Adaptive MPI

Orion Lawlor Milind Bhandarkar L. V. Kalé
Parallel Programming Laboratory
University of Illinois at Urbana-Champaign
olawlor@acm.org,{bhandark,l-kale1}@uiuc.edu

May 9, 2002

Abstract

"Adaptive MPI", or AMPI, implements virtual MPI processors, several of which may reside on a single physical processor. This virtualization allows MPI applications to use an automatic migration-based load balancer, automatically overlap computation and communication, and provides several other benefits. In this paper, we present the design of and recent work on AMPI, its low-level and application performance, and some of the advanced capabilities enabled by virtualization.

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 adaptive 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. Algorithmic considerations often restrict the number of processors to a power of 2, or a cube (or both).

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 allows the programmer to use the most natural decomposition for his problem, rather than being restricted by the physical machine. So, for example, 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.

- *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.

- *Load balancing:* If one of the physical processors becomes overloaded, the runtime system can migrate a few of its virtual processors to less heavily loaded physical processors. Our runtime system can make this kind of load balancing decision based on automatic instrumentation, as explained in Section 2.

- *Large machine emulation:* A small physical machine, such as a single serial processor, can simulate a large virtual machine—see Section 4.2 for an example. This emulation can be useful in debugging, performance tuning, and testing.

- *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 "chunking" 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 4.3 for details.

- *Automatic checkpointing:* AMPI's virtualization allows applications to be checkpointed without additional user programming, as described in Section 3.1.

- *Multiple independent modules:* MPI programs normally transfer control from one module to another strictly via manual subroutine calls. AMPI allows different modules to execute on different virtual processors, which allows adaptively interleaved execution, as described in Section 3.2.

We first describe how our virtual processors are implemented and migrated. Section 3 describes the design and implementation strategies for specific features, such as checkpointing. We then present performance data showing that these adaptive features are affordable in real programs. Finally, we will demonstrate some adaptive capabilities quantitatively in the context of a conjugate gradient solver for sparse linear systems.

1.1 Prior Work

The virtualization concept embodied by AMPI is very old, and Fox et al. [?] 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.

There are several excellent, complete, publicly available non-virtualized implementations of MPI, such as MPICH [?] and LAM [?]. Many researchers have described partially virtualized MPI implementations for checkpointing, often built on top of one of the free implementations of MPI. Several workers have described fully virtualized MPI implementations for fault-tolerance, such as FT-MPI [?], MPI/FT [?], and StarFish

[?]. AMPI differs from these efforts in that we virtualize to improve performance and allow load balancing rather than solely for checkpointing or for fault tolerance. Some of these works also impose unacceptable runtime overheads or require extensive changes to the user code, problems AMPI largely manages to avoid.

An older version of AMPI was described by Bhandarkar et al. [?]. Our automatic load balancing framework was described in detail by Kalé et al. [?].

2 Design and 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 communicating objects, which are mapped to processors by the CHARM++ runtime. CHARM++ supports migration of objects by providing efficient forwarding of messages, when necessary. Migration can be used by the built-in measurement-based load balancing [?], adapting to changing load on workstation clusters [?], and even shrinking/expanding jobs for timeshared machines [?]. Migration presents interesting problems for basic and collective communication which are nicely solved by CHARM++ [?].

AMPI's virtual processors are implemented as CHARM++ "user-level" threads—threads which are created and scheduled by ordinary code, not by the operating system kernel. The advantages of user-level threads are fast[1] context switching, control over scheduling, and control over stack allocation. CHARM++'s user-level threads are scheduled non-preemptively.

CHARM++ natively supports object migration; but thread migration required several interesting additions to the runtime system, as described in the following sections.

2.1 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—consider the variable i below:

```
int foo(void) {
  int i;
  bar(&i);
  return i;
}
```

During the call to `bar`, the stack-allocated variable i cannot be moved, since its address is stored by `bar`.[2] In a distributed memory parallel machine, if the stack is moved to a new machine, it will almost undoubtably be allocated at a different location, so `bar`'s pointer to i will become dangling 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

[1]On a 1.8 GHz AMD AthlonXP, 0.45 microseconds per suspend/schedule/resume.

[2]The address might be stored in a machine register, `bar`'s pushed parameters, `bar`'s stack frame, or all three!

has the ability to map arbitrary pages in and out of memory. So in practice we merely need to `mmap` the appropriate address range into memory on the new machine and use it for our stack. Because this uses the hardware's built-in virtual memory support, when migration is not occurring this approach does not affect performance.

Of course, we must ensure that each thread allocates its stack at a globally unique range of addresses. This is accomplished by simply dividing the total virtual address space into p regions; threads created on processor i then get their stacks allocated from region i. This system thus has globally-unique memory addresses like a software shared memory system (DSM), but here the data movement is proactive—when a thread migrates, it takes all its data with it. This "isomalloc" approach to thread migration comes from PM^2 [?].

2.2 Isomalloc Heaps

Another obvious problem with migrating an arbitrary program is dynamically allocated storage—for example, the array h in:

```
int main(int argc,char *argv[]) {
  MPI_Init(&argc,&argv);
  int *h=new int[23];
  for (...) {
    ...
    h[i]=...
  }
}
```

Clearly, if this thread is migrated to a new processor, h must come along as well. But unlike the thread stack, which the system allocated, h's location is known only to the user program. The previous version of AMPI required the user to code a "pack/unpack" routine to capture all allocated heap data. This routine was fairly easy to write, but rather difficult to maintain. Worse, this tiny amount of extra code prevented a straightforward switch from ordinary MPI to AMPI.

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. This means 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". Because nearly all languages can be linked together with C code, even the C++ `new` and FORTRAN90 `ALLOCATE` runtime calls eventually result in a call to malloc. However, some implementations of these language runtimes perform caching of free memory blocks, which must be disabled.

Unfortunately, the isomalloc heap approach is only of limited use on 32-bit systems. Since the virtual address range on these machines is limited to 4GB, and since this space is divided among all processors when using the isomalloc approach, we run out of allocatable space very quickly. For example, dividing the 4GB address space[3] among 100 processors means each processor can only allocate 40MB of memory; a significant limit. Thus on 32-bit machines, the pack/unpack method is generally required. Ma-

[3]In fact the program code and operating system use some space, so even less is available.

chines with 64-bit pointers, which are becoming increasingly common, have many terabytes of free virtual address space and hence can fully benefit from isomalloc heaps.

2.3 Global Variables

Although not specified by the MPI standard, many actual MPI programs assume that global variables can be used independently on each processor. However, in AMPI, all the threads on a processor share a single set of global variables; and when a thread migrates, it leaves its global variables behind. This means many MPI programs cannot run unmodified under AMPI.

A simple solution is to manually remove *all* the global variables from the code. For example, all the formal globals can be collected into a single struct named "Global", which is then passed into each function. This process, though mechanical, is cumbersome and can indeed be automated.

AMPIzer [**?**] is our source-to-source translator that removes global variables from arbitrary FORTRAN77 or FORTRAN90 code. AMPIzer is based on the Polaris compiler front end [**?**]. For simple[4] uses of the heap, Ampizer can also generate a pack/unpack routine if isomalloc heaps are not desired.

2.4 Limitations

During migration, we do not preserve a thread's open files and sockets, environment variables, or signals. Because of these difficulties, threads are only migrated when they call the special API routine MPI_Migrate; the non-migration-safe features can be used at any other time. Thread migration between different architectures on a heterogeneous parallel machine is also not supported.[5]

[4] "Simple" in this context means no pointers stored inside dynamically allocated blocks.

3 Other features

3.1 Checkpoint and Restart

As Stellner describes in his paper on his checkpointing framework [**?**], 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 separate 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.

Because AMPI 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.

[5] Without extensive compiler support or a common virtual machine, heterogeneous thread migration appears impossible.

3.2 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.

3.3 Coexistence with Charm++

As described above, AMPI modules can be used in-process with other AMPI modules. AMPI modules can also coexist with other CHARM++ modules. For example, a program using another threaded CHARM++ framework, such as the CHARM++ Finite Element Method (FEM) Framework [**?**], or the CHARM++ collision detection system [**?**], can still use AMPI.

In this "bound" mode, a single thread of user code can make calls to the FEM and AMPI frameworks; when the thread migrates, the support data required by both frameworks automatically migrates as well. This keeps the users' code simple, since they do not have to synchronize two separate threads of control. Of course, it is also possible to run an FEM framework module and an AMPI framework module in their own separate sets of threads.

Operation	Serial	100baseT Cluster			Myrinet Cluster		Origin2000		IBM SP3	
	AMPI	AMPI	MPICH	LAM	AMPI	MPICH	AMPI	MPI	AMPI	MPI
Send/Recv	2.1	154.0	116.3	89.9	20.2	10.5	77.4	13.4	190.9	114.7
Repeated Send	0.9	39.8	1676.3	4.6	9.1	401.4	27.7	5.1	67.2	38.4
Barrier	4.1	304.8	135.6	164.0	40.5	16.9	158.4	2.1	376.2	120.2
Bcast	3.8	54.2	1599.9	15.2	10.8	4.7	64.2	4.1	110.7	45.1
Allreduce	6.1	357.2	241.5	192.2	52.2	22.1	158.7	23.2	412.7	127.0
Bandwidth	125.3	8.9	10.3	7.2	54.0	43.2	42.3	96.9	57.8	71.5

Table 1: Time for various MPI operations under different MPI implementations. All entries are times in microseconds, except bandwidth which is megabytes per second. All tests performed on two processors.

4 Performance

We have described the results from experiments involving real scientific applications running on AMPI in another work [?].

4.1 Low-level Performance

The times for various low-level MPI operations on various machines and MPI implementations are shown in Table 3.3. The serial machine is an AMD Athlon 1800XP running two AMPI virtual processors. The 100baseT cluster is a set of 4-way 500Mhz Pentium III SMP nodes running Linux, connected using switched fast Ethernet; AMPI is on UDP and MPICH is on p4. The Myrinet cluster is a set of 2-way 1GHz Pentium III SMP nodes running Linux, connected using a Myricom interconnect; AMPI and MPICH both ran on GM directly. The SGI Origin2000 is a single 50-processor 195MHz R10000 node; AMPI ran on the native SGI MPI implementation. IBM SP3 is a set of 8-way 375MHZ Power3 nodes; AMPI again ran on the native MPI implementation.

"Send/Recv" performs a simple ping-pong operation—one processor sends while the other receives, then receives while the other sends. "Repeated Send" is nearly the same, except one processor always sends while the other always receives, and reports the time as measured by the sending processor. This send overhead is important during broadcast-style communication exchanges. MPICH, both on Ethernet and GM, had extremely poor performance for this test. "Barrier", "Bcast", and "Allreduce" are simply the equivalent MPI operation, in this case on just two processors, and with the time measured from the root. "Bandwidth" is the end-to-end large-message transmission rate, as measured by the time to exchange a one-megabyte buffer.

AMPI was occasionally several times slower than the non-adaptive MPI implementations. Part of this is simply the fundamental cost of AMPI's virtualization; but part is simply our implementation and we should be able to soon show substantial improvements. In particular, the non-blocking operations AMPI requires, such as MPI_Isend, have very poor performance on many MPI implementations; we have begun experimenting with direct implementations for many parallel machines. Despite these low-level results, application performance under AMPI is often quite good.

4.2 Conjugate Gradient Solver

This application is a partial differential equation solver which uses a sparse, matrix-free form of the conjugate gradient method to solve the Poisson problem on a regular 2D grid. It is an iterative method, and typically performs several thousand steps during one solution.

Each (virtual) processor is responsible for computing the solution on a rectangular region of the mesh. Since the solution residual for a grid point depends on the solutions for its nearest four neighbors, each processor maintains a one-element-thick ghost region. In each step, messages are exchanged to fill these ghost regions, and there are two short global reductions. Like many scientific codes, this application is normally memory bandwidth bound.

Figure 1 shows the time per step of the solver on a single physical processor, while varying the number of virtual processors between 1 and 4096. Because AMPI's virtual processors are implemented as user-level threads, there is very little overhead in managing a large number of threads. On our Pentium IV system, with a relatively small cache but very fast RDRAM memory, simulating 100 virtual processors led to only a slight (10%) slowdown. However, on the Athlon and Pentium III Xenon, improved cache usage while simulating 100 virtual processors actually resulted in slightly *better* performance than using the single physical processor normally.

Figure 2 shows the time per step of the solver on an actual parallel machine, ASCI Red. This is with a larger mesh–6000x6000 elements, for a 36 million row matrix. As shown, the runtime cost for AMPI's virtualization is small.

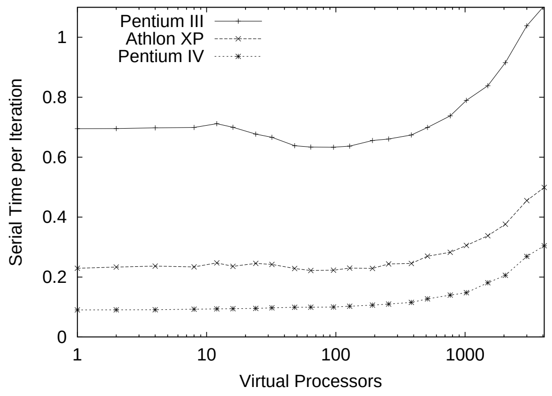

Figure 1: Time per step for the million-row conjugate gradient solver on one physical processor and up to 4096 virtual processors. The horizontal axis is logarithmic; the vertical axis is linear.

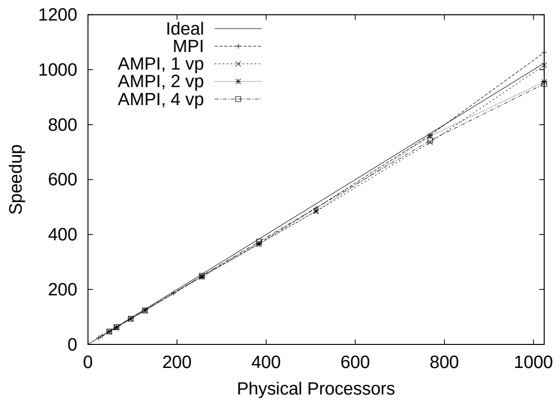

Figure 2: Speedup for the 36 million-row conjugate gradient solver on up to 1024 physical processors for MPI and AMPI. One timestep takes approximately 50ms on 1024 processors.

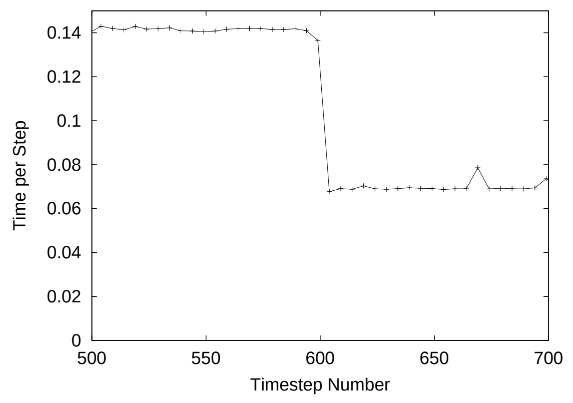

Figure 3: Time per step for the million-row conjugate gradient solver on a workstation cluster. Initially, the application runs on 16 machines. 16 new machines are made available at step 600, which immediately improves the throughput.

4.3 Shrink/Expand

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 3 shows the conjugate gradient solver described above responding to the availability of several new processors. The time per step drops dramatically as virtual processors are migrated onto the new physical processors.

5 Conclusions

We have presented AMPI, an adaptive implementation of MPI on top of CHARM++. AMPI implements migratable virtual MPI processors; and in particular allows the use of several virtual processors per 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.

We have much future work planned for AMPI. We hope to achieve full MPI1.1 standards conformance soon, and MPI2.0 shortly 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 need to be updated to provide more direct support for AMPI programs. We hope to extend our suite of automatic load balancing strategies to provide machine-topology specific strategies. Finally, we hope to apply our communication optimization libraries to programs running under AMPI.

References

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

[2] 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.

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

[4] Graham Fagg, Antonin Bukovsky, and Jack Dongarra. HARNESS and fault tolerant MPI. *Parallel Computing*, 27(11):1479–1496, October 2001.

[5] R. Batchu, J. Neelamegam, Z. Cui, M. Beddhu, A. Skjellum, Y. Dandass, and M. Apte. MPI/FT: Architecture and taxonomies for fault-tolerant, message-passing middleware for performance-portable parallel computing. In *Proceedings of the 1st IEEE International Symposium of Cluster Computing and the Grid*, May 2001.

[6] 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.

[7] Milind Bhandarkar, L. V. Kale, Eric de Sturler, and Jay Hoeflinger. Object-Based Adaptive Load Balancing for MPI Programs. In *Proceedings of the International Conference on Computational Science, San Francisco, CA, LNCS 2074*, pages 108–117, May 2001.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

[15] Georg Stellner. Cocheck: Checkpointing and process migration for mpi. In *Proceedings of the International Parallel and Dis-*

tributed Processing Symposium, pages 526–531. IEEE Computer Society Press, Los Alamitos, CA, 1996.

[16] 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.

[17] Orion Sky Lawlor and L.V. Kalé. A voxel-based parallel collision detection algorithm. In *Proceedings of International Conference in Supercomputing (to appear)*, 2002.