Optimizing Threaded MPI Execution on SMP Clusters

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

ABSTRACT

Our previous work has shown that using threads to execute MPI programs can yield great performance gain on multiprogrammed shared-memory machines. This paper investigates the design and implementation of a thread-based MPI system on SMP clusters. Our study indicates that with a proper design for threaded MPI execution, both point-to-point and collective communication performance can be improved substantially, compared to a process-based MPI implementation in a cluster environment. Our contribution includes a hierarchy-aware and adaptive communication scheme for threaded MPI execution and a thread-safe network device abstraction that uses event-driven synchronization and provides separated collective and point-to-point communication channels. This paper describes the implementation of our design and illustrates its performance advantage on a Linux SMP cluster.

Keywords

SMP clusters, MPI, Multi-threading, Communication optimization.

1. INTRODUCTION

With the commercial success of SMP architectures, SMP clusters with commodity components have been widely deployed for high performance computing due to the great economic advantage of clustering [1, 2]. MPI is a message-passing standard [14] widely used for high-performance parallel applications and has been implemented on a large array of computer systems. This paper studies fast execution of MPI programs on dedicated SMP clusters.

In the MPI paradigm, all the MPI nodes execute the same piece of program under separate address spaces. Each MPI node has a unique rank, which allows an MPI node to identify itself and communicate with its peers. As a result, global variables declared in an MPI program are private to each MPI node. It is natural to map an MPI node to a process. However, communication between processes have to go through operating system kernels, which could be very costly. Our previous studies [16, 18] show that process-based implementations can suffer large performance loss on multiprogrammed shared-memory machines (SMMs).

Mapping each MPI node to a thread opens the possibility of fast synchronization through address space sharing. This approach requires a compiler to transform an MPI program into a thread-safe form. As demonstrated in our previous TMPI work [16, 18], the above approach can deliver significant performance gain for a large class of MPI C programs on multiprogrammed SMMs.

Extending a threaded MPI implementation for a single SMM to support an SMP cluster is not straightforward. In an SMP cluster environment, processes (threads) within the same machine can communicate through shared memory while the communication between processes (threads) on different machines have to go through the network, which is normally several orders of magnitude slower than shared-memory access. Thus, in addition to mapping MPI nodes to threads within a single machine, it is important for an efficient MPI implementation to take advantage of such a two-level communication channel hierarchy.

The common intuition is that in a cluster environment, inter-node messaging delay dominates the performance of communication, thus the advantage of executing MPI nodes as threads diminishes. As will be shown later, our experience counters this intuition and multi-threading can yield great performance gain for MPI communication in an SMP cluster environment. This is because using threads not only speeds up the synchronization between threads on a single SMP node, it also greatly reduces the buffering and orchestration overhead for the communication among threads on different SMP nodes.

In this paper, we present the design and implementation of a thread-based MPI system on a Linux SMP cluster, and examine the benefits of multi-threading on such a platform. The key optimizations we propose are a hierarchy-aware and adaptive communication scheme for threaded MPI execution and a thread-safe network device abstraction that uses event-driven synchronization and provides separated collective and point-to-point communication channels. Event-driven synchronization among MPI nodes takes advantage of lightweight threads and eliminates the spinning overhead caused by busy polling. Channel separation allows more flexible and efficient design of collective communication primitives. The experiments we conduct is on a dedicated SMP cluster. We expect that greater performance gain can be demonstrated on a non-dedicated cluster and that is our future work.

The rest of the paper is organized as follows: Section 2 introduces the background and an overview of our thread-based MPI system (TMPI) on a cluster. Section 3 discusses the design of our TMPI system. Section 4 reports our performance studies. Section 5 concludes the paper.

2. BACKGROUND AND OVERVIEW

Our design is based on MPICH, which is known as a portable MPI implementation and achieves good performance across a wide range of architectures [8]. Our scheme is targeted at MPI programs that can be executed using threads. Thus we first give the type of MPI programs we address in this paper and briefly give an overview of the MPICH system. Then, we give a high-level overview of our thread-based MPI system (TMPI) on SMP clusters.

2.1 Using Threads to Execute MPI Programs

Using threads to execute MPI programs can improve performance portability of an MPI program and minimize the impact of multiprogramming due to fast context switch and efficient synchronization between threads. As shown in [16, 18], our experiments on an SGI Origin 2000 indicate that threaded MPI execution outperforms SGI's native MPI implementation by an order of magnitude on multiprogrammed SMMs.

As has been mentioned before, the need for compile-time transformation of an MPI program emerges from the process model used in the MPI paradigm. The major task of this procedure is called variable privatization. Basically we provide a per-thread copy of each global variable and insert statements in the original program to fetch each thread's private copy of that variable inside each function where global variables are referenced. Our algorithm is based on a general mechanism available in most thread libraries called thread-specific data or TSD. We extend TSD to make it feasible to run multi-threaded MPI programs. Note that our TSD-based transformation algorithm is able to support multithreading within a single MPI node. However, our current TMPI runtime system does not allow threads within a single MPI node call MPI functions simultaneously (`MPI_THREAD_SERIALIZED`). For detailed information, please refer to [18].

Not every MPI program can be transformed to map MPI nodes to threads. One major restriction affecting the applicability of threaded execution is that an MPI program cannot call low-level library functions which are not thread-safe (e.g. signals). Since most of the scientific programs do not involve such type of functions (particularly, MPI specification discourages the use of signals), our techniques are applicable to a large class of scientific applications. There are two other minor factors that need to be mentioned, which should not affect much the applicability of our techniques.

1. The total amount of memory used by all the MPI nodes running on one SMP node can fit in a single virtual address space. This should not be a problem considering 64-bit OS becomes more and more popular.
2. There is a fixed per-process file descriptor table size for most UNIX systems. Since UNIX's network library uses file descriptors to represent stream connections, applications might fail if the total number of files opened by all the MPI nodes on a single SMP is relatively large. Applications with a regular read-compute-write pattern can be modified by separating the file I/O from the program to circumvent this problem.

2.2 MPICH for SMP Clusters

MPICH follows a sophisticated layered design which is well-documented and discussed in the literature. Our design borrows some ideas from MPICH and we briefly summarize the architectural design of MPICH in this section.

The goal of MPICH's layered design is to make it easy and fast to port the system to new architectures, yet allow room for further tune-up by replacing a relatively small part of the software. As shown in Figure 1, MPICH's communication implementation consists of four layers. From bottom to top, they are:

1. Device layer. This includes various operating system facilities and software drivers for all kinds of communication devices.
2. ADI layer. This layer encapsulates the differences of various communication devices and provides a uniform interface to the upper layer. The ADI layer exports a point-to-point communication interface [7].
3. MPI point-to-point primitives. This is built directly upon the ADI layer. It also manages high-level MPI communication semantics such as contexts and communicators.
4. MPI collective primitives. This is built upon the point-to-point primitive layer. Messages share the same channel for both point-to-point communication and collective communication. MPICH uses special tagging to distinguish messages belong to a user point-to-point communication and internal messages for a collective operation.

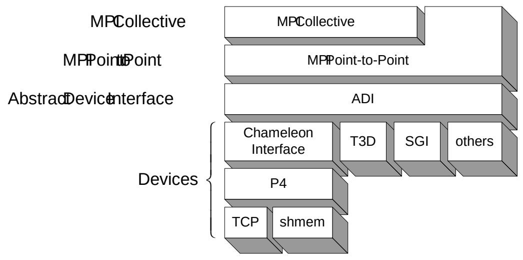

Figure 1: MPICH Communication Architecture.

To port MPICH to a different platform, it is only necessary to wrap the communication devices on the target system to provide the ADI interface. This design was mainly targeted to large parallel systems or networked clusters. It maps MPI nodes to individual processes. It is not easy to modify the current MPICH system to map each MPI node to a lightweight thread because the low-level layers of MPICH are not thread-safe. (Even though the latest MPICH release supports the MPI-2 standard, its MT level is actually `MPI_THREAD_SINGLE`.)

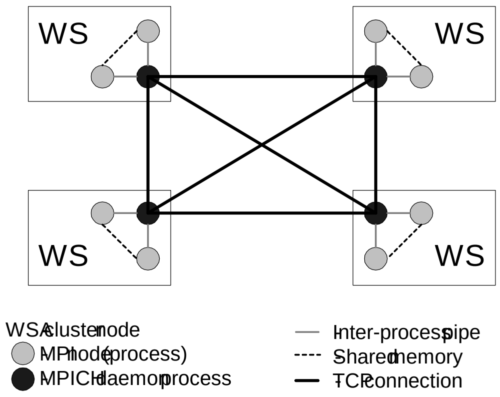

Figure 2: MPICH Using a Combination of TCP and Shared Memory.

As shown in Figure 1, the current support for SMP clusters in MPICH is basically a combination of two devices: a shared-memory device and a network device such as TCP/IP. Figure 2 shows a sample setup of such a configuration. In this example, there are 8 MPI nodes evenly scattered on 4 cluster nodes. There are also 4 MPI daemon processes, one on each cluster node, that are fully connected with each other. The daemon processes are necessary to drain the messages from TCP connections and to deliver messages across cluster-node boundaries. MPI nodes communicate with daemon processes through standard inter-process communication mechanisms such as domain sockets. MPI nodes on the same cluster node can also communicate through shared memory.

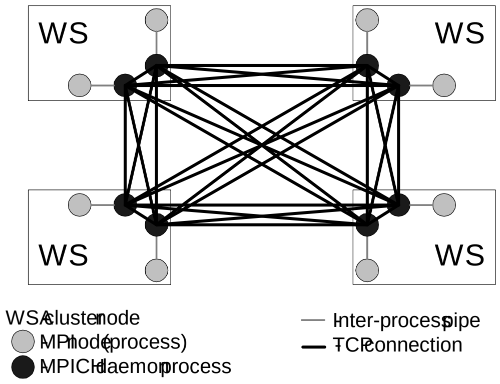

Figure 3: MPICH Using TCP Only.

One can also configure MPICH at compile time to make it completely oblivious to the shared-memory architecture inside each cluster node and use loopback devices to communicate among MPI nodes running on the same cluster node. In this case, the setup will look like Figure 3. In the example, we show the same number of MPI nodes with the same node distribution as in the previous configuration. What is different from the previous one is that there are now 8 daemon processes, one for each MPI node. Sending a message between MPI nodes on the same cluster node will go through the same path as sending a message between MPI nodes on different cluster nodes, but possibly faster.

2.3 Threaded MPI Execution on SMP Clusters

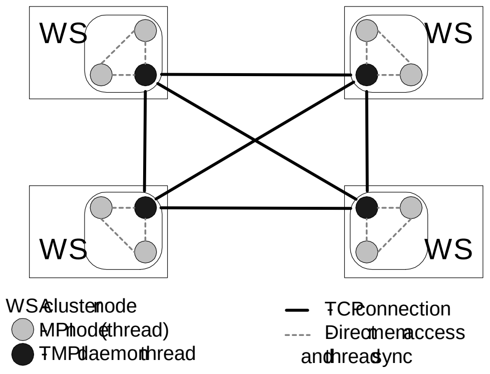

Figure 4: Communication Structure for Threaded MPI Execution.

In this section, we provide an overview of threaded MPI execution on SMP clusters and describe the potential advantages of TMPI. To facilitate the understanding, we take the same sample program used in Figure 2 and illustrate the setup of communication channels for TMPI (or any thread-based MPI system) in Figure 4. As we can see, we map MPI nodes on the same cluster node into threads inside the same process and we add an additional daemon thread in each process. Despite the apparent similarities to Figure 2, there are a number of differences between our design and MPICH.

1. In TMPI, the communication between MPI nodes on the same cluster node uses direct memory access instead of the shared-memory facility provided by operating systems.

2. In TMPI, the communication between the daemons and the MPI nodes also uses direct memory access instead of domain sockets.

3. Unlike a process-based design in which all the remote message send or receive has to be delegated to the daemon processes, in a thread-based MPI implementation, every MPI node can send to or receive from any remote MPI node directly.

As will be shown in later sections, these differences have an impact of the software design and provide potential performance gain of TMPI or any thread-based MPI systems. Additionally, TMPI gives us the following immediate advantages over process-based MPI systems. Comparing with an MPICH system that uses a mixing of TCP and shared memory (depicted in Figure 2):

1. For MPICH, usually shared memory is a limited system resource. There is an OS-imposed limit on the maximum size for a single piece of shared memory[1]; there is also a system-wide bound of the total size of all the shared memory. In fact, one test benchmark coming with MPICH failed because of limited shared-memory resource. A thread-based system doesn't suffer from this problem because threads in the same process can access the whole address space.

2. There is no easy way for MPICH to aggregate different types of devices. As a result, each MPI node has to use non-block polling to check if there are pending messages on either device, which could waste cpu cycles when the sender is not ready yet. Synchronizations among threads are more flexible and lightweight. Combined with our event-driven style daemons, all MPI nodes can freely choose busy spinning or blocking when waiting for a message.

3. Shared memory used in MPICH is a persistent system-wide resource. There is no automatic way to perform resource cleaning if the program doesn't clean it during its execution. Operating systems could run out of shared-memory descriptors when buggy user programs exit without calling proper resource cleanup functions and leave out garbage shared memory in the OS. Thus the reliability of user programs running on an MPICH-based cluster is more sensitive to this type of errors. For a thread-based system such as TMPI, there is no such a problem because shared address space access is a completely user-level operation.

Comparing with a thread-based MPI system, a purely TCP/IP-based MPICH implementation as depicted in Figure 3 has the following disadvantages:

1. There will be two more data copying to send a message between two MPI nodes on the same cluster node in MPICH. Synchronization between two MPI nodes also becomes more complicated than a thread-based system such as TMPI.

2. There will be a proliferation of daemon processes and network connections. This situation will get even worse for "fatter" cluster nodes (nodes with more processors) on which we run more MPI nodes.

2.4 Related Works

MPI on network clusters has also been studied in a number of other projects. LAM/MPI [13, 4] is a MPI system based upon a multicomputer management environment called Trollius [3]. It is different from MPICH in the sense that its design is specific for network clusters and that the lower level communication is provided by a stand-alone service through its unique Request Progression Interface. It does not address the issue of how to optimize MPI performance on a cluster of SMPs. Sun's MPI implementation [17] discusses how to optimize MPI's collective communication primitives on large scale SMP clusters, the focus of the work is on how to optimize collective operations on a single fat SMP node. MPI-StarT [9] made a couple of optimizations for SMP clusters by modifying MPICH's ADI layer. They propose a two-level broadcast scheme that takes advantage of the hierarchical communication channels. Our collective communication design extends their idea and is highly optimized for threaded MPI execution. MagPIe [10] optimizes MPI's collective communication primitives for clusters connected through a wide-area network. In MPI-FM [11] and MPI-AM [19, 12], they attempt to optimize the performance of lower level communication devices for MPI. Their techniques can be applied to our TMPI system. MPI-Lite [15], LPVM [20], and TPVM [6] study the problem of running message passing programs using threads on a single shared-memory machine.

To our knowledge, there is no research effort towards running MPI nodes using threads on SMP clusters. Our research complements the above work by focusing on taking advantage of executing each MPI node using a thread.

3. SYSTEM DESIGN AND IMPLEMENTATION

In this section, we detail the design and implementation of our thread-based MPI system – TMPI.

3.1 System Architecture

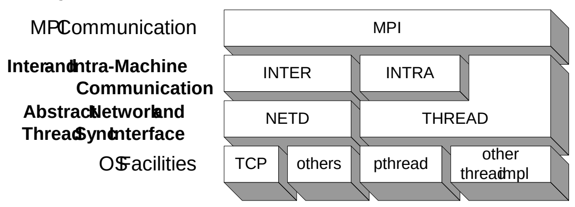

Figure 5: TMPI Communication Architecture.

The system architecture for MPI communication primitives is shown in Figure 5[2]. The four layers from bottom to top are:

1. Operating system facilities, such as TCP socket interface and pthread.

2. Network and thread synchronization abstraction layer. Potentially this layer allows for the portability of TMPI to other systems or performance tune-up by providing different implementations of network communication and thread synchronization. The thread synchronization abstraction is almost a direct mapping to pthread APIs, except that threads are launched in a batch style: create all the thread entities, start them through a single function call, and by the time that function call returns, all the threads will finish their execution. The network device abstraction is tailored to our threaded MPI execution model and is different from either the traditional socket interface or MPICH's ADI interface. We will talk about them in detail in Section 3.2 and Section 3.3.

[1]On the version of RedHat Linux 6.0 we installed (kernel version 2.2.15), this number is 4MB.

[2]There are some other synchronization primitives not shown in Figure 5, such as `compare-and-swap`.

3. Low level abstraction of communication management for threads on the same (INTRA) and different (INTER) cluster node(s). The intra-cluster-node communication manages the communication among threads on a single cluster node. The inter-cluster-node communication layer wraps the network abstraction interface and manages the logical communication on the multi-thread level. To be more specific, each thread has a local rank unique among threads on the same cluster node and a global rank unique among all the threads on all the cluster nodes. Given a global rank, we need to find on which cluster node the thread resides and what is its local rank on that cluster node as well; reverse lookup is also needed. Another functionality of the inter-cluster-node communication module is resource discover and allocation. The API allows for flexible placement of MPI nodes, e.g. how many cluster nodes should be used and what are the ranks of the MPI nodes placed on each cluster node. These decisions can be made anywhere from completely automatic or fully controlled by user supplied parameters.

4. MPI communication primitive implementation, including the notions of communicators, message tags and contexts. It is implemented upon three building blocks: intra- and inter-cluster-node communication, and thread interface. Particularly, all collective primitives are implemented with the awareness of the two-level communication hierarchy.

Despite some similarities to the MPICH design as shown in Figure 1, there are a couple of notable differences between the two systems:

- The top layer in TMPI is aware of different mechanisms to communicate between threads on the same or different cluster node(s) so that the implementation can organize the communication to take advantage of the two-level hierarchical communication channel.

- As will be further discussed in Section 3.3 and Section 3.4, in TMPI, collective communication primitives are not built upon point-to-point communication primitives. Instead, they are implemented independently in the top layer and the network device abstraction provides both point-to-point and collective communication among processes on different cluster nodes.

3.2 TMPI Network Device Abstraction

The network device abstraction in TMPI (called NETD) abstracts a network application as a group of relevant processes on different machines, each with a unique rank, and provides basic communication functionalities to the application. It is a thin layer that contains 28 core functions grouped into three categories:

Connection Management This includes the creation of processes on a number of cluster nodes and the setup/teardown of communication channels. All the relevant processes are fully connected with each other and can query about their own ID and the total number of the processes.

Collective Communication It provides collective communication among relevant processes created by NETD. The current implementation uses an adaptive algorithm to choose among three different spanning trees based on the number of the processes involved. When the size is small, a simple scatter tree is used; hypercube is used when the size is large; and balanced binary tree will be chosen when the size falls in the middle.

Point-to-Point Communication NETD has its unique point-to-point communication API. Each message contains a message header part and an optional payload part. The content of a message header is not interpreted by NETD. To receive a message, a caller must provide NETD a callback function (called a *message handle*). NETD buffers the message header and invokes the handle with the sender ID of the message and the message header. The message handle will be responsible for examining the header and performing necessary actions such as receiving the payload part if there is one. Such an interface is necessary to efficiently support MPI communication primitives in TMPI. An MPI node can receive a message from an unknown source (`MPI_ANY_SOURCE`). Such a situation is complicated in TMPI because normal network devices do not provide atomic receive operation. Thus if multiple threads wait for messages from the same port, a single logic message could be fragmented and received by different threads. To get around this problem, TMPI uses a daemon thread to receive all the incoming messages and invoke a message handle in the context of the daemon thread. That message handle will be responsible for decoding the actual source and destination, buffering the incoming data, and notifying the destination thread.

3.3 Separation of Point-to-Point and Collective Communication Channels in NETD

As we've mentioned before that in Figure 4, every thread in a process can access all the communication ports originated from that process. This feature inspired us into the idea of separating collective and point-to-point communication channels, which allows TMPI to take full advantage of the synchronous nature of collective communication and eliminate intermediate daemon overhead. For this reason, in Figure 4, each thick line actually represents two TCP connections, one dedicated for collective operations and the other for point-to-point communication. The daemon threads are only responsible for serializing incoming messages for the point-to-point communication.

The separation of point-to-point communication channels and collective communication channels is based on careful observations of MPI semantics. In MPI, receive operations are much more complicated than collective operations. Besides the wildcard receive operation we discussed before, it can be "out-of-order" with the notion of message tags, and "asynchronous" with the notion of non-block receives. Thus a daemon thread is required not only for serialization, but also for buffering purposes. Due to limited buffering in most network devices, if there is no daemon threads actively drains incoming messages, deadlock situations that are not permitted in MPI standard could happen. Figure 6 shows such an example where node 0 and node 1 would be blocked at the `MPI_Bsend` statement without the presence of daemon threads even if there is enough buffer space available.

For collective communication, MPI operations are never "out-of-order", and always "synchronous" in the sense that a collective operation will not be completed until all the MPI nodes involved reaches the same rendezvous point[3]. Further more, the structure of the spanning tree is determined at runtime before each operation.

MPICH does not separate collective and point-point communication channels, and all high-level MPI collective communication primitives are implemented on the top of point-to-point communication layer. As a result, all collective operations go through point-to-point communication daemons, which cause unnecessary overhead. Separation of point-to-point and collective communication channels could benefit a process-based MPI implementation as well; however it may not be as effective as in TMPI because two processes on the same cluster node cannot directly share the same network communication ports (e.g. sockets).

3.4 Hierarchy-Aware Collective Communication Design

The collective communication in TMPI are implemented as a series of intra- and inter-cluster-node collective communication. For examples, `MPI_Bcast` will be an inter-bcast followed by an intra-bcast and `MPI_Allreduce` will be an intra-reduce followed by an inter-reduce, then an inter-bcast and finished with an intra-bcast. The intra-cluster-node collective communication takes advantage of address space sharing and is implemented based on an efficient lock-free FIFO queue algorithm. In this way, collective communication can take full advantage of the two-level communication hierarchy.

Conceptually, having the above two-phase collective communication is the same as building a spanning tree in two steps. This idea was first mentioned in the MPI-StarT project [9]. Essentially there is a designated root MPI node on each cluster node that forms a spanning tree connecting all the cluster nodes. All the other MPI nodes connect to the local root on the same cluster node to form the whole spanning tree. Such a spanning tree will have exactly $n-1$ edges that cross cluster-node boundaries (called "network edges") where n is the number of cluster nodes involved in the communication.

Noted that MPICH which uses the shared memory setting actually does not take the two-step approach and builds a spanning tree directly from the given number of MPI nodes without knowing the distribution of MPI nodes on cluster nodes. As a result, a spanning tree for collective communication in MPICH may have more network edges. Figure 7 compares the spanning trees for a sample MPI program of size 9 running on 3 cluster nodes. As we can see, TMPI (the left part) results in 2 network edges and MPICH (the right part) has 5 network edges.

[3]It is possible to employ techniques such as pipelining and asynchronous operations to optimize collective operations. However, such optimizations are not required in the MPI standard and its effectiveness is not very eminent in real applications based on our experience.

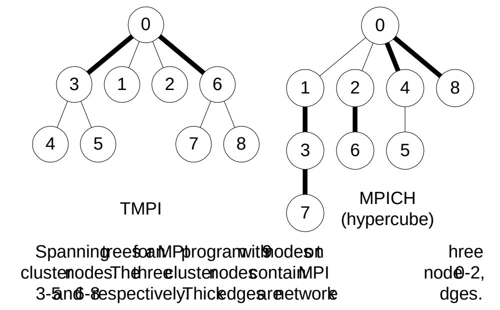
0
3
1
2
6
4
5
7
8
TMPI
0
1
2
4
8
3
6
5
7
MPICH
(hypercube)
Spanning trees for an MPI program with 9 nodes on
cluster nodes The three cluster nodes contain MPI
3-5 and 6-8 respectively Thick edges are network
hree
node 0-2,
dges.

Figure 7: Collective Communication Spanning Trees in TMPI and MPICH.

3.5 Adaptive Buffer Management in Point-to-Point Communication

The point-to-point communication in TMPI bears a lot of similarities to MPICH design. Conceptually, each MPI node has a receive queue and an unexpected-message queue. When a sender comes before the corresponding receiver, a send request handle will be deposited to the receiver's unexpected-message queue; similarly, if a receiver comes before the corresponding sender, a receive request handle will be stored in the receive queue. When a pair of sender and receiver are on different cluster nodes, the daemon thread on the receiver side will act on behalf of the remote sender.

One difficult problem facing this design is the temporary buffering of a message when the corresponding receiver is not ready yet. For intra-cluster-node point-to-point communication, we can always block the sender till the receiver comes if there is no internal buffer space available. However, when a sender sends a message to a remote receiver, it does not know whether there will be sufficient buffer space to hold the message. Because the message size in MPI could be arbitrarily large, a traditional conservative solution is the three-phase asynchronous message-passing protocol [5].

In TMPI, address space sharing and fast synchronization among threads lead us to an efficient adaptive buffering solution, which is a combination of an optimistic "eager-pushing" protocol and a three-phase protocol. Basically, the sender needs to make a guess based on the message size whether to transfer the actual message data with the send request metadata (the eager-pushing protocol) or send only the metadata first and send out data later when the receiver is ready (the three-phase protocol). The remote daemon on the receiver side will acknowledge whether the sender has made a correct guess or not.

Figure 8 shows the three simplified cases in TMPI's inter-cluster-node point-to-point communication protocol and we provide a detailed description in the following. Note that the figures do not accommodate the cases for "synchronous" or "ready" send operations. In Figure 8 (a), the sender sends the request with the actual data. On the receiver side, either the receiver already arrives or there are still internal buffer space available, so the daemon accepts the data, store the send request information and the actual data into the proper queue, notify the sender that the data are accepted, and wake up the receiver if necessary. The sender-side dae-

Node 0	Node 1

`MPI_Bsend(buf, BIG_SIZE, type, 1, ...);`
`MPI_Recv(buf, BIG_SIZE, type, 1, ...);`
	`MPI_Bsend(buf, BIG_SIZE, type, 0, ...);`
	`MPI_Recv(buf, BIG_SIZE, type, 0, ...);`

Figure 6: Possible deadlock without daemon threads.

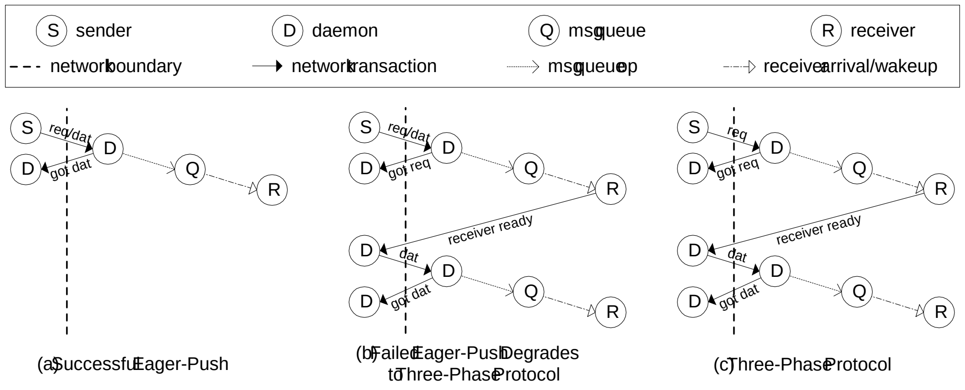

Figure 8: Point-to-Point Communication Between Nodes on Different Cluster Nodes.

mon receives that confirmation and frees the data on behalf of the sender. In Figure 8 (b), the sender still sends out the data with the request, but this time the receiver-side daemon cannot accept the incoming data. The daemon receives and discards the data, store only the request metadata in the unexpected-message queue. Later on, when the receiver arrives, it discovers the partially completed send request and asks for the associated data from the sender. The sender-side daemon then sends the data to the receiver side. When the receiver-side daemon receives the data, it saves the data to the receiver-supplied user buffer and tells the sender that the data are received. Subsequently, the sender-side daemon will deallocate the buffer upon receiving the acknowledge. Note that the actual data are transferred twice in this case. In Figure 8 (c), the sender decides to send the request part and data part separately. The whole flow is essentially the same as Figure 8 except that the actual data are only transfered once. This design allows for optimal performance when the sender makes the correct guess and functions correctly with degraded performance when it guesses wrong.

The remaining issue is to decide how the sender should switch between the eager-push and three-phase protocols. The decision is complicated by the fact that the internal buffer space in TMPI is shared by all the MPI nodes in the same cluster node and aggregated among a series of requests. So, against the common intuition to choose “eager-pushing” protocol as often as possible, it could still be preferable not to buffer a request even though there is buffer space available. This is because the same amount of buffer space might be used to hold data from multiple subsequent requests. Like a traditional non-preemptive scheduling problem, we should favor short requests and any algorithm could be suboptimal due to the lack of future knowledge. In the current implementation of TMPI, the sender will send the data with a request if and only if the message size is under a statically defined threshold. On the receiver side, the receiver daemon greedily buffers incoming data whenever possible. The current threshold is set to 100KBytes based on some empirical study.

Finally, allowing a receive daemon to send messages could result in a deadlock condition if not designed carefully. In TMPI, the NETD layer supports both blocked and non-blocked send. A non-blocked send merely puts the message into a queue and there is a send daemon responsible for sending the message out (and deallocate the memory if necessary). Our receive daemon always uses non-blocked send.

3.6 Further Discussions

Benefit of address space sharing in TMPI.

One benefit of a thread-based MPI implementation is the potential saving of data copying through address-space sharing. For intra-cluster-node point-to-point communication, TMPI only needs at most one intermediate data copying while for MPICH, it takes two intermediate data copying with shared memory or three intermediate data copying without shared memory. For inter-cluster-node point-to-point communication, since the daemons at either the sender- and receiver-side can access the sender- and receiver-buffer in TMPI, it takes zero to two intermediate data copying. For MPICH, it always needs three intermediate data copying. Additionally, data only need to move across process boundaries once in TMPI while data need to be transfered across process boundaries three times in MPICH. Since the two involving processes must be synchronized to transfer data from one to the other, MPICH performance is more sensitive to the OS scheduling in a multiprogrammed environment.

TMPI scalability.

The presence of a single receive daemon thread to handle all the incoming messages is a potential bottleneck in terms of scalability. In TMPI, it is possible to configure multiple daemons and all the incoming connections are partitioned among multiple daemons. However, currently it is just a compile-time parameter and we plan to study how to adaptively choose the number of daemons at runtime in the future. We can also create more instances of non-block send daemons or make it be responsible for messages send operations initiated by MPI nodes. This could be beneficial when there is a sudden surge of outgoing data and we do not want to block the sender threads because of that. In the current small scale settings, none of these configurations is necessary. Our point here is that the TMPI design is scalable to accommodate large clusters with fat SMP nodes.

4. EXPERIMENTS

In this section, we evaluate the effectiveness of the proposed optimization techniques for threaded MPI implementation versus a process-based implementation. We choose MPICH as a reference process-based implementation due to its wide acceptance. We should emphasize that the experiments are not meant to illustrate whether there are any flaws in the MPICH design. Instead, as discussed in the previous sections, we want to show the potential advantages of threaded MPI execution in an SMP cluster environments and in fact most of the optimization techniques in TMPI are only possible with a multi-threaded MPI implementation.

We have implemented a prototype system for Linux SMP clusters. It includes 45 MPI functions (MPI 1.1 standard) as listed in Appendix A. It does not yet support heterogeneous architectures[4], nor does it support user-defined data types or layouts. The MPICH version we use contains all functions in MPI 1.1 standard and provides partial support for MPI-2 standard. However, adding those functions is a relatively independent task and should not affect our experimental results.

All the experiments are conducted on a cluster of six quad-Xeon 500MHz SMPs, each with 1GB main memory. Each cluster node has two fast Ethernet cards connected with Lucent Canjun switch. The operating system is RedHat Linux 6.0 running Linux kernel version 2.2.15 with channel-bonding enabled[5].

4.1 Micro-Benchmark Performance for Point-to-Point Communication

In this section, we use micro-benchmarks to make fine-grain analysis of point-to-point communication sub-systems in both TMPI and MPICH.

Ping-pong test. First, we use the ping-pong benchmark to access the performance of point-to-point communication. We vary the data size from 4 Bytes to 1 Mega-bytes. As a common practice, we choose different metrics for small and large messages. For messages with size smaller than 1 Kilo-bytes, we report the performance in terms of round-trip delay; and for messages lager than 1 Kilo-bytes, we report the transfer-rate defined as total amount of bytes sent/received divided by the round-trip time.

[4]Note that MPICH detects whether the underlying system is homogeneous at the start-up time and no data conversion overhead is incurred if it is.

[5]Channel-bonding allows a single TCP connection to utilize multiple network cards, which could improve network bandwidth but does not help to reduce network delay. In our case, the achievable raw bandwidth will be 200Mbps between each pair of cluster nodes.

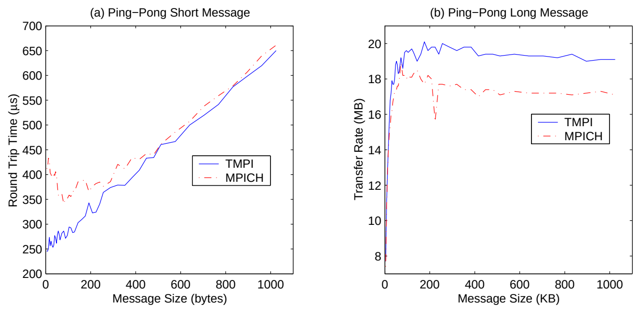

Figure 9: Inter-Cluster-Node Ping-Ping Performance.

Figure 9 shows the ping-pong performance for two MPI nodes on different cluster nodes. As we can see, when the message size is small, TMPI performs slightly better than MPICH, except when messages are very small, in which case TMPI's saving from inter-process data transfer overhead (such as system calls and context switches) becomes evident. When the message size becomes larger, TMPI has about a constant 2MB bandwidth advantage over MPICH due to the saving from data copying.

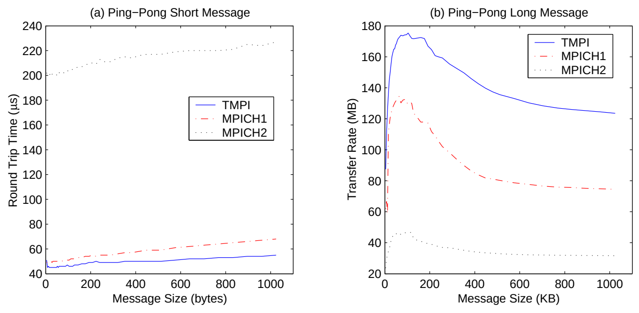

Figure 10: Intra-Cluster-Node Ping-Pong Performance. Two versions of MPICH are used: MPICH with shared memory (MPICH1) and MPICH without shared memory (MPICH2)

To access the impact of shared memory for both thread- and process-based MPI systems, Figure 10 shows the ping-pong performance for two MPI nodes on the same cluster node. We compare the performance for three MPI systems: TMPI, MPICH with shared memory (MPICH1) and MPICH without using shared memory (MPICH2). It is evident that ignoring the underlying shared-memory architecture yields much worse performance for MPICH2 comparing with the other two systems. TMPI's advantage over MPICH1 mainly comes from the saving of intermediate memory copying (for long messages) and fast synchronization among threads (for short messages). As a result, TMPI performance nearly doubled that of MPICH with shared memory. Note that both TMPI and MPICH1's performance drops after reaching a peak transfer rate, which is likely caused by underlying memory contention.

One-way message pipelining. The second benchmark is a simple one-way send-recv test, in which a sender keeps sending messages to the same receiver. Through this benchmark, we examine the impact of synchronization among MPI nodes. We compare the average time it takes for the sender to complete each send operation for short messages and still use the transfer rate metrics for large messages.

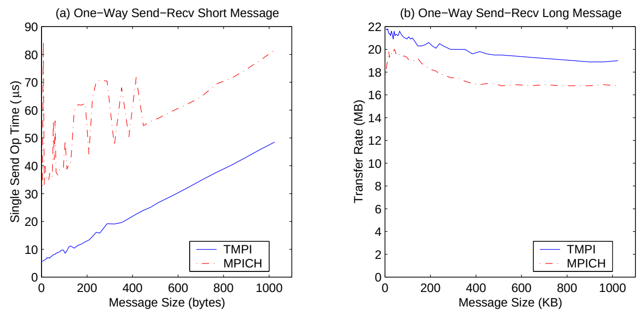

Figure 11: Inter-Cluster-Node One-Way Send-Recv Performance.

Figure 11 shows the one-way send-recv performance for two MPI nodes on different cluster nodes. Theoretically speaking, both thread- and process-based MPI implementations should yield similar performance for short messages because the average time for each send operation should be equal to the single trip time divided by the number of outstanding messages on the fly, which is basically limited by the network bandwidth. However, as mentioned before, in a process-based MPI implementation, inter-cluster-node message passing needs to travel through a local daemon and a remote daemon. Each time data need to be copied to these daemon's local buffer. Unless the sender, the receiver and the daemons are precisely synchronized, either buffer could be full or empty and cause the stall of the pipeline even though there might still be bandwidth available. As can be seen from Figure 11, MPICH performance is very unstable when message size is small due to the difficulty of synchronization among the processes. When message size becomes large, there are fewer outstanding messages and the performance is less sensitive to synchronization. Additionally, we can also see that TMPI has a $30\mu s$ advantage over MPICH for short messages and a 2MB bandwidth advantage for large messages due to the saving from extra data copying needed for a process-based MPI system.

As a comparison, we repeat the one-way send-recv test for two MPI nodes on the same cluster node. The result is shown in Figure 12. We again compare among three MPI systems with the same notation as in Figure 10. As we expected, MPICH1 performs almost identical with TMPI for small messages and only slightly poorer than TMPI for large messages, because there is no intermediate daemon processes along the data path in either system and that the cost of an extra data copying in MPICH1 is amortized among multiple outstanding requests. On the other hand, MPICH2 shows some irregular behavior for a certain range of message sizes. Note that in Figure 12 (a), we have to use a different scale to accommodate the performance curve of MPICH2. When the message size is below 800 bytes, a single send operation takes 3–4ms to complete, but when the message size goes beyond

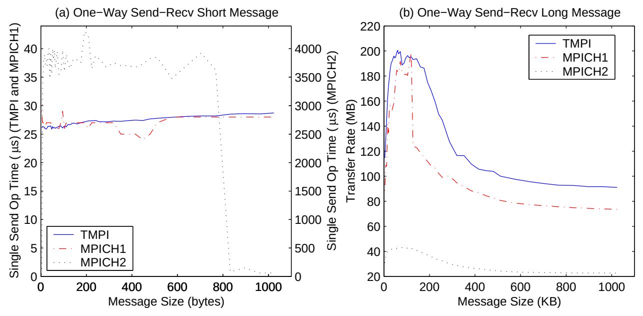

Figure 12: Intra-Cluster-Node One-Way Send-Recv Performance. Two versions of MPICH are used: MPICH with shared memory (MPICH1) and MPICH without shared memory (MPICH2)

that, it falls to a normal range between $50\mu s$ and $150\mu s$. We are not able to identify the exact source of the problem, but we think it might have something to do with resource contentions in the OS level. Regardless of this glitch, the performance of MPICH2 is much worse than the other two MPI systems, again due to its ignorance of the underlying shared memory.

4.2 Micro-Benchmark Performance for Collective Communication

To compare the performance of the collective communication primitives, we run three micro-benchmarks, each of which calls `MPI_Reduce`, `MPI_Bcast`, or `MPI_Allreduce` a number of times respectively and we compare the average time for each operation. The data volume involved in these benchmarks are very small, so the cost mainly comes from synchronization. We run these benchmarks with three different settings: 4 MPI nodes on the same cluster node (4×1), 4 MPI nodes scattered on 4 different cluster nodes (1×4), and 16 MPI nodes scattered on 4 MPI nodes (4×4). For `MPI_Bcast` and `MPI_Reduce`, we further use three different variations with regard to the root nodes in different iterations in each test: always stay the same (same); rotate among all the MPI nodes (rotate); or repeat using the same root for a couple of times and then do a root shift (combo). A number of conclusions can be drawn from the experiment results shown in Figure 13:

1. In most cases, MPICH2 performs the worst among the three MPI systems (except for the 1×4 case, in which MPICH1 and MPICH2 performance are almost the same), which signifies the importance of taking advantage of shared memory for an MPI system in an SMP cluster environment.

2. TMPI is up to 71 times faster for `MPI_Bcast` and 77 times faster for `MPI_Reduce` than MPICH1 which exploits shared memory within each SMP. Among other factors including address space sharing and hierarchy-aware algorithms, the performance gain mainly comes from the separation of collective and point-to-point communication channels, Theoretically speaking, an `MPI_Bcast` or `MPI_Reduce` operation should always be faster than an `MPI_Allreduce` operation. However, our

unit: μs		`MPI_Reduce`			`MPI_Bcast`			`MPI_Allreduce`		
Node Distr.	Root	TMPI	MPICH1	MPICH2	TMPI	MPICH1	MPICH2	TMPI	MPICH1	MPICH2
4×1	same	9	121	4384	10	137	7913	160	175	627
	rotate	33	81	3699	29	91	4238			
	combo	25	102	3436	17	32	966			
1×4	same	28	1999	1844	21	1610	1551	571	675	775
	rotate	146	1944	1878	164	1774	1834			
	combo	167	1977	1854	43	409	392			
4×4	same	39	2532	4809	56	2792	10246	736	1412	19914
	rotate	161	1718	8566	216	2204	8036			
	combo	141	2242	8515	62	489	2054			

Figure 13: Collective Communication Performance. The numbers shown in the table are the average time (μs) for each operation. We run each benchmark on three MPI systems: TMPI, MPICH with shared memory (MPICH1) and MPICH without shared memory (MPICH2). For `MPI_Reduce` and `MPI_Bcast`, we test cases when the root is always the same (same), when the root rotates among all the MPI nodes (rotate), or when we fix the root for a number of times then do a rotate (combo). The notation $a \times b$ used in the node distribution means we use b cluster nodes, each of which has a MPI nodes.

experiments show that there are cases for MPICH that an `MPI_Bcast` or `MPI_Reduce` operation performs worse than an `MPI_Allreduce` operation. Such an anomaly is caused by MPICH's design of collective communication on top of point-to-point communication. Messages for collective communication still have to go through daemons, get stored in message queues, and be matched by traversing the queues. When there are many outstanding requests, the cost of queue operations become expensive due to contentions, and the daemons could become a bottleneck. `MPI_Allreduce` test does not suffer from this problem because all the MPI nodes are synchronized and there is at most one outstanding request. On the other hand, by separating the communication channels of point-to-point and collective communication, TMPI does not show such an anomaly.

3. For TMPI, the "same root" tests perform much better than "rotating root" tests for the 1×4 and 4×4 cases. This means that TMPI can take better advantage of message pipelining due to address space sharing and its elimination of intermediate daemons.

4. Figure 13 also evidences the advantage of hierarchy-aware communication design. For example, in `MPI_Bcast` test, the 4×4 performance is roughly equal to the summation of the 4×1 case and the 1×4 case, since in TMPI, a broadcast in the 4×4 case is a 1×4 broadcast followed by a 4×1 broadcast on each individual cluster node. On the other hand, without using the two-level spanning tree, MPICH1's 4×4 performance is about 10%–60% worse than the summation of the 4×1 case and the 1×4 case. Similar conclusions also hold for `MPI_Reduce` and `MPI_Allreduce`.

4.3 Macro-Benchmark Performance

In this section, we use two application kernel benchmarks to further access the effectiveness of TMPI optimizations. The kernel benchmarks we use are Matrix-Multiplication (MM) and Gaussian-Elimination (GE) which perform computation-intensive linear algebra computation. MM consists mostly `MPI_Bsend` and `MPI_Recv`, and GE `MPI_Bcast`. We run MM with 1 to 16 MPI nodes and GE with 1 to 24 MPI nodes. The detailed node distribution is shown in Figure 14.

MM		GE			
Node #	Distr.	Nodes #	Distr.	Nodes #	Distr.
1	1×1	1	1×1	9	3×3
4	2×2	2	2×1	12	3×4
9	3×3	4	2×2	18	3×6
16	4×4	6	2×3	24	4×6
		8	3×2+2×1		

Figure 14: Distribution of MPI Nodes. $a \times b$ means we use b cluster nodes, each of which has a MPI nodes. We ensure that the number of cluster nodes and the number of MPI nodes on each cluster node will not decrease with the increase of total number of MPI nodes.

We compare TMPI with MPICH which uses shared memory on each SMP. The performance results are depicted in Figure 15. As we can see, when the number of MPI Nodes is small ($\leq$4), TMPI and MPICH perform similarly. However, when the number of MPI nodes becomes large and when there are more cluster nodes involved, TMPI shows better scalability than MPICH. For MM, TMPI has a constant 150MFLOP advantage over MPICH, which mainly comes from the saving of intermediate copying. For GE, neither system can keep linear speed-up and MPICH performance even degradates after reaching the peak at 12 MPI nodes. TMPI outperforms MPICH more than 100% when there are 4 to 6 cluster nodes involved. This indeed verifies the advantages of TMPI in an SMP cluster environment.

The reason that TMPI gains much more advantage in the GE case compared to the MM case is because GE calls the MPI broadcasting function while MM only uses point-to–point communication. As we have demonstrated in the previous section, TMPI outperforms MPICH more substantially in collective communication.

5. CONCLUDING REMARKS

In this paper, we have presented the design and implementation of a thread-based MPI system for SMP clusters. Our contributions include a novel network device abstraction interface tailored for threaded MPI execution on SMP clusters and hierarchy-aware communication. We proposed a number of optimization techniques including the separation of

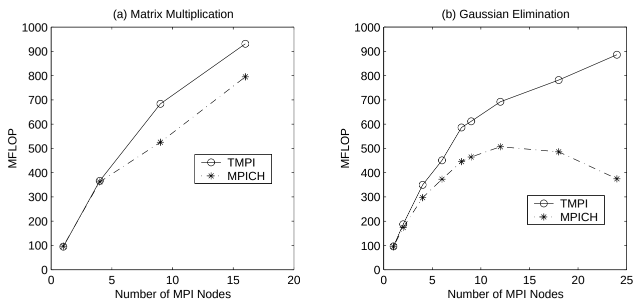

Figure 15: Macro-Benchmark Performance.

point-to-point and collective communication channels, adaptive buffering, and event-driven synchronization by taking advantage of multi-threading. Through micro- and macro-benchmark testing, our experiments show that TMPI can outperform MPICH substantially.

It should be noted that TMPI optimization is targeted at a class of C programs while MPICH is designed for general MPI programs. The experiments confirm that even in a cluster environment for which inter-node network latency is relatively high, exploiting thread-based MPI execution on each SMP can deliver substantial performance gains for global communication through fast and light-weight synchronization. Our experiments focus on a dedicated cluster and our future work is to study performance in a multiprogrammed environment for which thread-based synchronization may achieve more performance gain.

6. ACKNOWLEDGEMENTS

This work was supported in part by NSF ACIR-0082666, ACIR-0086061 and CCR-9702640. We would like to thank Lingkun Chu for his help on cluster administration. We would also like to thank Kai Shen and the anonymous referees for their valuable comments.

7. REFERENCES

[1] T. Anderson, D.Culler, D. Patterson, et al. A case for networks of workstations: NOW. *IEEE Micro*, 1995.

[2] Donald J. Becker, Thomas Sterling, Daniel Savarese, John E. Dorband, Udaya A. Ranawak, and Charles V. Packer. BEOWULF: a parallel workstation for scientific computation. In *Proceedings of International Conference on Parallel Processing*, 1995. More Beowulf papers are in http://www.beowulf.org.

[3] G. Burns, , V. Radiya, R. Daoud, and R. Machiraju. All About Trollius, 1990. Ohio Supercomputer Center.

[4] G. Burns, R. Daoud, and J. Vaigl. LAM: An Open Cluster Environment for MPI. Ohio Supercomputer Center.

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

[6] A. Ferrari and V. Sunderam. TPVM: distributed concurrent computing with lightweight processes. In *Proceedings of IEEE High Performance Distributed Computing*, pages 211–218, Washington, D.C., August 1995. IEEE.

[7] W. Gropp and E. Lusk. An Abstract Device Definition to Support the Implementation of a High-Level Point-to-Point Message Passing Interface. Technical Report MCS-P392-1193, Argonne National Laboratory, 1994.

[8] 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.

[9] P. Husbands and J. C. Hoe. MPI-StarT: Delivering Network Performance to Numerical Applications. In *Proceedings of ACM/IEEE SuperComputing '98*, New York, November 1998. ACM/IEEE.

[10] Thilo Kielmann, Rutger F. H. Hofman, Henri E. Bal, Aske Plaat, and Raoul A. F. Bhoedjang. MagPIe: MPI's collective communication operations for clustered wide area systems. In *Proceedings of ACM Symposium on Principles & Practice of Parallel Programming (PPoPP)*, pages 131–140, 1999.

[11] M. Lauria and A. Chien. MPI-FM: Higher Performance MPI on Workstation Clusters . *Journal of Parallel and Distributed Computing*, pages 4–18, January 1997.

[12] S. Lumetta, A. Mainwaring, and D. Culler. Multi-Protocol Active Messages on a Cluster of SMP's. In *Proceedings of ACM/IEEE SuperComputing '97*, New York, 1997. ACM/IEEE.

[13] L. MPI. Home page of lam — local area multicomputer. "http://www.lsc.nd.edu/lam/".

[14] MPI-Forum. MPI Forum, 1999. http://www.mpi-forum.org.

[15] S. Prakash and R. Bagrodia. MPI-SIM: using parallel simulation to evaluate MPI programs. In *Proceedings of Winter simulation*, pages 467–474, Washington, DC., December 1998.

[16] K. Shen, H. Tang, and T. Yang. Adaptive two-level thread Management for fast MPI execution on shared memory machines. In *Proceedings of ACM/IEEE SuperComputing '99*, New York, November 1999. ACM/IEEE. Available from www.cs.ucsb.edu/research/tmpi.

[17] S. Sistare, R. Vaart, and E. Loh. Optimization of MPI Collectives on Clusters of Large-Scale SMP's . In *Proceedings of ACM/IEEE SuperComputing '99*, New York, November 1999. ACM/IEEE.

[18] H. Tang, K. Shen, and T. Yang. Program Transformation and Runtime Support for Threaded MPI Execution on Shared Memory Machines. *ACM Transactions on Programming Languages and Systems*, 2000. An earlier version appeared in Proc. of ACM Symposium on Principles & Practice of Parallel Programming (PPoPP), 1999. Pages 107-118.

[19] F. Wong and D. Culler. Message passing interface on active messages. "http://now.cs.berkeley.edu/Fastcomm/MPI/".

[20] H. Zhou and A. Geist. LPVM: a step towards multithread PVM. *Concurrency - Practice and Experience*, 1997.

APPENDIX

A. A LIST OF MPI FUNCTIONS IMPLEMENTED IN TMPI

MPI_Init
MPI_Finalize
MPI_Comm_size
MPI_Comm_rank
MPI_Comm_compare
MPI_Barrier
MPI_Bcast
MPI_Reduce
MPI_Allreduce
MPI_Buffer_attach
MPI_Buffer_detach
MPI_Send
MPI_Isend
MPI_Send_init
MPI_Bsend
MPI_Ibsend
MPI_Bsend_init
MPI_Rsend
MPI_Irsend
MPI_Rsend_init
MPI_Ssend
MPI_Issend
MPI_Ssend_init

MPI_Recv
MPI_Irecv
MPI_Recv_init
MPI_Sendrecv
MPI_Sendrecv_replace
MPI_Startall
MPI_Testall
MPI_Testany
MPI_Testsome
MPI_Waitall
MPI_Waitany
MPI_Waitsome
MPI_Start
MPI_Wait
MPI_Cancel
MPI_Test
MPI_Request_free
MPI_Get_version
MPI_Get_processor_name
MPI_Wtime
MPI_Abort
MPI_Get_count