Tolerating Message Latency through the Early Release of Blocked Receives

Jian Ke[1], Martin Burtscher[1], and Evan Speight[2]

[1] Computer Systems Laboratory, School of Electrical & Computer Engineering,
Cornell University, Ithaca, NY 14853, USA
{jke, burtscher}@csl.cornell.edu

[2] Novel System Architectures, IBM Austin Research Lab,
Austin, TX 78758, USA
speight@us.ibm.com

Abstract. Large message latencies often lead to poor performance of parallel applications. In this paper, we investigate a latency-tolerating technique that immediately releases all blocking receives, even when the message has not yet (completely) arrived, and enforces execution correctness through page protection. This approach eliminates false message data dependencies on incoming messages and allows the computation to proceed as early as possible. We implement and evaluate our early-release technique in the context of an MPI runtime library. The results show that the execution speed of MPI applications improves by up to 60% when early release is enabled. Our approach also enables faster and easier parallel programming as it frees programmers from adopting more complex nonblocking receives and from tuning message sizes to explicitly reduce false message data dependencies.

1 Introduction

Clusters of workstations can provide low-cost parallel computing platforms that achieve reasonable performance on a wide range of applications reaching from databases to scientific algorithms. To enable parallel application portability between various cluster architectures, several message-passing libraries have been designed. The Message Passing Interface (MPI) standard [10] is perhaps the most widely used of these libraries. MPI provides a rich set of interfaces for operations such as point-to-point communication, collective communication, and synchronization.

Sending and receiving messages is the basic MPI communication mechanism. The simplest receive operation has the following syntax: *MPI_Recv(buf, count, dtype, source, tag, comm, status)*. It specifies that a message of *count* elements of data type *dtype* with a tag of (*tag*, *comm*) should be received from the *source* process and stored in the buffer *buf*. The *status* returns a success or error code as well as the *source* and *tag* of the received message if the receiver specifies a wildcard *source*/*tag*. The *MPI_Recv* call blocks until the message has been completely received. The MPI standard also defines a non-blocking receive operation, which basically splits *MPI_Recv* into two calls, *MPI_Irecv* and *MPI_Wait*. The *MPI_Irecv* call returns right away whether or not the message has been received, and the *MPI_Wait* call blocks until the entire message is present. This allows application writers to insert useful computation

between the *MPI_Irecv* and *MPI_Wait* calls to hide part of the message latency by overlapping the communication with necessary computation.

In both cases, the computation cannot proceed past the blocking call (*MPI_Recv* or *MPI_Wait*). In our library, we immediately release (unblock) all blocked calls (*MPI_Recv* and *MPI_Wait*) even when the corresponding message has not yet been completely received, and prevent the application from reading the unfinished part of the message data through page protection. *Our early-release technique automatically delays the blocking for as long as possible, i.e., until the message data is actually used by the application, and eliminates the false message data dependency implied by the blocking calls.* As such, it provides the following benefits:

- It allows the computation to continue on the partially received message data instead of waiting for the full message to complete, thus overlapping the communication with the computation.
- All blocking receives are automatically made non-blocking. The message blocking is delayed as much as possible, benefiting even nonblocking receives with sub-optimally placed *MPI_Wait* calls.
- Programmers no longer need to worry about when and how to use the non-blocking MPI calls, nor do they need to intentionally dissect a large message into multiple smaller messages. This reduces the development time of parallel applications. In addition, the resulting code is more intuitive and easier to understand and maintain, while at the same time providing or exceeding the performance of more complex code.

We implemented the early-release technique in our erMPI runtime library. Applications linked with our library instantly benefit from early release without any modification. erMPI currently supports the forty most commonly-used MPI functions, which is enough to cover the vast majority of MPI applications.

There has been much work on improving the performance of MPI runtime libraries. TMPI [12], TOMPI [1] and Tern [6] provide fast messaging between processes co-located on the same node via shared memory semantics that are hidden from the application writer. Tern [6] dynamically maps computation threads to processors according to custom thread migration policies to improve load balancing and to minimize inter-node communication for SMP clusters. Some implementations [9, 11] take advantage of user-level networks such as VIA [2] or InfiniBand [4] to drastically reduce the messaging overhead, thus reducing small-message latency. Other researchers have investigated ways to improve the performance of collective communication operations in MPI [5, 13]. Prior work by the authors has explored using message compression to increase the effective message bandwidth [7] and message prefetching to hide the communication time [8].

This paper is organized as follows. Section 2 introduces our erMPI library and describes the early-release implementation of blocked receives. Section 3 presents the experimental evaluation methodology. Section 4 discusses results obtained on two supercomputers. Section 5 presents conclusions and avenues for future work.

2 Implementation

2.1 The erMPI Library

We have implemented a commonly used subset of forty MPI functions in our erMPI library, covering most point-to-point communications, collective communications, and communicator creation APIs in the MPI specification [10]. The library is written in C and provides an interface for linking with FORTRAN applications. erMPI utilizes TCP as the underlying network protocol and creates one TCP connection between every two communicating MPI processes. Each process has one application thread as well as one message thread to handle sending to and receiving from all communication channels.

2.2 Early-Release Mechanism

The messaging thread creates an alias page block for each message receive buffer posted via an *MPI_Recv* or *MPI_Irecv* call and stores incoming message data via these alias pages. The application thread making the call to *MPI_Recv* or *MPI_Wait* never blocks when calling these routines, which is a slight departure from the spirit of these calls. However, if the message has not arrived or is only partially complete, the application thread protects the unfinished pages of the message receive buffer and immediately returns from the *MPI_Recv* or *MPI_Wait* call that would otherwise have blocked. Thus, computation can continue until the application thread touches a protected page, which causes an access exception, and the application is then blocked until the data for that page is available.

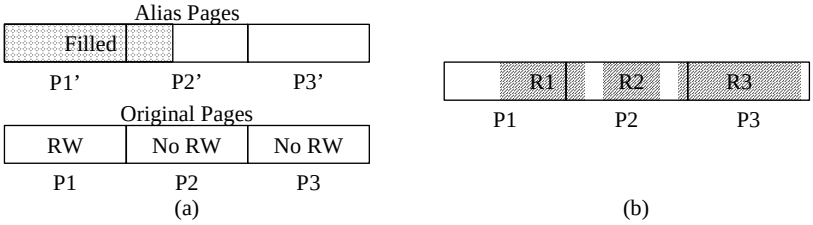


Fig. 1. Receive page examples

Figure 1 (a) shows an example of a receive buffer consisting of three pages. The virtual pages P_i and P_i' ($i \in \{1,2,3\}$) are mapped to the same physical page. The incoming message data is stored into the receive buffer through the alias pages P_i', which are created from the original buffer pages passed to *MPI_Recv* and are never protected. When the application thread calls *MPI_Recv* or *MPI_Wait* in this example, it will notice that P_1 is completely filled and therefore protects only P_2 and P_3, which will be granted *ReadWrite* access again by the messaging thread as soon as those pages are filled. The application thread returns from the *MPI_Recv* or *MPI_Wait* call without waiting for the completion of P_2 and P_3.

2.3 Implementation Issues

Shared Receive Pages. Figure 1 (b) depicts three outstanding receives, R_1, R_2 and R_3. All three receive buffers include part of page P_2. To handle such cases, we maintain a page protection count for shared receive pages to enforce the correct protection action. A page's protection count is incremented for each early-release protection. Note that a page only needs to actually be protected when the page protection count is increased from zero to one. If all three receives are released early, P_2's page protection count will be three. The count is decreased by one as soon as one of the receives completes the page. Once the count reaches zero, the page is unprotected.

There are at most two shared pages for each receive operation, one at the head of and the other at the tail of the receive buffer. For efficiency reasons, we log the page protection counts of all head and tail pages in a hash table.

Alias Page Creation. Most modern operating systems allow multiple virtual pages to be mapped to the same physical page and expose this function via system calls such as *mmap* in Unix and *MapViewOfFile* in Windows NT.

Creating an alias page is an expensive operation. To facilitate alias page reuse, we store the alias page description in a hash table. Each entry in the hash table records the starting addresses of both the original and the alias page blocks and the page block length. We hash the starting address of the original page block to index the hash table. A new alias page block is created if there is no hit or if the existing alias block is too small; otherwise a preexisting block is reused. Alias page blocks are allocated at a 16-page granularity. The page size is 4 kB in our system.

Send Operation. It is important that the protected pages be accessed only by the application thread running in user mode. If these pages are touched by a kernel or subsystem thread or in kernel mode, it may be impossible to catch and handle the access exceptions gracefully. This can happen when a send buffer shares a page with a receive buffer and the send buffer is passed to the operating system. To prevent this scenario, we also use alias page blocks for sends.

2.4 Portability and MPI Standard Relaxation

Even though we evaluate our early-release technique on Windows with TCP as the underlying network protocol, it can be similarly implemented on other systems, as long as the following requirements are met:

- The OS supports page protection calls and access violation handling.
- The network protocol can access the protected receive buffer. This is possible if the network subsystem has direct access to the physical pages or if alias pages can be used to interface with the communication protocol.
- The MPI library can be notified when a partial message arrives. This allows the protected pages to be unprotected as early as possible.

MPI_Recv returns the receive completion status in the *status* structure. It usually includes the matching send's *source* and *tag* and indicates whether the receive is a success. If a wild card *source* or *tag* is specified and the call is early released, the matching send's *source* or *tag* is typically not known. In such a case, we delay the

early release until this information is available. We always return a receive success in the *status* field and force the program to terminate should an error occur.

2.5 Other Issues

Message Unpacking. In our sample applications, messages are received into the destination buffers directly, allowing the computation to proceed past the receive operation and to work on the partially received message data. For applications that first receive messages into an intermediate buffer and then unpack the message data once they have been fully received, the early-released application thread would cause an access exception and halt the execution right away due to the message unpacking step, limiting the potential of overlapping useful computation with communication.

Since unpacking adds an extra copy operation and increases the messaging latency, it should be avoided whenever possible. More advanced scatter receive operations provide better alternatives for advanced programmers and parallelizing compilers. Another possible solution is to unpack the message as needed in the computation phase instead of unpacking the whole message right after the message receive.

Correctness. To guarantee execution correctness, an early-released application thread is not allowed to affect any other application thread before all early-released receives are at least partially completed. This means that new messages are not allowed to leave an MPI process if there exists unresolved early-released receives. Otherwise, a causality loop could be formed where an early-released application thread sends a message to another MPI process, which in turn sends a message that matches the early-released receive.

3 Evaluation Methods

3.1 Systems

We performed all measurements on the Velocity + (Vplus) and the Velocity II (V2) clusters at the Cornell Theory Center [3]. Both clusters run Microsoft Windows 2000 Advanced Server. The cluster configurations are listed below.

- Vplus consists of 64 dual-processor nodes with 733 MHz Intel Pentium III processors, 256 kB L2 cache per processor and 2 GB RAM per node. The network is 100Mbps Ethernet, interconnected by 3Com 3300 24-port switches.
- V2 consists of 128 dual-processor nodes with 2.4 GHz Intel Pentium 4 processors, 512 kB L2 cache per processor and 2 GB RAM per node. The network is Force10 Gigabit Ethernet interconnected by a Force10 E1200 switch.

3.2 Applications

We evaluate the performance of early release on three representative scientific applications: PES, N-body, and M3. In general, we see small performance improvements on benchmark applications due to the message unpacking effects.

PES is an iterative 2-D Poisson solver. Each process is assigned an equal number of contiguous rows. In each iteration, every process updates its assigned rows, sends the first and last row to its top and bottom neighbors, respectively, and receives from them two ghost rows that are needed for updating the first and last row in the next iteration. We fix the two corner elements (0,0), (N-1, N-1) to 1.0 and the other two corner elements (0, N-1), (N-1, 0) to 0.0 as boundary conditions.

N-Body simulates the movement of particles under pair-wise forces between them. All particles are evenly distributed among the available processes for the force computations and the position updates. After updating the states of all assigned particles, each process sends its updated particle information to all other processes for the force computation in the next time step.

M3 is a matrix-matrix-multiplication application. In each iteration, a master process generates a random matrix A_i (emulating a data collection process), distributes slices of the matrix to slave processes for computation, and then gathers the results from all slave processes. Each slave process stores a transposed transform matrix B, which is broadcast once from the master process to all slaves when the computation starts. Each slave process first receives matrix A_{ip}, which is part of matrix A_i, then computes matrix $C_{ip} = A_{ip}*B$ and sends C_{ip} to the master. Note that this parallelization scheme is by no means the most efficient algorithm for multiplying matrices.

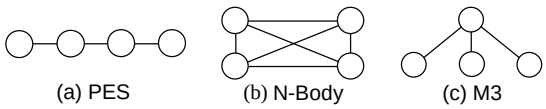


Fig. 2. Communication patterns

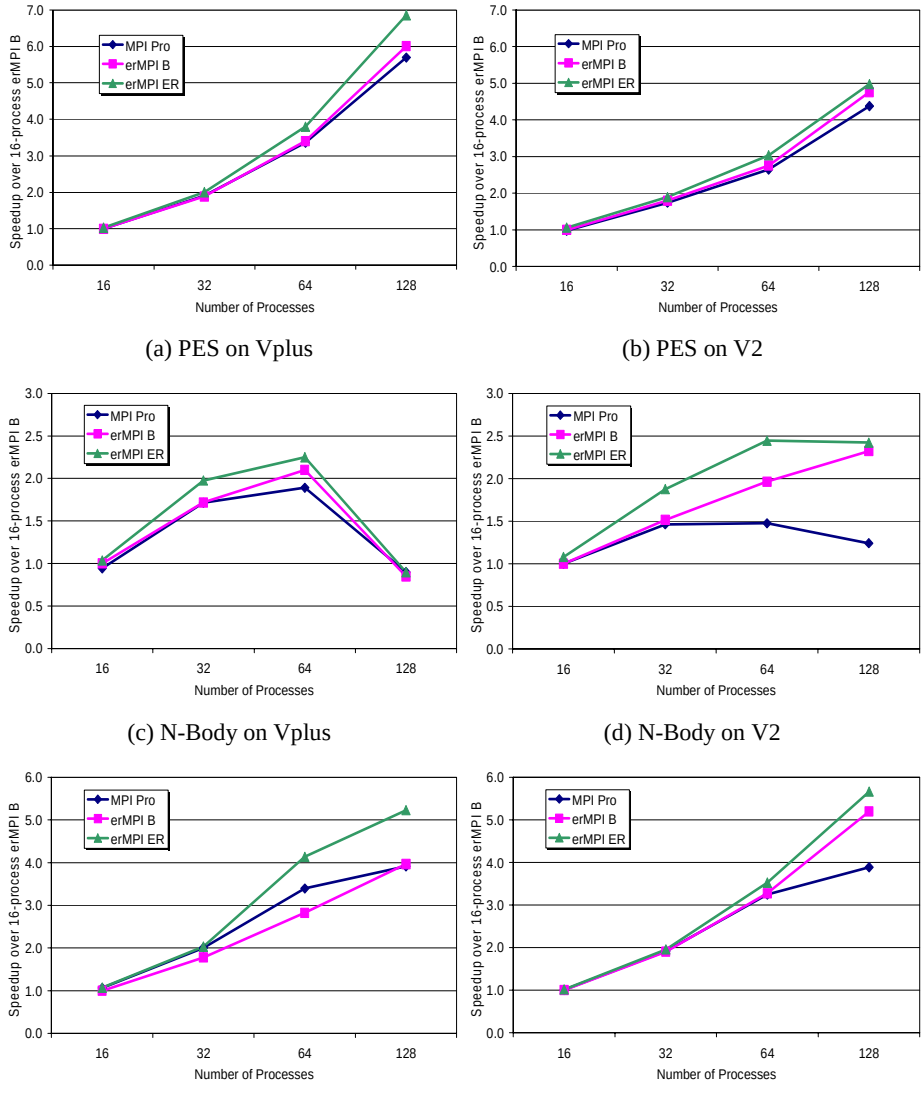
The communication patterns of these three applications for four-process runs are shown in Figure 2. The circles represent processes and the lines represent the communication between processes; each PES process only communicates with at most two neighboring processes; each N-Body process communicates with every other process; and each M3 slave process communicates with the master process. The messaging calls used are *MPI_Send*, *MPI_Irecv*, *MPI_Wait* and *MPI_Waitall*.

Table 1. Problem size and message size information

Program	Problem Size			Message Size (64-process run)		
	Size A	Size B	Size C	Size A	Size B	Size C
PES	5120X5120, 2000	10240X10240, 1000	20480X20480, 500	40 kB	80 kB	160 kB
N-Body	10240, 200	20480, 100	40960, 50	9 kB	18 kB	35 kB
M3	1024X1024, 400	2048X2048, 40	4096X4096, 20	128 kB	512 kB	2 MB

Table 1 lists the three problem sizes we used for each application. Size A is the smallest and Size C is the largest. In the “Problem Size” columns, the number before the comma is the matrix size for PES and M3 and the number of particles for N-Body; the number after the comma is the number of iterations or simulation time steps. We have adjusted the number of iterations so that the runtimes are reasonable. We run these applications with 16, 32, 64 and 128 processes and two processes per node. The

resulting message sizes for 64-process runs are shown in the "Message Size" columns. We obtained the runtimes with three MPI libraries. *MPI-Pro* is the default MPI library on both clusters. The *erMPI-B* is the baseline version of our erMPI library, in which the early release of receives is disabled. *erMPI-ER* is the same library but with early release turned on.



(a) PES on Vplus (b) PES on V2

(c) N-Body on Vplus (d) N-Body on V2

(e) M3 on Vplus (f) M3 on V2

Fig. 3. Scaling comparisons

4 Results

4.1 Scaling Comparison

Figure 3 (a - f) plots the scaling with problem size B of PES, N-Body and M3. Each application has two subgraphs, the left one shows Vplus and the right one V2 results. Each subgraph plots the execution speeds of the three MPI libraries against the number of processes used. The execution speeds are normalized to the 16-process run of the *erMPI* baseline library.

For a fixed problem size, the communication-to-computation ratio increases as the problem is partitioned among an increasing number of processors, which leads to worsening of parallel efficiency and scalability.

For PES (Figure 3 (a, b)), the erMPI early-release library scales the best among the three MPI libraries. The erMPI baseline library also scales better than MPI-Pro. PES scales better on the Vplus cluster than on V2. It appears that the higher processing power of V2 leads to a higher communication-to-computation time ratio and hence worse scalability.

N-Body is a communication intensive application. The communication dominates the computation as the number of processes increases. Figure 3 (c, d) shows that the MPI-Pro speedups start to saturate at around 32 processes and degrade at 128 processes. V2 has a higher network throughput and thus performs better than Vplus. Our erMPI library scales, with and without early release, to 64 processes.

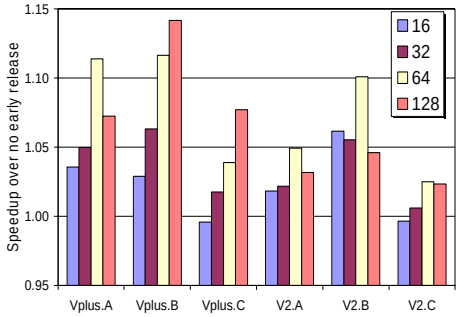
In Figure 3 (e, f), MPI-Pro performs better than the erMPI baseline for 32 and 64 processes on Vplus, but worse than the erMPI baseline for 128 processes on V2. The speedups in the remaining cases are roughly equal. erMPI with early release performs significantly better than both the baseline and MPI-Pro, especially for 64- and 128-process runs.

4.2 Early-Release Speedup

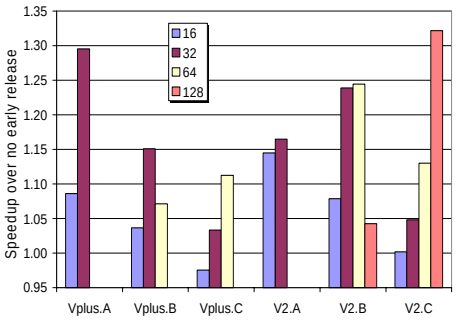
The scaling results from the previous section show that our baseline is comparable (superior in most cases) to MPI-Pro. In this section, we focus on the performance improvement of the early-release technique over the baseline. The speedups over the baseline erMPI library are plotted in Figure 4 (a - c) for the three applications. The labels along the x-axis indicate both the cluster and the problem size. Each group of bars shows results for runs with 16, 32, 64 and 128 processes. For the few non-scalable runs that take longer than the runs with fewer processes, the performance improvement over the baseline is meaningless and is left out of the figure.

We see that the speedups of a given problem size and cluster usually increase as the number of processes increases, as is the case for Vplus.B and Vplus.C with PES; for Vplus.C and V2.C with N-Body; and for Vplus.C, V2.A and V2.B with M3. The same trend holds in the other cases except for the last one or two bars. This is due to the increasing communication-to-computation ratio as the number of processes increases. Early release has little potential for performance improvement in cases where the communication time is minimal. On the other hand, when the communication-to-computation ratio becomes too large, the speedup decreases in some cases. There are

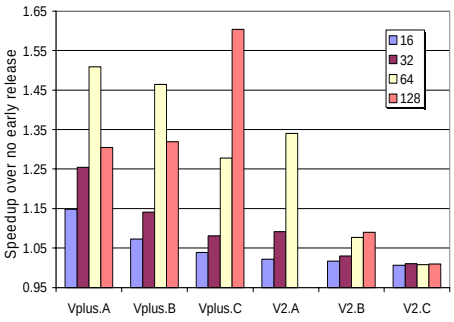
two reasons for this behavior. First, when receives are released early, application threads that proceed past the receive operations may send more data into the communication network, which worsens the network resource contention in communication-intensive cases. Second, as the communication-to-computation ratios increase past a certain level, the remaining computation is small enough that overlapping it with communication provides little performance benefit.



(a) PES



(b) N-Body



(c) M3

Fig. 4. Speedups due to early release

The same reasoning explains the speedup trends for varying problem sizes and clusters. As the problem size decreases, the communication-to-computation ratios increase and lead to higher early-release speedups. This behavior can be seen in the PES 16-process runs on Vplus, the N-Body 16- and 32-process runs on Vplus and 16-process runs on V2, and the M3 16-, 32- and 64-process runs on both Vplus and V2. The V2 cluster has a relative faster network (bandwidth) than Vplus and hence the potential for speedups due to early release is smaller. Indeed, V2 demonstrates a smaller performance improvement in most cases, except for N-Body sizes B and C, where the two negative effects start to impact the early-release speedups.

Four cases of PES show early-release speedups of over 10%. N-Body exhibits speedups of up to 32%, with four cases being over 20%. M3 reaches over 30% speedup in six cases, with a maximum speedup of 60%.

4.3 Early-Release Overhead and Benefit

The early-release overhead includes the creation of the alias page blocks and the page protection for unfinished messages. Since the alias page blocks are reused in our implementation, the overhead is amortized over multiple iterations and is negligible. Table 2 compares the page protection plus unprotection time on V2 with the raw transfer time over a 1 Gbps network. The cost of page protection is much smaller than the message communication latency. Most importantly, there is little penalty to the run time since the application thread would have been blocked waiting for the incoming message to complete anyway.

Table 2. Page protection overhead

Message Size	4 kB	16 kB	64 kB	256 kB
Protection Cost	3.0 µs	3.0 µs	3.3 µs	4.5 µs
1 Gbps Transfer Time	33 µs	131 µs	0.5 ms	2.1 ms

Parallel applications frequently consist of a loop with a communication phase and a computation phase. When a process receives multiple messages from multiple senders in the communication phase, often the computation following the communication need not access some of the received messages for a while. For example, the PES process receives two messages in the communication phase, one from the process "above" and the other from the process "below" it. The message data from the lower neighbor is accessed only at the end of computation phase, thus blocking for its completion at the end of the communication phase is not necessary. The same is true for N-Body as each process receives messages from multiple processes in the communication phase. Our early-release technique eliminates this false message data dependency and delays the blocking until the message data is indeed accessed. The reduced blocking time is most pronounced in the presence of load imbalance or processes running out of lock-step.

Each M3 slave process receives only one message in the communication phase, so the above effect does not appear. Nevertheless a similar false data dependency is eliminated by the early-release technique; the computation can start on the partially finished message data, maximally overlapping the communication with computation.

5 Conclusions and Future Work

In this paper, we present and evaluate a technique to release blocked message receives early. Our early-release approach automatically delays the blocking of message receives as long as possible to maximize the degree of overlapping of communication with computation, effectively hiding a portion of the message latency. The performance improvement depends on the communication-to-computation ratio and the extent of false message data dependencies of each application. Measurements with our erMPI library show an average early-release speedup of 11% on two supercomputing clusters for three applications with different communication patterns.

In future work, we plan to eliminate the message unpacking step for some benchmark applications and study the early-release performance on these highly tuned applications. Future research may also explore the usage of a finer early-release granularity to further improve the performance.

6 Acknowledgements

This work was supported in part by the National Science Foundation under Grant No. 0125987. This research was conducted using the resources of the Cornell Theory Center, which receives funding from Cornell University, New York State, federal agencies, foundations, and corporate partners.

References

1 E. D. Demaine, "A Threads-Only MPI Implementation for the Development of Parallel Programs," *Intl. Symp. on High Perf. Comp. Systems*, 7/1997, pp. 153-163.
2 D. Dunning, G. Regnier, G. McApline, D. Cameron, B. Shubert, F. Berry, A. Merritt, E. Gronke and C. Dodd, "The Virtual Interface Architecture," *IEEE Micro*, 3/1998, pp. 66-76.
3 http://www.tc.cornell.edu/
4 Infiniband Trade Association, *Infiniband Architecture Specification, Release 1.0*, Oct. 2000.
5 A. Karwande, X. Yuan and D. K. Lowenthal, "CC-MPI: A Compiled Communication Capable MPI Prototype for Ethernet Switched Clusters," *The Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 6/2003, pp. 95-106.
6 J. Ke, "Adapting parallel program execution in cluster computers through thread migration," *M.S. Thesis, Cornell University*, 2003.
7 J. Ke, M. Burtscher and E. Speight, "Runtime Compression of MPI Messages to Improve the Performance and Scalability of Parallel Applications," *Supercomputing*, 11/2004.
8 J. Ke, M. Burtscher and E. Speight, "Reducing Communication Time through Message Prefetching," *Intl. Conf. on Parallel and Distributed Processing Techniques and Applications*, 6/2005.
9 J. Liu, J. Wu, S. P. Kini, P. Wyckoff and D. K. Panda, "High Performance RDMA-Based MPI Implementation over InfiniBand," *Intl. Conf. on Supercomputing*, 6/2003, pp. 295-304.
10 MPI Forum, "MPI: A Message-Passing Interface Standard," *The Intl. J. of Supercomputer Applications and High Performance Computing*, 8(3/4):165-414, 1994.
11 E. Speight, H. Abdel-Shafi, and J. K. Bennett, "Realizing the Performance Potential of the Virtual Interface Architecture," *Intl. Conf. on Supercomputing*, 6/1999, pp. 184-192.
12 H. Tang and T. Yang, "Optimizing Threaded MPI Execution on SMP Clusters," *Intl. Conf. on Supercomputing*, 6/2001, pp. 381-392.
13 R. Thakur and W. Gropp, "Improving the Performance of Collective Operations in MPICH," *European PVM/MPI Users' Group Conference*, 9/2003, pp. 257-267.