamic ones. The CHARM++ performance analysis tools are being updated to provide more direct support for AMPI programs. Another potential future direction is to use compiler analysis to identify live variables at migration time to reduce communication volume. Finally, we plan to extend our suite of automatic load balancing strategies to provide machine-topology specific strategies, useful for modern machines such as BlueGene.

References

[1] Laxmikant V. Kalé. The virtualization model of parallel programming : Runtime optimizations and the state of art. In *LACSI 2002*, Albuquerque, October 2002.

[2] Chao Huang, Orion Lawlor, and L. V. Kalé. Adaptive MPI. In *Proceedings of the 16th International Workshop*

on Languages and Compilers for Parallel Computing (LCPC 2003), LNCS 2958, pages 306–322, College Station, Texas, October 2003.

[3] G.C. Fox, R.D. Williams, and P.C. Messina. *Parallel Computing Works*. Morgan Kaufman, 1994.

[4] V.K.Naik, Sanjeev K. Setia, and Mark S. Squillante. Processor allocation in multiprogrammed distributed-memory parallel computer systems. *Journal of Parallel and Distributed Computing*, 1997.

[5] L.V. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.

[6] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. Mpich: A high-performance, portable implementation of the mpi message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[7] Greg Burns, Raja Daoud, and J. Vaigl. Lam: An open cluster environment for mpi. In *Proceedings of Supercomputing Symposium 1994, Toronto, Canada*, 1994.

[8] J. Liu, J. Wu, and D. K. Panda. High performance rdma-based mpi implementation over infiniband. *Int'l Journal of Parallel Programming*, 2004.

[9] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004.

[10] Orion Sky Lawlor and L. V. Kalé. Supporting dynamic parallel object arrays. *Concurrency and Computation: Practice and Experience*, 15:371–393, 2003.

[11] Gengbin Zheng. *Achieving High Performance on Extremely Large Parallel Machines: Performance Prediction and Load Balancing*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.

[12] L. V. Kale, Milind Bhandarkar, and Robert Brunner. Runtime Support for Adaptive Load Balancing. In J. Rolim, editor, *Lecture Notes in Computer Science, Proceedings of 4th Workshop on Runtime Systems for Parallel Programming (RTSPP) Cancun - Mexico*, volume 1800, pages 1152–1159, March 2000.

[13] L. V. Kale, Sameer Kumar, and Krishnan Vardarajan. A Framework for Collective Personalized Communication. In *Proceedings of IPDPS'03*, Nice, France, April 2003.

[14] Sameer Kumar and L. V. Kale. Scaling collective multicast on fat-tree networks. In *ICPADS*, Newport Beach, CA, July 2004.

[15] Chao Huang. System support for checkpoint and restart of charm++ and ampi applications. Master's thesis, Dept. of Computer Science, University of Illinois, 2004.

[16] Gengbin Zheng, Lixia Shi, and Laxmikant V. Kalé. Ftc-charm++: An in-memory checkpoint-based fault tolerant runtime for charm++ and mpi. In *2004 IEEE International Conference on Cluster Computing*, San Dieago, CA, September 2004.

[17] Sayantan Chakravorty and L. V. Kale. A fault tolerant protocol for massively parallel machines. In *FTPDS Workshop for IPDPS 2004*. IEEE Press, 2004.

[18] Sayantan Chakravorty, Celso Mendes and L. V. Kale. Proactive fault tolerance in large systems. In *HPCRI Workshop in conjunction with HPCA 2005*, 2005.

[19] Rob F. Van der Wijngaart and Haoqiang Jin. Nas parallel benchmarks, multi-zone versions. Technical Report NAS Technical Report NAS-03-010, July 2003.

[20] Haoqiang Jin and Rob F. Van der Wijngaart. Performance characteristics of the multi-zone nas parallel benchmarks. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2004.

[21] Milind Bhandarkar and L. V. Kalé. A Parallel Framework for Explicit FEM. In M. Valero, V. K. Prasanna, and S. Vajpeyam, editors, *Proceedings of the International Conference on High Performance Computing (HiPC 2000), Lecture Notes in Computer Science*, volume 1970, pages 385–395. Springer Verlag, December 2000.

[22] Philippe H. Geubelle and Jeffrey S. Baylor. Impact-induced delamination of composites: a 2d simulation. *Composites Part B: Engineering*, 29(5):589–602, 1998.