Reducing Communication Time through Message Prefetching

Jian Ke and Martin Burtscher
Computer Systems Laboratory
School of Electrical & Computer Engineering
Cornell University, Ithaca, NY 14853
{jke, burtscher}@csl.cornell.edu

Evan Speight
Novel System Architectures
IBM Austin Research Lab
Austin, TX 78758
speight@us.ibm.com

Abstract – The latency of large messages often leads to poor performance of parallel applications. In this paper, we investigate a novel latency reduction technique where message receivers prefetch messages from senders before the matching sends are called. When the send is finally called, only the parts of the message that have changed since the prefetch need to be transmitted, resulting in a smaller message. Our message prefetching technique initiates communication while the sender is still in the computation phase and thus overlaps computation with communication to hide part of the message latency. We implement and evaluate our technique in the context of an MPI runtime library. The results show that the execution speed of five MPI applications improves by up to 24% when message prefetching is enabled.

Keywords: MPI, message prefetching, page protection, page version, latency reduction

1.0 Introduction

Utilizing clusters of workstations with high-speed interconnection networks for parallel computation can deliver supercomputing performance on a broad range of applications at a fraction of the cost of specialized hardware.

To enable portability between the wide variety of cluster architectures, while at the same time taking advantage of the specifics of the underlying network protocol and hardware, several message-passing libraries have been designed. The Message Passing Interface (MPI) standard [6] is one of the most widely used of these libraries. MPI provides a rich set of operations for point-to-point communication, collective communication, and synchronization operations.

The basic MPI receive operation has the following syntax: *MPI_Recv (buf, count, dtype, source, tag, comm, status),* where *buf* specifies the receive buffer, *count* is the number of elements to be received, *dtype* is the data type, *source* specifies the message sender and (*tag, comm*) are used to match a send operation with a corresponding receive operation. 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 a matching message has completely arrived. The MPI standard also includes a non-blocking receive operation, *MPI_Irecv*, which returns immediately whether or not a message has been received. Applications later call *MPI_Wait* to wait for message completion. This allows useful computation to be inserted between the *MPI_Irecv* and *MPI_Wait* calls, providing the opportunity to hide part of the message latency by overlapping the communication with necessary computation.

We introduce a mechanism that allows message contents to be prefetched from sending processes by receive operations, even before the send operation has been posted. Figure 1 visually compares our message prefetching approach with a conventional receive, where the vertical pattern represents the computation phase, the horizontal pattern represents the communication phase and the grid pattern depicts overlapped communication and computation phases. When the receiver blocks either inside *MPI_Recv* or *MPI_Wait*, our message prefetching implementation predicts the data buffer that will be sent by the next send operation, and prefetches a portion of the data from the sender even before a matching *MPI_Send* is called. When the matching *MPI_Send* is finally called, a shorter message can be sent since part of the message data has already been delivered to the receiver. Hence, our message prefetching technique has the potential to hide some of the message latency and to improve the performance of communication-intensive parallel applications.

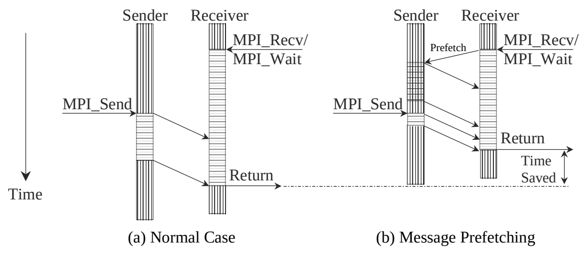

Figure 1. Messaging step comparison

We have implemented this message prefetching technique in our pfMPI runtime library. Applications linked with this library may see performance benefits without recompilation. pfMPI currently supports forty commonly-used MPI functions, enough to cover the vast majority of MPI applications.

Previous work has investigated various techniques to improve the performance of MPI libraries. TMPI [8] and TOMPI [1] deliver fast messaging between processes co-located on the same node via shared memory semantics that are hidden from the application programmer. Other MPI implementations [5, 7] exploit user-level networks such as VIA [2] or InfiniBand [4] to drastically decrease the overhead of sending messages, thus reducing small-message latency. There has also been research on improving the efficiency of collective communication operations [9, 10]. Prior work by the authors employs message compression to increase the effective network bandwidth and thus improve the overall application performance and scalability [11].

Prefetching has previously been studied for reducing data access latency in memory hierarchies [14, 16], web pages [15], and distributed shared memory systems [17, 18]. To our knowledge, this paper is the first to present a viable runtime prefetching technique for message-passing environments.

The rest of this paper is organized as follows. Section 2 introduces our pfMPI library and describes the implementation of message prefetching. Section 3 presents the experimental evaluation methodology. Section 4 discusses results obtained with our library on a supercomputer cluster at the Cornell Theory Center. Section 5 presents conclusions and avenues for future work.

2.0 Implementation

2.1 The pfMPI Library

We have implemented a commonly used subset of forty MPI functions in our pfMPI library, covering most point-to-point communications, collective communications, and communicator creation APIs in the MPI specification [6]. The library is written in C and provides an interface for linking with FORTRAN applications. pfMPI utilizes TCP as the underlying network protocol and creates one TCP connection between every two communicating MPI processes. Each process creates a message thread to handle sending to and receiving from all communication channels, as well as handling all prefetching requests and replies.

2.2 Message Prefetching Implementation

When a receiver blocks on an *MPI_Recv* call, our library attempts to prefetch some message data before the matching send is called so that less data needs to be transferred once the matching *MPI_Send* is finally executed. If partial message data has already started to arrive, no prefetch is requested.

The message communication time can be modeled as $l_0 + l_1*n$, where l_0 is the message startup time, l_1 is the per byte transfer time and n is the message length. Prefetching reduces the l_1*n term. For small messages, l_0 dominates and prefetching does not help much. Therefore, a prefetch is requested only if the message size is larger than a predefined threshold (4 kB in our implementation).

A prefetch request consists of a virtual address and size record that predicts the matching send buffer in the sender process. The prediction is learned from previous sends for each receive. To facilitate the prediction, the send buffer's virtual address is included in every normal MPI message. When a receive completes, the information of the receive and the matching send is logged in a hash table (the prefetch prediction table) so that a send buffer prediction can be made later for prefetch requests. The hash table key entry is <*receiveBufferAddress*, *receiveTag*, *receiveSource*> and the value entry is <*sendBufferAddress*, *sendSize*, *count*>, where '*count*' is the number of times the same *sendBufferAddress* and *sendSize* have been observed. We make a send buffer prediction only when the count is larger than two to suppress unreliable predictions.

When a message thread receives a prefetch request, it does not simply return the specified buffer data in full. Rather, it first tries to guess how much data is already complete and sends only the completed portion. The current implementation utilizes virtual page protection to estimate the amount of completed data. We use a hash table to maintain a page version number for every page whose protection was ever changed.

When a page's protection is changed to *ReadOnly*, a subsequent write to that page triggers a page access exception and the exception handler increases the page version number and changes the page protection back to *ReadWrite*. At the end of each normal *MPI_Send*, the middle and the last page of the send buffer are set to *ReadOnly* and both pages' versions are recorded in the *lastSendVersion* hash table. If the current page version of both the middle and the last page of the send buffer are different from the versions at the last send (as is determined by looking up the *lastSendVersion* hash table), the entire send buffer is pre-sent. If only the middle page version has changed, only data up to that page is pre-sent. Otherwise, nothing is pre-sent. The *lastSendVersion* hash table key is <*sendBufferAddress*, *sendBufferSize*, *sendDestination*>, and the value entry is <*middlePageVersion*, *lastPageVersion*>.

For *MPI_Send* to work correctly without having to resend all of the pre-sent data, we need to know which, if any, of the pre-sent pages have been modified by the application since the time of the pre-send. We do this by keeping an array of version numbers of the pre-sent pages. Pages that have not been modified since the pre-send are left out of the "normal" *MPI_Send* message, thus making it shorter and reducing the transfer time.

The prefetched data is directly written into the receive buffer that originated the prefetch unless some of the requested message has already arrived (late prefetch). When a prefetch is late, the sender will usually notice this event (the current page version is equal to the *lastSendVersion*) and drop the prefetch request unless the sender fills the send buffer again (e.g., in the next iteration). In this scenario, the next *MPI_Send* could use the pre-sent data requested for a previous send. Therefore, the prefetched data is buffered at the receiver if the prefetch is late. In all cases, the prefetched data is kept at the receiver until an *MPI_Send* message that exploits the pre-sent data arrives or a new prefetch request reaches the sender, guaranteeing that no further *MPI_Send* will use the buffered prefetch data. The pre-sent data is used by only one *MPI_Send* to match the one-time use guarantee from the receiver.

In our sample applications, messages are sent from the source buffers directly to facilitate message prefetching. For applications that first pack the message data into an intermediate buffer before sending, the prefetch requests will most likely find the buffer is not filled with new data and hence the prefetch request will be dropped, limiting the potential of starting sends early. A possible solution is to pack the message as it is generated in the computation phase instead of packing the entire message right before the message send.

3.0 Evaluation Methods

3.1 System

We performed all measurements on the *Velocity+* cluster at the Cornell Theory Center [3]. This cluster 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 operating system is Microsoft Windows

2000 Advanced Server. The network is 100Mbps Ethernet, interconnected by 3Com 3300 24-port switches.

3.2 Applications

We evaluate the performance of message prefetching on five representative scientific applications: PES, M3, N-body, FT, and IS.

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.

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. The slave processes store 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.

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 next time step.

FT and IS are the Fourier Transform and Integer Sort programs from the NAS NPB benchmark [12, 13]. The remaining NAS NPB benchmarks are left out due to the message packing effect discussed at the end of Section 2.2.

The communication patterns of the five 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 M3 slave process communicates with the master process; and each N-Body, FT and IS process communicates with every other process. The pseudo code for PES, M3 and N-Body is given in Appendix I.

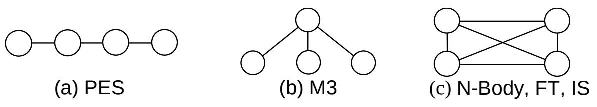

Figure 2. Communication patterns

Table 1 lists the problem size for each application. The number before the comma is the matrix size for PES, M3 and FT and the number of particles or integers for N-Body and IS; the number after the comma is the number of iterations or simulation time steps.

Table 1. Problem sizes

Program	Problem Size
PES	5120 X 5120, 2000
M3	1024 X 1024, 400
N-Body	10240, 200
FT	512 X 512 X 512, 20
IS	134217728, 10

4.0 Results

We run the five applications with 8, 16, 32, and 64 processes and one process per node. The runtimes are obtained with two MPI libraries, the baseline version of our pfMPI library in which message prefetching is disabled and the same library but with message prefetching turned on. The average runtimes are listed in Table 2. FT and IS scale only to 16 processes, so the runtimes for 64 processes were not collected.

The speedups over the baseline library are plotted in Figure 3 for the five applications. Each group of bars shows results for runs with 8, 16, 32, and 64 processes. We see that the speedups usually increase as the number of processes increases. This is due to the increasing communication-to-computation ratio as the number of processes increases, i.e., the same percentage of communication time reduction corresponds to a larger percentage of the runtime reduction.

Table 2. Runtime in seconds

Appl.	MPI Lib.	8	16	32	64
PES	Baseline	737	387	207.5	117.0
	Prefetch	735	385	204.1	114.0
M3	Baseline	1491	1142	793	738
	Prefetch	1339	939	675	593
N-Body	Baseline	944	551	483	336
	Prefetch	950	554	447	312
FT	Baseline	2451	1378	1475	--
	Prefetch	2382	1296	1305	--
IS	Baseline	130.7	91.0	143.9	--
	Prefetch	133.0	81.2	127.0	--

The PES process communicates with only two neighbors and its communication-to-computation ratio is the lowest of all applications. This explains why its speedup is relatively small.

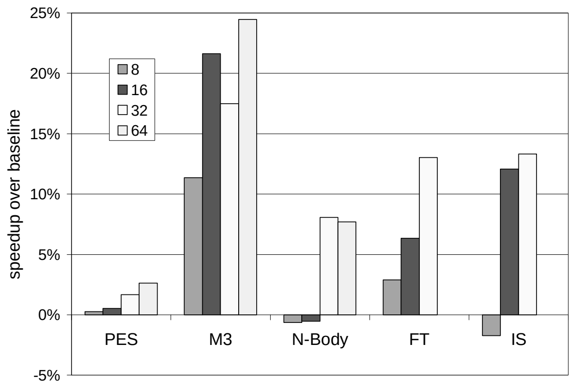

Figure 3. Speedups due to message prefetching

M3 has a communication-to-computation ratio that is larger than PES' but smaller than that of the other three applications. When the first slave process finishes the current iteration, it sends the result to the master process (the *MPI_Gather* call in the pseudo code), and then proceeds to wait for the next iteration (the *MPI_Scatter* call in the pseudo code). This wait triggers a message-prefetching request to the master process whose prefetch request handler pre-sends the data while the application thread waits to gather the results from all the slave processes.

N-Body, FT and IS have a message count per communication phase that is proportional to the square of the number of total processes, i.e., everyone sends and receives from every other process. Thus, the message prefetching interactions are much more complex. Overall, message prefetching delivers a runtime improvement of 6% to 12% in most cases. The slowdowns due to message prefetching in smaller runs are 0.5% for N-Body's 8- and 16-process runs and 1.7% for IS' 8-process run, which is much smaller than the performance improvement of 8.1% for N-Body's 32-process run and 12.1% for IS' 16-process run.

5.0 Conclusions and Future Work

In this paper, we present and evaluate a message prefetching technique for message-passing systems. Our MPI library starts the communication early, i.e., before *MPI_Send* is called, thus overlapping the computation with useful communication to hide some of the message latency. The performance improvement depends on the communication-to-computation ratio, the load balancing, and the communication pattern of the application. Measurements with our library show speedups between 2.6% and 24% on five applications running on a cluster with 64 nodes.

In future work, we plan to add heuristics to stop sending prefetches for receives that tend to prefetch late or where the pre-sent send buffer is written by the sender again before the *MPI_Send*.

6.0 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.

7.0 References

[1] E. D. Demaine, “A Threads-Only MPI Implementation for the Development of Parallel Programs,” *International Symposium on High Performance Computing Systems*, July 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*, March/April 1998, pp. 66-76.

[3] http://www.tc.cornell.edu/

[4] Infiniband Trade Association, *Infiniband Architecture Specification, Release 1.0*, October 2000.

[5] J. Liu, J. Wu, S. P. Kini, P. Wyckoff, and D. K. Panda, “High Performance RDMA-Based MPI Implementation over InfiniBand,” *International Conference on Supercomputing*, June 2003, pp. 295-304.

[6] MPI Forum, “MPI: A Message-Passing Interface Standard,” *The International Journal of Supercomputer Applications and High Performance Computing*, 8(3/4):165-414, 1994.

[7] E. Speight, H. Abdel-Shafi, and J. K. Bennett, “Realizing the Performance Potential of the Virtual Interface Architecture,” *International Conference on Supercomputing*, June 1999, pp. 184-192.

[8] H. Tang and T. Yang, “Optimizing Threaded MPI Execution on SMP Clusters,” *International Conference on Supercomputing*, June 2001, pp. 381-392.

[9] R. Thakur and W. Gropp, “Improving the Performance of Collective Operations in MPICH,” *European PVM/MPI Users' Group Conference*, September 2003, pp. 257-267.

[10] A. Karwande, X. Yuan and D. K. Lowenthal, “CC-MPI: A Compiled Communication Capable MPI Prototype for Ethernet Switched Clusters,” *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, June 2003, pp. 95-106.

[11] J. Ke, M. Burtscher, and E. Speight, “Runtime Compression of MPI Messages to Improve the Performance and Scalability of Parallel Applications,” *SC2004 High-Performance Computing, Networking and Storage Conference*, November 2004.

[12] D. Bailey, T. Harris, W. Saphir, R. v.d. Wijngaart, A. Woo, and M. Yarrow, “The NAS Parallel Benchmarks 2.0,” *Technical Report NAS-95-020, NASA Ames Research Center*, 1995.

[13] T. Tabe and Q. F. Stout, “The use of the MPI communication library in the NAS Parallel Benchmark,” *Technical Report CSE-TR-386-99, Department of Computer Science, University of Michigan*, 1999.

[14] A. J. Smith, “Sequential Program Prefetching in Memory Hierarchies”, *IEEE Computer*, December 1978, pp. 7-21.

[15] V. N. Padmanabhan and J. C. Mogul, “Using Predictive Prefetching to Improve World Wide Web Latency”, *ACM SIGCOMM Computer Communication Review*, July 1996, pp. 22-36.

[16] T. C. Mowry, “Tolerating Latency in Multiprocessors through Compiler-Inserted Prefetching”, *ACM Transactions on Computer Systems*, Vol. 16, No. 1, February 1998, pp. 55-92.

[17] R. Bianchini, R. Pinto, and C. L. Amorim, “Data Prefetching for Software DSMs”, *International Conference on Supercomputing*, July 1998, pp. 385-392.

[18] E. Speight and M. Burtscher, “Delphi: Prediction-Based Page Prefetching to Improve the Performance of Shared Virtual Memory Systems”, *The International Conference on Parallel and Distributed Processing Techniques and Applications*, June 2002, pp. 49-55.

8.0 Appendix I

PES Pseudo Code:

```
MPI_Comm_rank (MPI_COMM_WORLD, & myRank);
MPI_Comm_size (MPI_COMM_WORLD, & numProcesses);

for (int i = 0; i < numIterations; i ++)
{

    // post nonblocking receives
    if (myRank > 0)
       request1 = MPI_Irecv (source = myRank - 1);
    if (myRank < numProcesses –1)
       request2 = MPI_Irecv (source = myRank + 1);

    // send to two neighbor processes
    if (myRank > 0)
       MPI_Send (dest = myRank – 1);
    if (myRank < numProcesses – 1)
       MPI_Send (dest = myRank + 1);

    // wait for the receive completion
    if (myRank > 0) MPI_Wait (request1);
    if (myRank < numProcesses – 1) MPI_Wait (request2);

    Compute ();
 }
```

M3 Pseudo Code:

```
for (int i = 0; i < numIterations; i ++)
{
    MPI_Scatter (); // master distributes work to slaves

    if (myRank == root)
       FillMatrixAWithNewData (); // master
    else
       Compute (); // slaves

    // master collects results from slaves
    MPI_Gather ();
    if (myRank == root) WriteResults ();
}
```

N-Body Pseudo Code:

```
for (int i = 0; i < numIterations; i ++)
{
    MPI_Allgather (); // exchange local particles
    UpdateLocalParticles ();
}
```