A Threads-Only MPI Implementation for the Development of Parallel Programs

Erik D. Demaine
Dept. of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1
eddemaine@uwaterloo.ca

Abstract

In this paper, we present a threads-only implementation of MPI, called TOMPI, that allows efficient development of parallel programs on a workstation. The communication and context-switching overhead is reduced significantly compared to existing MPI implementations, by the use of threads and shared memory in place of UNIX processes and often sockets. Results demonstrate the scalability of TOMPI in comparison to two MPICH devices. We provide a C/C++ preprocessor to modify the semantics of global variables to appear as if each thread has its own address space. This makes TOMPI a true MPI implementation, that is, MPI programs can be run in TOMPI without modification.

Keywords: threads, MPI, parallel programming.

1 Introduction

The development phase of parallel programming, including testing, debugging, and performance tuning, is the most difficult and time-consuming. Supercomputing resources are usually unavailable for development, since they are expensive and often not required. With the advent of the message-passing standard MPI (Message Passing Interface) [1], it is possible to develop parallel programs on a workstation, and later run them on a supercomputer with minimal changes to the code. Unfortunately, no MPI implementations are designed specifically for parallel-program development on a single workstation.

Implementations supporting networks of workstations use operating-system (e.g., UNIX) processes and communicate between processes via sockets. In principle, these implementations could be used on a single workstation by simulating each processor with an operating-system process. Unfortunately, this is an inefficient way to simulate a highly parallel computer. First, UNIX processes involve a large memory overhead, especially in the case of older operating systems where each stores its own copy of the program code. They also induce a significant context-switching overhead, which has a large negative effect on highly parallel executions. Second, UNIX-domain sockets are inefficient on a single computer, compared to shared-memory communication. This slows down fine-grain applications in particular, which are common since we are likely developing code to be run on a supercomputer with fast communication. Finally, most implementations use polling communication, which does not scale well to several processes running on a single processor.

For this reason, we developed an MPI implementation designed specifically for a single workstation. It uses a single UNIX process and multiple threads, one for each MPI process, thereby minimizing memory and context-switching overheads. The single address space allows efficient communication, which we exploit. The result is an efficient "testbed" for parallel-program development on a workstation. It can even take advantage of modern SMP workstations (that is, those with multiple processors and a shared memory), for improved performance.

Of course, threads do not have the same semantics as processes. In particular, global data is shared by the threads, presenting a potential conflict. To remedy this problem, we provide a preprocessor for C and C++ to automatically convert global variables into thread-specific globals, thereby providing the same semantics as processes. The preprocessor is designed to require only slight modifications for nearly any extension to C.

The rest of this paper is organized as follows. Section 2 surveys related work in the literature. In Section 3, we discuss our threads-only implementation of MPI. We describe the preprocessor in Section 4. Finally, we provide experimental results and conclude in Sections 5 and 6, respectively.

2 Related Work

WinMPI [2] is an implementation of MPI for Microsoft Windows 3.1. Its goals are similar to ours; it was designed as a tool for easy and inexpensive parallel-program development on a single processor. WinMPI does not support threads for reduced overhead, requires extra memory copies (even for synchronous communication), only supports up to 16 processes, and requires several changes to existing MPI programs. These limitations, caused by the Windows environment, restrict WinMPI's usefulness.

Shared-memory MPI implementations, typically designed for multiprocessor systems, exploit the communication facilities available in a single workstation. However, memory, context-switching, and polling overheads are still major concerns. We compare our system to MPICH's shared-memory implementation [3] in Section 5.

LPVM [4] and TPVM [5] are two thread systems for the PVM (Parallel Virtual Machine) [6] message-passing system. LPVM modifies the PVM interface to be thread-safe, whereas TPVM builds on top of the current PVM. LPVM is more related to our work because it only supports a single computer. It differs in that it does not address the global-variable problem, likely because the idea of testing and debugging parallel programs on a workstation was not considered. Ignoring this, both LPVM and TPVM require the user to change her/his program for using a threads-only system, since their interfaces differ from PVM. Hence, these systems are useful for writing message-passing programs that exploit threads, but do not solve the problem of parallel-program development on a workstation.

MPI is much more attractive for a threads-only system since the interface is thread-safe. The only portable implementation of MPI supporting threads, Chant [7], is in fact built on top of a process-based MPI implementation, an approach similar to TPVM. Users of Chant must specify both process and thread ids in message-passing calls, and so Chant's programming interface is not MPI (although it is similar to MPI). This is unfortunate for testing and debugging MPI programs on a workstation, since users must modify their programs before and after using a threads-only system based on Chant. Again, Chant does not address the global-variable problem.

An internal NEC implementation of MPI, done in Tokyo, for the NEC SX-4 parallel-vector machine is the only project we know of that addresses the global-variable problem. The compiler (by default) converts all global variables into thread-specific data [8]. Since communication is through shared memory, the implementation can then use threads as MPI processes, thereby achieving greater efficiency as we have noted.

3 Threads-Only MPI Implementation

In this section, we present our threads-only MPI implementation for developing parallel programs on a workstation, called *TOMPI*. TOMPI is currently only a partial implementation, supporting most point-to-point and some collective communication. We plan to support all MPI features soon.

TOMPI supports POSIX threads (pthreads) [9] and Solaris threads [10]. It should be simple to port to other thread packages, since we use an intermediate layer for all thread calls. This layer is implemented as a set of macros so that it induces no additional runtime overhead. The only required features are

- spawning a thread defined by a function with a specified argument,
- thread-specific data,
- binary semaphores (mutexes), and
- condition variables.

By using threads, we minimize memory and context-switching overhead, thereby allowing one to use hundreds of MPI processes on a single workstation. What remains in the implementation is to optimize communication so that fine-grain applications can execute efficiently. This is the topic of the rest of this section.

Each thread has a message queue, a mutex, and a condition variable. We say that a thread is in the *critical section* of a queue if it has locked the corresponding mutex. A thread can *wait* on a condition variable, causing it to block until another thread executes a *notify* on the condition variable. These operations are typically done within the queue's critical section to avoid races [10]. We say that a thread p notifies another thread q if p executes a notify on q's condition variable.

The basic protocol is as follows. The sender "posts" the fact that it has a message by placing the message (i.e., its data, size, datatype, source, tag, and context) in the destination's queue, and notifies the destination. This is done in the critical section of the destination's queue. If the send operation is synchronous (that is, it does not complete until a matching receive is executed), then the sender also stores a unique pointer. For non-blocking sends (which return immediately, not waiting for the operation to complete), this represents the `MPI_Request` maintained by the user; otherwise, it is a pointer to a structure maintained locally by the blocking routine. If the receiver notices such a pointer, it places a reply on the source's queue that contains this pointer for matching purposes. The receiver does so in the critical section of the source's queue, in addition to notifying the source.

The standard send operation (`MPI_Send`) is synchronous to avoid more than one copy of the data. This is possible without inducing much overhead since the threads share an address space, and hence the receiver can access the send buffer. This synchronous behavior is also permissible by the MPI standard, for the very reason of optimization. On the other hand, if the destination does not have a matching receive posted, or if the message is short, it may make more sense to buffer the message to allow the sending thread to continue. We will consider such choices in the future.

We decided that data copying and dynamic allocation should be avoided as much as possible because they are slow operations. For this reason, we developed a queue structure that has the following properties:

- minimal dynamic allocation, that is, additional storage is only allocated when the queue is full, and this allocation is done in blocks;
- constant-time insertions and constant-time deletions in arbitrary positions[1]; and
- incremental searches.

A conventional linked list would support these operations in the desired time bounds, but cannot be allocated in a block manner. One cannot simply use a (dynamically growing) array to store the queue elements in order, since holes may be created by arbitrary deletes, resulting in expensive shifts. Instead, we must embed a linked list into an array, that is, use a parallel "next" array that gives the index of the next element in the list. Unfortunately, this causes insert operations to become expensive, since they must either search the entire array to find a blank location or waste parts of the allocated array. To alleviate this problem, we maintain a list of the unused elements in the array (commonly referred to as a *free list*); this can be done in constant time per operation.

A thread often wishes to wait for a matching message or set of messages to arrive in its queue. We do not want threads adding messages to the queue to use this information. Rather, the blocked thread should deal with this, and so threads adding to a particular queue simply notify the owner thread. However, we do not want the blocked thread to search through the entire queue every time it changes, unless the definition of a "matching message" changes.

For this reason, we maintain the "oldest not-yet-examined message" for each queue. The insert procedure sets this to the inserted node, unless it is already set. Before any MPI routine attempts to search the queue for the first time, it resets the "oldest not-yet-examined message" to the head of the queue (note that it must do so within the queue's critical section). The search routine only looks at the oldest not-yet-examined message, and any messages after that in the queue. Since messages are always added to the tail, the search routine will look at every new message.

With this queue structure, we achieve highly efficient (indeed, likely near-optimal) point-to-point communication. We are investigating additional structures to improve collective communication, which should be possible in a shared address space.

4 Preprocessor

We clearly want to be able to bring existing MPI applications into the TOMPI development environment without changing the code. The difficultly is that threads do not have the same semantics as processes, rather they share a common address space. This means that global variables and variables static to functions are not duplicated for each MPI process as the

[1]While only an add-to-tail operation is needed for an MPI implementation, elements may be deleted (received) in an arbitrary order by specifying a source, tag, and/or context.

user would expect[2]. Such semantics are provided by a concept known as *thread-specific data* (TSD) [10].

Let us concentrate on TSD as provided by POSIX threads, since Solaris threads are nearly identical in this respect. Thread-specific data is represented by variables of type `pthread_key_t`, which we will refer to as *keys*. `pthread_key_create` creates a new key, and should be called by only one of the threads that wish to use the TSD. Keys are generally stored in global variables to make them accessible to all threads. `pthread_getspecific` returns the value corresponding to the specified key, which has the size of a `void *` pointer and is initialized to `NULL`. `pthread_setspecific` sets the data corresponding to the given key to the specified value.

To avoid manually changing the code, we provide a preprocessor that converts C/C++ global-variable declarations and references into the appropriate TSD calls. This means that the user can develop code in the MPI disjoint-address-space model, and yet test on a workstation using a physically shared address space to minimize overhead. The preprocessor is automatically called at the appropriate point in the provided `mpicc` script. Hence, TOMPI is an MPI implementation, except that all code must be compiled with `mpicc`.

The preprocessor is run after the C preprocessor (`cpp`) and before the compiler (e.g., `cc`). It currently supports C and C++, and is designed to support other extensions to C with only minor changes. This is possible since the preprocessor does not have to understand much of the language. Other than the language's token structure, it needs to keep track of the current nesting level (that is, whether we are in the global section or not) and watch for variable/type declarations. For each variable declaration, it only needs to know the variable's type and name, and whether it is `static`, `extern`, and/or `const`. It can skip over variable initializers, variable dimensions, and statements that do not begin with a declaration specifier[3]. Statements that begin with a declaration specifier are guaranteed to be a variable/type declaration, a function prototype, or a function definition; it is fairly easy to distinguish between them.

Note that the preprocessor must watch for type declarations (`typedef`s) to distinguish between the variable declaration `type (i)` and the function call `func (i)`. It does not, however, need to know the actual type of a variable; rather it can use the "type alias" given in the variable declaration, which will later be expanded by the compiler.

An example of the result of the preprocessor is given in Figure 1. Including the original declarations is the only way, in general, to initialize dynamically allocated variables without significantly changing the code. For each global variable or static variable of a function, we add a declaration of the TSD key. We replace each reference to such a variable with a dereference of a call to `get_global`. This routine is outlined as follows:

`get_global` (address of key, data, n)

1. If key is `NULL`, do the following in a critical section:

   If key is still `NULL`, create it.

2. If the data corresponding to key is `NULL`, replace it with a dynamically allocated block of n bytes, initialized to data.
3. Return the address of the data corresponding to key.

[2]Note that I/O, on the other hand, has the same semantics as UNIX processes, and hence is not a concern.

[3]From the ANSI C specification (ANSI/ISO 9899-1990 standard), a declaration specifier is either a type specifier or a storage-class specifier (`typedef`, `extern`, `static`, `auto`, or `register`).

```
const int max = 256;
int rank;
int main (int argc, char *argv[]) {
   int size;
   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD, &rank);
   MPI_Comm_size (MPI_COMM_WORLD, &size);
   ...
}
int f (void) {
   static int i = 0;
   return i++;
}
```

(a)

```
const int max = 256;
int rank; Key rank_key = NULL;
int main (int argc, char *argv[]) {
   int size;
   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD,
      &(*((int *) get_global (&rank_key,
         &rank, sizeof (rank)))));
   MPI_Comm_size (MPI_COMM_WORLD, &size);
   ...
}
int f (void) {
   static int i = 0; static Key i_key = NULL;
   return (*((int *) get_global (&i_key,
      &i, sizeof (i))))++;
}
```

(b)

Figure 1: *Example of the preprocessor. (a) Input. (b) Output.* `Key` *represents the key data-type, depending on the underlying thread system. To simplify the example, we omit the include files, use a simple naming scheme, and exclude a level of indirection that is necessary because keys cannot be initialized to* `NULL`.

We initially test if key is `NULL` to avoid executing a critical section for every call to `get_global`.

Note that the line breaks are preserved so that the line numbers reported by the compiler from errors and warnings are correct.

Let us now give an idea of the speed of the preprocessor. Other than code for the lexer and hash tables, the code for the preprocessor is 16,734 bytes consisting of 488 lines. Running this through the C preprocessor (expanding macros and include statements) yields 42,132 bytes and 4,055 lines in 0.074 seconds of CPU time. Our preprocessor converts this result in 0.197 seconds. Hence, the preprocessor runs in time similar to the C preprocessor if we account for the increase in file size.

5 Results

In this section, we demonstrate the efficiency of TOMPI. We first use two uniprocessor systems for comparison to MPICH and evaluating scalability. We then use a two-processor SMP to evaluate the effectiveness of TOMPI on a multiprocessor. The system specifications are as follows:

- 66-MHz IBM RS/6000 model 25T running AIX 4.2.0 with 64 megabytes of memory
- 166-MHz Sun Ultra 1/170E running SunOS 5.5.1 with 128 megabytes of memory
- Sun SPARC 10 SMP running SunOS 5.4 with two 75-MHz processors and 128 megabytes of memory

MPICH [3] is a free portable implementation of MPI based on an *abstract device specification*. For a single processor, two devices are applicable, `ch_p4` and `ch_shmem`. They use sockets and shared memory, respectively, for communication.

Message Size	MPICH		TOMPI
	p4	shmem	
0	2,769	9,950	161
1	2,188	9,850	163
10	2,181	9,925	165
100	2,237	9,900	166
1,000	2,726	9,799	189
10,000	5,806	9,749	579
100,000	120,244	10,940	3,245
1,000,000	1,311,691	132,713	29,771

Table 1: *Comparison between TOMPI and MPICH* `ch_p4` *(socket) and* `ch_shmem` *(shared memory) devices for point-to-point messages. Times are in microseconds on the RS/6000.*

Message Size	MPICH		TOMPI
	p4	shmem	
0	4,182	100,011	24
1	4,191	101,312	24
10	3,974	100,012	24
100	4,340	100,013	25
1,000	1,214	100,012	30
10,000	758	99,843	79
100,000	400,059	100,012	572
1,000,000	4,100,617	300,044	5,638

Table 2: *Comparison between TOMPI and MPICH* `ch_p4` *(socket) and* `ch_shmem` *(shared memory) devices for point-to-point messages. Times are in microseconds on the Ultra.*

First, let us examine point-to-point communication. We implemented a ping-pong benchmark to evaluate the cost of passing a single message via `MPI_Send/Recv`. The benchmark uses a combination of means and medians [11] to obtain high accuracy and reduce experimental error. A message is sent back and forth several times, and the time to do this is divided by the number of messages sent (*mean*), improving the accuracy of the wall clock. This is done several times, and the *median* of the results is taken to discard outliers. We have also used the MPICH `mpptest` benchmark [3] and obtained similar results.

Results for various message sizes on the RS/6000 are given in Table 1. One surprising result is that the `ch_p4` device has a lower latency than the `ch_shmem` device. We believe this to be because the `ch_shmem` device was not designed for multiple processes on each processor. Indeed, both devices use polling, which is inefficient in this benchmark since processes wait most of the time, explaining why TOMPI ranks so high in this benchmark. `ch_p4` polls on a socket, whereas `ch_shmem` polls on a variable, which is a finer-grain operation, hence resulting in a greater performance hit from polling.

Similar results for the Ultra are given in Table 2. Because the Ultra processor is much faster than the RS/6000, the processes block for more clock cycles, resulting in more polling in MPICH. This explains why MPICH performs so poorly on this machine. It also explains why `ch_shmem` has an even worse latency than `ch_p4`, compared to the RS/6000, since `ch_shmem` uses a finer-grain polling mechanism.

Processes	Communication		MPICH		TOMPI
	Count	Volume	p4	shmem	
1	0	0	86.4	87.0	86.3
2	999	8	94.6	93.0	93.6
4	2,994	24	106.7	105.5	103.8
8	6,972	56	133.6	120.1	113.3
16	14,880	119	257.3	155.4	124.4
32	30,504	244	—	456.7	147.0
64	60,984	488	—	755.1	184.9
128	118,872	952	—	—	253.8
256	222,360	1,781	—	—	381.7
512	380,184	3,044	—	—	752.0

Table 3: *Comparison of Cholesky factorization for various numbers of processes. Times are in seconds on the RS/6000 for a* 1000×1000 *double-precision matrix. Volume is the total data communicated in millions of bytes.*

It is likely more important to examine the performance of a real application. We ran a parallel dense Cholesky-factorization algorithm on various numbers of processes but with the same problem size (a 1000×1000 matrix). This would be considered a small problem on a supercomputer, but is reasonable for testing the algorithm on a workstation (which has limited resources).

Let us first consider the RS/6000 results (Table 3). Because the Cholesky algorithm (fan in) significantly overlaps computation with communication, processes do little blocking. This explains why the MPICH and TOMPI results are much closer; polling is not much of a factor. In this sense, TOMPI has somewhat of an unfair disadvantage in this comparison. In addition, TOMPI uses synchronous communication (whereas MPICH uses buffering), which we believe slows down the algorithm somewhat. Finally, for runs with many threads, TOMPI's queues occasionally overflow and resize. Despite this unfairness, TOMPI scales much better than the two MPICH devices, which was our goal. Using threads we were able to run more MPI processes, but even where an MPICH device could compete, TOMPI did much better. For example, TOMPI can run 128 processes for the time it takes `ch_p4` to run 16, or it can run 512 in the time it takes `ch_shmem` to run 64.

The results on the Ultra (Table 4) are favorable for TOMPI even with few processes. This is likely because the Ultra has a much faster processor; on the serial Cholesky-factorization benchmark, it is slightly over three times faster than the RS/6000. Hence, the factorization algorithm blocks more often, resulting in a major loss of performance for MPICH from polling. Also note that `ch_shmem` performs consistently worse than `ch_p4` because of `ch_shmem`'s finer-grain polling.

Let us now examine the scalability of TOMPI. Ideally, the execution time would be the same for any number of processes. This is not the case because of the communication and context-switching overhead. The former is high in this communication-intensive algorithm (Table 3, columns 2–3); for example, with 512 processes, approximately four bytes are communicated for every floating-point operation in the computation. (There are 665,668,000 floating-point operations, regardless of the number of processes.)

We can expect to have lower context-switching overhead with user-level threads, which can speed up context switches by an order of magnitude [12]. AIX 4.2 (the operating system on

Processes	Communication		MPICH		TOMPI
	Count	Volume	p4	shmem	
1	0	0	28.3	28.2	28.2
2	999	8	30.0	99.8	28.5
4	2,994	24	44.1	162.3	28.9
8	6,972	56	59.7	254.1	29.8
16	14,880	119	117.7	445.5	31.1
32	30,504	244	149.3	958.1	34.4
64	60,984	488	171.4	1255.6	40.2
128	118,872	952	—	—	51.2
256	222,360	1,781	—	—	70.9
512	380,184	3,044	—	—	104.7
1024	499,500	4,000	—	—	140.4

Table 4: *Comparison of Cholesky factorization for various numbers of processes. Times are in seconds on the Ultra. Again, the matrix is* 1000×1000*, and volume is measured in millions of bytes.*

the RS/6000) only supports system-level threads. On the other hand, Solaris allows the user to mix both user-level and system-level threads by setting the level of "concurrency." The results in Table 4 represent the default of user-level threads, explaining why the Ultra scales so much better than the RS/6000.

Let us now turn to the SMP results for Cholesky factorization (Table 5). This illustrates how TOMPI can be used to test and debug parallel programs on modern (multiprocessor) workstations. We get significant speedup, although it is limited by the parallelism of the application.

We could not use 32 or more threads on two processors, because of a bug in Solaris's thread-specific data. Unpredictably, the thread-specific data pointers are modified to point to random locations. Since the run is sufficiently long, threads eventually lose track of `MPI_COMM_WORLD` and their local queue. We are currently writing our own version of thread-specific data for Solaris threads. This will be at least as efficient as the built-in mechanism, since the set of thread ids is an interval of integers and we can hence use a simple global array. Initial experimentation indicates that this will fix the problem.

An important issue is the number of threads we can run on a workstation. We believe that the AIX operating system imposes restrictions on the number of threads, limiting us to tests involving at most 512. Solaris threads, on the other hand, are primarily user-level, and hence should pose no artificial restriction. By default, however, each Solaris thread allocates a megabyte of virtual memory for its stack. We have modified the stack size of the Cholesky-factorization program to as low as 10 kilobytes per thread, allowing hundreds of thousands of threads to be run. The timing for this number of threads is not useful (and hence not shown) for a 1000×1000 matrix, where increasing past 1000 threads does not increase parallelism.

In comparison, MPICH is limited in the number of possible processes because of the overhead and limits of UNIX processes. On the RS/6000, the `ch_shmem` device can use somewhat more processes, likely because it does not use sockets, which impose additional limits.

We should note that the memory requirements of TOMPI are low. For example, the RS/6000 runs were satisfied with 32 megabytes of memory, which we consider to be the smallest amount of memory commonly found in modern workstations. The only exception is the

Processes	Communication		# Processors	
	Count	Volume	1	2
1	0	0	57.9	57.9
2	999	8	58.6	35.6
4	2,994	24	60.7	37.3
8	6,972	56	64.8	39.2
16	14,880	119	69.1	43.6
32	30,504	244	80.3	—
64	60,984	488	103.4	—
128	118,872	952	145.7	—
256	222,360	1,781	226.8	—
512	380,184	3,044	362.5	—
1024	499,500	4,000	498.8	—

Table 5: *1- and 2-processor runs of Cholesky factorization for various numbers of threads. Times are in seconds on the SMP. Again, the matrix is* 1000×1000*, and volume is measured in millions of bytes.*

512-thread run, which swapped slightly, yielding 1,122 seconds; using a 64-megabyte machine gives the 752 seconds in Table 3.

6 Conclusion

In this paper, we have described a threads-only implementation of MPI (TOMPI) for parallel-program development on a workstation. While the implementation is partial, it supports the commonly used MPI features. The communication is designed to be efficient by using a lightweight dynamic queue structure and synchronous communication by default (avoiding an extra memory copy). Since TOMPI is built on top of POSIX or Solaris threads, it avoids memory and context-switching overhead involved in UNIX processes, allowing many MPI processes on a system with little memory. Results show that the system also scales well in terms of performance. The provided C/C++ preprocessor modifies global variables to have the same semantics as if each thread had its own address space.

In the future, we plan to extend TOMPI to efficiently support the entire MPI standard. We would then like to incorporate TOMPI into an existing MPI implementation such as MPICH, to efficiently support more processes than processors in a distributed system. We also plan to implement Fortran support, in particular for the preprocessor. Finally, we will consider supporting more user-level threads packages to reduce context-switching overhead on systems other than Solaris.

Acknowledgment

We thank the referees for their comments on this work. David Taylor provided valuable comments on this paper, and access to the RS/6000's. Alan George and Bruno Preiss provided access to the SMP and the Ultra, respectively. We would like to thank Jonathan Eckstein, Ewing Lusk, and Peter Sanders for initial discussions on this work.

This work was supported by the Natural Sciences and Engineering Research Council (NSERC).

References

[1] Message Passing Interface Forum, "MPI: A message-passing interface standard," *The International Journal of Supercomputer Applications and High Performance Computing*, vol. 8, no. 3/4, 1994.

[2] Joerg Meyer, "Message-passing interface for Microsoft Windows 3.1," M.S. thesis, Department of Computer Science, University of Nebraska, Omaha, Nebraska, December 1994.

[3] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum, "A high-performance, portable implementation of the MPI Message Passing Interface standard," *Parallel Computing*, To appear.

[4] Honbo Zhou and Al Geist, "LPVM: A step towards multithread PVM," November 1995, World Wide Web. `http://www.epm.ornl.gov/~zhou/lpvm.ps`.

[5] Adam Ferrari and V. S. Sunderam, "TPVM: A threads-based interface and subsystem for PVM," Tech. Rep. CSTR-940802, Dept. of Mathematics and Computer Science, Emory University, Atlanta, GA 30322, August 1994.

[6] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam, *PVM: Parallel Virtual Machine — A User's Guide and Tutorial for Networked Parallel Computing*, The MIT Press, Cambridge, MA, 1994.

[7] Matthew Haines, David Cronk, and Piyush Mehrotra, "On the design of Chant: A talking threads package," in *Proceedings of the 1994 Supercomputing Conference*, Washington, DC, November 1994, pp. 350–359, IEEE Computer Society Press.

[8] Ewing Lusk, "Personal communication," March 1997.

[9] "Information Technology — Portable Operating System Interface (POSIXR) — Part 1: System Application: Program Interface (API)," Tech. Rep. 9945-1, ISO/IEC, 1996, IEEE/ANSI Standard 1003.1, 1996 Edition. Includes threads interface (1003.1c-1995).

[10] SunSoft, Inc., *Solaris Multithreaded Programming Guide*, Sun Microsystems Press/Prentice Hall, 1995.

[11] Erik D. Demaine, "Evaluating the performance of parallel programs in a pseudo-parallel MPI environment," in *Proceedings of the 10th International Symposium on High Performance Computing Systems (HPCS'96)*, Ottawa, Ontario, Canada, June 1996, Carleton University Press and Oxford University Press.

[12] Matthew Haines, Piyush Mehrotra, and David Cronk, "Chant: Lightweight threads in a distributed memory environment," Submitted to *Journal of Parallel and Distributed Programming*, 1996, World Wide Web. `http://www.cs.uwyo.edu/~haines/research/chant/jpdc.ps`.