MPI-StarT: Delivering Network Performance to Numerical Applications

Parry Husbands
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
tel: 617-253-8862
fax: 617-253-6652
`parry@lcs.mit.edu`
`http://theory.lcs.mit.edu/~parry`

James C. Hoe
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
tel: 617-253-8862
fax: 617-253-6652
`jhoe@lcs.mit.edu`
`http://www.csg.lcs.mit.edu/~jhoe`

Abstract:

We describe an MPI implementation for a cluster of SMPs interconnected by a high-performance interconnect. This work is a collaboration between a numerical applications programmer and a cluster interconnect architect. The collaboration started with the modest goal of satisfying the communication needs of a specific numerical application, MITMatlab. However, by supporting the MPI standard MPI-StarT readily extends support to a host of applications. MPI-StarT is derived from MPICH by developing a custom implementation of the Channel Interface. Some changes in MPICH's ADI and Protocol Layers are also necessary for correct and optimal operation.

MPI-StarT relies on the host SMPs' shared memory mechanism for intra-SMP communication. Inter-SMP communication is supported through StarT-X. The StarT-X NIU allows a cluster of PCI-equipped host platforms to communicate over the Arctic Switch Fabric. Currently, StarT-X is utilized by a cluster of SUN E5000 SMPs as well as a cluster of Intel Pentium-II workstations. On a SUN E5000 with StarT-X, a processor can send and receive a 64-byte message in less than 0.4 and 3.5 usec respectively and incur less than 5.6 usec user-to-user one-way latency. StarT-X's remote memory-to-memory DMA mechanism can transfer large data

Permission to make digital/hard copy of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SC '98, Orlando, FL
(c) 1998 IEEE 0-89792-984-X/98/0011 $3.50

blocks at 60 MByte/sec between SUN E5000s.

This paper outlines our effort to preserve and deliver this level of communication performance through MPI-StarT to user applications. We have studied the requirements of MITMatlab and the capabilities of StarT-X and have formulated an implementation strategy for the Channel Interface. In this paper, we discuss some performance and correctness issues and their resolutions in MPI-StarT. The correctness issues range from the handling of arbitrarily large message sizes to deadlock-free support of nonblocking MPI operations. Performance optimizations include a shared-memory-based transport mechanism for intra-SMP communication and a broadcast mechanism that is aware of the performance difference between intra-SMP and the slower inter-SMP communication.

We characterize the performance of MPI-StarT on a cluster of SUN E5000s. On SUN E5000s, MPI processes within the same SMP can communicate at over 150 MByte/sec using shared memory. When communicating between SMPs over StarT-X, MPI-StarT has a peak bandwidth of 56 MByte/sec. While fine-tuning of MPI-StarT is ongoing, we demonstrate that MPI-StarT is effective in enabling the speedup of MITMatlab on a cluster of SMPs by reporting on the performance of some representative numerical operations.

Keywords:
MPI, MPICH, MITMatlab, StarT-X, performance, clustering, SMP

1. Introduction

Developing a scientific/numerical application is a daunting task that requires specialized expertise in its respective field. Without an appropriate parallel programming environment, the effort to parallelize such an application could become equally demanding. Fortunately, MPI's[9] simplifying high-level abstraction partially shelters computational scientists from the nuisance of low-level execution and communication management. MPI's standardized interface also allows users to recoup the one-time programming efforts on a variety of platforms over several generations of hardware improvements. However, whether these advantages ultimately materialize still largely depends on whether an MPI implementation can meet the communication requirements of the applications.

The performance of an MPI implementation is clearly bounded by the performance of the underlying communication substrate. However, a high-performance substrate does not automatically guarantee a high performance implementation. In this paper, we describe our experience in developing MPI-StarT, an MPI implementation for a cluster of SMPs interconnected by the StarT-X cluster interconnect. StarT-X allows a cluster of PCI-equipped host platforms to communicate with an order-of-magnitude better performance than a conventional local area network. MPI-StarT's implementation is centered around preserving and delivering StarT-X's communication performance to user applications. MPI-StarT represents a collaboration between a numerical applications programmer and StarT-X's architect. The collaboration started with the modest goal to satisfy the communication needs of MITMatlab[5]. However, by supporting the MPI standard, MPI-StarT has been successful in extending support to other MPI applications.

In the next section, we begin our discussion of MPI-StarT by presenting StarT-X, the communication substrate. Section 3 presents an overview of MPI-StarT's implementation. Section 4 then highlights the specific correctness and performance issues we had to resolve in MPI-StarT. Sections 5 and 6 report MPI-StarT's benchmarked performance as well as its operational performance in MITMatlab. In Section 7, we briefly survey the related work before concluding in Section 8.

2. StarT-X Mechanisms and Performance

The StarT-X cluster interconnect[4] supports user-level message passing over a cluster of PCI-equipped host platforms. Salient features of the StarT-X cluster interconnect are:

- Three message-passing mechanisms
- Two message priorities
- Options for FIFO or non-FIFO ordered message delivery

We present StarT-X's PIO (Programmed I/O) and RDMA (Remote Direct Memory Access) operations, the two most relevant mechanisms in the implementation of MPI-StarT. For fine-grained message-passing applications, StarT-X provides a user-level PIO interface optimized to reduce user-to-user latency. The PIO mode implements a simple FIFO-based network abstraction similar to the CM-5's data network interface[11]. The user communicates by exchanging messages which each contain two 32-bit header words followed by a variable size payload of between 2 and 22 32-bit words. StarT-X provides two pairs of transmit and receive queues to handle high and low-priority messages separately. High priority message traffic can block the progress of low-priority messages, but not vice versa. Table 1 summarizes the PIO message passing performance between two SUN E5000s. Rows (a) and (b) report the default performance when sending and receiving 16-byte and 64-byte messages using 8-byte loads and stores. However, row (c) shows that a 64-byte message can be transferred at nearly the same cost as a 16-byte message if the UltraSPARC's ldda and stda instructions (64-byte loads and stores) are used for burst PIO operations.

	PIO Instructions	Message Size	Send Overhead	Send Bandwidth	Receive Overhead	Receive Bandwidth	Round Trip Latency
unit			(usec)	(MByte/sec)	(usec)	(MByte/sec)	(usec)
(a)	ldd/std	16	0.6	26.6	3.0	5.3	5.5
(b)	ldd/std	64	1.4	45.7	11.3	5.7	14.6
(c)	ldda/stda	64	0.4	160	3.5	18.3	5.6

Table 1: Performance Characteristics of PIO Message Passing: (a) 16-byte Messages (b) 64-byte Messages (c) 64-byte Messages using stda/ldda

In StarT-X, RDMA provides maximum bandwidth for large block transfers in coarse-grained applications. The StarT-X hardware packetizes and transfers large memory blocks between two hosts without the CPU’s assistance. The raw transfer rate between two SUN E5000 SMPs approaches 60 MByte/sec for data blocks larger than 16 KByte. However, three caveats affect the actual performance. First, the source and destination memory region for RDMA have to be specially allocated from a finite region of pinned physical memory managed by StarT-X’s device driver. Copying is required when the desired transfer does not start from or end in a pre-allocated RDMA buffer or if the transfer size is larger the maximum RDMA buffer size. Secondly, the RDMA mechanism is connection-based where the sender must use PIO message passing to arrange for exclusive right to transfer to a receiver. Lastly, RDMA operations conflict with high-priority PIO-mode message passing. As we describe below, these features of the interface significantly affect our design of MPI-StarT.

3. MPI-StarT Overview

For our MPI implementation, we decided to use the Channel Interface to MPICH[2]. Our decision was primarily influenced by the fact that, with the Channel Interface and MPICH, it is possible to build a complete, portable version of MPI by implementing only a few low-level messaging primitives. Table 2 shows a simplified view of the organization of MPICH. The Channel Interface is only responsible for sending and receiving arrays of bytes. The ADI and Protocol Layers are charged with translating MPI’s operations into these basic functions. In addition to providing an implementation of the Channel Interface, we also made some changes to the ADI and Protocol Layers (discussed in Section 4) to ensure correct and efficient operation.

User Calls (e.g. MPI_Send)
ADI
Protocol Layer
Channel Interface

Table 2: MPICH Organization

The Channel Interface makes a distinction between short and long messages. For short messages (called Control Messages), we used PIO messages, while RDMA transfers were employed for long messages. This was motivated entirely by the fact that StarT-X's PIO mode incurs less overhead and consequently is faster than RDMA for short message sizes. Currently, we cross over to RDMA for messages larger than 1 KByte.

Because our primary target platforms all consisted of SMPs, we included a shared memory transfer facility in the Channel layer for MPI processes that run on the same SMP. This is important for performance as using the network to send messages locally not only wastes a precious resource, but is considerably slower. MPI-StarT's inter and intra-SMP bandwidth between two processes differ by over factor of two. The difference is further magnified when inter-SMP StarT-X bandwidth must be shared by multiple simultaneous transfers. To support SMPs, we also incorporated locking mechanisms that ensure exclusive access to the StarT-X NIU when multiple processes are running on an SMP.

4. MPI-StarT Implementation Issues

Because RDMA transfers can only take place from special system buffers, one memory-to-memory copy is needed for each operation (both send and receive). In our implementation, we overlap copying to/from the system buffers with the RDMA operation so we can hide much of the copying overhead between user and system space. Any additional copies would only hurt performance, most notably on systems with lower main memory bandwidths.

When using StarT-X's RDMA mode, the processes at both ends of the transfer have to synchronize. In addition, if a process commits to an RDMA receive, its processor is utilized (with copying to user space) until the entire message is received and the RDMA cannot be used for receiving another message. This leads to potential deadlocks, particularly with MPI programs that use nonblocking operations. This was solved by using the Rendezvous Protocol (one of many provided implementations of the Protocol Layer) and modifying it slightly with simple tie-breaking rules that enable pairs of processes to exchange messages. With the Rendezvous Protocol control messages are exchanged before the large transfer is attempted and so both sender and receiver can synchronize. In addition, the Channel Interface receive is always called with a user space receive buffer, which removes the need for buffering large messages in the implementation.

It was also necessary to include support for sending and receiving arbitrarily large messages. With a 16-MByte system buffer for RDMA messages, we need to divide a large transfer into a series of 16-MByte transfers. Again the use of the Rendezvous Protocol removed the need for extra buffers and copying.

Many applications, (MITMatlab, in particular), make extensive use of collective operations (such as MPI_Bcast). In the current version of MPICH, when MPI_Bcast is called, a broadcast tree is generated assuming the bandwidths between all pairs of nodes are identical. With our hardware, it is better to minimize the number of times the network is used. We therefore implemented a two-stage broadcast where the data is first sent out to each SMP and then distributed within the SMPs. This resulted in a large improvement in broadcast's performance. This optimization can benefit the other collective operations as well. With the proliferation of SMP clusters we believe multi-protocol MPI implementations will be the norm. To address this issue in general, it would be useful if the higher level MPI operations (broadcast, for example) could dynamically create these data distribution trees based on knowledge of the relative performance of the point-to-point links.

Another issue that is tied to the multi-protocol nature of MPI-StarT is the choice of crossover size from control to large messages. For network messages, we switch over from PIO to RDMA at 1 Kbyte. However this crossover point is not optimal for the two shared memory protocols that we use. We therefore changed the Channel layer slightly to allow for different crossovers based on the locations of the communicating processes.

5. MPI-StarT Performance

We have characterized the performance of MPI-StarT on a cluster of SUN E5000 8-processor SMPs. Our benchmark measures the time to ping-pong S-bytes of data 100 times between two MPI processes. Graphs (a) and (b) in Figure 1 plot the observed bandwidth. Graph (a) shows the intra-SMP MPI bandwidth over the shared memory bus of an otherwise unloaded SUN E5000. Graph (b) reports the MPI bandwidth between two SUN E5000s. In both inter and intra-SMP communication, MPI-StarT approaches the peak hardware performance for large transfers. For small messages we have a latency of 18 microseconds for shared memory messages and 32 microseconds for network messages. Table 3 shows the improvement in broadcast performance due to our optimization. It measures the total time for a round of 1 MByte broadcasts (every process broadcasts a 1 MByte message in turn).

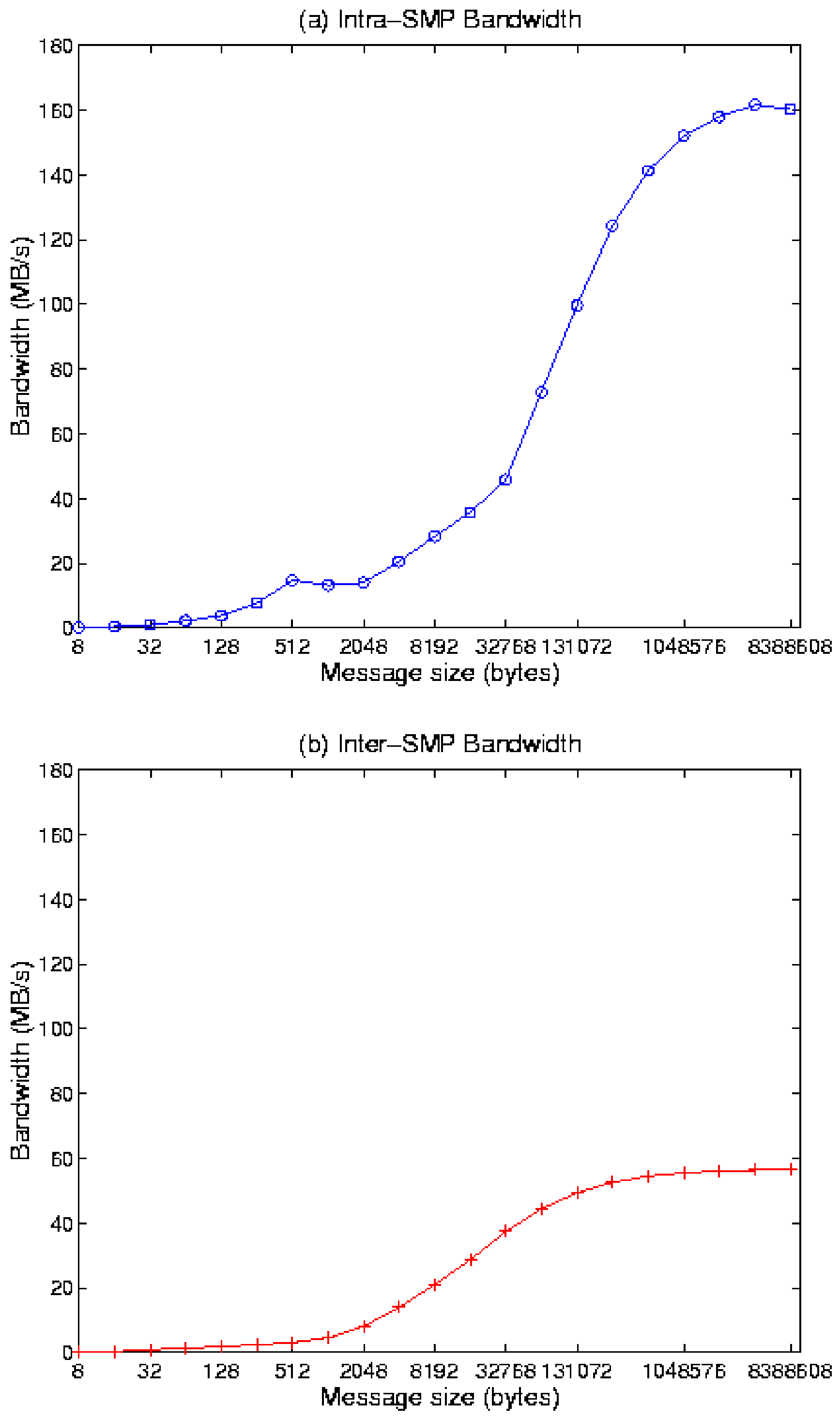


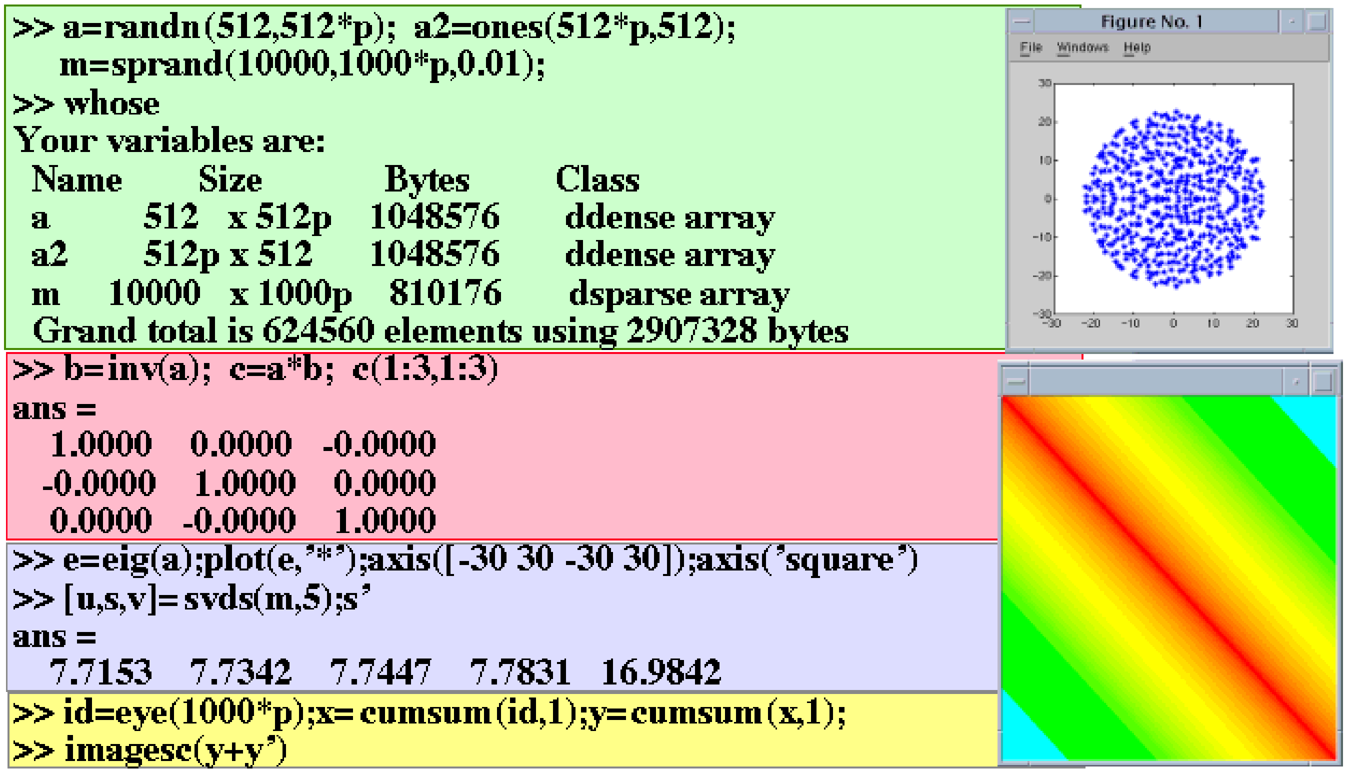
Figure 1: MPI-StarT Bandwidth (a) within an SMP (b) between SMPs

Broadcast Round Time (sec)		
P	Optimized	Original
1+1	0.148	0.147
2+2	0.466	0.781
4+4	1.158	2.420
8+8	3.591	6.934

Table 3: MPI-StarT Broadcast Performance ("P=1+1" means a total of 2 processors on 2 machines)

6. Operational Performance in MITMatlab

In order to fully test the performance gains from using MPI-StartT, we profiled a sample operation in a numerical application. MITMatlab[5] is a system that enables users to transparently work on large data sets within Matlab[8]. Figure 2 is a screen capture of an interactive MITMatlab session. MITMatlab is based on the Parallel Problems Server, a stand-alone program that provides a mechanism for running distributed-memory algorithms on large data sets. MITMatlab provides an easy way for Matlab users to interact with this external server.

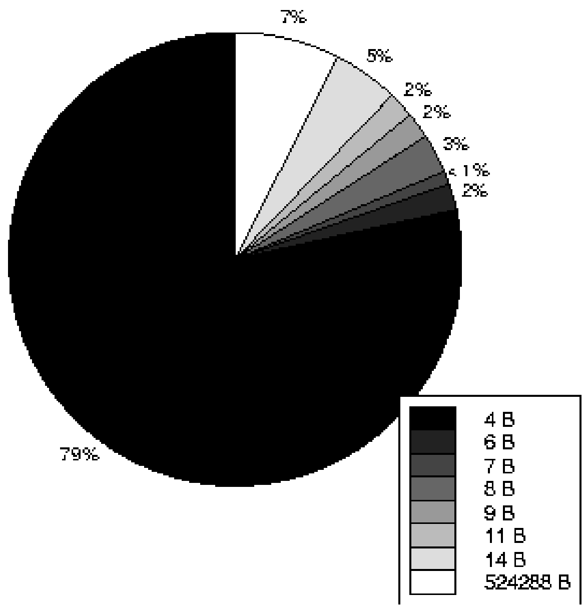


```
>> a=randn(512,512*p);  a2=ones(512*p,512);
    m=sprand(10000,1000*p,0.01);
>> whose
Your variables are:
  Name        Size            Bytes       Class
  a          512   x 512p    1048576     ddense array
  a2         512p x 512      1048576     ddense array
  m      10000   x 1000p    810176      dsparse array
  Grand total is 624560 elements using 2907328 bytes
>> b=inv(a);  c=a*b;  c(1:3,1:3)
ans =
    1.0000    0.0000   -0.0000
   -0.0000    1.0000    0.0000
    0.0000   -0.0000    1.0000
>> e=eig(a);plot(e,'*');axis([-30 30 -30 30]);axis('square')
>> [u,s,v]=svds(m,5);s'
ans =
    7.7153    7.7342    7.7447    7.7831   16.9842
>> id=eye(1000*p);x=cumsum(id,1);y=cumsum(x,1);
>> imagesc(y+y')
```

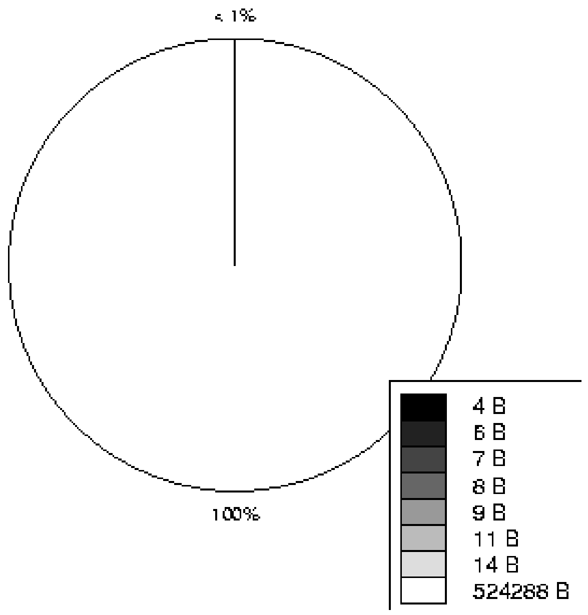
Figure 2: A Screen Capture of MITMatlab

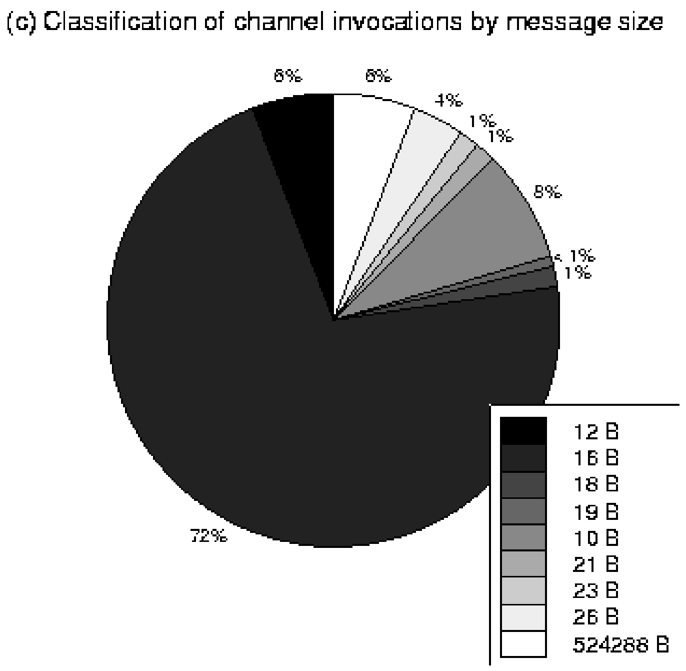
A typical computation might involve sparse matrices with over 40 million nonzero elements. Therefore, with the exception of server protocol messages, we expect many large transfers. To empirically verify this, we profiled a sample server operation to see the distribution of messages generated. The profiled operation is a common parallel matrix multiplication algorithm. The communication profile of a 1Kx1K matrix multiply on 8 processors split between 2 machines is shown in Figure 3. Pie charts (a) and (c) report the distribution of MPI broadcast and Channel Interface invocations by message size, Pie charts (b) and (d) show the distribution when weighted by the message size. Although, graphs (a) and (c) indicate the bulk of the invocations involves short messages, graphs (b) and (d) show that, in terms of bytes, the lion's share of the network utilization (and hence time) comes from large messages. We can thus conclude that it is important to have good performance on large transfers for such coarse-grained operations (one of the strengths of our implementation).

(a) Classification of MPI broadcasts by message size



(b) Classification of bytes broadcast by message size





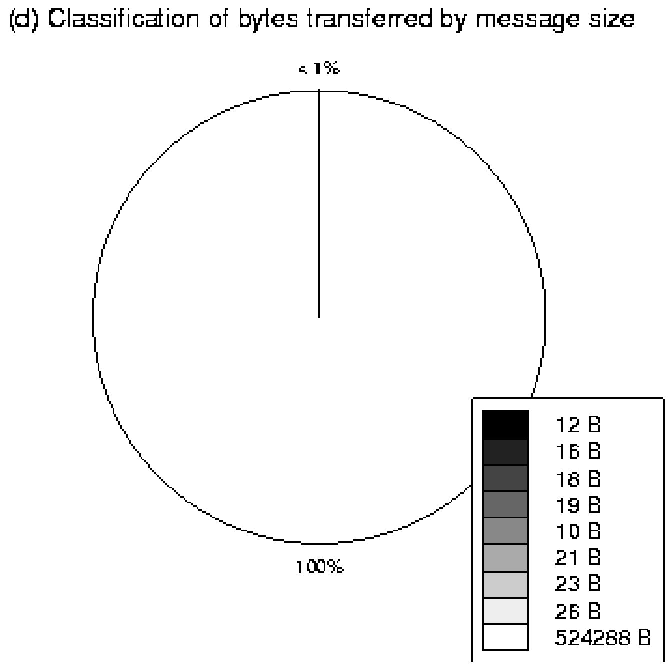


Figure 3: MPI Messages Profiled by Message Size

While we are continuing to improve our implementation, we see that MPI-StarT is effective in supporting MITMatlab on a cluster of SMPs. Table 4 summarizes the wall-clock time for matrix multiplication of N x N column-distributed matrices of single-precision floating-point numbers using two to twenty-four processors of a SUN E5000 SMP cluster. When testing on fewer than eight processors, we measure the performance both when the processors are on the same SMP and when the processors are divided between two SMPs. Before modifying the MPICH's broadcast implementation, our initial performance when using more than one SMP was dismal. However, by modifying the broadcast operation to manage the different inter and intra-SMP bandwidths, the current implementation is capable of speeding up the execution of MITMatlab on configurations of up to 24 processors over three SMPs. A similar level of performance and scalability cannot be reached on the same cluster using SUN's MPI library that supports communication over 100 Mbit Ethernet.

	Matrix Size		
	1Kx1K	2Kx2K	4Kx4K
Time	(sec)	(sec)	(sec)
P=1+1	9.7	69.5	NA
P=2	9.6	69.4	NA
P=2+2	4.8	35.1	403.5
P=4	4.7	35.0	402.9
P=4+4	2.6	17.5	204.2
P=8	3.0	17.7	207.8
P=8+8	2.0	10.6	102.9
P=8+8+8	3.0	9.5	79.1

Table 4: Wall-clock Time for N x N Matrix Multiplication on P Processors. ("P = 8 + 8 + 8" means a total of 24 processors on 3 SMPs.

Matrix multiplication is not the only operation that demonstrates speedups with MPI-StarT. Table 5 shows the performance of the sparse singular value decomposition routine in MITMatlab (taken from PARPACK [7]) on a SUN E5000 cluster. They show that MPI-StarT keeps up with SUN's MPI (using shared memory) no matter where the processes are placed.

MPI-StarT SVD Performance	
P	Time (sec)
1+1	297.6
2	295.3
2 (SUN)	297.7
2+2	173.9
4	171.8
4 (SUN)	167.9
4+4	125.0
8	129.2
8 (SUN)	128.9

Table 5: SVD Performance on P processors. These tests found the first 5 singular triplets of a random 10K x 10K sparse matrix with approximately 1 million nonzero elements. The (SUN) numbers used SUN's MPI.

7. Related Work

Vendor-supplied MPI libraries are available on nearly all commercial parallel platforms. Great efforts go into tuning these libraries for the best possible performance on a specific architecture. On the other hand, numerous other MPI packages solely rely on TCP or UDP to support clustering of stand-alone platforms over a standard local area network. These implementations are highly portable, but are hindered by the performance of the local area network and the large software overhead in the protocol and the operating system. MPI-FM[6], MPI-BIP [10] , and MPI-StarT all overcome this hurdle by augmenting a cluster with a higher performance user-level communication substrate. MPI-FM is based on Myrinet[1] and has reported a maximum bandwidth of 70 MByte/sec (for 64 Kbyte messages but 38 MB/sec for 1Mbyte messages) and a minimum latency of 17 microseconds on a cluster of x86 PCs. MPI-BIP is also based on Myrinet and achieves a maximum bandwidth of 113 MB/s and minimum latency of 9 microseconds between two Pentium Pro workstations but is not multi-protocol.

8. Summary

This paper focused on the issues in implementing a high-performance MPI layer. We based our implementation on MPICH by developing a custom Channel Interface for

StarT-X and by modifying a small portion of MPICH's Protocol and ADI Layers. We have observed encouraging results in employing MPI-StarT to support MITMatlab on a cluster of SUN E5000 SMPs with StarT-X. Our current implementation can make the full bandwidth of the hardware available to user applications, but we are still working on several optimizations to improve the latency of small messages. We intend to use faster locking primitives and fewer network messages to reduce the overhead. We are also collaborating with the authors of MITgcmUV[3], a global climate model, to bring up their simulation code on a StarT-X cluster of Intel PII personal computers.

Acknowledgments

This paper describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Parry Husbands is supported by a fellowship from Sun Microsystems and is supervised by Professor Alan Edelman. James C. Hoe's research is supervised by Professor Arvind of the Laboratory for Computer Science and is funded in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-92-J-1310 and Ft. Huachuca contract DABT63-95-C-0150.

References

1 N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. Su. Myrinet - A Gigabit-per-second Local-Area Network. IEEE Micro, February 1995.

2 W. Gropp and E. Lusk. MPICH working note: Creating a new MPICH device using the Channel Interface. Technical Report ANL/MCS-TM-213, Argonne National Laboratory, 1995.

3 C. Hill and J. Marshall. Application of a parallel Navier-Stokes model to ocean circulation. Parallel Computational Fluid Dynamics: Implementations and Results Using Parallel Computers, pages 545-552, New York, 1995.

4 J. C. Hoe. StarT-X: A one-man-year Exercise in Network Interface Engineering. In Proceedings of Hot Interconnects VI, August 1998.

5 P. Husbands and C. L. Isbell. The Parallel Problems Server: A Client-Server Model for Large Scale Scientific Computation. In Proceedings of VECPAR'98, 1998.

6 M. Lauria and A. Chien. MPI-FM: High Performance MPI on Workstation Clusters. Parallel and Distributed Computing, pages 4-18, January 1997.

7 K. J. Maschoff and D. C. Sorensen. A Portable Implementation of ARPACK for Distributed Memory Parallel Computers. In Preliminary Proceedings of the Copper Mountain Conference on Iterative Methods. 1996

8 MATLAB. http://www.mathworks.com/products/matlab/.

9 Message Passing Interface Forum. MPI: A Message Passing Interface Standard, 1.1 edition, June 1995.

10 L. Prylli and B. Tourancheau. BIP: A New Protocol Designed for High Performance Networking on Myrinet. In Workshop PC-NOW, IPPSIS/SPDP98. 1998.

11 Thinking Machines Corporation, 245 First Street, Cambridge, MA02142, USA. Connection Machine CM-5 Technical Summary, November 1993.

Author Biography

Parry Husbands is a Ph.D. student in Computer Science at the Massachusetts Institute of Technology. He received his S.M. from MIT in 1994 and B.Sc. in Math and Computer Science from the University of Toronto in 1992. His research interests are primarily in the development of user-friendly scientific computing tools.

James C. Hoe is a Ph.D. student in Electrical Engineering and Computer Science at the Massachusetts Institute of Technology. He received his S.M. from MIT in 1994 and B.S. in EECS from the University of California at Berkeley in 1992. His research interests are in computer architecture, hardware synthesis, and network and network interfaces. He is currently working on developing a hardware synthesis tool that accepts high-level behavioral descriptions in Term Rewriting Systems.