Compile/Run-time Support for Threaded MPI Execution on Multiprogrammed Shared Memory Machines

Hong Tang, Kai Shen, and Tao Yang
Department of Computer Science
University of California
Santa Barbara, CA 93106
{htang, kshen, tyang}@cs.ucsb.edu
http://www.cs.ucsb.edu/research/tmpi

Abstract

MPI is a message-passing standard widely used for developing high-performance parallel applications. Because of the restriction in the MPI computation model, conventional implementations on shared memory machines map each MPI node to an OS process, which suffers serious performance degradation in the presence of multiprogramming, especially when a space/time sharing policy is employed in OS job scheduling. In this paper, we study compile-time and run-time support for MPI by using threads and demonstrate our optimization techniques for executing a large class of MPI programs written in C. The compile-time transformation adopts thread-specific data structures to eliminate the use of global and static variables in C code. The run-time support includes an efficient point-to-point communication protocol based on a novel lock-free queue management scheme. Our experiments on an SGI Origin 2000 show that our MPI prototype called TMPI using the proposed techniques is competitive with SGI's native MPI implementation in a dedicated environment, and it has significant performance advantages with up to a 23-fold improvement in a multiprogrammed environment.

1 Introduction

MPI is a message-passing standard [3] widely used for developing high-performance parallel applications. There are a number of reasons that people use MPI on shared memory machines (SMMs). First, new applications may be required to integrate with existing MPI programs. Second, code using MPI is portable to any parallel machine without platform restriction. This is especially important for future computing infrastructures such as information power grids [1, 13], where resource availability, including platforms, dynamically changes for running submitted jobs. Third, even though shared memory programming is easier for developing a prototype of parallel applications, it is hard to fully exploit the underlying architecture without careful consideration of data placement and synchronization protocols. On the other hand, performance tuning for SPMD-based MPI code on large SMMs is normally easier since partitioned code and data exhibit good data locality.

MPICH [17] is a portable implementation of MPI that delivers good performance across a wide range of architectures. For SMMs, either a vendor has its own implementation or uses MPICH. Efficient execution of MPI code on an SMM is not easy since the MPI programming model does not take advantages of the underlying architecture. MPI uses the process concept and global variables in an MPI program are non-sharable among MPI nodes. As a result, a conventional MPI implementation has to use heavy-weight processes for code execution and synchronization. There are two reasons that process-based MPI implementations suffer severe performance degradation on multiprogrammed SMMs. First, it has been widely acknowledged in the OS community that space/time sharing which dynamically partitions processors among applications is preferable [10, 20, 32, 34]. The modern operating systems such as Solaris 2.6 and IRIX 6.5 have adopted such a policy in parallel job scheduling. Therefore, the number of processors allocated to an MPI job can be smaller than requested. In some cases, the number of assigned processors may change dynamically. Thus, multiprogramming imposes great disadvantages for MPI jobs because process context switch and synchronization are expensive. Secondly, without sharing space among processes, message passing between two MPI nodes must go through the system buffer and buffer copying degrades the communication efficiency of MPI code [1].

To appear in the Proceedings of 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99).

[1] An earlier version of SGI MPI enforced that the address space of each MPI process is shared with every other. However, SGI eventually

In this paper, we propose compile-time and run-time techniques that allow a large class of MPI C code to be executed as threads on SMMs. The compile-time code preprocessing eliminates global and static variables using thread-specific data structures, which results in safe execution of MPI code. The run-time techniques proposed in this paper are focused on efficient lock-free point-to-point communication.

We assume that readers are familiar with the MPI standard and will not present its definitions. Section 2 describes our current assumptions and related work. Section 3 discusses compile-time preprocessing that produces thread-safe MPI code. Section 4 discusses the run-time support for multi-threaded execution. Section 5 presents our lock-free management for point-to-point communication. Section 6 presents the experimental results on the SGI Origin 2000. Section 7 concludes the paper.

2 Assumptions and Related Work

Our first goal is to convert an MPI program (called *source program* later on) to be "thread-safe" so that the new program (called *target program* later on) will yield the same result as the source program when it is executed by multiple threads. To avoid confusion, the term "MPI node" is used to refer to an MPI running unit and the term "MPI process" is only used when we want to emphasize that an MPI node is actually a process. In the current work, we have made several assumptions. 1) The total memory used by all the nodes can fit in the address space of a process. 2) The total number of files opened by all the nodes can fit in one process's open file table. 3) The source program does not involve low-level system calls which are not thread-safe such as signals. 4) Each MPI node does not spawn new threads. Most programs written in MPI, however, should meet our assumptions and we found no exception in any of the MPI test programs we collected.

We assume that basic synchronization primitives such as *read-modify-write* and *compare-and-swap* [18] are available and we use them for lock-free synchronization management. Actually, all modern microprocessors either directly support these primitives or provide LL/SC [18] for software implementation.

The importance of integrating multi-threading and communication on distributed memory systems has been identified in previous work such as the Nexus project [14]. Earlier attempts to run message-passing code on shared-memory machines include the LPVM [35] and TPVM [12] projects. Both projects do not address how a PVM program can be executed in a multi-threaded environment without changing the programming interface. Most of previous MPI research is focused on distributed memory machines or workstation clusters, e.g. [9]. The MPI-SIM project [6, 7] has used multi-threading to simulate MPI execution on distributed memory machines as we will discuss in Section 3.1. Thread safety is addressed in [3, 26, 29]. However, their concern is how multiple threads can be invoked in each MPI node, but not how to execute each MPI node as a thread. These studies are useful for us to relax our assumptions in the future.

Previous work has also illustrated the importance of lock-free management for reducing synchronization contention and unnecessary delay due to locks [4, 5, 18, 21, 22]. Lock-free synchronization has also been used in the process-based SGI implementation [17]. Theoretically speaking, some concepts of SGI's design could be applied to our case after considerations for thread-based execution. However, as a proprietary implementation, SGI's MPI design is not documented and its source code is not available to public. The SGI design uses undocumented low-level functions and hardware support specific to the SGI architecture, which may not be general or suitable for other machines. Also, their design uses busy-waiting when a process is waiting for events [27], which is not desirable for multiprogrammed environments [19, 24]. Lock-free studies in [4, 5, 18, 21, 22] either restrict their queue model to be FIFO or FILO, which are not sufficient for MPI point-to-point communication, or are too general with unnecessary overhead for MPI. A lock-free study for MPICH is conducted in a version for the NEC shared-memory vector machines and Cray T3D [16, 8, 2], using single-slotted buffers for the ADI-layer communication. Their studies are still process-based and use the layered communication management which is a portable solution with overhead higher than our scheme. In terms of lock-free management, our scheme is more sophisticated with greater concurrency and better efficiency since our queues can be of arbitrary lengths and allow concurrent access by a sender and a receiver.

Our study is leveraged by previous research in OS job scheduling on multiprogrammed SMMs [10, 20, 32, 34, 33]. These studies show that multiprogramming makes efficient use of system resources and space/time sharing is the most viable solution, outperforming other alternatives such as time sharing and co-scheduling [24], for achieving high throughputs. The current version of OS in both SGI and SUN multiprocessors support space/time sharing policies.

3 Compile-time Support

The basic transformation needed to make the execution thread-safe for MPI C code is elimination of global and static variables. In an MPI program, each node can keep a copy of its own *permanent variables* – variables

gave up this design due to insufficient address space and software incompatibility [27].

allocated statically in a heap, such as global variables and local static variables. If such a program is executed by multiple threads without any transformation, then all threads will access the same copy of permanent variables. To preserve the semantics of a source MPI program, it is necessary to make a "private" copy of each permanent variable for each thread.

3.1 Possible Solutions

Below we discuss three possible solutions and examples for each of them are illustrated in Figure 1. The `main()` routine of a source program listed in Column 1 is converted into a new routine called `usrMain()` and another routine called `thr_main()` is created, which does certain initialization work and then calls `userMain()`. This routine `thr_main()` is used by the run-time system to spawn threads based on the number of MPI nodes requested by the user. We discuss and compare these solutions in details as follows.

The first solution illustrated in the second column of Figure 1 is called *parameter passing*. The basic idea is that all permanent variables in the source program are dynamically allocated and initialized by each thread before it executes the user's main program. Pointers to those variables are passed to functions that need to access them. There is no overhead other than parameter passing, which can usually be done quite efficiently. The problem is that such an approach is not general and the transformation could fail for some cases.

The second solution, which is used in [7], is called *array replication*. The preprocessor re-declares each permanent variable with an additional dimension, whose size is equal to the total number of threads. There are several problems with this approach. First, the number of threads cannot be determined in advance at compile time. MPI-SIM [7] uses an upper limit to allocate space and thus the space cost may be excessive. Second, even though the space of global variables could be allocated dynamically, the initialization of static and global variables must be conducted before thread spawning. As a result, function- or block-specific static variables and related type definitions must be moved out from their original lexical scopes, which violates the C programming semantics. It is possible to provide a complicated renaming scheme to eliminate type and variable name conflicts, but the target program would be very difficult to read. Finally, false sharing may occur in this scheme when the size of a permanent variable is small or not aligned to cache line size [25, 11].

Because of the above considerations, we have used the third approach based on *thread-specific data (TDS)*, a mechanism available in POSIX threads [23]. Briefly speaking, TSD allows each thread to associate a private value with a common key which is a small integer. Given the same key value, TSD can store/retrieve a thread's own copy of data. In our scheme, each permanent variable is replaced with a permanent key of the same lexical scope. Each thread dynamically allocates space for all permanent variables, initializes those variables for only once, and associates the reference of those variables with their corresponding keys. For each function that refers a permanent variable, this reference is changed to a call that retrieves the value of this variable using the corresponding key. Such a transformation is general and its correctness not difficult to prove. There will be no false sharing problem even for keys, because keys are never altered after initialization. Notice that certain thread systems such as SGI's SPROC thread library do not provide the TSD capability; however, it is still relatively easy to implement such a mechanism. In fact, we wrote TSD functions for the SGI's SPROC library. In the example of Figure 1, two TSD functions are used. Function `setval(int key, void *val)` associates value "`val`" to a key marked as "`key`" and function `void *getval(int key)` gets the value associated with "`key`". In this example, a key is allocated statically. In our implementation, keys are dynamically allocated.

3.2 TSD-based Transformation

We have implemented a preprocessor for ANSI C (1989) to perform the TSD-based transformation. The actual transformation uses dynamic key allocation and is more complex than the example in Figure 1 since interaction among multiple files needs to be considered and type definitions and permanent variable definitions could appear in any place including the body of functions and loops. We briefly discuss three cases in handling transformation.

- **Case 1: Global permanent variables.** If a variable is defined/declared as a global variable (not within any function), then it will be replaced by a corresponding key declaration. The key is seen by all threads and is used to access the memory associated with the key. This key is initialized before threads are spawned. In the `thr_main()` routine, a proper amount of space for this variable is allocated, initialized and then attached to this thread-specific key. Notice that `thr_main()` is the entry function spawned by the run-time system in creating multiple MPI threads; thus the space allocated for this variable is thread-specific.

- **Case 2: Static variables local to a control block.** A control block in C is a sequence of code delimited by "{" and "}". Static variables must be initialized (if specified) at the first time when the corresponding control block is invoked and the lexical scope of those static variables should be within

Source Program	Parameter passing	Array Replication	TSD
			`typedef int KEY;`
`static int i=1;`		`static int Vi[Nproc];`	`static KEY key_i=1;`

	`int thr_main()`
	`{`
	`...`
	`int *pi=malloc(sizeof(int));`
	`*pi=1;`
	
	`...`
	`usrMain(pi);`
	`}`
		`int thr_main(int tid)`
		`{`
		`...`
		
		`Vi[tid]=1;`
		
		`...`
		`usrMain(tid);`
		`}`
			`int thr_main()`
			`{`
			`...`
			`int *pi=malloc(sizeof(int));`
			`*pi=1;`
			`setval(key_i, pi);`
			`...`
			`usrMain();`
			`}`

`int main()`
`{`
 	
`i++;`
`return i;`
`}`
	`int usrMain(int *pi)`
	`{`
	
	`(*pi)++;`
	`return (*pi);`
	`}`
		`int usrMain(int myid)`
		`{`
		
		`Vi[myid]++;`
		`return Vi[myid];`
		`}`
			`int usrMain()`
			`{`
			`int *pi=getval(key_i);`
			
			`(*pi)++;`
			`return (*pi);`
			`}`

Figure 1: An example of code transformation. Column 1 is the original code. Columns 2 to 4 are target code generated by three preprocessing techniques, respectively.

```
if (key_V==0) {
    int new_key=key_create();
    compare_and_swap(&key_V, 0, new_key);
}
if (getval(key_V)==NULL) {
    T tmp=I;
    void *m=malloc(sizeof(tmp));
    memcpy(m, &tmp, sizeof(tmp));
    setval(key_V, m);
}
```

Figure 2: Target code generated for a static variable definition "`static T V = I;`".

this block. The procedure of key initialization and space allocation is similar to Case 1; however, the key has to be initialized by the first thread that executes the control block. The corresponding space has to be allocated and initialized by each thread when they reach the control block for the first time. Multiple threads may access the same control block during key creation and space initialization, so an atomic operation `compare_and_swap` is needed. More specifically, consider a statement for defining a static variable, `static T V = I;` where T is a type, V is the variable name, and I is an initialization phrase. This statement is replaced with "`static int key_V=0;`" and Figure 2 lists pseudo-code inserted at the beginning of a control block where this static variable is effective. Note that in the code, function `key_create()` generates a new key and the initial value associated with a new key is always `NULL`.

- **Case 3: Locally-declared permanent variables.** For a global variable declared locally within a control block using `extern`, the mapping is rather easy. The corresponding key is declared as `extern` in the same location.

For all three cases, the reference to a permanent variable in source MPI code is transformed in the same way. First, a pointer of proper type is declared and dynamically initialized to the reference of the permanent variable at the beginning of the control block where the variable is in effect. Then the reference to this variable in an expression is replaced with the dereference expression of that pointer, as illustrated in Figure 1, Column 4. The overhead of such indirect permanent variable access is insignificant in practice. For the experiments described in Section 6, the overhead of such indirection is no more than 0.1% of total execution time.

4 Run-time Support for Threaded Execution

The intrinsic difference between the thread model and the process model has a big impact on the design of run-time support. An obvious advantage of multi-threaded execution is the low context switch cost. Besides, inter-thread communication can be made faster by directly accessing threads' buffers between a sender and a receiver. Memory sharing among processes is usually restricted to a small address space, which is not flexible or cost-effective to satisfy MPI communication semantics. Advanced OS features may be used to force sharing of a large address space among processes; however, such an implementation becomes problematic, especially because it may not be portable even after OS or architecture upgrading [27]. As a result, process-based implementation requires that inter-process communication go through an intermediate system buffer as illustrated in Figure 3(a). Thus a thread-based run-time system can potentially reduce the number of some memory copy operations.

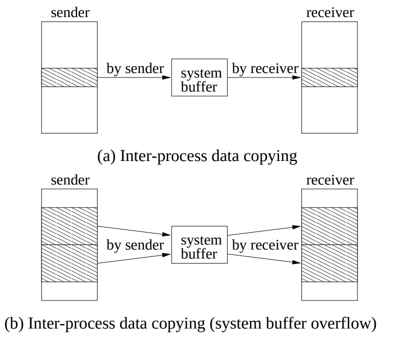

Figure 3: Illustration of inter-process message passing.

Notice that in our implementation, if message send is posted earlier than the receive operation, we choose not to let the sender block and wait for the receiver, in order to yield more concurrency. This choice affects when memory copying can be saved. We list three typical situations in which copy saving can take effect. 1) **Message send is posted later than message receive.** In this case, a thread-based system can directly copy data from the sender's user buffer to the receiver's user buffer. 2) **Buffered send operations.** MPI allows a program to specify a piece of user memory as the message buffer. In buffered send operation (`MPI_Bsend()`), if send is posted earlier than receive, the sender's message will be temporarily copied to the user-allocated buffer area before it is finally copied to the destination's buffer. For process-based execution, since the user-allocated message buffer is not accessible to other processes, an intermediate copy from the user-allocated buffer to the shared system buffer is still necessary. 3) **System buffer overflow.** If the message size exceeds the size of free space in system buffer, then the send operation must block and wait for the corresponding receive operation. In thread-based execution, a receiver can directly copy data from a sender's buffer. But in the process-based environment, the source buffer has to be copied in fragments to fit in the system buffer and then to the destination buffer. Figure 3(b) illustrates that copying needs to be done twice because the size of a message is twice as large as the buffer size.

The thread model also allows us the flexibility in design of a lock-free communication protocol to further expedite message passing. A key design goal is to minimize the use of atomic compare-and-swap or read-modify-write instructions in achieving lock-free synchronization. This is because those operations are much more expensive than plain memory operations, especially on RISC machines in which memory bus is stalled during an atomic operation. For example, on the Origin 2000 our measurement shows that plain memory access is 20 times faster than compare-and-swap and 17 times faster than read-modify-write. Our broadcasting queue management is based on previous lock-free FIFO queue studies [18, 22].

Finally, in our design and implementation, we adopt a spin-block strategy [19, 24] when a thread needs to wait for certain events.

In next section, we will discuss our point-to-point communication protocol which is specifically designed for threaded MPI execution.

5 Lock-free Management for Point-to-point Communication

Previous lock-free techniques [5, 18, 21, 22] are normally designed for FIFO or FILO queues, which are too restrictive to be applied for MPI point-to-point communication. MPI provides a very rich set of functions for message passing. An MPI node can select messages to receive by specifying a tag. For messages of the same tag, they must be received in a FIFO order. A receive operation can also specify a wildcard tag `MPI_ANY_TAG` or source node `MPI_ANY_SOURCE` in message matching. All send and receive primitives have both blocked and non-blocked versions. For a send operation, there are four send modes: standard, buffered, synchronized and ready. A detailed specification of these primitives can be found in [3, 30]. Such a specification calls for a more generic queue model. On the other hand, as will be shown later, by keeping the lock free queue model specific to MPI, a simple, efficient but correct implementation is still possible.

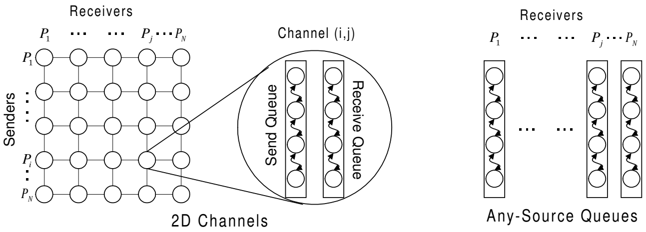

Figure 4: The communication architecture.

Let N be the number of MPI nodes. Our point-to-point communication layer consists of $N \times N$ *channels.* Each channel is designated for one sender-receiver pair and the channel from node P_i to P_j is different from the channel from P_j to P_i. Each channel contains a send queue and a receive queue. There are also additional N queues for handling receive requests with `MPI_ANY_SOURCE` as source nodes because those requests do not belong to any channel. We call these queues *Any-Source queues (ASqueue).* The entire communication architecture is depicted in Figure 4.

We define a send request issued by node s to be *matchable* with a receive request issued by node r if: 1) the destination node in the send request is r; **and** 2) the source node in the receive request is s or `MPI_ANY_SOURCE`; **and** 3) the tag in the send request matches the tag in the receive request or the tag in the receive request is `MPI_ANY_TAG`. In the simplest case of a send/receive operation, if the sender comes first, it will post the request handle[2] in the send queue, and later the receiver will match the request. If a receive request is posted first, the corresponding receive handle is inserted in a proper receive queue.

Our design is quite different from the layered design in MPICH. For the shared memory implementation of MPICH [17, 16], $N \times N$ single-slotted buffers are used for message passing in a lower layer. In a high layer, each process has three queues: one for send, one for receive, and one for unexpected messages. Thus messages from a sender with different destinations are placed in one send queue, similarly receive handles for obtaining messages from different sources are posted in the same receive queue. This design is portable for both SMMs and distributed memory machines. However, it may suffer high multiplexing cost when there are many queued messages with different destinations or sources.

The rest of this section is organized as follows. Section 5.1 presents the underlying lock-free queue model. Section 5.2 gives the protocol itself. Section 5.3 discusses the correctness of this protocol.

5.1 A Lock-free Queue Model

As we mentioned above, our point-to-point communication design contains $2N^2+N$ queues. Each queue is represented by a doubly-linked list. There are three types of operations performed on each queue: 1) put a handle into the end of a queue; 2) remove a handle from a queue (the position can be in any place.); 3) search (probe) a handle for matching a message. Previous lock-free research [18, 21, 22] usually assumes multiple-writers and multiple-readers for a queue, which complicates lock-free management. We have simplified the access model in our case to one-writer and multiple-readers, which gives us flexibility in queue management for better efficiency.

In our design, each queue has a *master* (or *owner*) and the structure of a queue can only be modified by its master. Thus a master performs the first two types of operations mentioned above. A thread other than the owner, when visiting a queue, is called a *slave* of this queue. A slave can only perform the third type of the operations (probe). In a channel from P_i to P_j, the send queue is owned by P_i and the receive queue is owned by P_j. Each ASqueue is owned by the MPI node which buffers its receive requests with the *any-source* wildcard.

Read/write contention can still occur when a master is trying to remove a handle while a slave is traversing the queue. Removing an interior handle by a master needs careful design because some slaves may still hold a reference and can result in invalid memory references. Herlihy [18] proposed a solution to such a problem by using accurate reference counting for each handle. Namely, each handle in a queue keeps the number of slaves that hold references to this handle. A handle will not be unlinked from the queue if its reference count is not zero. Then when a slave scans through a queue, it needs to decrease or increase the reference count of a handle using an atomic operation. Such an atomic operation requires at least one two-word compare-and-swap and two atomic additions [22], which is apparently too expensive. Another solution is to use a two-pass algorithm [22] which marks a handle as dead in the first pass and then removes it in the second pass. This approach is still not efficient because of multiple passes. We introduce the *conservative reference counting (CRC)* method that uses the total number of slaves which are traversing the queue to approximate the number of live references to each handle. Using such a conservative approximation, we only need to maintain one global reference counter and perform one atomic operation when a slave starts or finishes a probe operation. This conservative approximation works well with small overhead if the contention is not very intensive, which is actually true for most computation-intensive MPI applications.

Another optimization strategy called *semi-removal* is used in our scheme during handle deletion. Its goal is to minimize the chance of visiting a deleted handle by future traversers and thus reduce searching cost. If a handle to be removed is still referenced by some traverser, this handle has to be "garbage-collected" at a later time, which means other traversers may still visit this handle. To eliminate such false visits, we introduce three states for a handle: *alive* when it is linked in the queue, *dead* when it is not, and *semi-alive* when a handle is referenced by some traverser but will not be visited for future traversers. While the CRC of a queue is not zero, a handle to be removed is marked as semi-alive by only updating links from its neighboring handles. In this way, this handle is bypassed in the doubly-link list and is not visible to the future traversers. Note that this handle still keeps its link fields to its neighboring handles in the queue. All semi-alive items will eventually be declared as dead once the master finds that the CRC drops to zero. This method is called "semi-removal" in contrast to "safe-removal" in which the removal of a handle is deferred until removing is completely safe.

Figure 5 illustrates steps of our CRC method with semi-removal (Column 2) and those of the accurate reference counting method with safe-removal (Column 3). In this example, initially the queue contains four handles a, b, c, and d, and the master wants to remove b and c while at the same time a slave comes to probe the queue. Note that the reference counting in column 3 is

[2] A handle is a small data structure carrying the description of the send/receive request such as message tag and size.

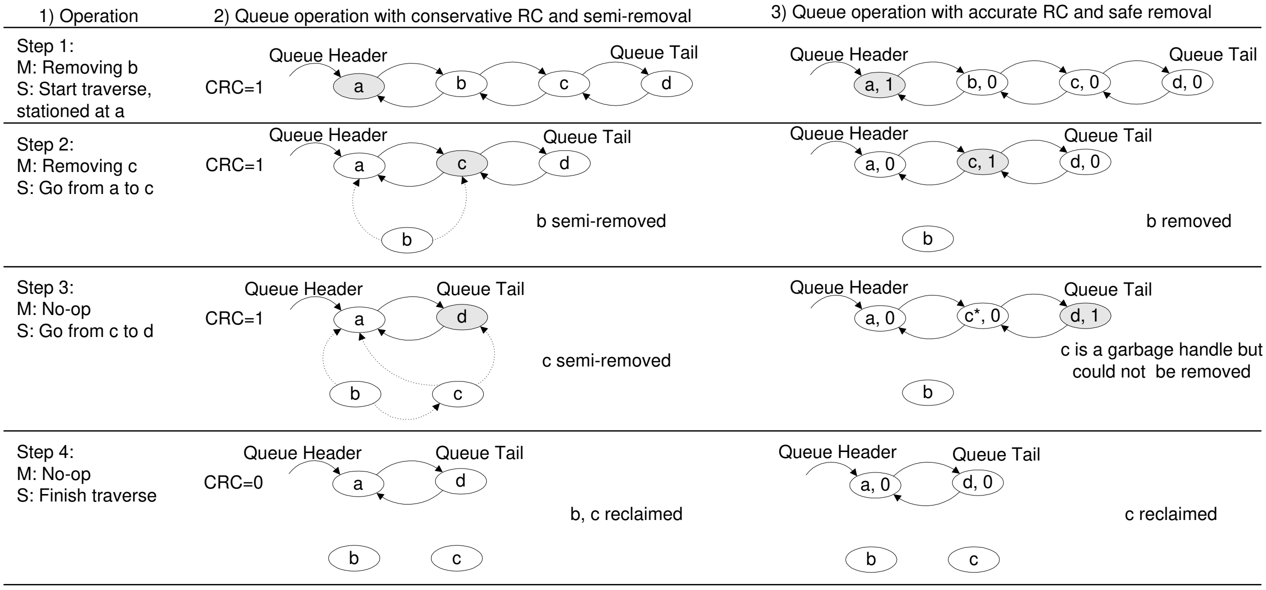

Figure 5: An example of conservative reference counting with semi-removal (column 2) compared to accurate reference counting with safe-removal (column 3). Column 1 lists actions taken by the master (marked as "M") and the salve (marked as "S"). Handles in shade are station points of the slave at each step. For accurate reference counting, the reference count is also shown within each handle.

marked within each handle, next to the handle name. For this figure, we can see that the average queue length (over all steps) in Column 2 is smaller than Column 3, which demonstrates the advantages of our method.

We have examined the effectiveness of our method by using several micro-benchmarks which involve intensive queue operations. Our method outperforms the accurate reference counting with safe removal by 10–20% in terms of average queue access times.

5.2 A Lock-free Point-to-point Communication Protocol

Our point-to-point communication protocol is best described as "enqueue-and-probe". The execution flow of a send or receive operation is described in Figure 6. For each operation with request $R1$, it enqueues $R1$ into an appropriate queue. Then it probes the corresponding queues for a matchable request. If it finds a matchable request $R2$, it marks $R2$ as `MATCHED` and then proceeds with the message passing. Notice that a flag is set by atomic subroutine compare_and_swap() to ensure that only one request operation can succeed in matching the same handle. For systems that do not support sequential consistency, a memory barrier is needed between enqueuing and probing to make sure that enqueuing completes execution before probing. Otherwise, out-of-order memory access and weak memory consistency in a modern multiprocessor system can cause a problem and the basic properties of our protocol studied in Section 5.3 may not be valid.

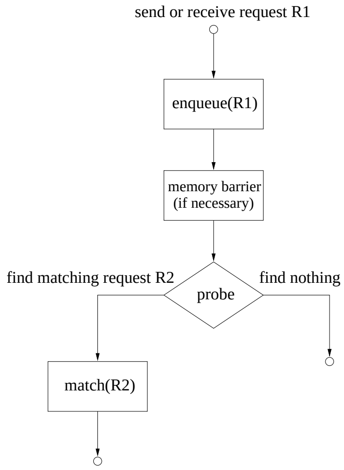

Figure 6: Execution flow of a send or receive operation.

Both send and receive operations have the same execution flow depicted in Figure 6 and their enqueue and probe procedures are described as follows.

- **Enqueue in receive operation:** If a receive request has a specific source node, the receiver adds the receive handle to the end of the receive queue. If the receive request uses the *any-source* wildcard, the receiver adds this handle to the ASqueue it owns. Notice that an enqueued handle is attached with a timestamp which is used to ensure the FIFO receive order.
- **Probe in receive operation:** If the receive re-

quest specifies a source node, the receiver probes the send queue in the corresponding channel to find the first matchable handle in that queue. If the receive request uses the *any-source* wildcard, the receiver probes all N send queues destined to this receiver in a random order (to ensure fairness). Notice that probing succeeds when the first matchable handle is found because no order is defined in MPI for send requests issued from different senders.

- **Enqueue in send operation:** The sender adds a send handle to the end of the send queue in the corresponding channel.
- **Probe in send operation:** The sender probes the receive queue in the corresponding channel and the ASqueue owned by the receiver to find the first matchable receive handle. If it succeeds in only one of those two queues, it returns the request handle it finds. If it finds matchable requests in both queues, it will use their timestamps to select the earlier request.

Since a flag is used to ensure that concurrent probings to the same handle cannot succeed simultaneously, it is impossible that several sender-probe operations match the same receive handle in a queue. It is however possible that when probing of a send operation finds a matchable receive handle in a queue, the probing of this receive request has found another send handle. To avoid this mismatch, the probing of a send operation must check the probing result of this matchable receive request and it may give up this receive handle if there is a conflict. Similarly, a conflict can arise when a receiver-probe operation finds a send handle while the probing of this send handle finds another receive handle. Thus the probing of a receive operation must wait until this matchable send request completes its probing and check the consistency. We call the above strategy *mismatch detection*. Finally, there is another case which needs special handling. If both the sender and the receiver find each other matchable at the same time, we only allow the receiver to proceed with message passing and make the sender yield as if it did not find the matchable receive request.

5.3 Correctness Studies

Our point-to-point message passing primitives such as blocking or non-blocking communication are built on the top of the above protocol. In [31], we have proven that our protocol satisfies the following three basic properties. One can use these properties to ensure the correctness of higher level communication primitives.

- **No double matching.** One send (receive) request can only successfully match one receive (send) request.
- **No message loss.** There exists no case such that two matchable send-receive requests are pending in their queues forever.
- **No message reordering.** There exists no case such that the execution order of send requests issued in one MPI node is different from the execution order of receive operations that are issued in another MPI node and match these messages.

6 Experimental Studies

The purpose of the experiments is to study if the thread-based execution can gain great performance advantages in non-dedicated environments and be competitive with the process-based MPI execution in dedicated environments. By "dedicated", we mean that the load of a machine is light and an MPI job can run on a requested number of processors without preemption. Being competitive in dedicated situations is important since a machine may swing dynamically between non-dedicated and dedicated states. Another purpose of our experiments is to examine the effectiveness of address-sharing through multi-threading for reducing memory copy and the lock-free communication management. All the experiments are conducted on an SGI Origin 2000 at UCSB with 32 195MHz MIPS R10000 processors and 2GB memory.

We have implemented a prototype called TMPI on SGI machines to demonstrate the effectiveness of our techniques. The architecture of the run-time system is shown in Figure 7. It contains three layers. The lowest layer provides support for several common facilities such as buffer and synchronization management, the middle layer is the implementation of various basic communication primitives and the top layer translates the MPI interface to the internal format.

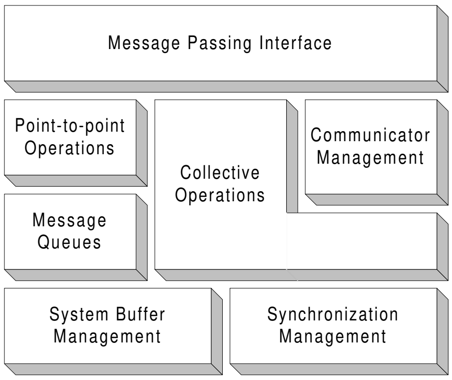

Figure 7: Run-time system architecture of TMPI.

We use the IRIX SPROC library because the performance of IRIX Pthreads is not competitive with SPROC.

The current prototype includes 27 MPI functions (MPI 1.1 Standard) for point-to-point and collective communications, which are listed in the appendix of this paper. We have focused on optimization and performance tuning for the point-to-point communication. Currently the broadcast and reduction functions are implemented using lock-free central data structures, and the barrier function is implemented directly using a lower-level IRIX barrier function. We have not fully optimized those collective functions. This should not affect the results we obtained through the experiments. We compare the performance of our prototype with the SGI's native implementation and the MPICH. Note that both SGI MPI and MPICH have implemented all MPI 1.1 functions; however those additional functions are independent and integrating them into TMPI should not effect our experimental results.

6.1 A Performance Comparison in Dedicated Environments

The characteristics of the four test benchmarks we have used are listed in Table 1. Two of them are kernel benchmarks written in C for dense matrix multiplication using Canon's method and a linear equation solver using Gaussian Elimination. Two of them (Sweep3D and Heat) are from the ASCI application benchmark collection at Lawrence Livermore and Los Alamos National Labs. HEAT is written in Fortran and we use utility f2c to produce a C version for our test. Sweep3D also uses Fortran. However, f2c cannot convert it because it uses an automatic array feature. We have manually modified its communication layer to call C MPI functions and eliminated one global variable used in its Fortran code. Thus, our code transformation is applied only to the C portion of this code.

Figure 8 depicts the overall performance of TMPI, SGI and MPICH in a dedicated environment measured by the wall clock time. We run the experiments multiple times and report the average, when every MPI node has exclusive access to a physical processor without interfered by other users. We do not have experimental results for 32 nodes because the Origin 2000 machine at UCSB has always been busy. For MM, GE and HEAT, we list megaflop numbers achieved since this information is reported by the programs. For Sweep3D, we list the parallel time speedup compared to single-node performance.

From the result shown in Figure 8, we can see that TMPI is competitive with SGI MPI. The reason is that a process-based implementation does not suffer process context switching overhead if each MPI node has exclusive access to its physical processor. For the MM benchmark, TMPI outperforms SGI by around 100%. We use the SGI SpeedShop tool to study the execution time breakdown of MM and the results are listed in

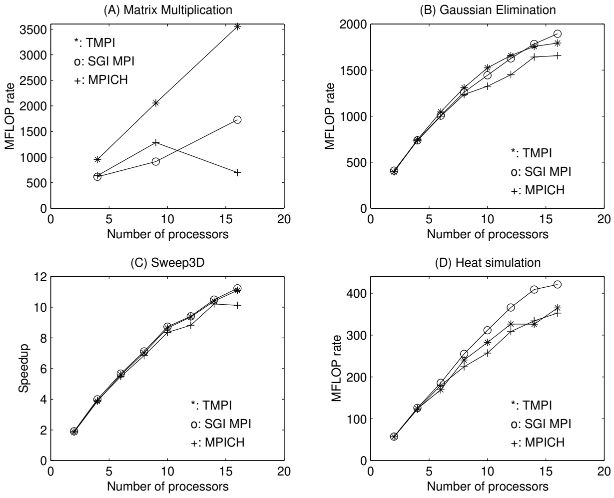

Figure 8: Overall performance in dedicated environments.

Table 2. We can see that TMPI spends half as much memory copy time as SGI MPI because most of the communication operations in MM are buffered send and fewer copying is needed in TMPI as explained in Section 4. Memory copying alone still cannot explain the large performance difference and so we have further isolated the synchronization cost, which is the time spent waiting for matching messages. We observe a large difference in synchronization cost between TMPI and MPICH. Synchronization cost for SGI MPI is unavailable due to lack of access to its source code. One reason for such a large difference is the message multiplexing/demultiplexing overhead in MPICH as explained in Section 5. The other reason is that communication volume in MM is large and system buffer can overflow during computation. For a process based implementation, data has to be fragmented to fit into the system buffer and copied to the receiver several times; while in TMPI, a sender blocks until a receiver copies the entire message. For the HEAT benchmark, SGI can outperform TMPI by around 25% when the number of processors becomes large. This is because the SGI version is highly optimized and can take advantages of more low-level OS/hardware support for which we do not have access. For the GE and Sweep3D, SGI and TMPI are about the same.

6.2 A Performance Comparison in Non-dedicated Environments

In a non-dedicated environment, the number of processors allocated to an MPI job can be smaller than the requested amount and can vary from time to time. Since we do not have control over the OS scheduler, we cannot fairly compare different MPI systems without fixing processor resources. Our evaluation methodology is to

Benchmark	Function	Code size	#permanent variables	MPI operations
GE	Gaussian Elimination	324 lines	11	mostly `MPI_Bcast`
MM	Matrix multiplication	233 lines	14	mostly `MPI_Bsend`
Sweep3D	3D Neutron transport	2247 lines	7	mixed, mostly recv/send
HEAT	3D Diffusion PDE solver	4189 lines	274	mixed, mostly recv/send

Table 1: Characteristics of the tested benchmarks.

	Kernel computation	Memory copy	Other cost (including synchronization)	Synchronization
TMPI	11.14 sec	0.82 sec	1.50 sec	0.09 sec
SGI MPI	11.29 sec	1.79 sec	7.30 sec	-
MPICH	11.21 sec	1.24 sec	7.01 sec	4.96 sec

Table 2: Execution time breakdown for 1152×1152 Matrix Multiplication on 4 processors. ”-” means data unavailable due to lack of access to SGI MPI source code.

create a repeatable non-dedicated setting on dedicated processors so that the MPICH and SGI versions can be compared with TMPI. What we did was to manually assign a fixed number of MPI nodes to each idle physical processor[3], then vary this number to check performance sensitivity.

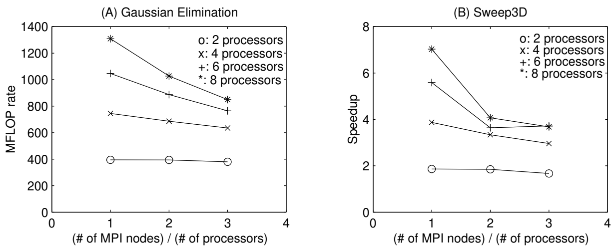

Figure 9: Performance degradation of TMPI in non-dedicated environments.

Figure 9 shows the performance degradation of TMPI when the number of MPI nodes on each processor increases. We can see that the degradation is fairly small when running no more than 4 processors. When the number of physical processors is increased to 8, TMPI can still sustain reasonable performance even though more communication is needed with more MPI nodes. MPICH and SGI MPI however, exhibit fairly poor performance when multiple MPI nodes share one processor. Tables 3 lists the performance ratio of TMPI to SGI MPI, which is the megaflop or speedup number of the TMPI code divided by that of the SGI MPI. Tables 4 lists the performance ratio of TMPI to MPICH. We do not report the data for MM and HEAT because the performance of MPICH and SGI deteriorates too fast when the number of MPI nodes per processor exceeds 1, which makes the comparison meaningless.

We can see that the performance ratios stay around one when $\frac{\text{\# of MPI nodes}}{\text{\# of processors}} = 1$, which indicates that all three implementations have similar performance in

[3]IRIX allows an SPROC thread be bound to a processor.

Benchmarks	GE			Sweep3D		
$\frac{\#ofMPInodes}{\#ofprocessors}$	1	2	3	1	2	3
2 processors	0.97	3.02	7.00	0.97	1.87	2.53
4 processors	1.01	5.00	11.93	0.97	3.12	5.19
6 processors	1.04	5.90	16.90	0.99	3.08	7.91
8 processors	1.04	7.23	23.56	0.99	3.99	8.36

Table 3: Performance ratio of TMPI to SGI MPI in a non-dedicated environment.

Benchmarks	GE			Sweep3D		
$\frac{\#ofMPInodes}{\#ofprocessors}$	1	2	3	1	2	3
2 processors	0.99	2.06	4.22	0.98	1.21	1.58
4 processors	1.01	3.06	6.94	0.99	1.55	2.29
6 processors	1.05	4.15	9.21	1.02	2.55	5.90
8 processors	1.06	3.31	10.07	1.03	2.64	5.25

Table 4: Performance ratios of TMPI to MPICH in a non-dedicated environment.

dedicated execution environments. When this node-per-processor ratio is increased to 2 or 3, TMPI can be 10-fold faster than MPICH and 23-fold faster than SGI MPI. This great performance gain is due to threads' low context switch cost and our less aggressive spin-block synchronization strategy. The SGI MPI has the poorest performance. It seems that the busy-waiting synchronization strategy in SGI MPI is more aggressive than MPICH, which leads to more contention when there are multiple nodes running on the same processor. Busy waiting, however, can deliver favorable performance in a dedicated environment.

6.3 Benefits of Address-sharing and Lock-free Management

Impact of data copying on point-to-point communication. We compare TMPI with SGI MPI and MPICH for point-to-point communication and examine the benefits of data copying due to address-sharing in TMPI. To isolate the performance gain due to the reduction in memory copying, we also compare TMPI with another version of TMPI (called TMPI_mem) which emulates the process-based communication strategy, i.e., double copying between user buffers and the system buffer. The micro-benchmark program we use does the "ping-pong" communication (`MPI_SEND()`), which sends the same data (using the same user data buffer) between two processors for over 2000 times. In order to avoid favoring our TMPI, we use standard send operations instead of buffered send.

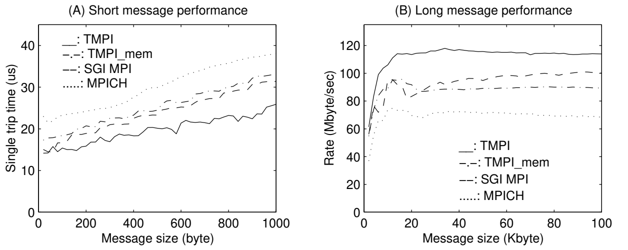

Figure 10: Communication performance of a ping-pong test program.

Figure 10 depicts the results for short and long messages. We use the single-trip operation time to measure short message performance and data transfer rate to measure long message performance because the message size does not play a dominant role in the overall performance for short messages. It is easy to observe that TMPI_mem shares a very similar performance curve with SGI MPI and the difference between them is relatively small, which reveals that the major performance difference between TMPI and SGI MPI is caused by saving on memory copy. And on average, TMPI is 16% faster than SGI MPI. TMPI is also 46% faster than MPICH, which is due to both saving on memory copy and our lock-free communication management. SGI MPI is slightly better than TMPI_mem, which shows that communication performance of SGI MPI is good in general if the advantage of address space sharing is taken away. Another interesting point in Figure 10(B) is that all three implementations except TMPI have a similar surge when message size is around 10K. This is because they have similar caching behavior. TMPI has a different memory access pattern since some memory copy operations are eliminated.

Effectiveness of the lock-free communication management. We assess the gain due to the introduction of lock-free message queue management by comparing it with a lock-based message queue implementation, called TMPI_lock. In the lock-based implementation, each channel has its own lock. The message sender first acquires the lock, then checks the corresponding receive queue. If it finds the matching handle, it releases the lock and processes the message passing; otherwise it enqueues itself into the send queue and then releases the lock. The receiver proceeds in a similar way. We use the same "ping-pong" benchmark in this experiment.

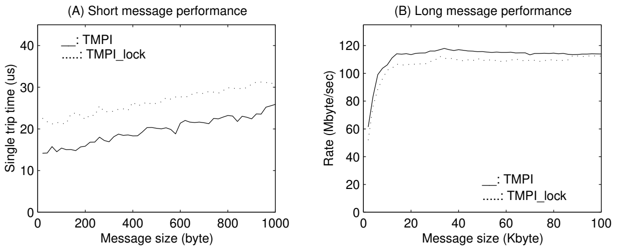

Figure 11: Effectiveness of lock-free management in point-to-point communication.

Figure 11 shows the experimental results for short and long messages. We can see that TMPI cost is constantly smaller than TMPI_lock by $5 - 6\mu s$ for short messages, which is a 35% overhead reduction. For long messages, its impact on data transfer rate will become smaller as the message size becomes very large. This is expected because the memory copy operations count for most of the overhead for long messages in this micro-benchmark.

7 Concluding Remarks

The main contribution of our work is the development of compile-time and run-time techniques for optimizing the execution of MPI code using threads. These include TSD-based transformation and an efficient and provably-correct, point-to-point communication protocol with a novel lock-free queuing scheme. These techniques are applicable to most of MPI applications, considering that MPI is mainly used in the scientific computing and engineering community.

The experiments indicate that our thread-based implementation TMPI using the proposed techniques can obtain large performance gains in a multiprogrammed environment with up to a 23-fold improvement compared to SGI MPI for the tested cases. TMPI is also competitive with SGI MPI in a dedicated environment, even though SGI MPI is highly optimized and takes advantage of SGI-specific low-level support [17]. The lock-free management is critical for minimizing communication overhead and it would be interesting to compare our design with the SGI's lock-free design, had it be documented.

The atomic operations used in our design should also be available in other SMMs such as SUN Enterprise. We plan to investigate this issue. We also plan to extend our compile-time support for C++/Fortran and examine the usefulness of our techniques for irregular computation with chaotic communication patterns [15, 28]. TMPI is a proof-of-concept system to demonstrate the effectiveness of our techniques, and we plan to add more MPI functions to TMPI.

Acknowledgment

This work was supported in part by NSF CCR-9702640 and by DARPA through UMD (ONR Contract Number N6600197C8534). We would like to thank Anurag Acharya, Rajive Bagrodia, Bobby Blumofe, Ewa Deelman, Bill Gropp, Eric Salo, and Ben Smith for their helpful comments, and Claus Jeppesen for his help in using Origin 2000 at UCSB.

References

[1] Information Power Grid. http://ipg.arc.nasa.gov/.

[2] MPI for NEC Supercomputers. http://www.ccrl-nece.technopark.gmd.de/~mpich/.

[3] MPI Forum. http://www.mpi-forum.org.

[4] T.E. Anderson. The Performance of Spin Lock Alternatives for Shared-memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[5] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread Scheduling for Multiprogrammed Multiprocessors. In *Proceedings of the 10th ACM Symposium on Parallel Algorithms and Architectures*, June 1998.

[6] R. Bagrodia, S. Docy, and A. Kahn. Parallel Simulation of Parallel File Systems and I/O Programs. In *Proc. of SuperComputing'97*.

[7] R. Bagrodia and S. Prakash. MPI-SIM: Using Parallel Simulation to Evaluate MPI Programs. In *Proc. of Winter Simulation Conference*, 1998.

[8] R. Brightwell and A. Skjellum. MPICH on the T3D: A Case Study of High Performance Message Passing. Technical report, Computer Sci. Dept., Mississippi State Univ, 1996.

[9] J. Bruck, D. Dolev, C. T. Ho, M. C. Rosu, and R. Strong. Efficient Message Passing Interface (MPI) for Parallel Computing on Clusters of Workstations. In *Proc. of 7th ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 64–73, 1995.

[10] M. Crovella, P. Das, C. Dubnicki, T. LeBlanc, and E. Markatos. Multiprogramming on Multiprocessors. In *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*, pages 590–597, December 1991.

[11] D. E. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture A Hardware/Software Approach*. Morgan Kaufmann Publishers, 1 edition, 1999.

[12] A. Ferrari and V. Sunderam. TPVM: Distributed Concurrent Computing with Lightweight Processes. In *Proc. of IEEE High Performance Distributed Computing*, pages 211–218, August 1995.

[13] I. Foster and C. Kesselman (Eds). *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.

[14] I. Foster, C. Kesselman, and S. Tuecke. The Nexus Approach to Integrating Multithreading and Communication. *J. Parallel and Distributed Computring*, (37):70–82, 1996.

[15] C. Fu and T. Yang. Space and Time Efficient Execution of Parallel Irregular Computations. In *Proceedings of ACM Symposium on Principles & Practice of Parallel Programming*, pages 57–68, June 1997.

[16] W. Gropp and E. Lusk. A high-performance MPI implementation on a shared-memory vector supercomputer. *Parallel Computing*, 22(11):1513–1526, January 1997.

[17] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-performance, Portable Implementation of The MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, September 1996.

[18] M. Herlihy. Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, 11(1):124–149, January 1991.

[19] L. I. Kontothanassis, R. W. Wisniewski, and M. L. Scott. Scheduler-Conscious Synchronization. *ACM Transactions on Computer Systems*, 1997.

[20] S. T. Leutenegger and M. K. Vernon. Performance of Multiprogrammed Multiprocessor Scheduling Algorithms. In *Proc. of ACM SIGMETRICS'90*, May 1990.

[21] S. S. Lumetta and D. E. Culler. Managing Concurrent Access for Shared Memory Active Messages. In *Proceedings of the International Parallel Processing Symposium*, April 1998.

[22] H. Massalin and C. Pu. A Lock-Free Multiprocessor OS Kernel. Technical Report CUCS-005-91, Computer Science Department, Columbia University, June 1991.

[23] B. Nichols, D. Buttlar, and J. P. Farrell. *Pthread Programming*. O'Reilly & Associates, 1 edition, 1996.

[24] J. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Proceedings of the Distributed Computing Systems Conf.*, pages 22–30, 1982.

[25] D. A. Patterson and J. L. Hennessy. *Computer Organization & Design*. Morgan Kaufmann Publishers, 2 edition, 1998.

[26] B. Protopopov and A. Skjellum. A Multi-threaded Message Passing Interface(MPI) Architecture: Performance and Program Issues. Technical report, Computer Science Department, Mississippi State Univ, 1998.

[27] E. Salo. Personal Communication, 1998.

[28] K. Shen, X. Jiao, and T. Yang. Elimination Forest Guided 2D Sparse LU Factorization. In *Proceedings of the 10th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 5–15, June 1998.

[29] A. Skjellum, B. Protopopov, and S. Hebert. A Thread Taxonomy for MPI. *MPIDC*, 1996.

[30] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference*. MIT Press, 1996.

[31] H. Tang, K. Shen, and T. Yang. Compile/Run-time Support for Threaded MPI Execution on Multiprogrammed Shared Memory Machines. Technical Report TRCS98-30, Computer Science Dept., UCSB, December 1998.

[32] A. Tucker and A. Gupta. Process Control and Scheduling Issues for Multiprogrammed Shared-memory Multiprocessors. In *the 12th ACM Symposium on Operating System Principles*, December 1989.

[33] K. K. Yue and D. J. Lilja. Dynamic Processor Allocation with the Solaris Operating System. In *Proceedings of the International Parallel Processing Symposium*, April 1998.

[34] J. Zahorjan and C. McCann. Processor Scheduling in Shared Memory Multiprocessors. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 214–225, May 1990.

[35] H. Zhou and A. Geist. LPVM: A Step Towards Multithread PVM. *Concurrency - Practice and Experience*, 1997.

A A List of MPI Functions Implemented in TMPI

```
MPI_Send()
MPI_Bsend()
MPI_Ssend()
MPI_Rsend()
MPI_Isend()
MPI_Ibsend()
MPI_Issend()
MPI_Irsend()
MPI_Send_init()
MPI_Bsend_init()
MPI_Ssend_init()
MPI_Rsend_init()
MPI_Recv()
MPI_Irecv()
MPI_Recv_init()
MPI_Sendrecv()
MPI_Sendrecv_replace()
MPI_Wait()
MPI_Waitall()
MPI_Request_free()
MPI_Comm_size()
MPI_Comm_rank()
MPI_Bcast()
MPI_Reduce()
MPI_Allreduce()
MPI_Wtime()
MPI_Barrier()
```