Program Transformation and Runtime Support for Threaded MPI Execution on Shared Memory Machines

HONG TANG, KAI SHEN, and TAO YANG
University of California, Santa Barbara

Parallel programs written in MPI have been widely used for developing high-performance applications on various platforms. Because of a restriction of the MPI computation model, conventional MPI implementations on shared memory machines map each MPI node to an OS process, which can suffer serious performance degradation in the presence of multiprogramming. This paper studies compile-time and runtime techniques for enhancing performance portability of MPI code running on multiprogrammed shared memory machines. The proposed techniques allow MPI nodes to be executed safely and efficiently as threads. Compile-time transformation eliminates global and static variables in C code using node-specific data. The runtime support includes an efficient and provably-correct communication protocol that uses lock-free data structure and takes advantage of address space sharing among threads. The experiments on 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 that it has significant performance advantages in a multiprogrammed environment.

Categories and Subject Descriptors: B.3.2 [**Memory Structures**]: Design Styles—*Shared memory*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; D.3.2 [**Programming Languages**]: Language Classifications—*Concurrent, distributed, and parallel languages*; D.3.4 [**Programming Languages**]: Processors—*Preprocessors; Run-time environments*; D.4.1 [**Operating Systems**]: Process Management—*Multiprocessing / multiprogramming / multitasking*; E.1 [**Data Structures**]: —*Lists, stacks, and queues*

General Terms: Design, Performance, Languages, Algorithms, Experimentation

Additional Key Words and Phrases: MPI, program transformation, lock-free synchronization, threaded execution, shared memory machines, multiprogrammed environments

1. INTRODUCTION

The Message-Passing Interface (MPI) [MPI-Forum 1999; Snir et al. 1996] is the de facto industry standard for developing high-performance parallel applications on various platforms. People use MPI on shared memory machines (SMMs) mainly because MPI programs with coarse grain parallelism can perform as well as other parallel programming models for SMMs such as threads or OpenMP, while retaining source level portability

This work has been partially supported by NSF CCR-9702640 and by DARPA through UMD (ONR Contract Number N6600197C8534).
A shorter version of this paper appeared in the Proceedings of 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99).
Authors' addresses: Department of Computer Science, University of California, Santa Barbara, CA, 93106; email: {htang, kshen, tyang}@cs.ucsb.edu.

for other platforms. However, supporting efficient execution of MPI code on an SMM is challenging because the MPI programming model does not take advantage of the underlying architecture. MPI uses the process concept, so global variables in an MPI program are not shared among MPI nodes. As a result, a conventional MPI implementation has to use heavy-weight processes for code execution and synchronization. Without sharing space among processes, passing messages between two MPI nodes must go through some system buffer and buffer copying degrades communication efficiency of MPI code[1].

SMMs are normally multiprogrammed, which can also impose great disadvantages for process-based MPI implementations. It has been widely acknowledged in the OS community that multiprogramming can yield higher throughput [Crovella et al. 1991; Leutenegger and Vernon 1990; Tucker and Gupta 1989; Zahorjan and McCann 1990; Ousterhout 1982]. It is possible to use a queuing system to execute jobs one by one. However, putting jobs in a batch queue is not feasible for applications that require interactive or real-time responses. There are various scheduling policies used for multiprogramming and a single MPI program typically runs well under the gang-scheduling policy [Feitelson 1997]. However, space/time sharing scheduling policies turn out to be popular because they are shown to be more effective for delivering high throughput [Crovella et al. 1991; Leutenegger and Vernon 1990; Tucker and Gupta 1989; Zahorjan and McCann 1990]. Modern operating systems such as Solaris 2.6 and IRIX 6.5 have adopted such a policy in their parallel job scheduling (see a discussion in Section 2 on gang-scheduling used in the earlier version of IRIX). As a consequence of such a scheduling policy, 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 during the execution. The performance of process-based MPI jobs is very sensitive to the variation of system load in such a setting because of frequent context switch and expensive synchronization.

Using light-weight threads to execute MPI nodes can improve the performance portability of an MPI program when running on an SMM under different OS scheduling policies and variable system loads. It can also facilitate efficient implementation of MPI communication primitives by taking advantage of address space sharing among threads. In this paper, we propose compile-time and runtime techniques that allow MPI nodes to be executed as threads. The compile-time code transformation eliminates global and static variables in an MPI program using node-specific data. The runtime techniques proposed in this paper are focused on lock-free point-to-point communication.

It should be noted that most of the application programs using MPI tend to be coarse-grained and they do not send small messages frequently. Thus a common intuition may be that performance of MPI code is less sensitive to multiprogramming and fast synchronization. However, our experiments in Section 6 show that this intuition is not correct and optimizing synchronization efficiency can greatly improve MPI code performance in a multiprogrammed environment.

The rest of this paper is organized as follows. Section 2 describes our current assumptions and related work. Section 3 discusses the compile-time transformation that produces thread-safe MPI code. Section 4 discusses our runtime support for threaded MPI execution. Section 5 presents our lock-free management for point-to-point communication. Section 6

[1] An earlier version of SGI MPI enforced that the address space of each MPI process is shared with every other. However, SGI eventually gave up this design due to insufficient address space and software incompatibility [Salo 1998].

presents the experimental results. Section 7 concludes the paper.

2. ASSUMPTIONS AND RELATED WORK

A parallel program that uses MPI contains a number of MPI nodes (or called processes in the literature). In a dedicated setting, each MPI node runs on one CPU and thus the degree of parallelism exploited during execution is the number of MPI nodes used. In a multiprogrammed environment, several MPI nodes may run on the same processor and using a light-weight thread to execute an MPI node provides better adaptiveness to load variation. The role of the program transformation is to ensure that each MPI node can be *safely* executed as a thread. Not every parallel program can be translated for thread-safe execution using our scheme. A restriction for our framework is that an MPI program does not call any low-level library function which is not thread-safe (e.g. signals). Most scientific programs do not involve such functions (MPI specification also discourages the use of signals) and most libraries for modern operating systems are designed to be thread-safe. Thus our techniques are applicable to a large class of scientific and engineering applications.

There are two other factors that need to be mentioned.

(1) The total memory used by all the MPI nodes must fit in a single virtual address space. This should not be a problem considering 64-bit OS now becomes more and more popular.
(2) The total number of files opened by all MPI nodes must fit in one process's open file table. This is adjustable by OS reconfiguration.

Our lock-free data structure design for thread communication makes the following assumption: each MPI node does not spawn multiple threads which call MPI functions simultaneously. As discussed in the MPI-2 standard [MPI-Forum 1999], there are four possible levels of thread support:

(1) Only one thread runs within each MPI node.
(2) Each MPI node may be multithreaded, but only the main thread will make MPI calls.
(3) There are multiple threads spawned, however the user program guarantees that MPI functions calls are serialized.
(4) No restriction.

The proposed lock-free data structure is intended for level-3 support and we plan to remove this restriction in the future work. It should be noticed that this assumption is only used in our runtime system design and does not affect the result of compile-time transformation. We also assume that basic synchronization primitives such as *read-modify-write* and *compare-and-swap* [Herlihy 1991] are available for implementing lock-free data structure. Actually, all modern microprocessors either directly support these primitives or provide atomic instructions such as load-linked/store-conditional (LL/SC) [Herlihy 1991] 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 [Foster et al. 1996]. Earlier attempts to run message-passing code on shared-memory machines include the LPVM [Zhou and Geist 1997] and TPVM [Ferrari and Sunderam 1995] 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 researches are focused on distributed memory machines or workstation clusters, e.g. [Bruck et al. 1997]. The MPI-SIM project [Prakash and Bagrodia 1998] has used multi-threading to simulate MPI execution on distributed memory machines for performance prediction as we will discuss in Section 3.1. Thread safety of MPI systems is addressed in [MPI-Forum 1999; Protopopov and Skjellum 1998; Skjellum et al. 1996] and recent commercial MPI products from SUN, IBM and SGI are thread-safe. 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.

Previous work has illustrated the importance of lock-free management for reducing synchronization contention and unnecessary delay due to locks [Anderson 1990; Arora et al. 1998; Herlihy 1991; Lumetta and Culler 1998; Massalin and Pu 1991]. Lock-free synchronization has also been used in the process-based SGI implementation [Gropp et al. 1996]. 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. Also, their design uses busy-waiting when a process is waiting for events [Salo 1998], which is not desirable for multiprogrammed environments [Kontothanassis et al. 1997; Ousterhout 1982]. Lock-free studies in [Anderson 1990; Arora et al. 1998; Herlihy 1991; Lumetta and Culler 1998; Massalin and Pu 1991] restrict their queue models to be either FIFO or FILO. These models are not sufficient for MPI point-to-point communication, and sometimes too general with unnecessary overhead for MPI. A study that attempts to use lock-free data structures for MPICH is conducted in a version for NEC shared-memory vector machines and Cray T3D [Gropp and Lusk 1997; Brightwell and Skjellum 1996; NEC 1999], in which they used single-slotted buffers for the ADI-layer communication. Their studies are still process-based and use layered communication management which is a portable solution with higher overhead 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 multiple MPI nodes.

Our study is leveraged by previous researches in OS job scheduling on multiprogrammed SMMs [Crovella et al. 1991; Leutenegger and Vernon 1990; Tucker and Gupta 1989; Zahorjan and McCann 1990; Yue and Lilja 1998]. These studies show that multiprogramming makes efficient use of system resources and space/time sharing is the most viable solution. Gang-scheduling [Feitelson 1997] is beneficial to a single MPI job, however, a hybrid strategy using space/time sharing can achieve higher throughput and shorter average response times. SGI OS adopts gang-scheduling in IRIX 6.4; however IRIX 6.5 changed the default scheduling to space/time sharing for shared memory applications. As explained in [NCSA 1999], SGI made this change because a gang-scheduled job cannot run until sufficient processors are available so that all members of the gang can be scheduled, and the turnaround time for a gang-scheduled job can be long. Also in IRIX 6.5, gang-scheduled jobs do not get priority over non-gang scheduled jobs. SGI MPI in IRIX 6.5 uses the default space/time scheduling and does not allow user to specify gang-scheduling (a mechanism that turns on gang-scheduling using `schedctl()` for an SPROC job does not work for this new SGI MPI version) [Salo 1998]. While it is still debatable which OS policies are preferable for a particular user environment, it is clear that different SMMs can employ different scheduling policies. Our goal is to allow an MPI program to perform well in the presence of multiprogramming under different scheduling policies.

The issues of performance portability were studied in [Jiang et al. 1997] for executing parallel programs written in threads which run well in hardware cache-coherent machines

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

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

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

Fig. 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.

but not in SVM systems (shared virtual memory) and their goal is to develop a general methodology that restructures applications manually through algorithmic or data structure enhancement. Our work focuses on automatic program transformation and system support for parallel programs using MPI.

3. PROGRAM TRANSFORMATION FOR THREADED MPI EXECUTION

The basic transformation that allows an MPI node to be executed safely as a thread is the elimination of global and static variables. In an MPI program, each node can keep a copy of its own *permanent variables* – variables allocated statically during compile time, such as global variables and local static variables. If such a program is executed by multiple MPI nodes as threads without any transformation, then all MPI nodes 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 MPI node.

3.1 Possible Solutions

There are 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 `umain()` and another routine called `tmain()` is created, which does certain initialization work and then calls `umain()`. This routine `tmain()` is used by the runtime 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, as 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 MPI node 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 could fail for some cases. A counter example is shown in Figure 2. After the transformation, function `foo()` carries an additional parameter to pass for global variable `x` while `foo2()` stays the same. Function `foo3()`

```
int x=0;
int foo(int a)
{    return a+x++; }
int foo2(int b)
{    return b>0?b:-b; }
int foo3(int u, int (*f)(int))
{    return (*f)(u); }
main()
{    printf("%d ", foo3(1, foo));
     printf("%d ", foo3(1, foo2));
}
```

Fig. 2. A counter example for parameter passing.

carries a function pointer and it may call `foo()` with argument `x` or call `foo2()` without any extra argument. As a result, it is very hard, if not impossible, for pointer analysis to predict whether `foo3()` should carry an additional argument in executing `*f()`.

The second solution, which is used by MPI-SIM [Prakash and Bagrodia 1998], is called *array replication*. The preprocessor re-declares each permanent variable with an additional dimension, whose size is equal to the total number of MPI nodes. There are three problems with this approach. First, the number of threads cannot be determined in advance at compile time. MPI-SIM [Prakash and Bagrodia 1998] uses an upper limit to allocate space and thus the space cost may be excessive. Secondly, even though space of global variables could be allocated dynamically, 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 recognized and moved out from their original lexical scopes. 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 not cache-line aligned [Patterson and Hennessy 1998; Culler et al. 1999].

Because of the above considerations, we have used the third approach based on NSD (node-specific data). This concept is derived from thread-specific data (TSD) available in POSIX threads [Nichols et al. 1996] [2]. Briefly speaking, TSD allows each thread to associate a private pointer sized value with a commonly shared (among all threads) key value which is a small integer. Given the same key value, TSD can store/retrieve a thread's own copy of data. By NSD, we mean the data is specific to each MPI node, but can be shared by all user-level threads spawned within this node. In our scheme, each permanent variable is replaced with a permanent key of the same lexical scope. Each MPI node dynamically allocates space for all permanent variables, initializes those variables for only once, and associates the reference of those variables with their corresponding keys. In the user program, for each function that refers to a permanent variable, this reference is changed to a call that retrieves the address of the node-specific copy of that permanent variable using the corresponding key. Such a transformation is general and the chance of false sharing is minimized because different nodes allocate their own node-specific data

[2] 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 SGI's SPROC thread.

separately.

In the example of Figure 1, two NSD 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 NSD-based Transformation

We have implemented a preprocessor for ANSI C (1989) [Kernighan and Ritchie 1988] to perform the NSD-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 or declared as a global variable (not within any function), then it will be replaced by a corresponding key declaration. This key is seen by all MPI nodes and is used to access the memory associated with the key. This key is initialized before MPI nodes are spawned. In the `tmain()` routine, a proper amount of space for this variable is allocated, initialized and then attached to this node-specific key. Notice that `tmain()` is the entry function spawned by the runtime system in creating multiple MPI nodes; thus the space allocated for this variable is node-specific.

— **Case 2: Static variables local to a control block.** A control block in C is a sequence of statements 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 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 MPI node that executes the control block. The corresponding space has to be allocated and initialized by each MPI node when it reaches the control block for the first time. Multiple MPI nodes 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 that defines a static variable, `static T V = I;` where T is its type, V is the variable name, and I is an initialization phrase. This statement is replaced with `static int key_V=0;` and Figure 3 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`. Also note that function `initval()` is similar to `setval()` except that it sets the value only when the key does not have an associated value yet. This is necessary because multiple threads in the same MPI node might attempt to allocate memory simultaneously.

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

```
if (key_V==0) {
    int new_key=key_create();
    compare_and_swap(&key_V, 0, new_key);
}
if ((p=getval(key_V))==NULL) {
    T tmp=I;
    void *m=malloc(sizeof(tmp));
    memcpy(m, &tmp, sizeof(tmp));
    if (initval(key_V, m)!=SUCCEED) {
        free(m);
        p=getval(key_V);
    } else {
        p=m;
    }
}
```

Fig. 3. Target code generated for a static variable definition `static T V = I;`.

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.

Correctness of the above transformation algorithm is not difficult to prove. However, when a C program links to some external library functions whose source code is not available, it is difficult to verify if the execution of the transformed program is thread-safe. This is the reason why we impose a constraint that such a program should only call thread-safe low-level functions as discussed in Section 2.

4. RUNTIME SUPPORT FOR THREADED EXECUTION

The intrinsic difference between the thread model and the process model has a big impact on the design of the runtime system. 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 [Salo 1998]. As a result, process-based implementation requires that inter-process communication go through an intermediate system buffer. Thus a thread-based runtime system can potentially reduce buffer overhead due to memory copying and overflow management.

Notice that in our implementation, if message sending is posted earlier than the corresponding receive operation, we choose not to let the sender block and wait for the receiver whenever possible, 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 sending is posted later than message receiving.** In this case, a thread-based system can directly copy data from the sender's user buffer to the receiver's user buffer.

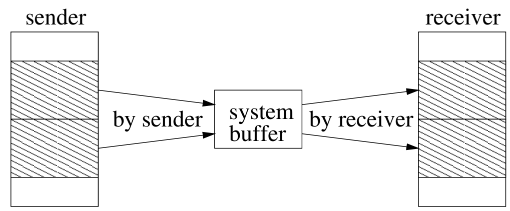

Fig. 4. Message fragmentation during system buffer overflow.

(2) **Buffered send operations.** MPI allows a program to specify a piece of user memory as a message buffer. In buffered send operation (`MPI_Bsend()`), if sending is posted earlier than receiving, the sender's message will be temporarily copied to the user-allocated buffer area before it is finally copied to the receiver's buffer. For a process-based implementation, 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 the system buffer, then the send operation must block and wait for the corresponding receive operation. In a thread-based implementation, a receiver can directly copy data from a sender's buffer. But in a 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 4 illustrates that copying needs to be done twice because the size of a message is twice as large as the free buffer size.

The thread model also gives great flexibility in the 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 these atomic operations can be potentially expensive due to multiple retries in the presence of contention.

MPI primitives include point-to-point communication and collective communication. Our focus in this paper is a point-to-point communication protocol which is specifically designed for threaded MPI execution and will be presented in next section. Our broadcasting queue management is based on previous lock-free FIFO queue studies [Herlihy 1991; Massalin and Pu 1991]. During event waiting, we adopt a spin-block strategy [Kontothanassis et al. 1997; Ousterhout 1982] when a thread needs to wait for certain events.

5. LOCK-FREE MANAGEMENT FOR POINT-TO-POINT COMMUNICATION

Previous lock-free techniques [Arora et al. 1998; Herlihy 1991; Lumetta and Culler 1998; Massalin and Pu 1991] 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 wild-card 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 modes: standard, buffered, synchronized and ready. A detailed specification of these primitives can be found in [MPI-Forum 1999; Snir et al. 1996]. Such a specification calls for a more generic queue

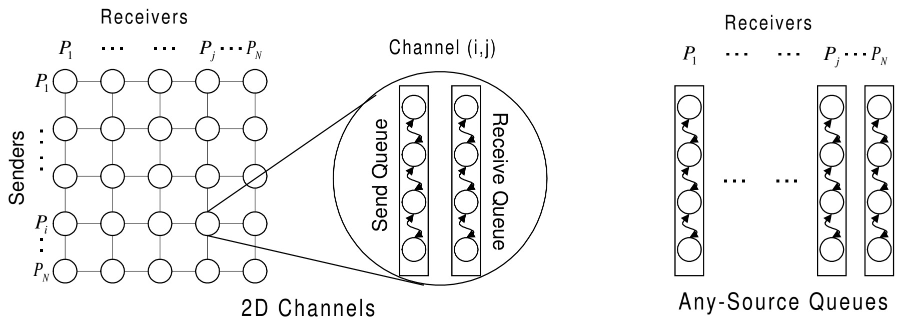

Fig. 5. The communication architecture.

model. On the other hand, as will be shown later, by keeping the lock free queue model specific to MPI, a simple, efficient, and correct implementation is still possible.

This section is organized as follows. Section 5.1 presents the communication architecture of TMPI. Section 5.2 presents the underlying lock-free queue model. Section 5.3 gives the protocol itself. Section 5.4 discusses the correctness of this protocol.

5.1 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 N additional 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 5.

We define a send request issued by node s to be *matchable* with a receive request issued by node r if:

(1) the destination name in the send request is r; **and**
(2) the source name 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`.

Communication based on this channel architecture can be accomplished efficiently. In the simplest case when sender P_i wants to send a message to receiver P_j, channel(i, j) will be used. If the sender comes first, it will post a send handle[3] in the send queue, and later the receiver will match this send handle. If a receive request is posted first, the corresponding receive handle is inserted in the receive queue. For an any-source receiving operation, the receiver may need to search N send queues to match a message with the proper tag. The details of our protocol using channels are described in Section 5.3. In terms of space cost, space overhead for each queue in this communication structure takes less than 50 bytes in our current implementation. For a large SMM with 256 processors, the channel architecture costs about 3MB in total. Even for an SMM with 1024 processors, the space overhead is 50MB (i.e. 48KB per processor) and is still quite small.

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

Our design is quite different from the layered design in MPICH. For the shared memory implementation of MPICH [Gropp et al. 1996; Gropp and Lusk 1997], $N \times N$ single-slotted buffers are used for message passing in a lower layer. In a higher layer, each process has three queues: one for sending, one for receiving, 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 for explicit point-to-point communication when there are many queued messages with different destinations or sources. On the other hand, the MPICH design may have a performance advantage for any-source receive. Let h be the total number of outstanding message handles inspected by TMPI for an any-source receive when such an inspection is necessary, the worst case cost of an any-source receive is $O(h + N)$ for TMPI while it is $O(h)$ for MPICH. This is because TMPI may scan all N queues even some of them are empty while MPICH only maintains one receive queue for all message handles with different sources. Thus our design seeks to optimize explicit point-to-point communication while imposing small overhead on any-source receive operations. We opt for this design trade-off because any-source receive operation is not frequently used in applications and benchmarks we have seen and the value N limited by the number of processors in an SMM is not too large in practice.

5.2 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 could be three types of operations performed on each queue:

—Put a handle into the end of a queue;

—Remove a handle from a queue (it can be in any place of the queue);

—Search (probe) a handle for matching a message.

Previous lock-free researches [Herlihy 1991; Lumetta and Culler 1998; Massalin and Pu 1991] usually assume 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 more 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 the 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 who buffers its receive requests with the *any-source* wild-card. It is worth emphasizing that slave nodes are only restricted not to modify the structure of a queue. They are still able to modify the content of queue nodes through atomic instructions (to avoid simultaneous modification).

Read/write contention can still occur when a master is trying to remove an interior handle while a slave is traversing the queue owned by this master. If we allow the master to proceed the remove operation in this case, the traversing slave may hold a reference to the removed garbage item, which can result in invalid memory access later. Herlihy [Herlihy 1991] proposed a solution to this problem by using accurate reference count 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 move the reference pointer and decrease or increase the reference count of handles using atomic operations. Each step of the traverse requires at least one two-word *compare-and-swap* and two atomic additions [Massalin and Pu 1991], which is very expensive. Another solution is to use a two-pass algorithm [Massalin and Pu 1991] in which the master 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 count (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 for each queue and perform one atomic operation when a slave starts or finishes a probe operation. This conservative approximation works well with small overhead if 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 the searching cost. If a handle to be removed is still referenced by some slave, 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 it is completely safe, i.e. the dead item's reference count is zero.

Figure 6 illustrates steps of our CRC method with semi-removal (Column 2) and those of the accurate reference count 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 count in column 3 is 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 count with safe removal by 10–20% in terms of average queue access times.

5.3 A 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 depicted in Figure 7. For each operation with request $R1$, the algorithm 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 the message passing. Notice that a flag is set by an atomic *compare-and-swap* instruction to ensure that only one request operation

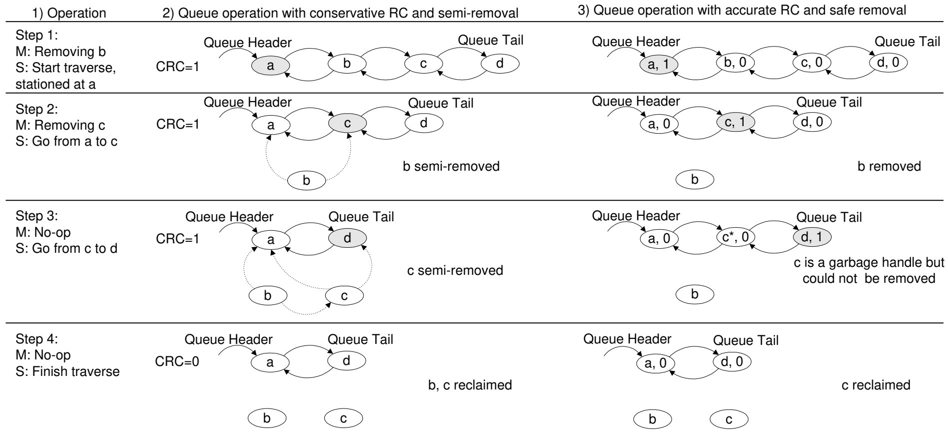

Fig. 6. An example of conservative reference count with semi-removal (column 2) compared to accurate reference count with safe-removal (column 3). Column 1 lists actions taken by the master (marked as M) and a salve (marked as S). Handles in shade are station points of the slave at each step. For accurate reference count, the reference count is also shown within each handle.

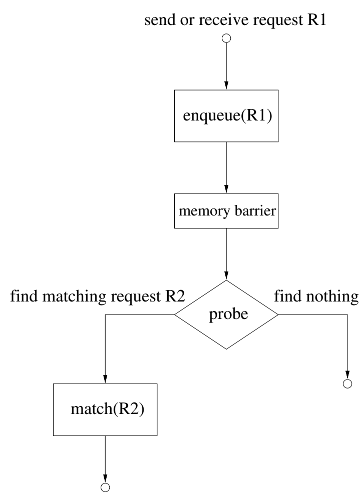

Fig. 7. Execution flow of a send or receive operation.

can succeed in matching the same handle. For systems that do not support sequential consistency, a memory barrier is needed between the enqueuing phase and the probing phase to make sure that the enqueuing completes execution before the 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 5.4 may not be valid.

Both send and receive operations have the same execution flow depicted in Figure 7 and their enqueue and probe procedures are described as follows.

— **Enqueue in a receive operation:** If a receive request has a specific source name, the receiver adds the receive handle to the end of the receive queue in the channel corresponding to the (sender, receiver) pair. If the receive request uses the *any-source* wild-card, the receiver adds this handle to the ASqueue it owns. Notice that an enqueued handle is attached with a logical time-stamp which is used to ensure the FIFO receive order.

— **Probe in a receive operation:** If the receive request specifies a source name, 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* wild-card, the receiver searches 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 a send operation:** The sender adds a send handle to the end of the send queue in the corresponding channel.

— **Probe in a 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 time-stamps 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 the probing of a send operation finds a matchable receive handle in a queue, the probing of this receive request may find 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.

Figure 8 shows the state transition graph of this point-to-point communication protocol. In the figure, the life cycle of a handle starts from state NEW and ends in state DEAD. In the NEW state, the handle is just created and not linked in the queue. After the enqueue phase, the handle goes to state PROBE. Depending on the result of the probe phase, the handle goes to either state PENDING or MATCHING. At the PENDING state, the handle will be matched by the peer (or may be cancelled by the owner). After a successful probe and mismatch detection, the handle will go to the intermediate MATCHING state to perform the actual message passing operation. However, as mentioned before, it might happen that the peer also moves to the MATCHING state, at which case we let the sender to yield and the receiver to proceed. That's why we have two arcs from MATCHING state to FREE state. FREE state simply means that the handle is no longer in use and can be removed by the owner. Eventually, the handle gets removed from the queue and goes to the DEAD state, at which point it is safe to be discarded or recycled later. Transitions in solid lines are triggered by the owner and those in dashed lines are triggered by the peer. If a handle can transit from one state to other states by both owner actions and peer actions,

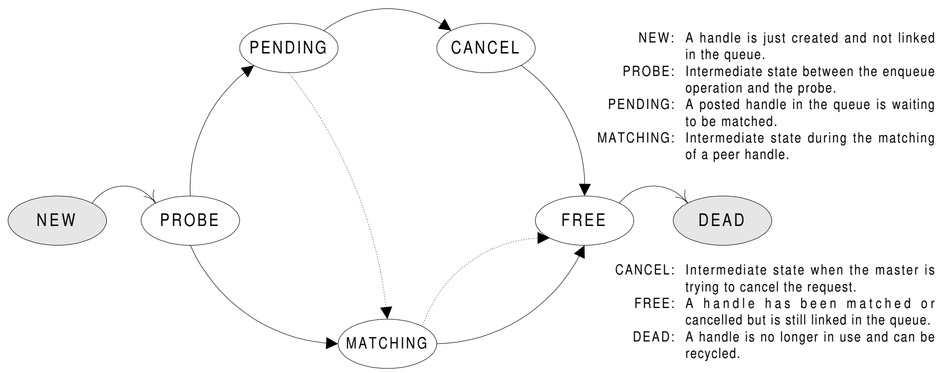

Fig. 8. The state transition graph of the point-to-point communication protocol. Transitions in solid lines are triggered by the owner and those in dashed lines are triggered by the peer.

then the alteration of the state flag must be done by using *compare-and-swap*.

5.4 Correctness Studies

Our point-to-point message passing primitives such as blocked or non-blocked communication are built on the top of the above protocol. A complete study of the correctness on message-passing behavior of an MPI program using our protocol relies on the characteristics of the program (e.g. deadlock-free). We however in this section provide three basic properties of our protocol and one can use these properties to ensure the correctness of higher level communication primitives. These properties address three basic issues:

— **No double matching.** One send (receive) request can only successfully match one receive (send) request.

— **Progress.** There couldn't be such a case that two matchable send-receive requests are pending in their queues forever.

— **Ordered delivery.** There couldn't exist such a case that the execution order of sending 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.

THEOREM 5.4.1. **(No double matching)** *Let two send requests be $S1$, $S2$ and two receive requests be $R1$, $R2$. Neither of the following two cases exists:*

—*Case 1: $S1$ and $S2$ are matched with $R1$.*

—*Case 2: $R1$ and $R2$ are matched with $S1$.*

PROOF. If Case 1 is true, there are three sub-cases.

— *Case 1.1: Probing of both $S1$ and $S2$ finds $R1$.* This is impossible since only one probing can succeed in matching the same handle due to the use of an atomic *compare-and-swap* instruction to modify the state flag in the handle.

— *Case 1.2: Probing of $S1$ finds $R1$ while probing of $R1$ finds $S2$.* This cannot happen since our mismatch-detection strategy ensures that $S1$'s probe compares its result with $R1$'s probing result. If $R1$'s probe matches $S2$ instead of $S1$, then $S1$ must give up this matching and it should not match $R1$.

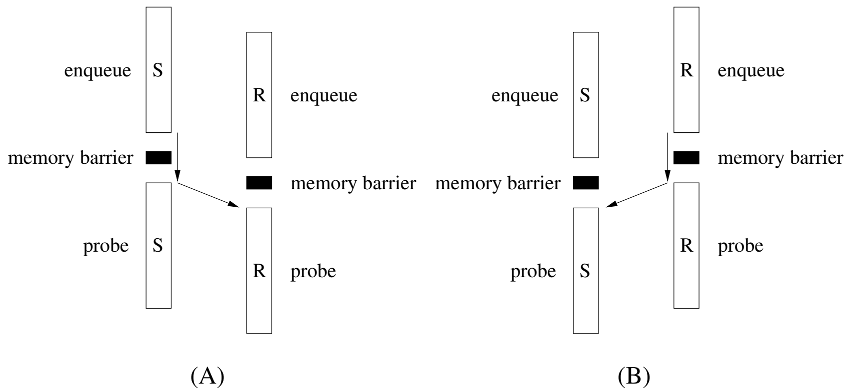

Fig. 9. Illustration for the proof of Theorem 5.4.2.

— *Case 1.3: Probing of* $S2$ *finds* $R1$ *while probing of* $R1$ *finds* $S1$. The proof is similar to Case 1.2. $S2$'s probing result must be consistent with $R1$'s probing result.

We can use a similar argument to show that Case 2 cannot be true. □

In our proofs for the second and third properties, we measure the starting and end time of an enqueuing or probing operation using a natural clock. Notice that this global time-stamp is only used for the proof purpose and it is not feasible to *explicitly* obtain such a time-stamp because each processor uses its own local clock for instruction execution. We define *Start*(e) as the time when any enqueue or probe operation e starts its first instruction on a processor. *End*(e) is the time when all instructions for e are completed, including all outstanding memory operations.

We will also use term *succeed* in the proof. We say a send (or receive) request *succeeds* if its corresponding send (or receive) operation matches a matchable request or it is matched by another receive (or send) operation.

THEOREM 5.4.2. **(Progress)** *There couldn't be such a case that two matchable requests* S *and* R *are pending in their queues after a program completes its execution.*

PROOF. We prove it by contradiction. Assume there exists a pair of matchable requests S and R in a given execution, neither of them succeeds at the end of program execution. Let S_{enq} and S_{probe} be S's enqueue and probe operation respectively. Let R_{enq} and R_{probe} be R's enqueue and probe operation respectively. Then there two possible situations.

— *Start*$(S_{\text{probe}}) \leq$ *Start*(R_{probe}).
As illustrated in Figure 9(A), since there is a memory barrier issued between S_{enq} and S_{probe}, we know *End*$(S_{\text{enq}}) <$ *Start*(S_{probe}). Therefore, *End*$(S_{\text{enq}}) <$ *Start*(R_{probe}) which means S is enqueued before R's probe is issued. Then at least R's probe can find this send handle. Using Theorem 5.4.1, either R succeeds in matching S or S has found another send request. This contradicts the assumption that neither S nor R succeeds.

— *Start*$(S_{\text{probe}}) >$ *Start*(R_{probe}).
As illustrated in Figure 9(B), the proof for this case is similar to the above case. We can show that R is enqueued before S's probe is issued, then we can induce a contradiction.

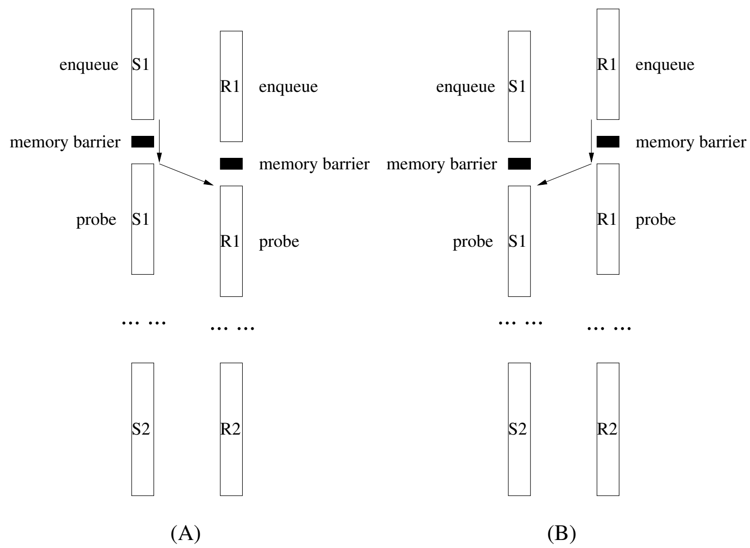

Fig. 10. Illustration for the proof of Theorem 5.4.3.

□

THEOREM 5.4.3. **(Ordered delivery)** *Let two send requests be $S1$, $S2$ and two receive requests be $R1$, $R2$. The following case does not exist:*

—*$S1$ and $S2$ are issued by the same sender and $S1$ is issued before $S2$;* **and**

—*$R1$ and $R2$ are issued by the same receiver and $R1$ is issued before $R2$;* **and**

—*$S1$ is matchable with $R1$;* **and**

—*$S1$ and $R2$ are matched together and $S2$ and $R1$ are matched together during program execution.*

PROOF. We prove it by contradiction. Assume there exists such a case. Let $S1_{\text{enq}}$ and $S1_{\text{probe}}$ be $S1$'s enqueue and probe operation respectively. Let $R1_{\text{enq}}$ and $R1_{\text{probe}}$ be $R1$'s enqueue and probe operation respectively. Then there are two possible situations.

— *Start*$(S1_{\text{probe}}) \leq$ *Start*$(R1_{\text{probe}})$.
As illustrated in Figure 10(A), since there is a memory barrier between $S1_{\text{enq}}$ and $S1_{\text{probe}}$, we know that *End*$(S1_{\text{enq}}) <$ *Start*$(S1_{\text{probe}})$. Therefore, *End*$(S1_{\text{enq}}) <$ *Start*$(R1_{\text{probe}})$, which means $S1$ is enqueued before the start of $R1$'s probe. Since $S1$ is matched with $R2$, this matching happens after $R1$'s probe because $R2$ is issued after $R1$ by the same receiver. This infers that $S1$ is enqueued but has not matched $R2$ or been matched by $R2$ when $R1$'s probe is issued. This leads to the result that $S1$ will be matched with $R1$. By Theorem 5.4.1, $S1$ and $R2$ cannot be matched together.

— *Start*$(S1_{\text{probe}}) >$ *Start*$(R1_{\text{probe}})$.
As illustrated in Figure 10(B), the proof is similar to the above case. We can show that $R1$ is enqueued but not matched when $S1$'s probe is issued, which also leads to the result that only $S1$ and $R1$ can be matched with each other.

□

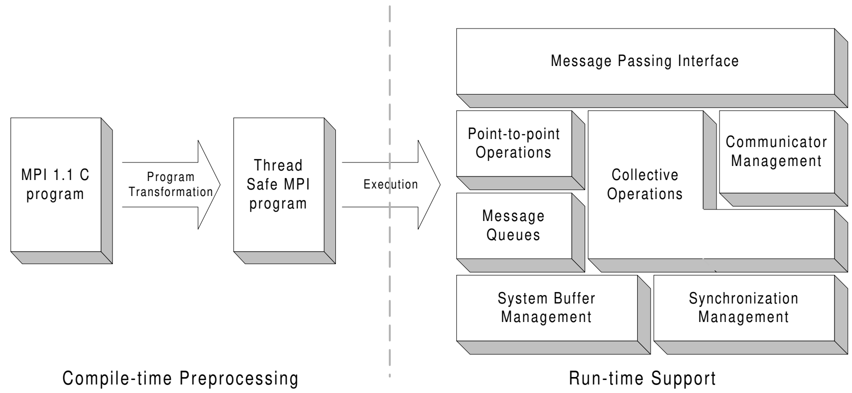

Fig. 11. System architecture of TMPI.

6. EXPERIMENTAL STUDIES

The main purpose of the experiments is to study if thread-based execution can gain great performance advantages in non-dedicated environments and be competitive against 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. Another purpose of our experiments is to examine the impact of address space sharing in reducing buffering overhead and the effectiveness of lock-free management. Most of the experiments are conducted on an SGI Origin 2000 at UCSB with 32 195MHz MIPS R10000 processors and 2GB memory. Some experiments are conducted on another Origin 2000 machine with 250MHz MIPS R10000 at NCSA.

We have implemented a prototype system called TMPI on SGI machines to demonstrate the effectiveness of our techniques. The architecture of TMPI is shown in Figure 11. Its runtime system 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.

We use the IRIX SPROC library because performance of IRIX Pthreads is not competitive with SPROC. The current prototype includes 29 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 the optimization and performance tuning for 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 barrier function in IRIX. We have not fully optimized those collective functions and our result shows that the impact is small in the sense that we are still able to demonstrate the advantages of multithreading. We compare the performance of our prototype with SGI's native implementation and 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 affect our experimental results.

The characteristics of the five test benchmarks we have used are listed in Table I. Two of them are kernel benchmarks written in C. One is a dense matrix multiplication using Cannon's method [Cannon 1969] and the other is a linear equation solver using Gaussian

Table I. Characteristics of the tested benchmarks.

Benchmark	Function	Problem Size	#Lines	#Perm Vars	MPI Ops
GE	Gaussian Elimination	2880×2880	324	11	`MPI_Bcast`
MM	Matrix multiplication	1440×1440	233	14	`MPI_Bsend`
Sweep3D	3D Neutron transport	50×50×50	2247	7	mixed
HEAT	3D Diffusion PDE	50×50×50	4189	274	mixed
CG	Conjugate Gradient	14000×14000	2489	32	mixed

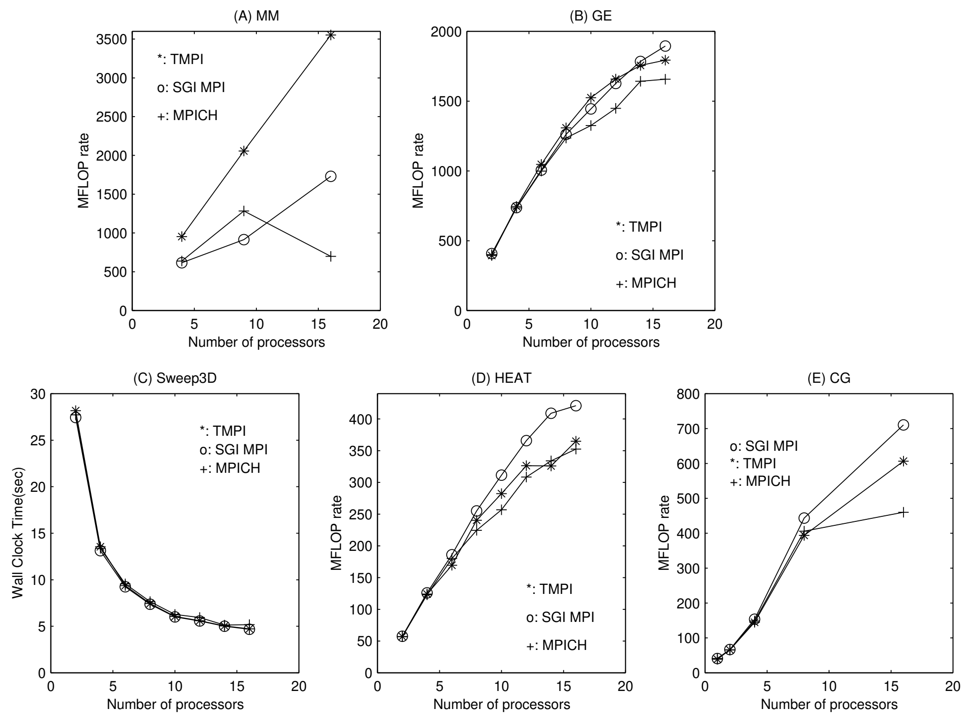

Fig. 12. Overall performance in dedicated environments at UCSB.

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 used a utility (f2c) to produce a C version for our test. The performance of the transformed program is about 70% to 80% of the original program's performance. 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 global variables used in its Fortran code. CG is also written in Fortran and comes from the NASA Numerical Aerospace Simulation parallel benchmark. Its global and static variables are also eliminated manually (with the aid of a Fortran transformation tool currently under development).

6.1 A Performance Comparison in Dedicated Environments

Figure 12 depicts the overall performance of TMPI, SGI and MPICH in a dedicated environment at UCSB. The y axes are either megaflop rates or wall-clock times reported by the

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

(seconds)	Kernel	Memory copy	Other cost (sync included)	Synchronization
TMPI	11.14	0.82	1.50	0.09
SGI MPI	11.29	1.79	7.30	-
MPICH	11.21	1.24	7.01	4.96

benchmarks. Notice that all benchmarks report megaflop rates, which are calculated using wall-clock elapse time, except for Sweep3D, which only reports wall-clock elapse time. We run the experiments three times and report the average. Every MPI node has exclusive access to a physical processor without interfered by other users in these experiments. We do not have experimental results for 32 nodes because the Origin 2000 machine at UCSB has always been busy.

From the result shown in Figure 12, 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 underlying physical processor. For GE and Sweep3D, SGI and TMPI are about the same. For HEAT and CG benchmark, SGI can outperform TMPI by 10-25% when the number of processors becomes large. For MM, TMPI outperforms SGI by around 100%. We used SGI's SpeedShop tool to study the execution time breakdown of MM and the results are listed in Table II. 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, so we further isolated the synchronization cost, which is the time spent in 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 message size in MM is large and system buffer may 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.

To further examine the scalability and competitiveness of TMPI, we have conducted additional experiments in an Origin 2000 machine at NCSA using up to 64 processors. Figure 13 shows the performance of three benchmarks using TMPI and SGI MPI. The Origin machine at NCSA has a clock rate faster than that at UCSB and thus three benchmarks have better performance at NCSA. Notice that TMPI is relatively slower than SGI MPI when the number of processors becomes 32 or 64 in the GE case. This is because after all SGI MPI has been fully optimized during the last few years [Gropp et al. 1996] and our collective communication implementation is not fully optimized (e.g. it uses central control and does not consider network topology). While there is room to further improve our implementation, the overall performance of current TMPI is still competitive to SGI.

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

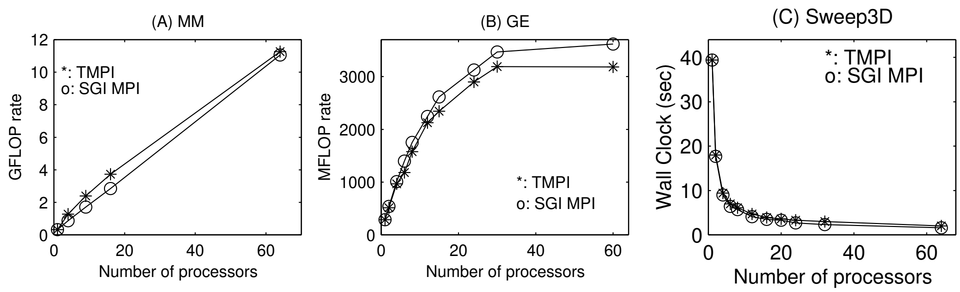

Fig. 13. Overall performance in dedicated environments at NCSA.

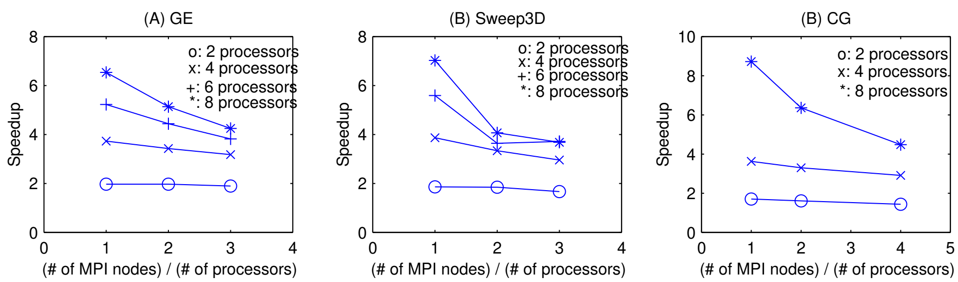

Fig. 14. Megaflop rates or speedups of TMPI code in non-dedicated environments.

fixing processor resources. Our evaluation methodology is to 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[4], then vary this number to check performance sensitivity.

Figure 14 shows the speedup of TMPI code for three benchmarks when the number of MPI nodes per processor increases. Performance degradation is fairly small when the number of MPI nodes is not more than 12. When this number increases to 24 or 32, TMPI can still sustain reasonable performance on 8 processors despite the increased communication overhead. Note for CG, it requires the number of MPI nodes to be a power of 2, so we do not have data for 6 processors and we tested multiprogramming degree of 1, 2, and 4. We do not list data for SGI MPI and MPICH because their performance deteriorates too fast when the number of MPI nodes per processor exceeds 1.

Tables III lists the performance ratio of TMPI to SGI MPI, which is the megaflop or speedup number of TMPI divided by that of SGI MPI. Tables IV lists the performance

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

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

Benchmarks	GE			Sweep3D			CG		
# of MPI nodes / # of processors	1	2	3	1	2	3	1	2	4
2 processors	0.97	3.02	7.00	0.97	1.87	2.53	1.00	2.36	5.58
4 processors	1.01	5.00	11.93	0.97	3.12	5.19	1.15	3.99	15.01
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	0.96	19.87	>50

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

Benchmarks	GE			Sweep3D			CG		
$\frac{\text{\# of MPI nodes}}{\text{\# of processors}}$	1	2	3	1	2	3	1	2	4
2 processors	0.99	2.06	4.22	0.98	1.21	1.58	1.01	3.84	15.31
4 processors	1.01	3.06	6.94	0.99	1.55	2.29	0.98	11.43	27.98
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	0.94	17.82	>50

ratio of TMPI to MPICH. We can see that 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 dedicated execution environments. When this node-per-processor ratio is increased to 2 or 3, TMPI can be 28-fold faster than MPICH and 23-fold faster than SGI MPI (besides the >50 case[5]).

To explain why TMPI outperforms SGI MPI significantly in a multiprogrammed environment, we again used SpeedShop to study the execution time breakdown for GE and SWEEP3D. We run 3 MPI nodes per processor with a total of 8 processors. Execution times reported in Table V are accumulated *virtual process times*[6]. As can be seen from Table V, for both GE and SWEEP3D, the kernel computation times for both versions are roughly the same. However, for SGI MPI, both programs incur substantially more overhead in synchronization and queue management. The saving from memory copy through address space sharing is limited (though obvious) compared with the saving by TMPI in synchronization and queue management. It seems that the synchronization strategy used in SGI MPI can significantly hurt MPI program performance in a multiprogrammed environment, even though it can deliver good performance in a dedicated environment. SGI uses a busy-waiting strategy in their lock-free communication design [Salo 1998], which could be a partial reason. Due to the lack of access to their implementation, we cannot conclude whether such a strategy is inherent to their specific lock-free design.

Obtaining such a large improvement over SGI MPI and MPICH (e.g. when the multiplexing degree is 3) raises a question: Will parallel code be too slow and sequential execution would actually be better for the above tested cases? The answer is no if using TMPI in above cases and can be yes if using SGI MPI and MPICH. For example, if a user runs the Sweep3D program using 24 MPI nodes and the OS assigns it to 8 processors, the speedup obtained TMPI is around 4 based on Figure 14 while the speedup is less than 0.5 using SGI MPI and 0.75 using MPICH, since TMPI is 8.36 and 5.25 times faster respectively based on Tables III and IV.

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 space sharing in TMPI. To isolate the performance gain due to

[5]For SGI MPI and MPICH, when we run CG using 32 MPI nodes on 8 processors, they could not terminate even after 10 minutes (which is more than 50 times of the execution time of TMPI) and we killed them before they finish.

[6]The profiling tool `ssrun` interrupts the process every 1ms and checks which function body the program counter is pointing to. It then estimates the *virtual process time* spent in a certain function call based on the percentage of the samplings of which the program counter points to that function. This will exclude the time when the system is providing services, such as executing system calls, because the tool cannot interrupt a system call and check the PC. Certain precaution has to be taken when interpreting these data.

Table V. Execution time breakdown for GE and SWEEP3D by running 3 MPI nodes on each processor using a total number of 8 processors. Invoked functions are sorted into 5 categories: kernel computation, synchronization, queue management, memory copy and others.

GE	TMPI		SGIMPI	
	Time(Sec)	Percentage	TIME(Sec)	Percentage
Kernel	35.3	56.7%	34.7	1.0%
Sync.	23.2	37.2%	2912.4	84.3%
Queue Mng.	0.7	1.1%	368.5	10.7%
Memcpy	3.1	5.0%	6.9	0.2%
Others	0.0	0.0%	132.7	3.8%
Total	62.3	100%	3455.2	100%

SWEEP3D	TMPI		SGIMPI	
	Time(Sec)	Percentage	TIME(Sec)	Percentage
Kernel	47.8	54.3%	48.3	5.6%
Sync.	38.1	43.3%	722.8	84.5%
Queue Mng.	1.0	1.1%	83.4	9.7%
Memcpy	1.1	1.3%	1.4	0.2%
Others	0.0	0.0%	0.0	0.0%
Total	62.3	100%	855.9	100%

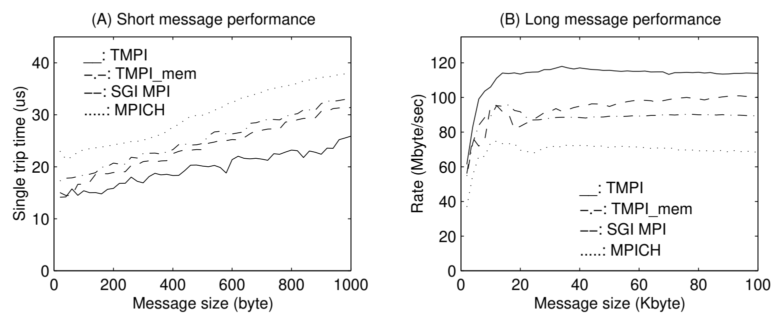

Fig. 15. Communication performance of a ping-pong microbenchmark. For short messages (left figure), the lower the curve is, the better the performance; for long messages (right figure), the higher the curve is, the better the performance.

the saving from memory copying, we also compare TMPI with another version of TMPI (called TMPI_mem) which emulates the process-based communication strategy, i.e., first copying from a sender's user buffer to the system buffer and then to a receiver's user buffer. The micro-benchmark program we use does the memory-to-memory "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 15 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 the saving from memory copy. And on average, TMPI is 16% faster than

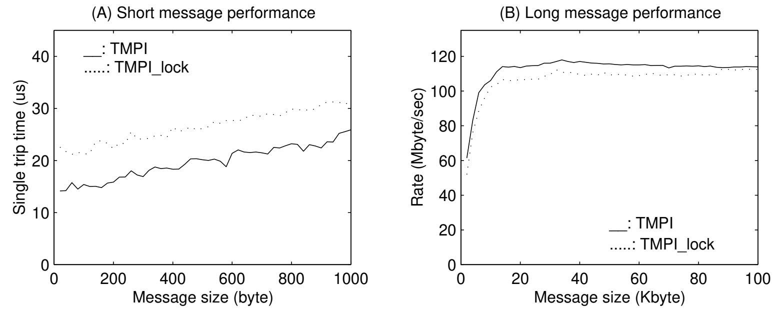

Fig. 16. Effectiveness of lock-free management in point-to-point communication.

SGI MPI. TMPI is also 46% faster than MPICH, which is due to both saving from 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 15(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 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 16 shows the experimental results for short and long messages. We can see that TMPI is constantly faster than TMPI_lock by 5-6μs 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 runtime techniques for optimizing execution of message-passing programs. These include NSD-based transformation for threaded execution and an efficient and provably-correct protocol for point-to-point communication with a novel lock-free queuing scheme. These techniques are applicable to most MPI applications, considering that MPI is mainly used in the scientific computing and engineering community. The experiments indicate that the TMPI prototype using the proposed techniques can obtain large performance gains in a multiprogrammed environment for the tested cases while it is competitive with SGI MPI in a dedicated environment.

The key advantage of using threads studied in this paper is to allow efficient design of inter-node communication through address space sharing and to allow MPI execution to be more adaptive to load variation under different OS scheduling policies. Another potential advantage is that when we use a user level thread to execute an MPI node, we can dynamically control the number of active kernel threads to match the number of available physical processors in order to minimize kernel level context switch cost. Recently [Shen et al. 1999] we have studied this idea and we find that minimizing unnecessary use of kernel-level threads in a multiprogrammed environment can lead to an additional 88% performance improvement.

TMPI is a proof-of-concept system intended for demonstrating the effectiveness of our techniques. Our current lock-free data structure does not allow multiple threads within each MPI node to call MPI functions concurrently and we plan to relax this restriction in the future. Our current implementation uses SGI machines and we are porting TMPI to PC Xeon SMPs. It should not be difficult to port our implementation to other SMM platforms by using Pthreads or by providing a thin thread system. Since SMM clusters become popular and MPI remains to be popular on such an architecture, we also plan to extend this work for SMM clusters.

APPENDIX: A List of MPI Functions Implemented in TMPI

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

ACKNOWLEDGEMENTS

We would like to thank anonymous referees, Anurag Acharya, Rajive Bagrodia, Bobby Blumofe, Ewa Deelman, Bill Gropp, and Eric Salo for their helpful comments, and Claus Jeppesen for his help in using Origin 2000 at UCSB.

REFERENCES

ANDERSON, T. E. 1990. The performance of spin lock alternatives for shared-money multiprocessors. *IEEE Trans. Parall. Distrib. Syst. 1,* 1 (Jan.), 6–16.

ARORA, N. S., BLUMOFE, R. D., AND PLAXTON, C. G. 1998. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the 10th Symposium on Parallel Algorithms and Architectures*. Puerto Vallarta, Mexico, 119–29.

BRIGHTWELL, R. AND SKJELLUM, A. 1996. MPICH on the T3D: a case study of high performance message passing. Tech. rep., Computer Sci. Dept., Mississippi State Univ.

BRUCK, J., DOLEV, D., HO, C.-T., ROŞU, M.-C., AND STRONG, R. 1997. Efficient message passing interface (MPI) for parallel computing on clusters of workstations. *Journal of Parallel and Distributed Computing 40,* 1 (10 Jan.), 19–34.

CANNON, L. E. 1969. A cellular computer to implement the kalman filter algorithm. Ph.D. thesis, Department of Electrical Engineering, Montana State University, Bozeman, MT. Available from UMI, Ann Arbor, MI.

CROVELLA, M., DAS, P., DUBNICKI, C., LEBLANC, T., AND MARKATOS, E. 1991. Multiprogramming on multiprocessors. In *Proceedings of the 3rd IEEE Symposium on Parallel and Distributed Processing*. IEEE, 590–597.

CULLER, D. E., SINGH, J. P., AND GUPTA, A. 1999. *Parallel Computer Architecture A Hardware/Software Approach*, 1 ed. Morgan Kaufmann Publishers, San Francisco, CA.

FEITELSON, D. 1997. Job scheduling in multiprogrammed parallel systems. Tech. Rep. Research Report RC 19790, IBM.

FERRARI, A. AND SUNDERAM, V. 1995. TPVM: distributed concurrent computing with lightweight processes. In *Proceedings of IEEE High Performance Distributed Computing*. IEEE, Washington, D.C., 211–218.

FOSTER, I., KESSELMAN, C., AND TUECKE, S. 1996. The Nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing 37,* 1 (25 Aug.), 70–82.

GROPP, W. AND LUSK, E. 1997. A high-performance MPI implementation on a shared-memory vector supercomputer. *Parallel Computing 22,* 11 (Jan.), 1513–1526.

GROPP, W., LUSK, E., DOSS, N., AND SKJELLUM, A. 1996. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing 22,* 6 (Sept.), 789–828.

HERLIHY, M. 1991. Wait-free synchronization. *ACM Trans. Program. Lang. Syst. 11,* 1 (Jan.), 124–149.

JIANG, D., SHAN, H., AND SINGH, J. P. 1997. Application restructuring and performance portability on shared virtual memory and hardware-coherent multiprocessors. In *Proceedings of the 6th ACM Symposium on Principles and Practice of Parallel Programming*. ACM, New York, 217–29.

KERNIGHAN, B. W. AND RITCHIE, D. M. 1988. *The C Programming Language* , 2 ed. Prentice Hall, Inc, Englewood Cliffs, NJ.

KONTOTHANASSIS, L. I., WISNIEWSKI, R. W., AND SCOTT, M. L. 1997. Scheduler-conscious synchronization. *ACM Trans. Comput. Syst. 15,* 1 (Feb.), 3–40.

LEUTENEGGER, S. T. AND VERNON, M. K. 1990. The performance of multiprogrammed multiprocessor scheduling algorithms. In *Proceedings of ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. New York, 226.

LUMETTA, S. S. AND CULLER, D. E. 1998. Managing concurrent access for shared memory active messages. In *Proceedings of the International Parallel Processing Symposium*. Orlando, Florida, 272–8.

MASSALIN, H. AND PU, C. 1991. A lock-free multiprocessor OS kernel. Tech. Rep. CUCS-005-91, Computer Science Department, Columbia University. June.

MPI-FORUM. 1999. MPI Forum. http://www.mpi-forum.org.

NCSA. 1999. NCSA note on SGI Origin 2000 IRIX 6.5. http://www.ncsa.uiuc.edu/SCD/Consulting/Tips/Scheduler.html.

NEC. 1999. MPI for NEC Supercomputers. http://www.ccrl-nece.technopark.gmd.de/˜mpich/.

NICHOLS, B., BUTTLAR, D., AND FARRELL, J. P. 1996. *Pthread Programming*, 1 ed. O'Reilly & Associates.

OUSTERHOUT, J. 1982. Scheduling techniques for concurrent systems. In *Proceedings of the 3rd International Conference of Distributed Computing Systems*. IEEE, 22–30.

PATTERSON, D. A. AND HENNESSY, J. L. 1998. *Computer Organization & Design*, 2 ed. Morgan Kaufmann Publishers, San Francisco, CA.

PRAKASH, S. AND BAGRODIA, R. 1998. MPI-SIM: using parallel simulation to evaluate MPI programs. In *Proceedings of Winter simulation*. Washington, DC., 467–474.

PROTOPOPOV, B. AND SKJELLUM, A. 1998. A multi-threaded message passing interface(MPI) architecture: performance and program issues. Tech. rep., Computer Science Department, Mississippi State Univ.

SALO, E. 1998. Personal communication.

SHEN, K., TANG, H., AND YANG, T. 1999. Adaptive two-level thread Management for fast MPI execution on shared memory machines. In *Proceedings of ACM/IEEE SuperComputing '99*. ACM/IEEE, New York. Will be available from www.cs.ucsb.edu/research/tmpi.

SKJELLUM, A., PROTOPOPOV, B., AND HEBERT, S. 1996. A thread taxonomy for MPI. *MPIDC*.

SNIR, M., OTTO, S., HUSS-LEDERMAN, S., WALKER, D., AND DONGARRA, J. 1996. *MPI: The Complete Reference*. MIT Press.

TUCKER, A. AND GUPTA, A. 1989. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating System Principles*. ACM, New York.

YUE, K. K. AND LILJA, D. J. 1998. Dynamic processor allocation with the Solaris operating system. In *Proceedings of the International Parallel Processing Symposium*. Orlando, Florida.

ZAHORJAN, J. AND MCCANN, C. 1990. Processor scheduling in shared memory multiprocessors. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. ACM, New York, 214–225.

ZHOU, H. AND GEIST, A. 1997. LPVM: a step towards multithread PVM. *Concurrency - Practice and Experience*.

Received July 6, 1999; revised February 28, 2000; accepted May 22, 2000