Supporting Adaptivity in MPI for Dynamic Parallel Applications [*]

Chao Huang, Gengbin Zheng, Laxmikant V. Kalé
Parallel Programming Laboratory
University of Illinois at Urbana-Champaign
{chuang10, gzheng, kale}@cs.uiuc.edu

May 24, 2007

Abstract

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 Adaptive MPI (AMPI), an adaptive implementation and extension of MPI with migratable threads. AMPI includes a powerful run-time support system that takes advantage of the freedom of mapping virtual MPI processes (VPs) onto processors. With this run-time system, AMPI supports such features as automatic adaptive overlap of communication and computation and automatic load balancing. It can also support other features such as checkpointing without additional user code, and the ability to shrink and expand the set of processors used by a job at runtime. In this paper, 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.

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, as exemplified in [1, 2]. Typical implementations of MPI, the industry standard for parallel programming on many platforms, do not support the dynamic nature of these applications well. As a result, programming productivity and parallel efficiency suffer. In this paper, we describe our research in exploring adaptivity support for such dynamic applications.

1.1 Motivation

Message Passing Interface (MPI) is a standard that specifies interface for a set of message passing functions. It has become the *de facto* standard for parallel programming on a wide range of platforms. There are several excellent, complete, publicly available implementations of MPI, such as MPICH [3] and MPI/LAM [4]. A more recent joint effort for open source high performance computing, Open MPI [5], has also attracted much attention in HPC circle. Many machine vendors also provide their own platform-native implementations of MPI.

Most typical and widely used implementations of MPI are highly optimized for message passing performance, as efficient communication is one of the most important design goals of the MPI Standard. Most implementations, however, fail to address on the important issue of supporting dynamic nature of parallel applications. Our research approaches this issue with adaptivity support for MPI paradigm.

[*]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.

1.2 Design Goals

To provide adaptivity support for MPI programming paradigm, we design our implementation with the following goals.

Adaptive overlap between communication and computation: One issue in MPI programming is that of overlapping communication with computation. When the program hits a receive statement, ideally the corresponding message should have already arrived, so that blocking the process and wasting CPU time is avoided. To achieve this, the programmer tries to move sends up and receives down, and fit some computation between sends and receives, giving time for communication to complete. Our implementation should allow adaptive overlap of communication and computation, improving program efficiency without inducing this kind of programming complexity.

Automatic load balancing: Many scientific applications have dynamically varying workload distribution. The computing hotspots can be constantly shifting, as exemplified in the fracture propagation simulation and adaptive mesh refinement methods. Load imbalance has an especially high performance impacts in parallel programs, because the slowest node dictates the overall performance of the whole system. Run-time load balancing should be effective, adaptive to the application as well as to the parallel platform, and be automated so that minimal user effort is required.

We explore our design alternatives with these goals in mind. We take an over-decomposition approach to adaptive overlapping and virtualize MPI processes with parallel flows of control. Design alternatives for parallel flows of control we examined include processes, kernel threads and user-level threads. On a wide range of platforms, we compared various aspects including maximum number of flows per processor, context switch overhead, and migratability [6]. We concluded that user-level thread is the best fit.

Load balancing requires migratability of our user-level threads as well as capability of the underlying run-time system to monitor the workload distribution dynamically. The run-time should also be able to observe the communication patterns happening in the system, so that it can advise communication-aware load balancing strategy. The implementation should be built on top of a run-time system with such capabilities.

1.3 Past Work

Several previous projects have put significant efforts into addressing this situation with various approaches. Some MPI implementations support multi-thread programming within one processor to expose an additional degree of concurrency and exploit overlapping between communication and computation. For instance, TMPI [7] uses multithreading for performance enhancement of multi-threaded MPI programs on shared memory machines. However, the full power of overlap between communication and computation is not realized with simple multithreading; the implementer needs to carefully decide on a set of related design questions including the choice of parallel flows of control and an intelligent scheduling mechanism.

Another effect is automatic load balancing. Traditionally, MPI programming takes two separate routes toward load balancing. The first is run-time process migration support. Early efforts in migratable MPI process include those in the CoCheck [8] and related Tool-Set project [9]. More recently, hybrid programming model with MPI+OpenMP [10] approaches the problem by redistributing OpenMP threads among MPI processes, but these approaches incur significant programming complexity overhead. As an alternative, library support for dynamic load balancing improves performance for a specific category of parallel applications. For example, Zoltan [11] is a programming toolkit that supports dynamic load balancing, but its interface is incompatible with that of MPI. Other library solutions for load balancing are limited to certain types of parallel applications, as exemplified by Chombo [12] library for adaptive mesh refinement applications. It is desirable to provide an integrated, efficient and automated load balancing mechanism for a wide range of applications.

Section 2 of this paper describes Adaptive MPI (AMPI), our implementation of MPI that achieves the design goals for adaptivity support. We go through challenging research issues in the implementation of AMPI, especially the ones associated with supporting migratable threads. In Section 3, we analyze AMPI's performance with a wide range of benchmarks and real-life parallel applications. Section 5 concludes the paper with discussion of possible future directions.

2 Design and Implementation

In this section we describe our adaptive implementation of MPI called Adaptive MPI (AMPI). AMPI began as a proof-of-principle project to demonstrate message passing can be efficiently supported with migratable threads and adaptive run-time system, and is now a full-fledged MPI implementation. We first outline the design of AMPI along with its Adaptive Run-Time System (ARTS). After that, we focus our discussion on the implementation of two key features of AMPI, virtual processes and migratable threads, that help achieving the two design goals stated in Section 1.2.

2.1 Adaptivity Support for MPI

AMPI supports adaptivity in MPI via 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 an ARTS maps these VPs onto P physical processors. The number of VPs 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 number, and with AMPI, V can still be a cube number even though P is prime. When $V = P$, the program executes the same way it would with any typical MPI implementation, and it enjoys only part of the benefits of AMPI, such as collective communication optimization. To take full advantage of the AMPI run-time system, we have V much larger than P.

AMPI offers an effective division of labor between the programmer and the run-time system. The program for each process still has the same syntax as specified in the MPI Standard. Further, not being restricted by the physical processors, the programmer is able to design more flexible partitioning that best fits the nature of the parallel problem. The run-time system, on the other hand, has the opportunity of adaptively mapping and re-mapping the programmer's virtual processors onto the physical machine.

Besides the performance benefits that we are going to measure in Section 3, this design helps AMPI programmers to practice good software engineering disciplines such as high cohesion and low coupling. High cohesion means any module in a program should be understandable as a meaningful unit and components of a module should be closely related to one another. Low coupling requires that different modules be understandable separately and have low interaction with one another. With MPI's traditional processor-centric programming model, it is often almost inevitable that programmers will violate these principles. With virtualized processes, on the other hand, programmers are given the freedom to partition and structure the parallel application in accordance with good software engineering principles.

For a real-life example, consider a version of the rocket simulation code developed at the Center for Simulation of Advanced Rockets (CSAR) at Illinois [13, 14]. Figures shown here are simplified representation of the application structure, which consists of two modules, *Rocflo* and *Rocsolid*. Rocflo module simulates the structure of fluid dynamics of the burning gas in a rocket booster, and the Rocsolid module models the structure of the solid fuel inside the booster. With typical MPI implementation, the fluid mesh and the solid meshes are required to be glued together on each processor, even though these two different meshes are decomposed separately by each module and have no logical connection, as shown in Figure 1. With AMPI's adaptivity support, the work-and-data units of each module with its own set of VPs of different sizes can be separated to allow natural expression of the algorithm (See Figure 2).

2.2 Adaptive Overlap via Virtual Processes

In MPI programming, having to block the CPU and wait for communication to complete can cause tremendous impact to performance. 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, and therefore it is important for the CPUs not to be blocked. Non-blocking point-to-point MPI calls such as `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"

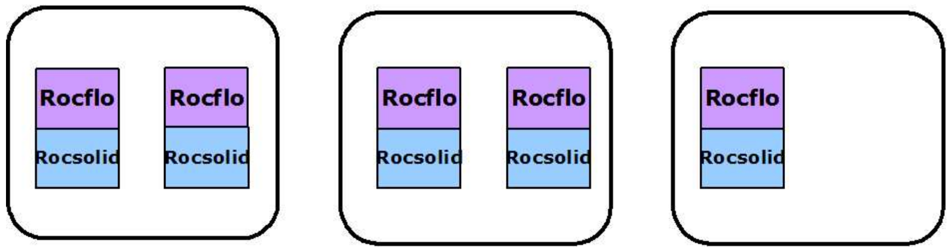


Figure 1: Structure of Rocket Simulation Code with Typical MPI Implementation

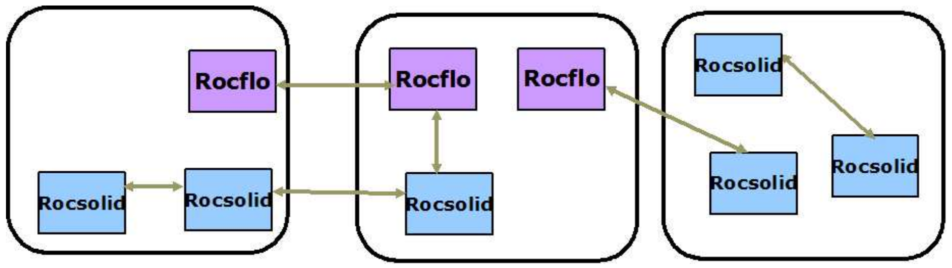


Figure 2: Structure of Rocket Simulation Code with AMPI's Adaptivity Support

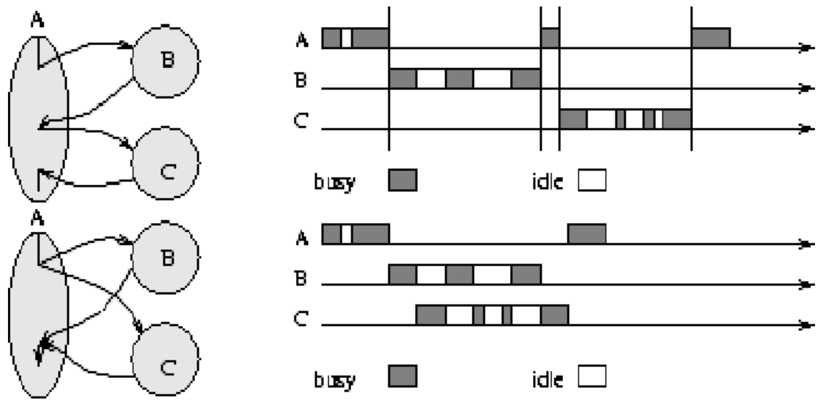


Figure 3: Adaptive Overlapping

to do within the very same process. Multithreading within the MPI process is another effort, yet the increase in efficiency comes at the price of additional programming complexity.

AMPI's adaptive run-time empowers the MPI program to adaptively overlap communication and computation. To illustrate this, we compare different scenarios of running the same MPI program without and with the adaptivity support. In Figure 3, there are three parallel modules A, B and C spread across all processors. A must call B and C, but there is no dependence between B and C. In traditional MPI style programming model, the programmer has to choose one module between B and C to call from A first on all processors. Only when the first chosen module returns can A call the remaining one on all processors. This model can be inefficient because when one module idles the CPU, for example, when waiting for communication to complete, other modules are not allowed to take over and do useful computations, even though there is absolutely no dependence between the modules.

With ARTS support, A can invoke B on all the VPs, initiating computation and sending out messages, and since there is no dependence between B and C, A can also start off C in a similar fashion, and thereby modules B and C can interleave their execution. When one module blocks due to communication or load imbalance, the other module can automatically overlap the idle time with computation, based on the availability of data, as illustrated in Figure 3. With non-blocking calls and careful programming, the programmer could achieve the same effects with traditional

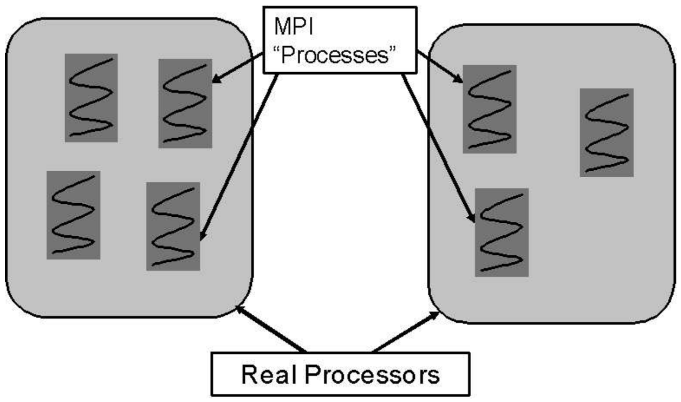


Figure 4: Implementation of AMPI Virtual Processors

MPI, but the price is additional programming complexity and a breach of modularity. Figure 10 from Section 3.1.3 has a more realistic visualization of adaptive overlap in an AMPI program.

2.2.1 Implementing Virtual Processes

AMPI implements its MPI processors as user-level threads bound to parallel objects, as illustrated in Figure 4. The rank of an AMPI VP corresponds to the index of the parallel object, and is independent of the rank of physical processor it resides on. As the parallel object and the user-level thread are bound together, they always migrate together, for example during a load balancing session.

The threads used in 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, control over scheduling, and control over stack allocation. Thus, it is feasible to run thousands of such threads on one physical processor [15]. The user-level threads we use have a overhead for a suspend/schedule/resume operation is 0.45 microsecond on a 1.8 GHz AMD AthlonXP and are scheduled non-preemptively.

Message passing between AMPI VPs is implemented as communication among the parallel objects, and the underlying messages are handled by the ARTS. The ARTS supports efficient routing and forwarding of messages; even the destination VP has migrated away from its initial residing processor, the AMPI message is delivered to the object regardless of its current physical location. The ARTS provides a scalable mechanism to determine the location of a given VP. First of all, the system maps any VP index into a home processor that always knows where the corresponding VP can be reached. When the VP migrates away, it updates its home processor of its current location. A message destined for it will still be sent to its home processor, which in turn forwards the message to its current residing processor. Since this forwarding is inefficient, the home processor will inform the sender of the VP's current location, advising it to send future messages directly to the new location. This mechanism avoids having a central registry which consumes enormous non-distributed storage and presents a serial bottleneck, or a broadcast-based mechanism that wastes bandwidth.

2.2.2 Handling Global Variables

While global variables in the MPI code cause no problem with traditional MPI implementations, where each process image contains a separate copy, they are not safe in AMPI's multi-threading paradigm. AMPI VPs are executed as user-level threads, many of which can run on one processor. Therefore, AMPI run-time needs to ensure thread safety of the global variables in the MPI code. AMPI offers two ways of achieving that.

The first solution is to privatize global variables in the MPI code at run-time level by creating a copy of the global variables for each of the user-level threads, namely the AMPI VPs. We implemented a *swap-global* scheme by analyzing the Executable and Linking Format (ELF) object files in a way similar to the Weaves run-time framework [16]. A dynamically linked ELF file format executable always accesses global variables via the Global Offset

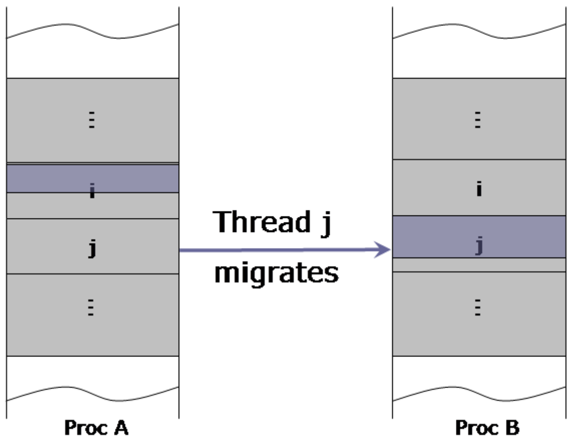


Figure 5: Migrating a Thread's Stack Allocated with Isomalloc

Table (GOT), which contains one pointer to each global variable. To make separate copies of the global variables, we then create separate copies of the GOT, one for each user-level thread. When the thread scheduler switches the threads, it also swaps the corresponding GOTs.

The second approach is global variable removal at code level. We can modify the MPI source code, collecting all the global variables into a non-global data structure and passing it into each function referencing any of the global variables. This process of global variable removal is mechanical and sometimes cumbersome. Fortunately, it can be automated by source-to-source translator, such as *AMPIzer* [17] based on Polaris [18] that privatizes global variables from arbitrary Fortran77 or Fortran90 code and generates necessary code for moving the data across processors. Similar tool for handling C/C++ MPI code will also be made available soon.

2.3 Automatic Load Balancing with Migratable Threads

AMPI's capability of dynamic load balancing adapts to the application behavior by migrating VPs across processors at run-time. We describe our support for efficiently migrating thread data, and then explain in details two of AMPI's key features including load balancing and system-level checkpointing.

2.3.1 Migrating Thread Data

One of the biggest challenges in migrating threads is to migrate thread data. Specifically, it is difficult to extract the useful stack and heap data allocated for a thread for migration, and to update the correct values of pointers contained in thread data, including function return addresses, frame pointers and pointer variables, on the new processor after migration. There are two different approaches available in AMPI [6].

The idea behind the first approach is to guarantee that the stack and heap of a thread will always have exactly the same address on the new processor as on the old processor. With this guarantee, no pointers need to be updated because all the references remain valid on the new processor. This idea was originally developed for thread migration in the PM2 run-time system [19].

The implementation of isomalloc stack and isomalloc heap is similar. First, a range of same virtual address area across all processors is reserved and split into P regions, each for one processor. Note that only virtual memory is assigned, and not any physical memory is actually allocated. When the thread allocate data on the stack or the heap, the slot of virtual space corresponding to its residing processor will be used. When the thread moves to a new processor, the run-time system simply copies over the iso-mallocated data in that slot, knowing the same addresses are guaranteed to be valid on the destination processor (See Figure 5).

While the isomalloc approach enables automatic thread data migration, it requires large virtual address space, especially when the number of threads is large. The second solution takes an alternative approach, requiring the

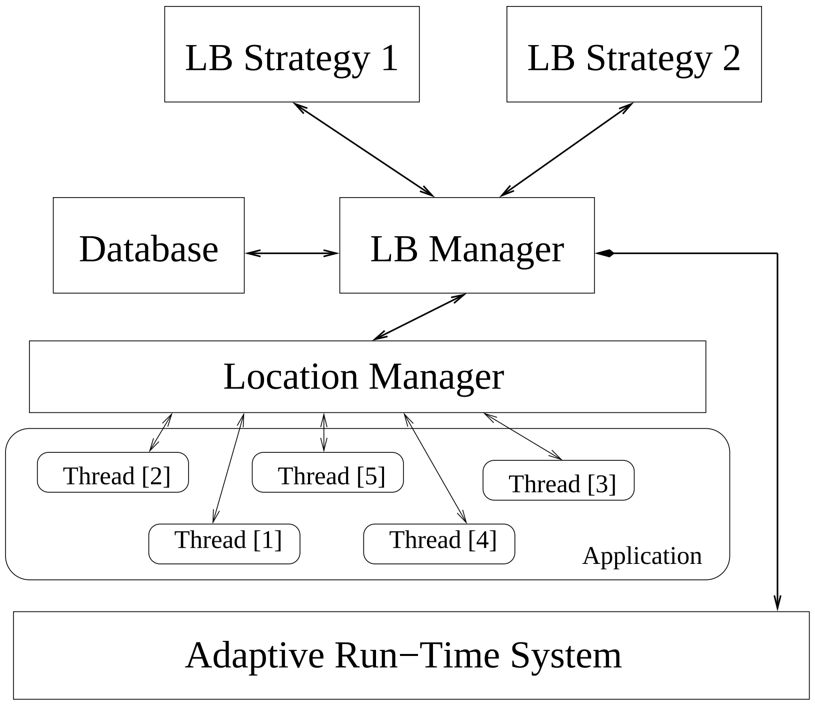


Figure 6: Components of the Load Balancing Framework

programmer to specify in code the data items to move at migration. As an extension to the system's PUP (Packing and UnPacking) framework, the programmer writes a PUP'er function to iterate through the data items chosen to be moved. This approach has the advantage of efficiency. Since the programmer has the best knowledge of the data items in the code, only those essential for restarting the thread execution on the new processor are moved.

2.3.2 Load Balancing Strategies

The challenge is for the ARTS to intelligently determine an effective remapping of threads for the future. In order for the load balancing algorithms to make better load balancing decisions, it is essential for the ARTS to provide most up-to-date application and system load information as input data. The AMPI run-time system exploits a simple heuristic called *Principle of Persistence* [20] to automate this process. Like the principle of locality, the principle of persistence is an empirical heuristic about parallel program behaviors. We observe that for most parallel programs expressed in terms of VPs, the computation loads and communication patterns tend to persist over time. This heuristic applies to many programs with dynamic behavior, including those using adaptive mesh refinement with abrupt but infrequent changes, and those simulating molecular dynamics with slow and gradual changes over time.

Based on the principle of persistence, our ARTS uses a measurement based load balancing scheme. Figure 6 illustrates the components of the load balancing framework on a single processor. At the top level of the figure are the load balancing strategies, from which the run-time can choose one to plug in and use. In the center is the LB Manager which plays the key role in load balancing. When informed by LB Manager to perform load balancing, strategies on each processor may retrieve information from the local LB Manager database about the current state of the processor and the threads currently assigned to it. Depending on the specific strategy, it may communicate with other processors to gather state information. With all information available, strategies determine when and where to migrate thread, and provide this information to the LB Manager, which supervises the movements of the threads, informing the strategies as threads arrive. When the moves are complete, the strategies signal the LB Manager to resume the threads. A variety of such strategies have been developed for applications with different dynamic behaviors. Some strategies are centralized, others fully distributed. Some use only computation load when making a decision, and others take into account communication patterns and even topology of the platform.

During execution, the LB Manager monitors the load behavior on each processor. It collects background load and idle time statistics into the LB Database, which is used by the LB Strategies for making load balancing decisions. The LB Manager also interacts with threads through Location Managers. Location Managers manage the location of the threads. They are responsible for monitoring the movement of threads and reporting chare threads' arrival and departure to the LB Manager. The Location Manager also monitors the execution of chare threads. When a particular thread is being executed, it notifies the LB Manager so that the manager may start the timing for its execution. The Location Manager also reports about communication initiated by the thread. The Location Manager knows how to migrate a thread under its control. Thus, when the LB Manager receives a request to migrate a thread, it forwards the request to the Location Manager.

2.3.3 Automatic Checkpointing

Based on migratable threads, the ARTS supports checkpoint/restart mechanisms with minimal user intervention required. Because the program comprises migratable parallel threads, the ARTS checkpoints the program by migrating all the threads from the processors to stable media: either hard disk drive or memory on peer nodes[21]. At restart phase, the threads are migrated back from the storage, run-time system information and user data restored, and the execution is restarted from where the checkpoint has happened[22].

It is important to note that the checkpoint/restart mechanism in the ARTS has benefits beyond fault tolerance. It also offers the capability of adapting to changing computing environment. Imagine if we lose 1 node out of a 1024-node partition in the middle of a long execution. We can immediately work around this failure and restart the checkpointed program, with the same number of VPs, on 1023 physical processors. Moreover, this concept can be extended to a shrink/expand feature, which allows an adaptive application executed on the ARTS to shrink or expand the set of physical nodes on which it runs at run time, adapting to changing load on workstation clusters, or enabling more flexible job scheduling for time-shared machines.

2.4 AMPI Interface Extension

Beyond what is specified in the MPI Standard, AMPI defines a set of extensions, including an extra option for running with virtual processes, and several additional calls for various extension functionalities. The extension functions' names start with *AMPI_* instead of the usual *MPI_*, and their syntax and behavior are explained as follows.

2.4.1 Running with Virtual Processes

When the user run an AMPI program the usual way with the *-np P* option, the program is launched on *P* processors, with one virtual process on each processor, hence without any adaptivity support. AMPI provides a *+vp VP* option to specify to total number of virtual processes to run on the *P* processors.

- *ampirun -np P pgm*: launch program *pgm* on *P* processors, with one virtual process on each processor. (*P* = *VP*)
- *ampirun -np P pgm +vp VP*: launch program *pgm* on *P* processors, with *VP* virtual processes.

2.4.2 Automatic Load Balancing Interface

AMPI provides three load balancing function calls as extension of the MPI Standard.

- *void AMPI_Migrate(void)*: collective call that signals possible load balancing point. This call suspends the execution of the virtual processes, even though the actual migration may or may not happen, depending on the LB Manager's decision.
- *void AMPI_Async_Migrate(void)*: collective call that starts load balancing session while allowing the application to continue, such that load balancing overlaps with computation. When the load balancing decision is available, the threads could be migrated asynchronously.

- *void AMPI_Setmigratable(int comm, int mig)*: collective call that enables or disables load balancing in a communicator according to the value of *mig*.
- *void AMPI_Migrateto(int destPE)*: local call to force migration of the calling VP to *destPE* without being directed by the LB Manager.

2.4.3 Automatic Checkpointing Interface

AMPI extension for checkpointing on disk and in peers' system memory includes two function calls.

- *void AMPI_Checkpoint(char *dname)*: collective call that initiates on-disk checkpointing of the current program into directory given by *dname*.
- *void AMPI_MemCheckpoint()*: collective call that initiates in-memory checkpointing. The peers on which the thread's data is saved are chosen by the run-time system.

2.4.4 Asynchronous Collective Communication Interface

Collective calls involves many or all MPI processes and are time-consuming. AMPI offers the flexibility of making non-blocking collective communication calls such as reduction and all-to-all. These calls help the user exploit the large gap between the elapsed time and CPU time of collective operations. Especially with the adaptivity support, when virtual processes in a communicator are blocked on a collective operation, the CPUs they reside on can be used by other virtual processes on the same physical processors.

The function interface is very much like the blocking counterpart, with an extra *MPI_Request* request* parameter returning the request handler.

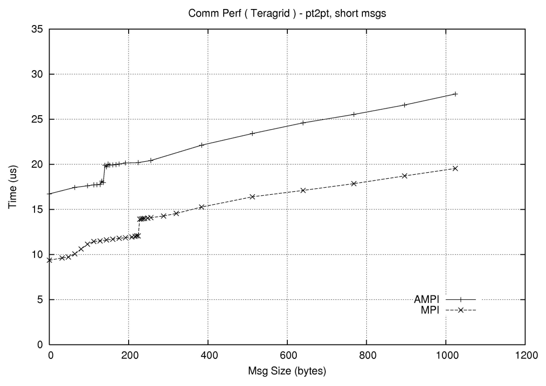
- *void AMPI_Ireduce(..., MPI_Request* request)*
- *void AMPI_Iallreduce(..., MPI_Request* request)*
- *void AMPI_Ialltoall(..., MPI_Request* request)*
- *void AMPI_Iallgather(..., MPI_Request* request)*

3 Performance Evaluation

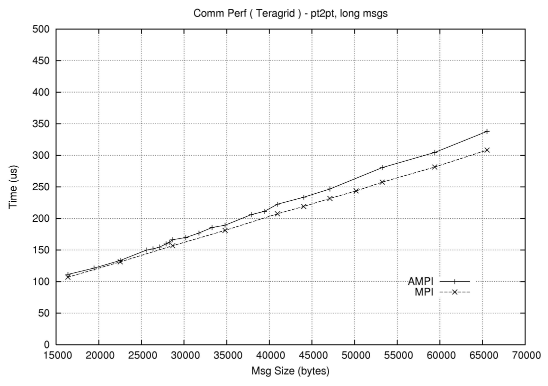
In this section, we present performance analysis of AMPI with various benchmarks and applications to demonstrate its advantages. Our main benchmarking platforms are the Turing Cluster with 640 dual Apple G5 nodes connected with Myrinet network and MPICH 1.2.6 at University of Illinois of Urbana-Champaign, NCSA's IA-64 TeraGrid Cluster with 888 dual Intel Itanium 2 nodes and Myrinet network installed with MPICH 1.2.5, NCSA's Tungsten Cluster with 1280 dual Intel Xeon nodes and Myrinet network with MPICH 1.2.5, and the Lemieux Cluster with 750 dual Alpha nodes and Quadrics network installed with MPICH 1.2.6 at Pittsburgh Supercomputer Center.

3.1 Comparison with MPI

As a competitive implementation of MPI, AMPI offers performance benefits and functionality extensions to MPI applications, especially those with dynamic nature. We now take a closer look at the comparison between AMPI and typical native MPI implementations, starting with a quantitative breakdown of the virtualization overheads of AMPI.



(a) Short Messages



(b) Long Messages

Figure 7: Point-to-point Performance on NCSA IA-64 Cluster

3.1.1 Virtualization Overheads

Because AMPI is implemented on top on ARTS, 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 pingpong style microbenchmark. Point-to-point performance on the benchmarking platforms are listed in Figures 8(a), 8(b) and 8(c). To explain the breakdown of the virtualization overhead in the ping-pong benchmark, 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 (See Figure 7(a)). For longer messages, we pay the overhead of extra message copying in order to support migratable threads (See Figure 7(b)). Active research work is being carried out to reduce overhead for both situations.

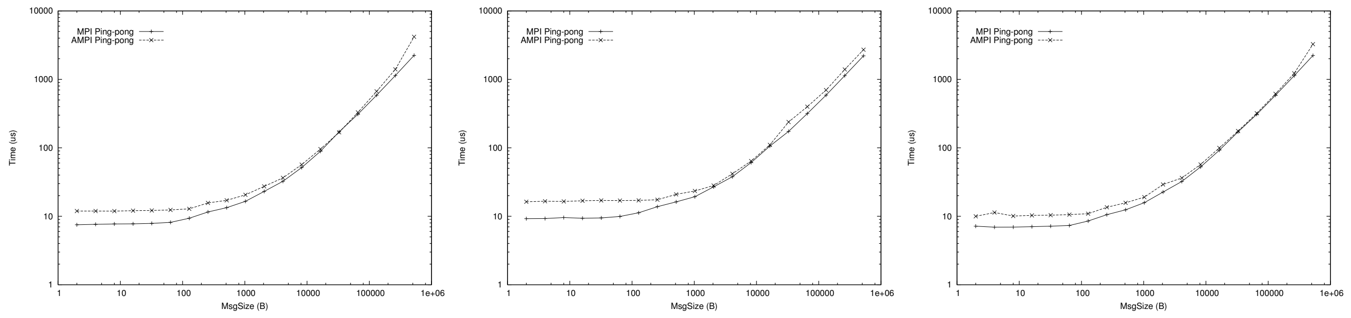
In practice, the virtualization overheads are usually hidden via appropriate subtask assignment. From our experiences, the few microseconds of virtualization overhead is negligible when the average work driven by one message is at hundreds of microseconds level, which is achieved by the choice of number of VPs per processor. It should be noted that such “good choice” has a fairly big tolerance, since the virtualization overhead is not very sensitive to the number of VPs per processor. 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 functionality extensions and performance advantages of AMPI do not come at an undue price in basic performance.

#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 1: 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	-	1786	-	701.7	-	-	367.3	-	-	212.2	-	917.0
AMPI	3633	1782	1367	696.9	710.5	705.0	363.6	387.4	385.2	206.9	250.0	921.7

Table 2: Timestep Time [ms] of 960^3 3D 7-point Stencil Calculation with AMPI v.s. Native MPI on NCSA IA-64 Cluster



(a) Turing (Apple G5) Cluster (b) NCSA IA-64 Cluster (c) Tungsten (Xeon) Cluster

Figure 8: Point-to-point Communication Time

3.1.2 Flexibility to Run

AMPI supports virtual MPI processes, thus giving the programmer the freedom to run multiple MPI processes on one physical processor. We illustrate this functionality extension with a more realistic benchmark, 3D stencil-type calculation. The 3D stencil calculation is a multiple timestepping calculation involving a group of regions in a mesh. At each timestep, every region 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.

With this 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 processor. 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.

On Lemieux clusters, 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 1 and 2. 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. For instance, during the development of the CSAR code, a specific software bug occurs only with 480 processor configuration run. Consequently, to debug it, the developers requires a 480 partition on parallel platforms to launch their problematic run. With AMPI, the 480 processor run can be performed on a much smaller partition that is easier to get, offering the developer with great productivity advantage.

3.1.3 Adaptive Overlapping

Performance benefit of adaptive overlapping arises from the fact that the actual CPU overhead in a blocking communication operation is typically smaller than the total elapsed time. We show this with the following multi-ping benchmarks. In the benchmark, processor A sends multiple ping messages to processor B, which responds with a short pong message after it has received all. This communication pattern differs from the usual ping-pong benchmark in that it fills the pipeline on a message's path from sender to receiver: sender CPU, sender NIC, interconnect, receiver NIC, and receiver CPU. The performance from multi-ping benchmark represents the limiting factor in the pipeline, namely the due price for a point-to-point communication. The gap between its curve and the ping-pong benchmark's curve is the time wasted on waiting for the communication to complete. Figure 8(a) shows big gap between the two curves, suggesting that much of the time usually attributed to communication can be utilized for useful computation. Many traditional MPI implementations, however, cannot take advantage of this gap easily.

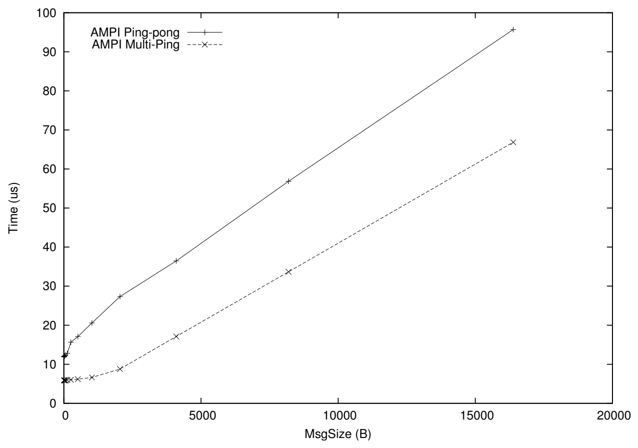


Figure 9: Performance of Ping-pong vs Multi-ping Benchmark on Turing (Apple G5) Cluster

K	VP=8	VP=16	VP=32	VP=64	MPICH
128	25.0	25.8	25.8	25.8	10.6
256	97.7	96.8	91.3	90.4	100.1
512	744.9	772.7	776.7	770.6	751.6
1024	7418	6545	5908	5894	7437

Table 3: Iteration Time [ms] of K^3 3D 7-point Stencil Calculation on 8 PEs of NCSA IA-64 Cluster

In AMPI, several VPs can be mapped onto one physical processor. This design 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, as illustrated by the following stencil calculation benchmark.

Table 3 shows the iteration time of 3D stencil calculations on 8 physical processors on NCSA IA-64 Cluster. The calculations are of different sizes K^3, and with AMPI of various VP numbers. It can be observed that the overall performance increases when more VPs are mapped on one processor in the given range. The underlying reason is illustrated by the projections visualization in Figure 10. The solid blocks represent computation and the gaps are idle time when CPU is waiting for communication to complete. As the number of VPs per processor 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.

It can also be observed that virtualization does not always result in performance improvement. As we introduce more VPs on one processor, virtualization overhead might overweigh the benefit from adaptive overlap and cause longer execution time. In Section 3.1, we will discuss in more detail how the benefit of overlap and virtualization overhead are correlated with the number of VPs per processor.

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.

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 the load balancing framework, AMPI supports automatic measurement-based dynamic load balancing and thread migration. In this section, we present the case studies of load balancing several MPI benchmarks and a real-world application.

NAS Parallel Benchmark (NPB) is a well known parallel benchmark suite. Its Multi-Zone version, LU-MZ,

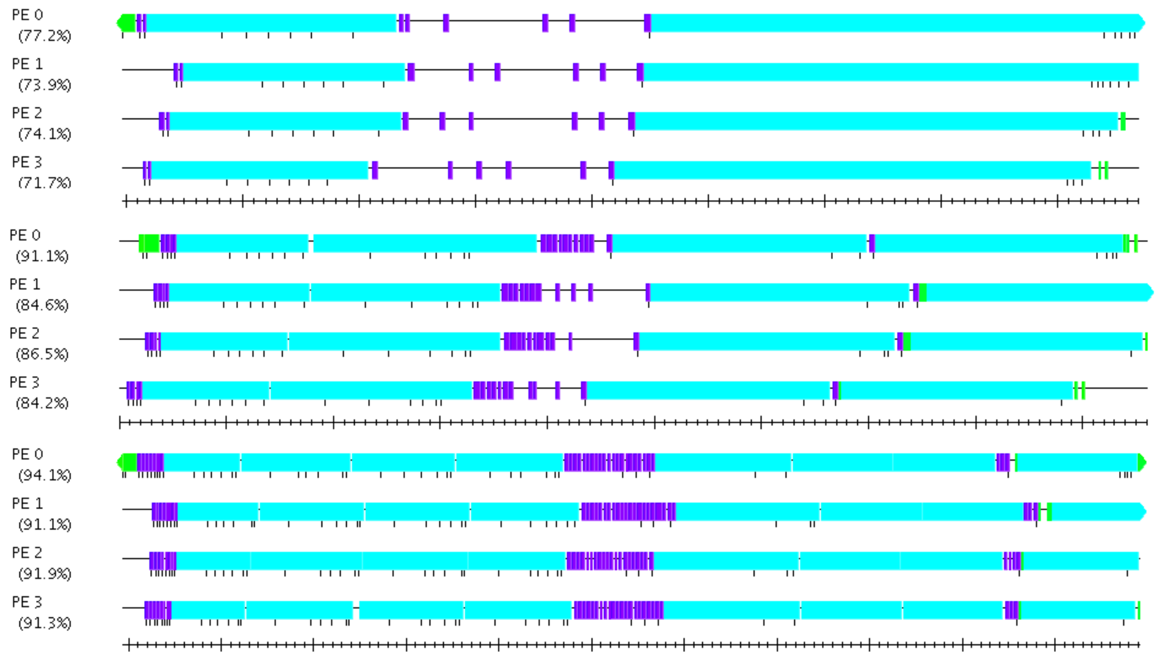


Figure 10: 7-point Stencil Timeline with 1, 2 and 4 VPs Per Processor

SP-MZ and BT-MZ, "solve discretized versions of the unsteady, compressible Navier-Stokes equations in three spatial dimensions"[23]. 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 [24]: 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.

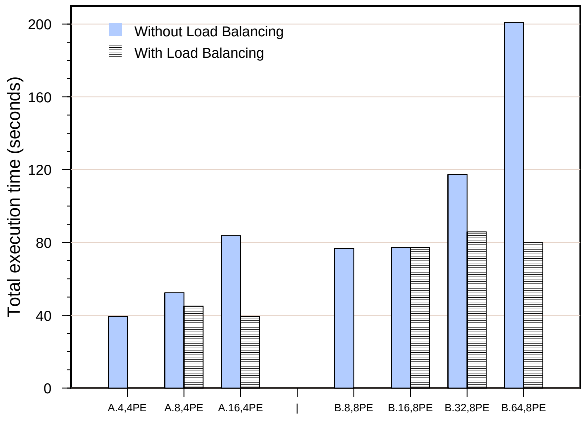


Figure 11: Load Balancing on NAS BT-MZ

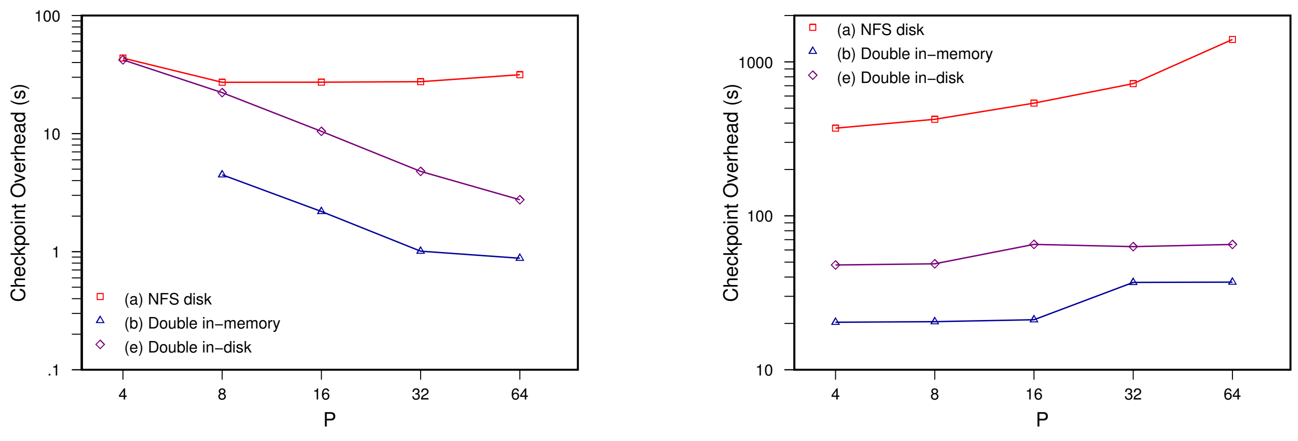
In this benchmark, we plant in the function call to trigger the automatic load balancing in AMPI run-time system. After 3 timesteps, when the run-time has collected sufficient information to advise the load balancer, the AMPI VPs are migrated from heavier-loaded processors onto lighter-loaded ones. The execution time is visualized in Figure 11.

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, having more VPs per processor allows the load balancing module to work more effectively, simply because there are more threads to move around if necessary. That is the reason that having number of VP much larger 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 experienced where such load balancing is useful. For the other type of application with dynamically varying workload, load balancer can be triggered periodically. The application example in Section 4.2 exemplifies this use.

3.3 Checkpoint Overhead

To illustrate the checkpoint overhead of AMPI, we perform our experiments with two NAS benchmarks FT and LU class B on the Turing cluster. The total amount of data to save is different: FT class B has nearly 2GB regardless of number of processor, while LU class B has saved data size roughly proportional to the number of processors, so the per processor data is nearly constant.



(a) FT Class B

(b) LU Class B

Figure 12: Checkpoint Overhead of NAS Benchmark on Turing Cluster

The results are shown in Figure 12(a) for FT and Figure 12(b) for LU. The x-axis is increasing number of processors, the y-axis is checkpoint time in second, and both axes are in logarithmic scale. In each figure, there are 3 curves representing 3 series of runs with the benchmark. The "NFS disk" scheme saves data to NFS disk. The "Double in-memory" and "Double in-disk" schemes saves data in peers' memory or local-disk to avoid NFS disk bottleneck, and uses double checkpointing to ensure single fault tolerance.

For both benchmarks, the NFS disk scheme is the most expensive. Thanks to RAID disks, it scales on 4 and 8 processors for FT-B, but beyond that, the NFS bottleneck becomes the limiting factor and the performance deteriorates as number of processors increases. The two double checkpointing runs utilize the fast Myrinet interconnect

	AMPI						MPI
P	16	30	60	120	240	480	480
Time(s)	15.33	8.41	5.02	3.01	1.66	2.415	2.732

Table 4: *Rocstar* Performance Comparison of 480-processor Dataset for Titan IV SRMU Rocket Motor on Apple Cluster

to transfer checkpoint data, and have similar scaling behavior. Because writing to local hard disk is not as fast as storing to peers' memory, we observe that the overhead of in-disk variation is always higher than its in-memory counterpart. With lower overhead from in-memory scheme, we can checkpoint the program more often, and hence reduce the work lost since last checkpoint when a fault occurs.

There is one interesting point to note here. In Figure 12(a) with FT class B, the double checkpoint scheme is unable to run on 4 processors, because the memory footprint for that specific benchmark (2GB) is too large for the machine. The same problem occurs on systems with relatively small per-node memory configuration, including IBM's BlueGene/L. In this scenario, the user has two potential solutions. First, the user can use the in-disk variation of double checkpoint scheme. However, when a local disk is unavailable, as on BlueGene/L, the on-disk checkpoint can still serve the purpose. Moreover, when the socket error detection, based on which the double checkpoint scheme is built, is missing, the on-disk scheme becomes the only choice.

4 Application Case Study

In this section, we discuss two parallel applications which benefit from AMPI. These two projects are part of our collaboration with scientific and engineering researchers, and the benefits are not limited to performance gains.

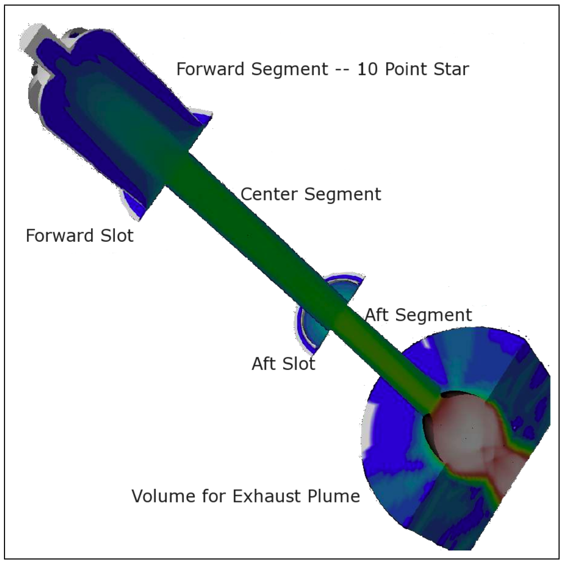
4.1 Rocstar

The Center for Simulation of Advanced Rockets (CSAR) is 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. CSAR consists of several dozen faculty and professional staff from a number of different engineering and science departments. The main CSAR simulation code consists of several major components in various domains, including a fluid dynamics simulation, for the hot gas flowing through and out of the rocket; a surface burning model for the solid propellant; a nonmatching but fully-coupled fluid/solid interface; and finally a finite-element solid mechanics simulation for the solid propellant and rocket casing. Each one of these components - fluids, burning, interface, and solids - began as an independently developed parallel MPI program.

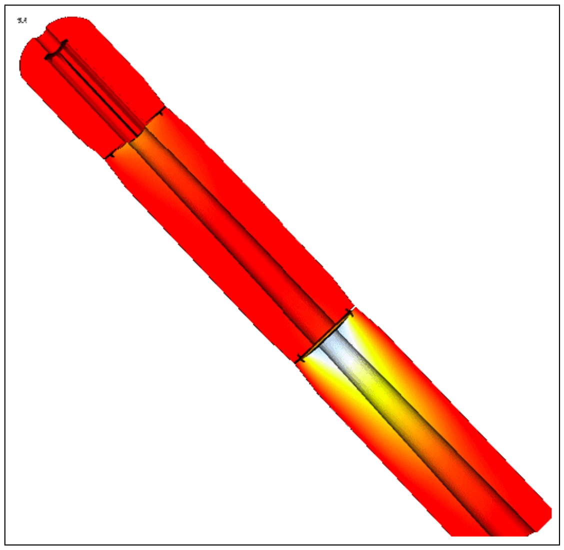
One of the most important early benefits CSAR found in using AMPI is the ability to run a partitioned set of input files on a different number of virtual processors than physical processors. For example, a CSAR developer was faced with an error in mesh motion that only appeared when a particular problem 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 32 physical processors of a local cluster, which made resolving the error much faster and easier.

Because each of the CSAR simulation components are developed independently, and each has its own parallel input format, there are difficult practical problems involved in simply preparing input meshes that are partitioned for the correct number of physical processors available. Using AMPI, CSAR developers can simply use a fixed number of virtual processors, which allows a wide range of physical processors to be used without repartitioning the problem's input files.

To demonstrate the performance benefits of virtualization using AMPI, we compared the performance of *Rocstar* using AMPI and MPICH/GM on different numbers of processors of the Turing Apple cluster with Myrinet



(a) Cutaway View of Fluids Domain



(b) Propellant Deformation After One Second

Figure 13: Titan IV Propellant Slumping Visualization

interconnect at CSAR. Our test used a 480-processor dataset of the Titan IV SRMU Prequalification Motor #1. This motor exploded during a static test firing on April 1, 1991 due to excessive deformation of the aft propellant segment just below the aft joint slot [25]. Figure 13 shows a cutaway view of the fluids domain and the propellant deformation, obtained from *Rocstar*'s 3-D simulations at nearly one second after ignition for an incompressible neoHookean material model.

We ran *Rocstar* using AMPI (implemented on the native GM library) on various number of physical processors ranging from 16 to 480 processors, and ran the same simulation with MPICH/GM on 480 processors. Table 4 shows the wall-clock times per iteration. The AMPI-based run outperformed the MPICH/GM-based by about 12% on 480 processors, demonstrating the efficiency of our AMPI implementation directly on top of the native GM library. Note that even better performance was obtained on 240 processors with two AMPI threads per physical processor. This virtualization allowed the AMPI run-time system to dynamically overlap communication with computation to exploit the otherwise idle CPU cycles while reducing interprocessor-communication overhead for the reduction in the number of physical processors, leading to a net performance gain for this test.

4.2 Fractography3D

Fractography3d is a dynamic 3D crack propagation simulation program to simulate pressure-driven crack propagation in structures. It was developed by Professor Philippe Guebelle of Department of Aerospace Engineering at University of Illinois and his students and collaboration with our group. Fractography3d code is implemented on the FEM framework [26] and AMPI.

For this experiment, the application simulates a crack propagation with a force and the process of elastic turning into plastic zone along the crack, as illustrated in Figure 14. The crack propagation simulation was run with 1000 AMPI virtual processors on 100 processors of the Turing Cluster.

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*. At each iteration of the simulation, pressure exerted upon the plastic

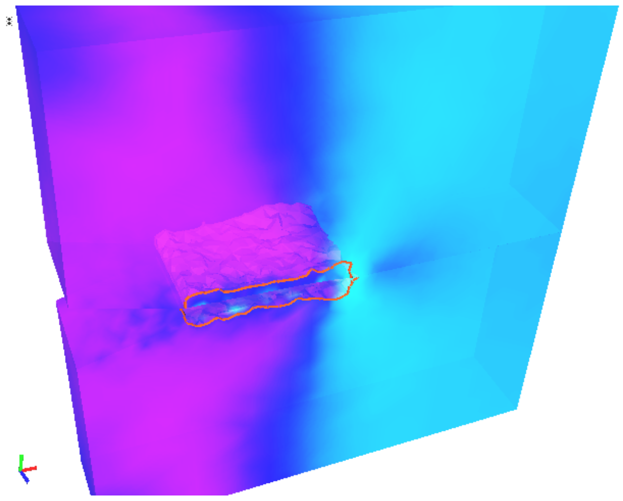


Figure 14: Fractography3D: Crack Propagation Visualization

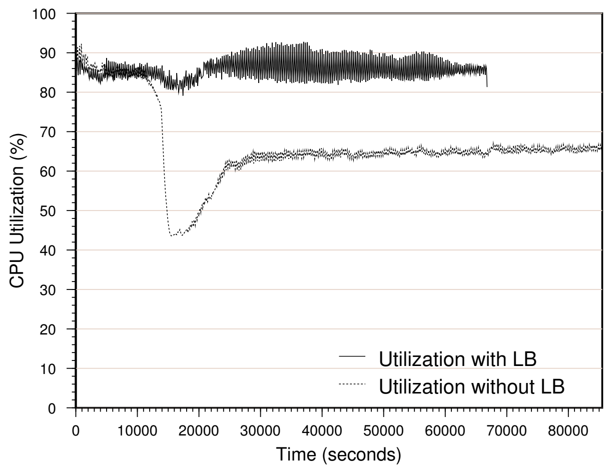
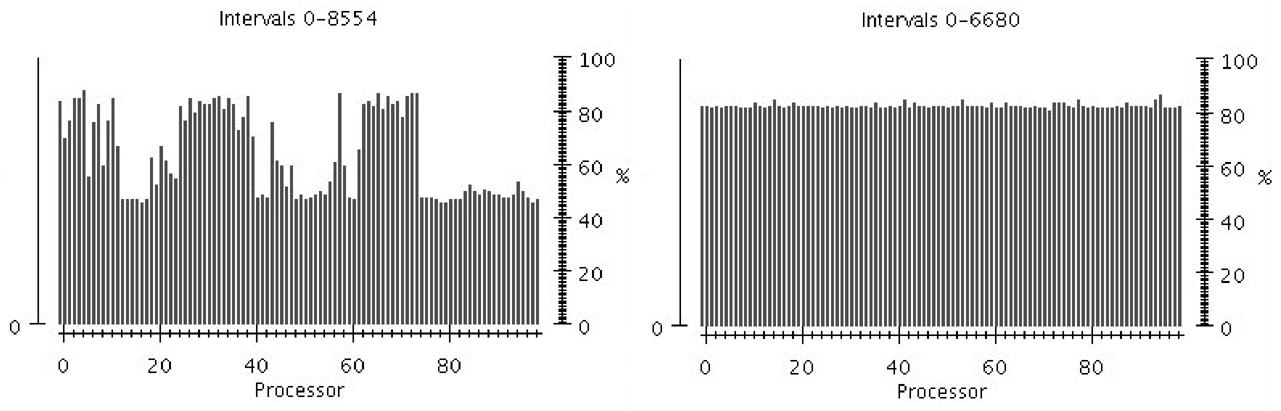


Figure 15: CPU Utilization Projections Graph of Fractography3D Over Time Without vs. With Load Balancing



(a) Without Load Balancing

(b) With Load Balancing

Figure 16: CPU Utilization Graphs of Fractography3D Across Processor Without vs. With Load Balancing

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.

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 15. 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 16. 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 15 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 run-time to migrate chunks from the overloaded processor to underloaded processors, leading to improved performance. As figure 15 shows, the overall CPU utilization on all processors throughout the entire simulation stays around 80-90%. The lower half of Figure 16 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.

5 Conclusions

We presented AMPI, an adaptive implementation of MPI on top of our adaptive run-time system. AMPI implements migratable virtual and light-weight MPI processors. 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 communication and computation, 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. Meanwhile AMPI is also an active research project; much future work is planned. AMPI already supports some 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. Our performance analysis tools such as Projections [27] are being updated to provide more direct support for AMPI programs. 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] Orion Lawlor, Sayantan Chakravorty, Terry Wilmarth, Nilesh Choudhury, Isaac Dooley, Gengbin Zheng, and Laxmikant Kale. Parfum: A parallel framework for unstructured meshes for scalable dynamic physics applications. *Engineering with Computers*, 22(3-4):215–235.

[2] Sameer Kumar, Chao Huang, Gheorghe Almasi, and Laxmikant V. Kalé. Achieving strong scaling with NAMD on Blue Gene/L. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2006*, April 2006.

[3] 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.

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

[5] 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.

[6] Gengbin Zheng, Orion Sky Lawlor, and Laxmikant V. Kalé. Multiple flows of control in migratable parallel programs. pages 435–444, Columbus, Ohio, August 2006. IEEE Computer Society.

[7] Hong Tang, Kai Shen, and Tao Yang. Program transformation and runtime support for threaded MPI execution on shared-memory machines. *ACM Transactions on Programming Languages and Systems*, 22(4):673–700, 2000.

[8] R. Wismller, T. Ludwig, A. Bode, R. Borgeest, S. Lamberts, M. Oberhuber, C. Rder, and G. Stellner. The tool-set project: Towards an integrated tool environment for parallel programming. In *Proceedings of Second Sino-German Workshop on Advanced Parallel Processing Technologies, APPT'97, Koblenz, Germany*, pages 9–16. Verlag Dietmar Folbach, September 1997.

[9] Georg Stellner. CoCheck: Checkpointing and Process Migration for MPI. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS '96)*, Honolulu, Hawaii, 1996.

[10] Lorna Smith and Mark Bull. Development of mixed mode mpi / openmp applications. *Scientific Programming*, 9(2-3/2001):83–98. Presented at Workshop on OpenMP Applications and Tools (WOMPAT 2000), San Diego, Calif., July 6-7, 2000.

[11] Karen D. Devine, Erik G. Boman, Robert T. Heaphy, Bruce A. Hendrickson, James D. Teresco, Jamal Faik, Joseph E. Flaherty, and Luis G. Gervasio. New challenges in dynamic load balancing. *Appl. Numer. Math.*, 52(2–3):133–152, 2005.

[12] P. Colella, D.T. Graves, T.J. Ligocki, D.F. Martin, D. Modiano, D.B. Serafini, and B. Van Straalen. Chombo Software Package for AMR Applications Design Document, 2003. http://seesar.lbl.gov/anag/chombo/ChomboDesign-1.4.pdf.

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

[14] 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.

[15] Neelam Saboo, Arun Kumar Singla, Joshua Mostkoff Unger, and L. V. Kalé. Emulating petaflops machines and blue gene. In *Workshop on Massively Parallel Processing (IPDPS'01)*, San Francisco, CA, April 2001.

[16] Joy Mukherjee and Srinidhi Varadarajan. Weaves: A framework for reconfigurable programming. *International Journal of Parallel Programming*, 33(2-3):279–305, 2005.

[17] Karthikeyan Mahesh. Ampizer: An mpi-ampi translator. Master's thesis, Computer Science Department, University of Illinois at Urbana-Champiagn, 2001.

[18] 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.

[19] 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. Lecture Notes in Computer Science 1586*, pages 496–510. Springer-Verlag, April 1999.

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

[21] Gengbin Zheng, Chao Huang, and Laxmikant V. Kalé. Performance evaluation of automatic checkpoint-based fault tolerance for ampi and charm++. *ACM SIGOPS Operating Systems Review: Operating and Runtime Systems for High-end Computing Systems*, 40(2), April 2006.

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

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

[24] 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.

[25] W. G. Wilson, J. M. Anderson, and M. Vander Meyden. Titan IV SRMU PQM-1 overview, 1992. AIAA Paper 92-3819.

[26] 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.

[27] Laxmikant V. Kale, Gengbin Zheng, Chee Wai Lee, and Sameer Kumar. Scaling applications to massively parallel machines using projections performance analysis tool. In *Future Generation Computer Systems Special Issue on: Large-Scale System Performance Modeling and Analysis*, volume 22, pages 347–358, February 2006.