Adaptive MPI

Chao Huang, Orion Lawlor, L. V. Kalé
chuang10@uiuc.edu, olawlor@acm.org, l-kale1@uiuc.edu

Parallel Programming Laboratory
University of Illinois at Urbana-Champaign

Abstract. Processor virtualization 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 processor virtualization. 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 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. This paper describes AMPI, its features, benchmarks that illustrate performance advantages and tradeoffs offered by AMPI, and application experiences.

1 Introduction

The new generation of parallel applications are complex, involve simulation of dynamically varying systems, use adaptive techniques such as multiple timestepping and adaptive refinements, and often involve multiple parallel modules. Typical implementations of the MPI do not support the dynamic nature of these applications well. As a result, programming productivity and parallel efficiency suffer. We present AMPI, an adaptive implementation of MPI, that is better suited for such applications, while still retaining the familiar programming model of MPI.

The basic idea behind AMPI is to separate the issue of mapping work to processors from that of identifying work to be done in parallel. Standard MPI programs divide the computation into P processes, one for each of the P processors. In contrast, an AMPI programmer divides the computation into a large number V of virtual processors, independent of the number of physical processors. The virtual processors are programmed in MPI as before. Physical processors are no longer visible to the programmer, as the responsibility for assigning virtual processors to physical processors is taken over by the runtime system. This provides an effective division of labor between the system and the programmer: the programmer decides *what* to do in parallel, and the runtime system decides *where* and *when* to do it. This division allows the programmer to use the most natural decomposition for their problems, rather than being restricted by the physical machine. For example, algorithmic considerations often restrict the number of processors to a power of 2, or a cube, but with AMPI, V can still be a cube even though P is prime.

Note that the number of virtual processors V is typically much larger than P. Using multiple virtual processors per physical processor brings several additional benefits.

1.1 Related Work

The virtualization concept embodied by AMPI is very old, and Fox et al. [1] make 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 [2] for data-parallel array based applications. CHARM++ is one of the earliest, if not the first, processor-virtualization system implemented on parallel machines[3, 4]. 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 [5] and MPI/LAM [6]. Many researchers have described their implementations for fault-tolerance via checkpoint/restart, often built on top of one of the free implementations of MPI like CoCheck[7] and StarFish [8]. AMPI differs from these efforts in that it provides full virtualization to improve performance and allow load balancing rather than solely for checkpointing or for fault tolerance.

Meanwhile there are plenty of efforts in implementing MPI nodes on top of lightweight threads. MPI-Lite [9] and TMPI [10] are two good examples. They have successfully used threaded execution to improve the performance of message passing programs, especially on SMP machines. Adaptive MPI, however, enables extra optimization with the capability of migrating the user-level threads that MPI processors are executed on.

The CHARM++/AMPI approach is to let the runtime system change the assignment of VPs to physical processors at runtime, thereby enabling a broad set of optimizations. In the next section, we motivate the project, providing an overview of the benefits. In Section 3 we describe how our virtual processors are implemented and migrated. Section 4 describes the design and implementation strategies for specific features, such as checkpointing and load-balancing. We then present performance data showing that these adaptive features are beneficial in complex applications, and affordable (i.e. present low overhead) in general. We will summarize our experience in using AMPI in several large applications.

2 Benefits of Virtualization

In [11], the author has discussed in detail the benefits of processor virtualization in parallel programming, and CHARM++ has indeed taken full advantage of these benefits. Adaptive MPI inherits most of the merits from CHARM++, while furnishing the common MPI programming environment. Here is a list of the benefits that we will detail in this paper.

- *Adaptive overlap 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 this kind of load balancing decision based on automatic instrumentation, as explained in Section 4.1.
- *Asynchronous interfaces to collective operations:* AMPI supports asynchronous, or non-blocking, interfaces to collective communication operations to allow the overlap between time-consuming collective operations with other useful computation. Section 4.2 describes this in detail.
- *Automatic checkpointing:* AMPI's virtualization allows applications to be checkpointed without additional user programming, as described in Section 4.3.
- *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.
- *Flexible usage of available processors:* The ability to migrate virtual processors can be used to adapt the computation if the available part of the physical machine changes. See Section 4.5 for details.

3 Adaptive MPI

3.1 AMPI Implementation

AMPI is built on CHARM++, and uses its communication facilities, load balancing strategies and threading model.

CHARM++ uses an object based model: programs consist of a collection of message driven objects mapped onto physical processors by CHARM++ runtime system. The objects communicate with other objects by invoking an asynchronous entry method on the remote object. Upon each of these asynchronous invocation, a message is generated and sent to the destination processor where the remote object resides. Adaptive MPI implements its MPI processors as CHARM++ "user-level" threads bound to CHARM++ communicating objects.

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 migration of objects via efficient data migration and message forwarding if necessary. Migration presents interesting problems for basic and collective communication which are effectively solved by the CHARM++ runtime system[12].

Migration can be used by the built-in measurement-based load balancing [13], adapting to changing load on workstation clusters [14], and even shrinking/expanding jobs for timeshared machines [15].

The threads used by AMPI are 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[1], control over scheduling, and control over stack allocation. Thus, it is feasible to run thousands of such threads on one physical processor (e.g. See [16]). CHARM++'s user-level threads are scheduled non-preemptively.

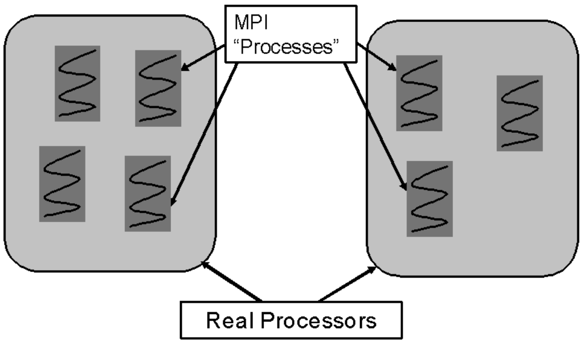

Fig. 1. An MPI process is implemented as a user-level thread, several of which can be mapped to one single physical processor. This virtualization enables several powerful features including automatic load balancing and adaptive overlapping.

3.2 Writing An AMPI Program

Writing an AMPI program is barely different from writing an ordinary MPI program. In fact, a legal MPI program is also a legal AMPI program. To take full advantage of the migration mechanism, however, there is one more issue to address: global variables.

Global variable is any variable that is stored at a fixed, preallocated location in memory. Although not specified by the MPI standard, many actual MPI programs assume that global variables can be used independently on each processor, i.e., global variable x on processor 1 can have a different value than that of global variable x on processor 2. However, in AMPI, all the threads on one processor share a single address space and thus a single set of global variables; and when a thread migrates, it leaves its global variables behind. Another problem is global variables shared on the same processor might be changed by other threads. Therefore, having global variables is disallowed in AMPI programming.

3.3 Converting MPI Programs To AMPI

If the MPI program uses global variables, it cannot run unmodified under AMPI, and we need to convert it to fit AMPI. As discussed in section 3.2, for thread safety, global variables need to be either removed or "privatized". To remove the global variables from

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

the code, one can collect all the formal globals into a single structure (allocated "type" in F90) named, say, "GlobalVars", which is then passed into each function.

To manually remove *all* the global variables is sometimes cumbersome, though mechanical. Fortunately this can be automated. *AMPIzer* [17] is our source-to-source translator based on Polaris[18] that privatizes global variables from arbitrary FORTRAN77 or FORTRAN90 code and generates necessary code for moving the data across processors.

4 Features

In this section, the key features that can help achieving higher parallel performance and alleviate the complexity of parallel programming will be discussed in detail.

4.1 Automatic Load Balancing

To achieve automatic dynamic load balancing without introducing an excessive amount of overhead poses fair challenges. CHARM++ addresses this issue with its integrated load balancing strategies, or Load Balancers[13]. The common mechanism they share is: during the execution of the program, a load balancing framework collects workload information on each physical processor in the background, and when the program hands over the control to a load balancer, it uses this information to redistribute the workload, and migrate the parallel objects between the processors as necessary.

As there are different answers to the questions of (1) what information to collect, (2) where the information is processed, and (3) how to design the redistribution scheme, there are different types of load balancing strategies. For the first question, some load balancers look at computation workload only, while others take inter-processor communication into consideration. For the second question, some load balancers contribute the information to a central agent in the system for processing, whereas others only have objects exchange information with their neighbors and make decisions locally. At the last link, some load balancers randomly redistribute the workload and hope for the best, as opposed to having deliberate algorithms to help determine the new distribution toward better balance. For more detail, please refer to [13] and CHARM++ manuals.

A key issue in automatic load balancing is to cleanly move objects from one processor to another processor. CHARM++ natively supports object migration; but in the context of AMPI, thread migration required several interesting additions to the runtime system, as described in the following sections.

Isomalloc Stacks A user-level thread, when suspended, consists of a stack and a set of preserved machine registers. During migration, the machine registers are simply copied to the new processor. The stack, unfortunately, is very difficult to move. In a distributed memory parallel machine, if the stack is moved to a new machine, it will almost undoubtedly be allocated at a different location, so existing pointers to addresses in the original stack would become invalid when the stack moves. We cannot reliably update all the pointers to stack-allocated variables, because these pointers are stored in machine registers and stack frames, whose layout is highly machine- and compiler-dependent.

Our solution is to ensure that even after a migration, a thread's stack will stay at the same address in memory that it had on the old processor. This means all the pointers embedded in the stack will still work properly. Luckily, any operating system with virtual memory support has the ability to map arbitrary pages in and out of memory. Therefore we merely need to `mmap` the appropriate address range into memory on the new machine and use it for our stack. To ensure that each thread allocates its stack at a globally unique range of addresses, the available virtual address space is divided into P regions, each for one thread respectively. This idea of "isomalloc" approach to thread migration is based on PM^2 [19].

Isomalloc Heaps Another obvious problem with migrating an arbitrary program is dynamically allocated storage. Unlike the thread stack, which the system allocated, dynamically allocated locations are known only to the user program.

The "isomalloc" strategy available in the latest version of AMPI uses the same virtual address allocation method used for stacks to allocate all heap data. Similarly, the user's heap data is given globally unique virtual addresses, so it can be moved to any running processor without changing its address. Thus migration is transparent to the user code, even for arbitrarily interlinked, dynamically allocated data structures. To do this, AMPI must intercept and handle all memory allocations done by the user code. On many UNIX systems, this can be done by providing our own implementation of `malloc`. Machines with 64-bit pointers, which are becoming increasingly common, support a large virtual address space and hence can fully benefit from isomalloc heaps.

Limitations During migration, we do not preserve a thread's open files and sockets, environment variables, or signals. However, threads are only migrated when they call the special API routine MPI_Migrate, so currently the non-migration-safe features can be used at any other time. The intention is to support these operations via a thread-safe AMPI specific API, which will work with migration, in the future. Thread migration between different architectures on a heterogeneous parallel machine is also not supported.[2]

4.2 Collective Communication Optimization

Collective communications are required in many scientific applications, as they are used in many basic operations like high dimensional FFT, LU-factorization and linear algebra operations. These communications involves many or all processors in the system, which makes them complex and time-consuming. AMPI uses the CHARM++ communication library[20, 21] to optimize its collective communication. This library uses two intelligent techniques in optimizing collective communications. For small messages, messages are combined and routed via intermediate processors to reduce the software overhead. For large messages, network contention, the dominant factor in the total cost, is lowered by smart sequencing of the messages based on the underlying network topology.

Beside the above optimization inherited from CHARM++, AMPI has its own improvement on the collective communication operations. If we take a closer look at the time spent on collective communications, only a small portion of the total time is

[2] This will require extensive compiler support or a common virtual machine. Alternatively, stack-copying threads along with user-supplied pack/unpack code can be used to support AMPI in heterogeneous environment.

software overhead, namely the time CPU spends on communication operations. Especially, a modern NIC with communication co-processor performs message management through remote DMA so that this operation requires very little CPU interference. On the other hand, the MPI standard defines collective operations like MPI_Alltoall and MPI_Allgather to be blocking, wasting the CPU time on waiting for the communication calls to return. To better utilize the computing power of CPU, we can make the collective operations non-blocking to allow useful computation while other MPI processors are waiting for slower collective operations.

In IBM MPI for AIX [22], the similar non-blocking collectives were implemented but not well benchmarked or documented. Our approach differs from IBM's in that we have more flexibility of overlapping, since the light-weight threads we use are easier to schedule to make full use of the physical processors.

4.3 Checkpoint and Restart

As Stellner describes in his paper on his checkpointing framework [23], process migration can easily be layered on top of any checkpointing system by simply rearranging the checkpoint files before restart. AMPI implements checkpointing in exactly the opposite way. In AMPI, rather than migration being a special kind of checkpoint/restart, checkpoint/restart is seen as a special kind of migration - migration to and from the disk.

A running AMPI thread checkpoints itself by calling `MPI_Checkpoint` with a directory name. Each thread drains its network queue, migrates a copy of itself into a file in that directory, and then continues normally. The checkpoint time is dominated by the cost of the I/O, since very little communication is required.

There are currently two ways to organize the checkpoint files: (1) All threads on the same physical processor will group into one single disk file to reduce the number of files to be created, (2) Each thread has its own file. In the second option, because AMPI system checkpoints threads rather than physical processors, an AMPI program may be restored on a larger or smaller number of physical processors than was it started on. Thus a checkpoint on 1000 processors can easily be restarted on 999 processors if, for example, a processor fails during the run.

4.4 Multi-module AMPI

Large scientific programs are often written in a modular fashion by combining multiple MPI modules into a single program. These MPI modules are often derived from independent MPI programs.

Current MPI programs transfer control from one module to another strictly via subroutine calls. Even if two modules are independent, idle time in one cannot be overlapped with computations in the other without breaking the abstraction boundaries between the two modules. In contrast, AMPI allows multiple separately developed modules to interleave execution based on the availability of messages. Each module may have its own "main", and its own flow of control. AMPI provides *cross-communicators* to communicate between such modules.

4.5 Shrink-Expand Capability

AMPI normally migrates virtual processors for load balance, but this capability can also be used to respond to the changing properties of the parallel machine. For example, Figure 2 shows the conjugate gradient solver responding to the availability of several new processors. The time per step drops dramatically as virtual processors are migrated onto the new physical processors.

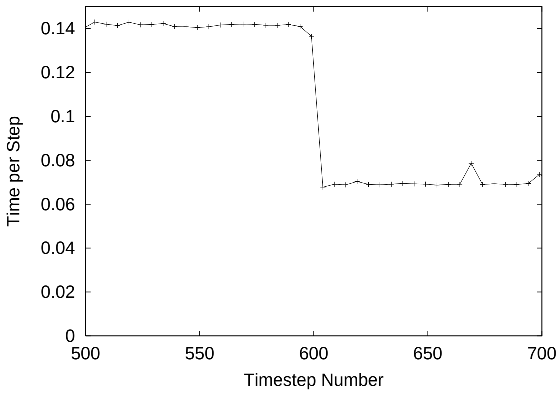

Fig. 2. Time per step for the million-row conjugate gradient solver on a workstation cluster. Initially, the application runs on 16 machines. 16 new machines are made available at step 600, which immediately improves the throughput.

5 AMPI Benchmarks

In this section we use several benchmarks to illustrate the aspects of performance improvement that AMPI is capable of. One of the basic benchmarks here is 2D grid-based 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 neighbors and does some computation based on the neighbors' data. The objects can be organized in a 2D or 3D mesh, and 1-away or 2-away neighbors may be involved. Depending on these different choices, the number of points in the stencil computation can range from 5 to 13. Although this is a simplified model of many applications, like fluid dynamics or heat dispersion simulation, it can well serve the purpose of demonstration. We have chosen Lemieux, the supercomputerat Pittsburgh Supercomputing Center [24], as the major benchmark platform.

5.1 Adaptive Overlapping

In Adaptive MPI, Virtual Processors are message-driven objects mapped onto physical processors. Several VPs can be mapped onto one physical processor, and the message passing among VPs is really communication between these objects.

We have explained this in Section 3.1. Now we will show the first benefit of virtualization: adaptive overlapping of computation with communication and it can improve the utilization of CPUs.

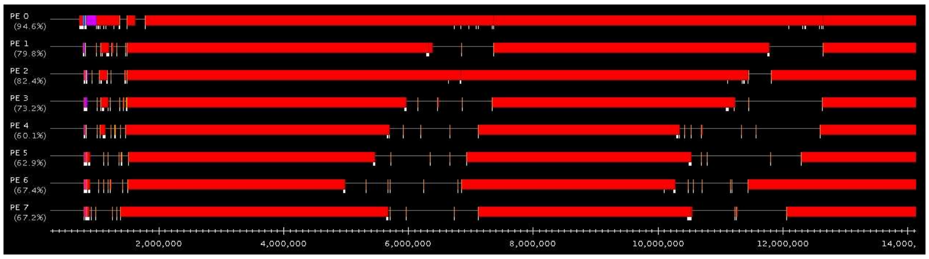

Fig. 3. Timeline of 1024^2 2D 5-point stencil calculation on Lemieux. No virtualization is used in this case: one VP per processor.

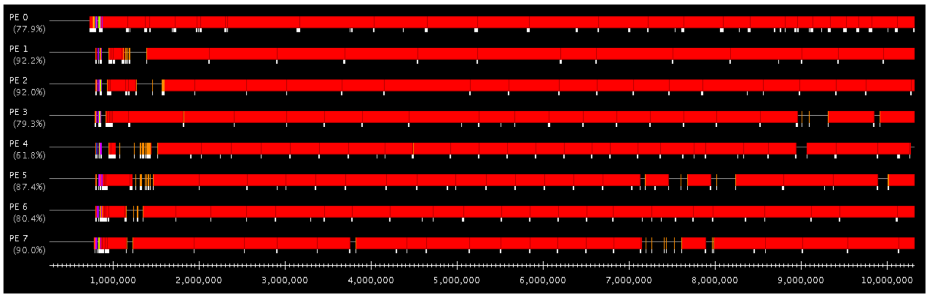

Fig. 4. Timeline of 1024^2 2D 5-point stencil calculation on Lemieux. Virtualization ratio is 8: eight VPs created on each processor.

Figures 3 and 4 are the timeline from the visualization tool for CHARM++: Projections[3]. In the timelines, x direction is time and y direction shows 8 physical processors. For each processor, the solid block means it is in use, while the gap between blocks is idle time. Figures shown are from 2 separate runs of 2D 5-point stencil calculation. In the first run, only one VP is created on each physical processor, so there is no virtualization allowed. In the second run, 8 VPs are created for each physical processor, with each VP taking less amount of computation, the total problem size is the same. In the displayed portion of the execution time, in Figure 3 we can see there are obvious gaps between blocks, and the overall utilization is around 70%. This illustrates the CPU time wasted while waiting for blocking communication to return. In Figure 4, however, the gaps of communication are filled with smaller chunks of computation: when one object is waiting for its communication to return, other objects on the processor can automatically take over and do their computation, eliminating the need for manual arrangement. With the adaptive overlapping of communication and computation, the average utilization of CPU is boosted to around 80%.

[3] Manual available at http://finesse.cs.uiuc.edu/manuals/

5.2 Automatic Load Balancing

In parallel programming, load imbalance is to be very carefully avoided. Unfortunately, load imbalance, especially dynamic load imbalance, appears frequently and is difficult to remove. For instance, consider a simulation on a mesh, where part of the mesh has a more complicated structure than the rest of the mesh, and the load within this mesh is imbalanced. As another example, when adaptive mesh refinement (AMR) is in use, hot-spots can arise where the mesh structure is highly refined. This dynamic type of load imbalance requires more programmer/system interference to remove. AMPI, using the automatic load balancing mechanism integrated in CHARM++ system, accomplishes the task of removing static and dynamic load imbalance automatically.

As a simple benchmark, we modified the 5-point stencil program by dividing the mesh in a 2D stencil calculation into 2 part: in the first 1/16 mesh, all objects do 2-away (13-point) calculation, while the rest do 1-away (5-point) calculation. The load on the 1/16 processors is thus much heavier than that on the rest 15/16. The program used 128 AMPI VPs on 16 processors.

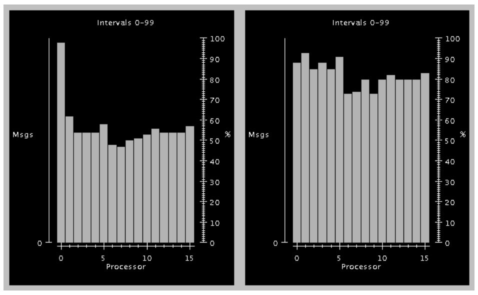

Fig. 5. Utilization of 16 processors before(Left) and after(Right) automatic load balancing in a non-uniform stencil calculation

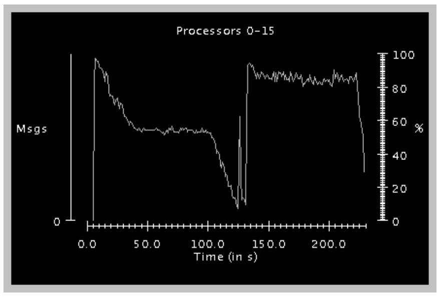

Fig. 6. Overall CPU utilization before and after automatic load balancing in a non-uniform stencil calculation

Although it is an artificial benchmark, it represents a common situation: very small fraction of overloaded processors potentially ruin the overall performance of all processors. The load balancer is employed to solve this problem, as shown in Figure 5 and 6. According to Figure 5, one of the 16 processors are overloaded while others are underloaded, with average utilization less than 60% before load balancing, while after load balancing, the variation of the workload is diminished and the overall utilization is about 20% higher. Correspondingly, the average time per iteration drops from 1.15ms to 0.85ms. Figure 6 demonstrates how the load balancer is activated and utilization increased from 55% to 85% approximately. Note that this load balancing is all automatically done by the system; there is no programmer interference needed at all.

5.3 Collective Communication Optimization

MPI standard defines the collective operations as blocking, which makes it impossible to overlap them with computation, because many or all processors are blocked waiting for the collective operation to return. In Section 4.2 we discussed the optimization of supporting non-blocking collective operations to allow overlapping. Now we illustrate how this feature can save the execution time in parallel applications.

In [25], a parallel algorithm for Quantum Molecular Dynamics is discussed. One complexity in the algorithm arises from 128 independent and concurrent sets of 3D FFTs. Although each of the FFT can be parallelized, overlapping between different sets of FFTs is difficult due to the all-to-all operation required for transposing data in each FFT. However, AMPI's non-blocking all-to-all operation allows the programmer to overlap the communication and computation from consecutive sets of FFT and save execution time.

To make a benchmark based on this application, we simplified the above problem. We do two independent sets of 2D FFT, each consisting of the one 1D FFT, transpose, and another 1D FFT. To pipeline the operations, we move the second 1D FFT of the first set after the transpose of the first set. In the blocking version, however, this pipelining is not gaining any performance, because the transpose, implemented as blocking all-to-all communication, stops any other computation from being done. In the non-blocking version, the second set is able to do real computation while the first set is waiting for its communication to complete.

Figure 7 demonstrates the effect of overlapping collective communications with computation. The y axis is different number of processors, for blocking version(labeled as MPI) and non-blocking version(labeled as AMPI) respectively, and the x axis is the execution time. Using distinct colors in the stacked bars, we denote the breakdown of the overhead for 1D FFT (computation), communication, and for non-blocking version, the waiting time for non-blocking operation, as discussed in Section 4.2.

It can be observed that the two versions have similar amounts of computation, but in terms of communication, the non-blocking version has advantage because part of its waiting time is reduced by overlapping it with computation. The AMPI bar is 10% - 20% shorter than the MPI bar, the amount of saving depending on the amount of possible overlap. This saving could be even larger if there is more computation for overlap.

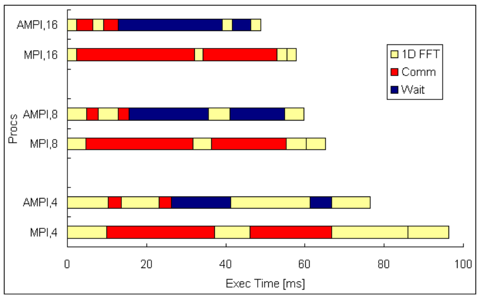

Fig. 7. Breakdown of execution time of 2D FFT benchmark on 4, 8, and 16 processors, with comparison between blocking(MPI) and non-blocking(AMPI) all-to-all operations.

# Procs	Native MPI	AMPI(1)	AMPI(K)
8	-	318.488	104.909
27	29.440	41.415	28.166
64	14.162	16.433	12.670
125	9.121	11.504	11.590
216	8.066	6.506*	8.365
512	5.519	6.486	5.645
1728	4.499	3.521*	-

Table 1. Execution time[ms] of 240^3 3D 7-point stencil calculation on Lemieux

5.4 Flexibility and Overhead

In this section we are going to show the flexibility virtualization provides, as well as the overhead virtualization incurs. Our benchmark is 240^3 3D 7-point stencil calculation.

First we run it with native MPI on Lemieux. Because the model of the program divides the job into K-cubed partitions, not surprisingly, the program runs only on a cube number of processors. On Adaptive MPI 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 visualized in Table 2. The performances on ative MPI and on Adaptive MPI appear to have very little difference. Note that on some "random" number of PEs, like 19 and 140, the native MPI program is not able to run, while AMPI handles the situation perfectly.

Now let's take a closer look at the speedup data of the same program running on native MPI, AMPI with 1 VP per processor and AMPI with multiple (K=4 - 10) VPs per processor. Table 1 displays the execution time of the same size problem running on increasing number of processors, with the best K values shown in AMPI(K) column.

Comparing the execution time of native MPI against AMPI, we find that although native MPI outperforms AMPI in many cases as expected, it does so by only a small amount. Thus, the flexibility and load balancing advantages of AMPI do not come at an undue price in basic performance[4]. In some cases, nevertheless, AMPI does a little better. For example AMPI(K) is faster than native MPI when number of processors is small. This is due to the caching effect; many VPs grouped on one processor will increase the locality of data as well as instructions. The advantage of this caching effect is shown in Table 1, where AMPI with virtualization outperforms AMPI(1) on smaller number of processors. When there are many processors involved, the cost of coordinating the VPs takes over and offset the caching effect. Two results (marked by "*" in Table 1) are anomalous, and we have not identified the underlying causes yet.

#PE	19	27	33	64	80	105	125	140	175	216	250	512
Native MPI	N/A	29.440	N/A	14.162	N/A	N/A	9.121	N/A	N/A	8.066	N/A	5.519
AMPI	42.410	30.528	24.646	15.635	12.621	10.935	10.776	10.616	9.388	8.626	7.549	5.464

Table 2. Execution time[ms] of AMPI v.s. Native MPI, of 240^3 3D 7-point stencil calculation on Lemieux

6 AMPI Experience: Rocket Simulation

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 from ten different engineering and science departments, as well as 18 professional staff. The main CSAR simulation code consists of four major components: a fluid

[4] A microbenchmark shows an average of 2μs for a context switch between the threads with which AMPI VPs are associated, on an 400MHz PIII Xeon processor.

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 often 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.

As the solid propellant burns away, each processor's portion of the problem domain changes, which will change the CPU and communication time required by that processor. The most important long-term benefit that the CSAR codes will derive from AMPI is the ability to adapt to this changing computation by migrating work between processors, taking advantage of the CHARM++ load balancing framework's demonstrated ability to optimize for load balance and communication efficiency. Because the CSAR components do not yet change the mesh structure during a run, and merely distort the existing mesh, the computation and communication patterns of the virtual MPI processors do not yet change. However, this mesh distortion breaks down after a relatively small amount of motion, so the ability to adjust the mesh to the changing problem domain is scheduled to be added soon.

Finally, the CSAR simulator's current main loop consists of one call to each of the simulation components in turn, in a one-at-a-time lockstep fashion. This means, for example, the fluid simulation must finish its timestep before the solids can begin its own. But because each component runs independently except at well-defined interface points, and AMPI allows multiple independent threads of execution, we will be able to improve performance by splitting the main loop into a set of cooperating threads. This would allow, for example, the fluid simulation thread to use the processor while the solid thread is blocked waiting for remote data or a solids synchronization. Separating each component should also improve our ability to optimize the communication balance across the machine, since currently the i'th fluids processor has no physical correspondence with the i'th solids processor.

In summary, AMPI has proven a useful tool for the CSAR simulation, from debugging to day-to-day operations to future plans.

7 Conclusions

We have presented AMPI, an adaptive implementation of MPI on top of CHARM++. 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 computation and communication, emulate large machines on small ones, and respond to a changing physical machine. Several applications are being developed using AMPI, including those in rocket simulation.

AMPI is an active research project; much future work is planned for AMPI. We expect to achieve full MPI-1.1 standards conformance soon, and MPI-2 thereafter. We are rapidly improving the performance of AMPI, and should soon be quite near that of non-migratable MPI. The CHARM++ performance analysis tools 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 future machines such as BlueGene/L.

References

1. G.C. Fox, R.D. Williams, and P.C. Messina. *Parallel Computing Works*. Morgan Kaufman, 1994.
2. 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.
3. 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.
4. L. V. Kale and Sanjeev Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, 1996.
5. 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.
6. Greg Burns, Raja Daoud, and J. Vaigl. Lam: An open cluster environment for mpi. In *Proceedings of Supercomputing Symposium 1994, Toronto, Canada*, 1994.
7. Georg Stellner. CoCheck: Checkpointing and Process Migration for MPI. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS '96)*, Honolulu, Hawaii, 1996.
8. Adnan Agbaria and Roy Friedman. StarFish: Fault-tolerant dynamic mpi programs on clusters of workstations. In *8th IEEE International Symposium on High Performance Distributed Computing*, 1999.
9. MPI-Lite, Parallel Computing Lab, University of California. http://may.cs.ucla.edu/projects/sesame/mpi_lite/mpi_lite.html.
10. 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.
11. Laxmikant V. Kalé. The virtualization model of parallel programming : Runtime optimizations and the state of art. In *LACSI 2002*, Albuquerque, October 2002.

12. O. Lawlor and L. V. Kalé. Supporting dynamic parallel object arrays. In *Proceedings of ACM 2001 Java Grande/ISCOPE Conference*, pages 21–29, Stanford, CA, Jun 2001.
13. L. V. Kale, Milind Bhandarkar, and Robert Brunner. Run-time 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.
14. Robert K. Brunner and Laxmikant V. Kalé. Adapting to load on workstation clusters. In *The Seventh Symposium on the Frontiers of Massively Parallel Computation*, pages 106–112. IEEE Computer Society Press, February 1999.
15. Laxmikant V. Kalé, Sameer Kumar, and Jayant DeSouza. An adaptive job scheduler for timeshared parallel machines. Technical Report 00-02, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, Sep 2000.
16. 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.
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. L. V. Kale, Sameer Kumar, and Krishnan Vardarajan. A framework for collective personalized communication, communicated to ipdps 2003. Technical Report 02-10, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, 2002.
21. L. V. Kale and Sameer Kumar. Scaling collective multicast on high performance clusters. Technical Report 03-04, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, 2003.
22. IBM Parallel Enviroment for AIX, MPI Subroutine Reference. http://publib.boulder.ibm.com/doc_link/en_US/a_doc_lib/sp34/pe/html/am107mst.html.
23. Georg Stellner. Cocheck: Checkpointing and process migration for mpi. In *Proceedings of the International Parallel and Distributed Processing Symposium*, pages 526–531. IEEE Computer Society Press, Los Alamitos, CA, 1996.
24. Lemieux, Pittsburgh Supercomputing Center. http://www.psc.edu/machines/tcs/lemieux.html.
25. Ramkumar Vadali, L. V. Kale, Glenn Martyna, and Mark Tuckerman. Scalable parallelization of ab initio molecular dynamics. Technical report, UIUC, Dept. of Computer Science, 2003.