Low Latency Message Passing on Workstation Clusters using SCRAMNet [1] [2]

Vijay Moorthy, Matthew G. Jacunski, Manoj Pillai, Peter P. Ware, Dhabaleswar K. Panda, Thomas W. Page Jr. and P. Sadayappan
Dept. of Computer and Information Science
The Ohio State University
Columbus, OH 43210
{moorthy|jacunski|pillai|ware|panda|page|saday@cis.ohio-state.edu}
Tel: (614)-292-5199, Fax: (614)-292-2911

V. Nagarajan and Johns Daniel
SYSTRAN Corporation
4126 Linden Avenue
Dayton, OH 45432
{nagu|jdaniel@systran.com}

Contact Author: Vijay Moorthy

Abstract:

Clusters of workstations are emerging as a popular platform for parallel and distributed computing. The performance of clusters of workstations depends greatly on the networks interconnecting them. In recent years many high performance interconnects have been developed. These interconnects provide high bandwidth. However, they also tend to have high communication latency. There is considerable ongoing research for providing message passing for clusters of workstations with low communication latency. SCRAMNet is an extremely low latency replicated non-coherent shared memory network. So far, this interconnect has been used only for real-time applications and very little work has been done to adapt it for use in cluster computing.

This paper reports our early experiences with using SCRAMNet for cluster computing. The basic features of SCRAMNet are presented. We have implemented a user-level zero-copy message passing protocol called the BillBoard Protocol (BBP). This protocol and its API are presented in detail. The one way latency for sending a 4-byte message between two nodes has been observed to be as little as 9.2 microseconds. Since the SCRAMNet interconnect supports hardware replication, it is possible to implement broadcast and multicast with almost the same latency as point-to-point communication. Using the BBP, the latency for broadcasting short messages to 4 nodes has been measured to be as low as 11.2 microseconds. Similarly, the latency for a 4-node barrier is as low as 44 microseconds. We have also built an MPI library on top of the BBP which makes use of broadcast support from the BBP to enhance broadcast ,multicast and barrier. Performance results at both levels (API and MPI) are compared with comparable implementations on other contemporary networks such as Fast Ethernet, ATM and Myrinet. These results demonstrate the potential of SCRAMNet as a high performance interconnect for building scalable workstation clusters supporting either message passing or distributed shared memory.

[1]This research is supported by an NSF-SBIR Grant DMI-9761318.
[2]SCRAMNet is a registered trademark of SYSTRAN Corporation

Contents

1 Introduction **3**

2 SCRAMNet Networking **4**

2.1 Overview 4

2.2 Advantages of using SCRAMNet for Cluster Computing 5

3 Design Goals for Development of Message Passing on SCRAMNet **5**

4 The BillBoard Protocol and the API **6**

4.1 BBP Shared Memory Organization 7

4.1.1 Data Partition 7

4.1.2 Control Partition 7

4.2 Communication Protocol 8

4.3 The API 9

5 Implementing MPI over SCRAMNet **9**

5.1 MPICH Architecture 10

5.2 MPI Collective Operations over SCRAMNet 11

5.2.1 MPI semantics for collective operations 11

5.2.2 Implementation of MPI_Bcast 12

5.2.3 Implementation of MPI_Barrier 12

6 Performance evaluation **13**

6.1 Point-to-Point Communication 13

6.2 Collective Communication 15

7 Related work **17**

8 Conclusions **17**

1 Introduction

There is a great interest today in using workstation clusters as a platform for parallel and distributed computing and a cost effective alternative to conventional parallel computers[1, 3, 19]. Unlike MPP's which use customized interconnection networks for communication [8], workstation clusters use commodity networking technologies. In recent years many new high performance networking technologies have emerged such as ATM [2], Fast Ethernet [15] and FiberChannel [20]. These networks often provide high bandwidth but also have high latency due to network latency and high software overhead. These networks were originally designed for high throughput, which they deliver in everyday network usage. However, for high performance in parallel processing, low latency is very critical. As a result considerable effort has been put into developing low latency networking technologies such as Myrinet [5] and messaging software such as Illinois Fast Messages [18] and U-Net [23].

SCRAMNet (Shared Common RAM Network) [6, 22] is a replicated non-coherent shared memory network. It was initially developed for systems to execute real-time applications that require low latency communication, such as aircraft simulators, industrial process control, virtual reality, telemetry and robotics. So far, there has not been much work done to adapt SCRAMNet for use in cluster computing. For these applications, SCRAMNet has been almost exclusively used for shared memory programming and little work has been done for providing message passing support on SCRAMNet. Message Passing has been increasing in popularity [17, 21] and MPI (Message Passing Interface) [17] has emerged as the *de facto* standard for portable parallel programming on a wide variety of systems ranging from MPP's to workstation clusters. In this paper, we present our early experiences with building a low latency message passing system for workstation clusters using SCRAMNet.

SCRAMNet consists of network interface cards (NICs) with memory banks connected via a dual ring. The NICs can be plugged into standard I/O buses such as PCI and S-Bus. The memory banks on the NICs can be mapped into the address spaces of workstations on which the NICs are connected. When a workstation writes to a memory location on its NIC, it is propagated through the ring and the write is reflected at the same location on the memory banks on the other NICs connected to the ring. The memory is shared, as a write by one node on the ring can be read by all others. However, the memory is not coherent, that is, if two nodes concurrently write to the same location, all the nodes may not see the writes in the same order. Since SCRAMNet uses a ring topology, the time taken for a write to propagate throughout the ring is bounded and predictable. Also, the latencies involved are very low - 250 ns per hop for short messages.

In this paper, we present the design details and performance evaluation of a low latency message passing protocol developed by us for workstation clusters using SCRAMNet. This protocol is named the BillBoard Protocol (BBP). It is a zero copy protocol in which messages are copied from user space directly to the NIC. It does not use locks and avoids race conditions by disallowing any location in the shared memory from being written by more than one process. We present a SCRAMNet Billboard Application Programming Interface (API) which provides rudimentary message passing on SCRAMNet for point-to-point communication as well as multicast/broadcast/barrier to support collective operations efficiently. We also describe our implementation of an MPI library for SCRAMNet, which is built on top of the BillBoard API. Unlike most implementations in which these operations are built on top of point-to-point communication primitives, this implementation allows MPI collective operations (broadcast and barrier) to be directly supported on top of the BillBoard API, to provide low latency collective operations.

A range of performance evaluation tests were carried out on an experimental test-bed to study the performance of the BBP and the MPI implementation. These results were also compared with the performance results obtained using other contemporary networking technologies such as Fast Ethernet, ATM and Myrinet. For point-to-point communication, SCRAMNet does very well for small messages. For messages less than 500 bytes, it is faster than all the above technologies. Sending a short 4-byte message takes only 9.2 μs at the API layer. The API layer broadcast takes very little additional time than point-to-point communication. At the MPI layer, barrier latency is an order of magnitude faster than that for Fast Ethernet and ATM. These results demonstrate that SCRAMNet technology can be used in workstation clusters to support extremely low latency point-to-point and collective communication.

This paper is organized as follows. Section 2 provides a brief overview of SCRAMNet technology. Section 3 discusses the requirements and methodology of our development of a message passing system for SCRAMNet. Section 4 describes the BillBoard Protocol and API. Section 5 describes our MPI implementation for SCRAMNet. Section 6 evaluates the performance of this system at various levels and compares it with other networking technologies. Section 7 describes our ongoing work and related work done elsewhere. Section 8 presents our conclusions.

2 SCRAMNet Networking

This section briefly describes the SCRAMNet networking and outlines the advantages for using it in workstation clusters.

2.1 Overview

SCRAMNet [6] is a replicated non-coherent shared memory network. It provides a limited amount of non-coherent shared memory across the cluster. The network consists of a set of workstations with NICs containing memory banks. Writes to any one memory bank are replicated on all the other NICs on the network. Though physically distributed, the memory is shared; writes by any node are visible to all other nodes. However, SCRAMNet shared memory is not coherent; different nodes may not see updates in the same order.

Physically, SCRAMNet is arranged as a ring of up to 256 nodes. Two neighboring nodes can be separated by a maximum of 300m. The maximum separation can be increased up to 3500m using 1300nm laser. The ring is normally made of fiber-optic cable which allows such distances but for shorter distances, coaxial cabling can also be used. A fiber optic bypass switch ensures continuity in the ring when not all nodes are powered on. For systems larger than 256 nodes, a hierarchy of rings is used to accommodate more nodes. SCRAMNet is extremely fast. The data transfer latency between two neighboring nodes is 250-800 nanoseconds, depending on the mode of transmission being used and packet size. SCRAMNet normally uses fixed 4-byte packets for transmission. It provides a maximum throughput of 6.5 MBytes/sec. Variable length packets with lengths varying from 4 bytes to 1 KByte can also be used. In this mode, the maximum throughput is 16.7 MBytes/sec. However data latencies are much higher when variable length packets are used.

SCRAMNet NICs are available with memory ranging from 4 KBytes to 8 MBytes, for a wide variety of I/O buses and backplanes such PCI, Sbus, EISA, ISA, PMC and VMEBus. No driver software is required to use these NICs as the data transfer is handled at the hardware level. Most OS's allow the shared memory on the NIC to be mapped to the virtual address space of the workstation. Once this mapping is done, any write to the memory on the NIC is automatically and immediately propagated to the other NICs on the ring network.

SCRAMNet uses Register Insertion for Media Access Control on the ring. At each node, there is a switch which multiplexes incoming messages from the incoming link on the network and the NIC to the outgoing link. If a message arrives on the network, while the node is in the process of injecting a message into the network, then the network message is queued. A message is generated whenever a host writes to a location in SCRAMNet memory. Multiple writes are put in a FIFO queue on the NIC and sent out one by one. Each message traverses the entire ring once and then is removed from the network by the source node.

This section provides only a brief overview. Readers are directed to [6] for details.

2.2 Advantages of using SCRAMNet for Cluster Computing

Shared memory and message passing are the two programming models available for parallel programming. Applications which require shared memory tend to use physically shared memory for communication. However, for large systems physically shared memory is very difficult to build. Larger systems, therefore tend to use message passing. However, message passing using conventional LAN networking technology is orders of magnitude slower. Distributed Shared Memory (DSM) systems try to provide an abstraction of shared memory on top of message passing systems. For a DSM system to work, it has to provide cache coherence and scaling DSM systems while maintaining cache coherence is a difficult problem [12].

SCRAMNet offers a middle path. It has a limited amount of non-coherent shared memory, but it is scalable. It does not have very high bandwidth but it has extremely low latency. In addition, message latencies are bounded. There is no overhead of protocol information to be added on messages and information. Since the SCRAMNet shared memory can be mapped to the virtual address space, data can be transferred from one node to another using a single store instruction, or an assignment statement in a high level language. For larger data transfers, programmed I/O or DMA can be used.

The next section describes the development of a message passing system on top of SCRAMNet.

3 Design Goals for Development of Message Passing on SCRAMNet

We have developed a protocol for message passing on SCRAMNet which we have named the BillBoard Protocol (BBP). The meaning of the name will be clearer as we describe the protocol in detail in the next section. We have also built a messaging library using this protocol for PCs connected with SCRAMNet and running *Linux*. In this section we present the requirements that drove the development of the protocol and the primitives provided by our messaging library for SCRAMNet.

There were several underlying design goals for message passing primitives on SCRAMNet:

- Provide efficient user-level message passing primitives that provide typical send/receive functionality to application programs.

- Allow for multiple processes on a single machine.

  A single workstation may have multiple processes running which need to communicate through SCRAMNet. The protocol should permit more than one process per node to share the same NIC.

- Provide feasibility for adding write protection.

  Messages are sent on SCRAMNet by writing to SCRAMNet memory. When there are multiple participating processes, the protocol should implement write protection.

- Provide in-order delivery between any two processes.

  Messages from a process A to process B must be delivered in the same order in which they are sent.

- Minimize accesses to the SCRAMNet card over the I/O bus.

  The processor in the workstation has to go through the I/O bus to read SCRAMNet memory. Since SCRAMNet memory can receive updates over the network, and not just the host processor, it cannot be cached. Also I/O buses are slower than system buses. So it takes the processor more time to access SCRAMNet memory than the main memory. The I/O bus is normally shared with other devices. Hence it can be a bottleneck, and accesses over the I/O bus must be minimized.

- Implementation of the protocol should not require any locks.

  If processes have to wait for locks, latencies can increase. Therefore, the protocol should be lock-free.

- Design memory access and use protocol such that applications may benefit from the inherent broadcast/multicast capability of SCRAMNet.

We followed these design goals to develop the BillBoard protocol. The next section describes the protocol and the associated API.

4 The BillBoard Protocol and the API

The BillBoard Protocol derives its name from the manner in which communication is done using this protocol. When a process has to send a message to one or more processes, it posts the message at one place, in the SCRAMNet shared memory. The message can then be read by one or more receivers. This method of transmission is akin to advertising on a billboard, hence the name. This protocol requires $O(n)$ storage for data and $O(n^2)$ storage for control information. This protocol has very low latency because no process has to wait for a lock to write in the shared memory, as any location is written by exactly one process.

In this section we describe how the SCRAMNet shared memory is used, how communication is performed, and the primitives provided by the library.

4.1 BBP Shared Memory Organization

The shared memory is divided equally among the participating processes. The maximum number of processes is therefore limited by the amount of shared memory available. Each process' memory block is further subdivided into a data partition and a control partition. Figure 1 shows the organization of the partitions for a sample system. The data partition is where messages are written. The control partition holds control information required for transfer of messages, described in detail below. When a process wants to send a message it allocates a buffer in its data partition, writes the message to that buffer, and then sets flags in the control partition of the receiving process to indicate a waiting message. The receiving process then reads the message buffer in the sender's data partition and sets flags in the sender's control partition to acknowledge receipt of the message.

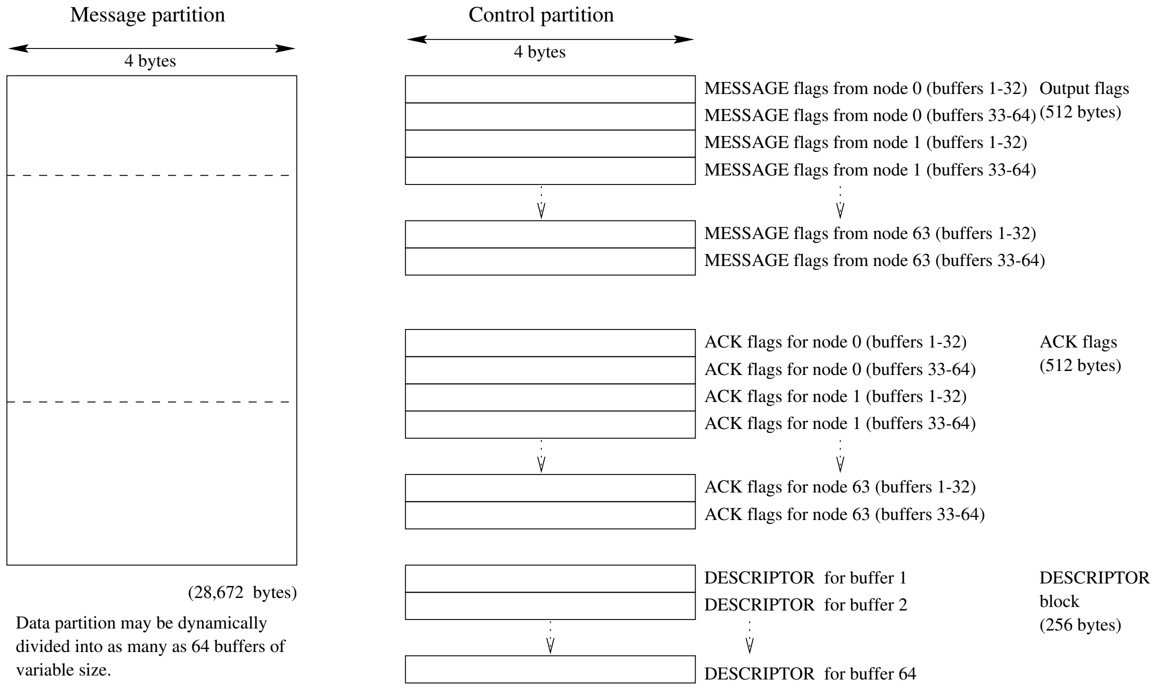

Figure 1: SCRAMNet memory organization for a sample system with 64 processes and 64 buffers per process and 2 MB of SCRAMNet memory.

4.1.1 Data Partition

The data partition is a heap from where memory is allocated for message buffers. The number of buffers, b is determined by the number of bits allocated in the control partition for various flags associated with these buffers.

4.1.2 Control Partition

The control partition contains the following :

- MESSAGE ready flags: n sets (corresponding to n participating processes) of b bits each indicating a specific block to be read by this node. Each set of flags starts at a word boundary.

- ACK flags: one for each buffer in the data partition, (b bits) indicating that the buffer has been read by receiving process(es), and that the buffer may be re-allocated. Each set of flags starts at a word boundary.

- DESCRIPTOR blocks: These contain the starting location, length, type and ID of each buffer in the data partition. Each descriptor is one word long.

The amount of SCRAMNet shared memory allocated to each process depends on the total number of participating processes. If M bytes is the total memory, then each process gets M/n bytes. This memory is to be divided into the data and control partitions. Since there are b buffer descriptors, then the control partition needs $n * (b \bmod 8)$ bytes for MESSAGE flags, $n * (b \bmod 8)$ bytes for ACK flags, and $4 * b$ bytes for buffer descriptors. Let C be the size of the control partition. Then,

$$C = 2 * n * (b \bmod 8) + 4 * b$$

It follows that, for this protocol to work, $C < (M/n)$ must hold. Let $D = (M/n) - C$ be the size of the data partition. Then, the average size of a message buffer B is D/b. For good performance, B should be somewhere in the range of the lengths of messages being sent. If B is not in this range, then it indicates an imbalance between the size of the data partition and the number of buffer descriptors in the control partition. If B is in the lower end of this range, then there is insufficient memory in the data partition and b should be decreased. If B is in the higher end of this range, then there are not enough buffer descriptors, so b should be increased. Ideally, B should be close to the actual mean of lengths of messages being sent. That would ensure high utilization of the available memory in the data partition as well as the buffer descriptors in the control partition.

4.2 Communication Protocol

Let us see how communication can occur using the above memory organization. Suppose process P_1 wants to send a message to process P_2. First P_1 tries to allocate a buffer in the data partition for the message transfer. If there is sufficient space, the next message buffer number is allocated for it. P_1 then writes the message buffer descriptor in P_1's control partition and writes the message in the allocated buffer. P_1 also writes the message flag in P_2's control partition corresponding to P_1 after toggling the bit corresponding to the allocated message buffer. During the receive operation, P_2 reads the message flags from its control partition in the shared memory and compares it with the previously stored values. Any changes indicate newly arrived messages, and their bit positions indicate the message buffer numbers. To receive the message, P_2 reads the offset and length of the message from P_1's control partition. Then it reads the appropriate buffer from P_1's data partition and toggles P_2's acknowledgement flag for that buffer in P_1's control partition. The message transfer is then complete.

In our protocol, only a single node on the SCRAMNet network is permitted to write to any particular memory location at any one time. Any node may write to its data partition, descriptor blocks, and the ACK and MESSAGE ready flags in other nodes' control partitions that are associated with this node. Essentially all memory allocated to a node may be written to by only that node except for MESSAGE flags and ACK flags associated with other nodes. Although many processes may be writing to one process' control partition concurrently, these writes do not interfere with each other as MESSAGE flag sets and ACK flag sets are always word aligned.

Since both the memory in the data partitions and the number of descriptors are limited, each process has to do garbage collection whenever it runs out of either. For garbage collection, a process reads its acknowledgement flags and then frees all those buffers that have been acknowledged.

The above protocol describes point-to-point communication on SCRAMNet. Since the data partition of every process is visible to all others, this protocol can be very easily extended for collective operations. To send a multicast or a broadcast message, a process allocates a buffer in its data partition, as before (and does garbage collection if required). The process writes the message to the allocated buffer, and then sets MESSAGE ready flags for more than one receiver (all the other processes in case of broadcast). Each extra receiver, adds only the cost of writing one more word to SCRAMNet memory at the sender. This is why it costs just a little more than sending a point-to-point message to do multicast or broadcast on SCRAMNet. Unlike point-to-point based algorithms for multicast such as the binomial tree algorithm [14], this type of multicast is a single step algorithm, which means that potentially, all the receivers could receive the multicast message simultaneously.

4.3 The API

The function of the API is to provide a bare-basics message passing interface on top of which more sophisticated layers can be built. The next section describes our implementation of MPI on top of this API. A process is identified by its rank. If there are *n* participating processes, then each process has a unique rank (or id) which is a number from 0 to *n-1*. All participating processes must first call an initialization function to set up SCRAMNet memory and persistent state information such as message flags. After initialization, a process may send or receive messages. Sources and destinations are specified by their ranks. User processes call a "Message Available" function that checks SCRAMNet memory for new messages from a specific node. The advantage of this is that a single memory access may detect many new messages. This has a significant impact on performance, as reading the memory over the I/O bus is the highest latency component in the transfer of messages between processes. There are just five functions in the API :

- `scr_init` : This function maps SCRAMNet memory to the process's virtual address space, allocates storage space, and initializes the persistent state information and the data and control partitions.
- `scr_Send` : Sends a message to one destination.
- `scr_MsgAvail` : Checks if any new messages have arrived from a particular source.
- `scr_Recv` : Receive a message from a specified source.
- `scr_Bcast` : Sends a message to more than one destination (Multicast).

5 Implementing MPI over SCRAMNet

MPI is a popular standard for writing portable parallel programs. It is well suited as a programming paradigm for distributed memory clusters. This section describes our implementation of MPI which is built on top of the BillBoard API.

Our implementation of MPI is a derivative of MPICH [9], a publicly available implementation of MPI from Argonne National Labs. Most commercial MPI implementations as well as many research implementations of MPI are derivatives of MPICH.

5.1 MPICH Architecture

MPICH has a layered architecture. Figure 2 shows the various layers in the MPICH stack with sample functions from each layer. The top two layers contain the bindings to the MPI functions. The point-to-point binding layer sits on top of the Abstract Device Interface (ADI) layer. Though complex functions such as collective communication operations, and topology mapping are built on top of the basic point-to-point layer, hooks are provided in MPICH to extend it so that these more complex operations can directly use any additional functionality provided by the underlying device. For instance, in the MPICH devices for the CRAY T3D and Intel NX, MPI collective operations can use the collective communication support provided by these systems.

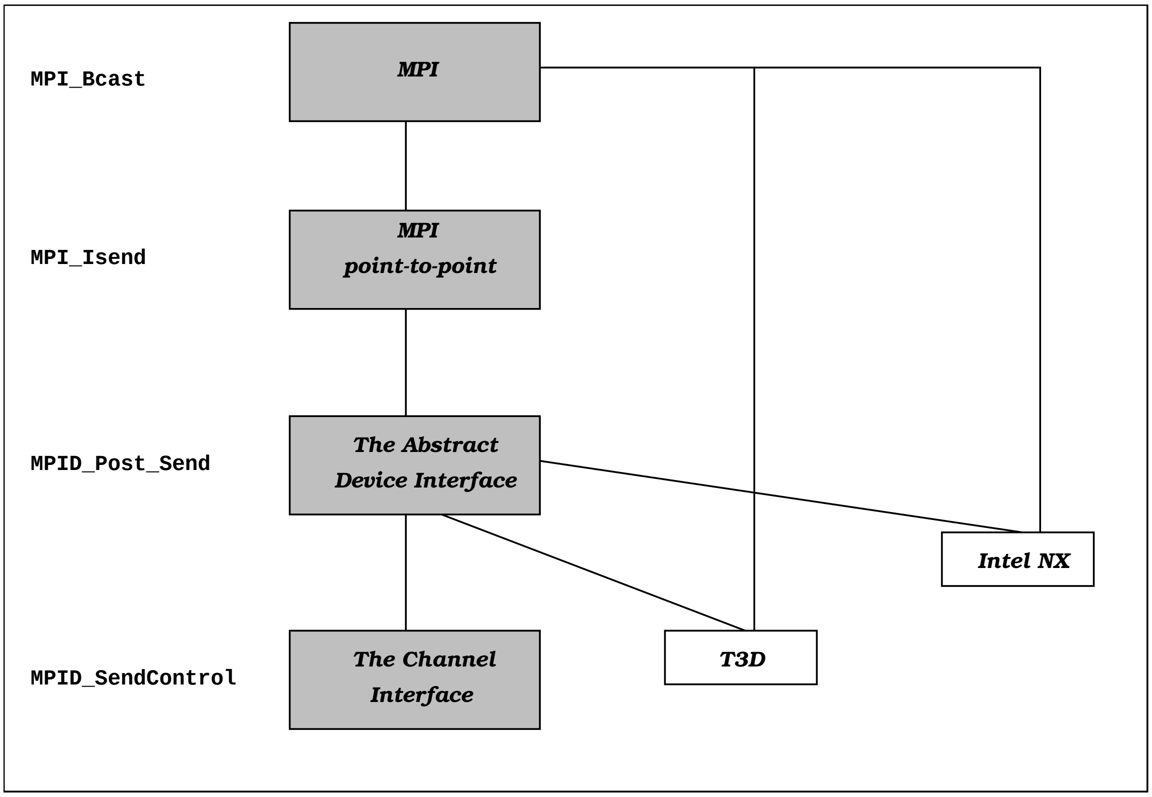

Figure 2: Overview of layered Architecture of MPICH.

The ADI is a set of macros and functions. All MPI functions are implemented on top of it. The ADI is the key to the portability of MPICH. Some devices may provide limited functionality while others may provide more complex functionality. The purpose of the ADI is to define an abstract device with a rich set of functions and then provide software emulation of functions not implemented in the underlying device.

The ADI provides functions for specifying messages to be sent and received, moving data between the MPI layer and message passing hardware, managing message queues and providing basic information about the execution environment. In particular, the ADI contains code for packetizing messages, attaching header information, managing multiple buffer policies, matching posted receives with incoming messages and queuing messages if necessary, and handling heterogeneous communication.

The ADI can be implemented on top of any device. For instance, in case of the CRAY T3D, it is implemented in terms of primitives provided by the T3D. One implementation of the ADI is in terms of a lower level interface called the channel interface. Because the channel interface is a much more rudimentary interface, it provides the fastest route to porting MPICH to a new system. The functionality required of this interface is just that provided by UNIX select, read and write. We have used this approach in our implementation and developed a SCRAMNet Channel layer device which is a minimal implementation of the Channel Interface.

5.2 MPI Collective Operations over SCRAMNet

MPI provides a rich set of collective communication operations. The collective operations in MPI are barrier, broadcast, gather, scatter, all-gather, complete exchange and global reduction [17]. All collective operations have a communicator argument that specifies the group of processes that are to be involved in the collective operation. MPICH provides an implementation that layers MPI collective communication operations on top of MPI point-to-point operations. However, some MPI collective operations, notably broadcast and barrier, can be implemented more efficiently if the underlying network has broadcast capability.

In our implementation, MPI Multicast and Barrier have the flexibility not to use point-to-point based algorithms. Instead, they can use the broadcast nature of the SCRAMNet hardware. We have implemented `MPI_Bcast` and `MPI_Barrier` using the multicast function provided by the BillBoard API as described in section 4.3. Fig 3 shows how these functions fit into the MPICH architecture.

5.2.1 MPI semantics for collective operations

The MPI standard specifies the following semantics for collective operations:

1. Collective operations in MPI take place in a different communication domain than point-to-point operations. Messages sent by collective operations do not interfere with and cannot be received as point-to-point messages.
2. Collective operations are blocking. Message buffers can be reused once the call returns.
3. Collective operations other than barrier do not necessarily synchronize the participating processes. Implementations are free to choose whether or not to synchronize processes during a collective operation. For instance, it is valid for an implementation to allow the root process of a broadcast to proceed as soon as the data of the broadcast has been buffered.
4. MPI requires that collective operations be matched in execution order. So, even though collective operations may not synchronize the processes, there is no ambiguity regarding the result of execution of an MPI program with collective operations.

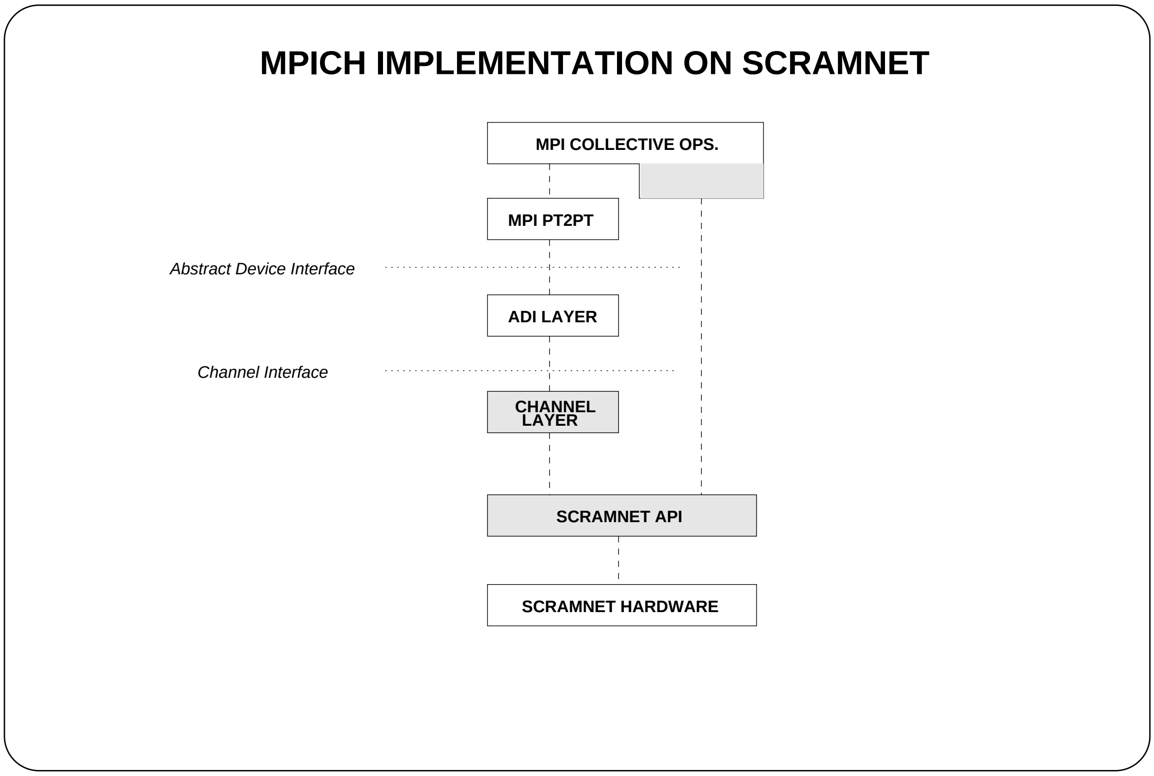

Figure 3: MPICH Implementation on SCRAMNet

We follow these semantics to implement MPI_Bcast and MPI_Barrier using the SCRAMNet API. These implementations are discussed below.

5.2.2 Implementation of MPI_Bcast

The process that is the root (or sender) of the MPI_Bcast determines the processes in the group. It then uses the multicast operation in the SCRAMNet API to broadcast the data to each process in the group. All receiving processes wait for a message from the root process. Our implementation of broadcast is not synchronizing. The root of the broadcast does not wait for other processes to arrive at the MPI_Bcast call. The root of the broadcast is the first to finish the MPI_Bcast call. Since the SCRAMNet API provides in-order delivery, multiple MPI_Bcast operations are matched in order.

5.2.3 Implementation of MPI_Barrier

In our implementation of an MPI_Barrier, the process with rank 0 acts as the co-ordinator. It waits for a null message from every other process in the group associated with the communicator. After it has received messages from all processes in the group, it uses the multicast operation in the SCRAMNet API to send a null message to all other processes in the group.

6 Performance evaluation

This section describes the performance of our message passing system at both API and MPI levels. Results for point-to-point as well as collective operations are presented. The measurements were taken on a network of 4 Dual Pentium II 300 MHz SMP boxes connected by SCRAMNet and running on LINUX version 2.0.30. The results are also compared with comparable implementations on other contemporary technologies namely, Fast Ethernet, ATM and Myrinet.

6.1 Point-to-Point Communication

We have done extensive experimentation to determine the point-to-point communication latency. Figure 4 shows the one-way latencies for sending messages on SCRAMNet at the API level and MPI level. It can be observed that for small messages, the latency is very small. The MPI layer adds a little overhead over the API layer. At the API layer, a 4-byte message can be sent in 9.2 μs. At the MPI layer a 4-byte message can be sent in 55 μs.

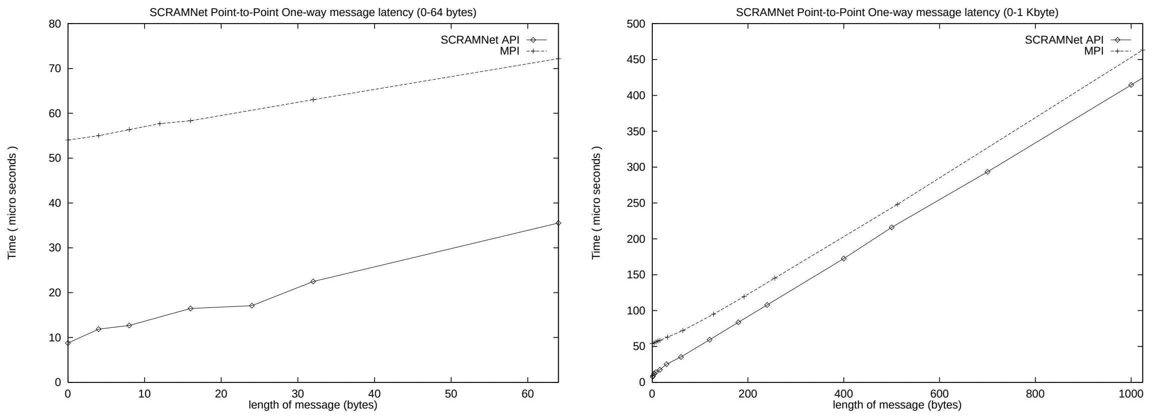

Figure 4: One-way Message latency for SCRAMNet Using BillBoard API and MPI

Figure 5 compares the performance of SCRAMNet with Fast Ethernet, ATM and Myrinet at the API layer. The measurements for Fast Ethernet and ATM were taken with TCP/IP. The measurements for myrinet were taken with the native myrinet API as well as TCP/IP. SCRAMNet does better than Fast Ethernet for message lengths up to several thousand bytes. It is better than ATM for message lengths less than 1000 bytes. It has lower latency than Myrinet using myrinet API for messages shorter than 500 bytes.

Figure 6 compares the performance of SCRAMNet with Fast Ethernet and ATM at the MPI level. The measurements shown for Fast Ethernet and ATM were taken using MPICH with TCP/IP. As in the API layer, we see that SCRAMNet does well for small messages and with each of the other two networks, there is a threshold beyond which the other network does better. SCRAMNet is faster than Fast Ethernet for messages shorter than 440 bytes and faster than ATM for messages shorter than 550 bytes.

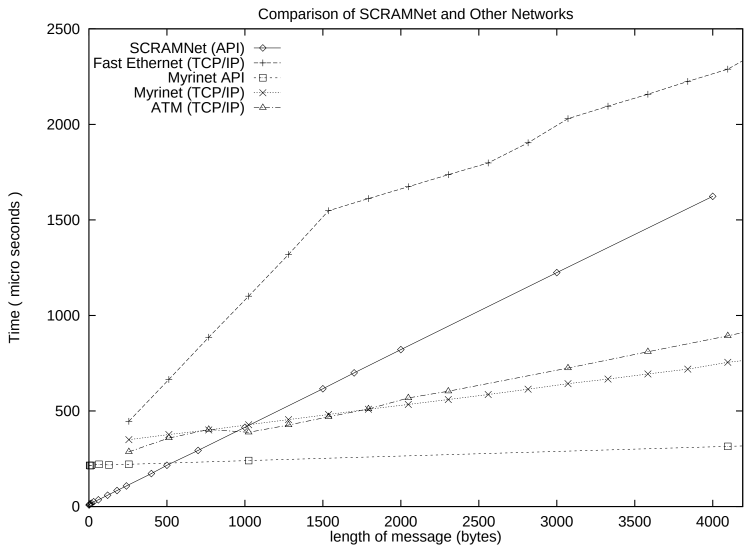

Figure 5: Comparison of SCRAMNET with other networks at the API layer

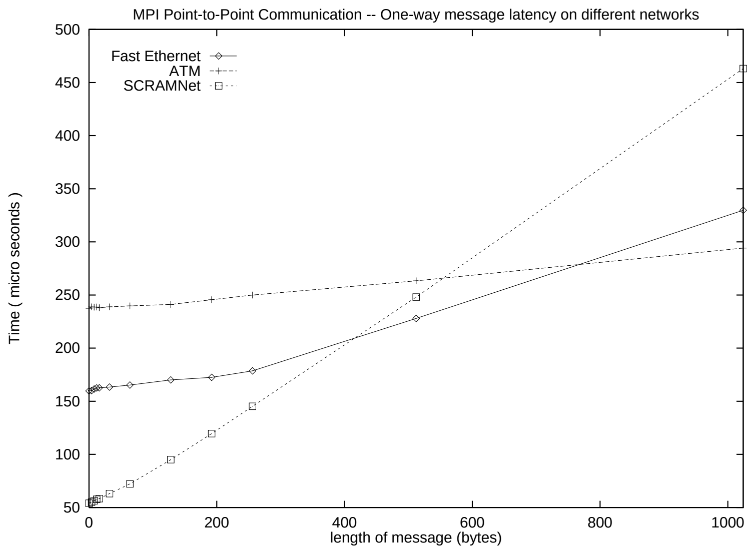

Figure 6: Comparison of SCRAMNET with other networks at the MPI layer

6.2 Collective Communication

Figure 7 shows the time taken to send a message to one receiver and to 3 receivers (broadcast). As can be seen the broadcast adds a very little overhead to the one receiver message latency.

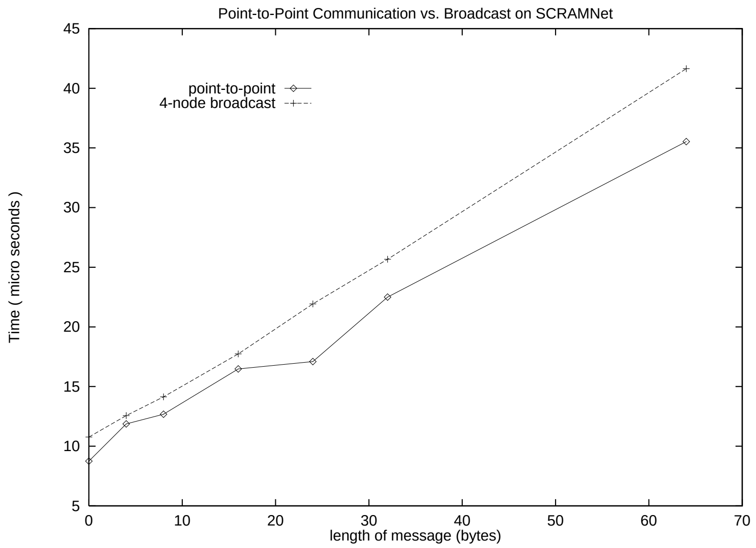

Figure 7: Comparison of SCRAMNET point-to-point and broadcast communication

Figure 8 compares the performance of MPI broadcast on Fast Ethernet and SCRAMNet. Two implementations of MPI_Bcast for SCRAMNet are shown - one using the standard point-to-point based algorithm in MPICH, and one using the API level multicast. Comparing the point-to-point based broadcast, we see a trend similar to that for point-to-point communication. For short message lengths, SCRAMNet does better than Fast Ethernet, but for messages longer than 400 bytes, Fast Ethernet does better. Our implementation of MPI_Bcast using the API level multicast is much faster. In this experiment, up to message lengths of 1 Kbyte, it is faster than Fast Ethernet. For very long messages, Fast Ethernet will do better.

Figure 9 shows the performance of Barrier on SCRAMNet. As in the case of MPI_Bcast two implementations of MPI_Barrier are shown, one using the standard MPICH point-to-point based algorithm, and one using the API level multicast. MPI_Barrier is more expensive on Fast Ethernet and ATM than SCRAMNet. For instance, a 3-node barrier takes 554 μs on Fast Ethernet and 660 μs on ATM, while it takes only 179 μs on SCRAMNet using the point-to-point algorithm. The implementation using the API level multicast takes only 37 μs, that is 15 times as fast as Fast Ethernet and 18 times as fast as ATM.

For collective operations we see a trend similar to that seen for point-to-point communication. The behavior of SCRAMNet vis-a-vis other networks differs for short and long messages. Since SCRAMNet has low software overhead, and very low network latency, message latencies are much lower when compared to other networks for short messages. Since the barrier operation involves very short messages, we see a good improvement for SCRAMNet over other networks. Short message length broadcast is also an order of magnitude better than Fast Ethernet since a broadcast costs just a little bit more than a point-to-point message.

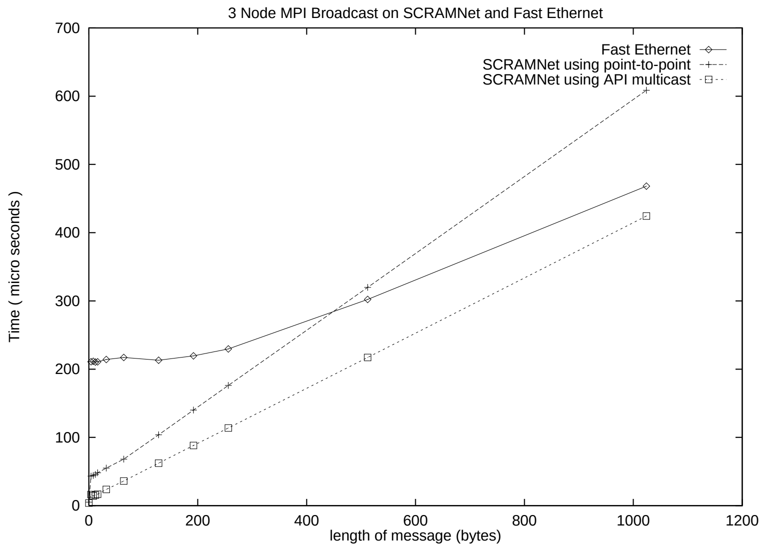

Figure 8: Comparison of MPI broadcast on SCRAMNet and Fast Ethernet

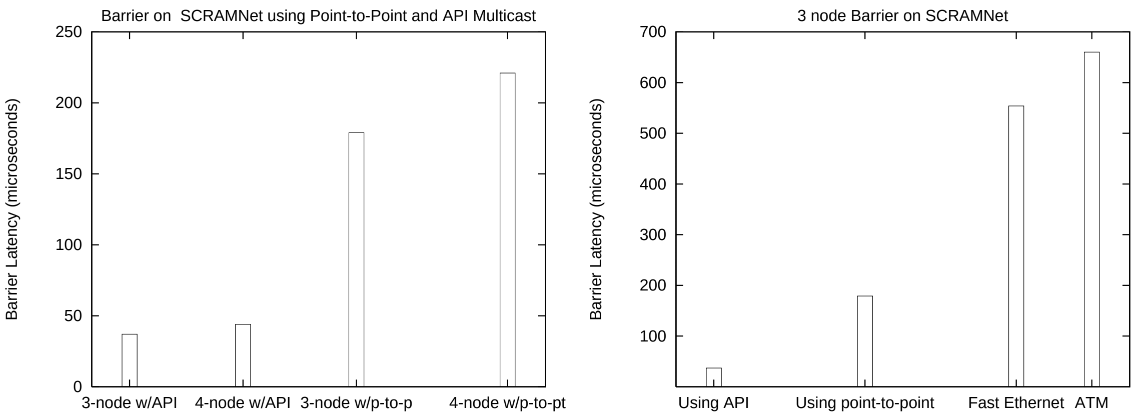

Figure 9: Performance of MPI Barrier on SCRAMNet

7 Related work

There has been recent research to provide low latency communication using using shared memory interconnects, mainly DEC Memory Channel [7, 13] , Princeton SHRIMP [4] and the IEEE SCI (Scalable Coherent Interconnect) [10] standard .

The DEC Memory Channel uses the same technology as the Encore Reflective memory. In Memory Channel, the virtual address space of a process is mapped to the virtual address space of another process so that a store instruction in the physical memory of one system is reflected in the physical memory of another. Sharing of memory in Memory Channel is one-to-one.

The Princeton SHRIMP multicomputer uses the routing network of Intel's Paragon systems and recently, the Myrinet network. It is a point-to-point interconnection and does not provide broadcast/multicast capability at the physical layer. It is not general purpose, as it requires the cache to snoop on DMA writes as well.

SCI provides a way to combine a cluster of processors and keep their caches coherent by providing a cache-based directory scheme. Though not initially developed for clusters, it has been extended to be used for providing CC-NUMA for workstation clusters.

There is also work being done to provide fast collective communication on workstation clusters using hardware support. PAPERS (Purdue's Adapter for Parallel Execution and Rapid Synchronization) [11] use specialized networks, called Bitwise Aggregate Networks, for fast collective communication.

There is work being done on using SCRAMNet as well. [16] from U. of Pittsburgh discusses synchronization mechanisms for using SCRAMNet.

In our research, we have tried to use a generic approach to support both point-to-point and collective communication using SCRAMNet.

8 Conclusions

We have built a message passing system on SCRAMNet. For short messages, SCRAMNet has been shown to perform better than other networks (Fast Ethernet, Myrinet and ATM) for point-to-point communication as well as broadcast and barrier. We have built a rudimentary API on SCRAMNet which provides basic message passing on SCRAMNet with very low latency. We have implemented MPI on top of the API to provide full fledged low latency message passing over SCRAMNet. Although, the results presented for collective operations are for small scale systems, due to lack of resources (PCs and Network Interface Cards) , these initial results promise good improvements for larger scale systems. These directions will help us improve performance of our MPI implementation further. We are also working on using SCRAMNet together with other networks such as Myrinet and ATM to design efficient communication subsystems for workstation clusters which have low latency as well as high bandwidth. The performance results show that SCRAMNet has characteristics complementary to those of existing networks. This makes SCRAMNet a good candidate for use with a high bandwidth network within the same cluster.

We are currently working along two directions to reduce MPI point-to-point latency even further. The first direction is to remove the Channel Interface layer by creating an Abstract Device Interface layer directly on top of the BillBoard API. The second direction is to incorporate an interrupt mechanism to enhance the communication performance for long messages. Currently, our MPI implementation uses polling to check for incoming messages. Polling requires memory access across the I/O bus which increases the receive overhead. Also, currently each layer has to manage received message queues due to the absence of an interrupt mechanism in the BBP layer. With these directions, we plan to develop a faster framework for message passing on SCRAMNet.

References

[1] T. Anderson, D. Culler, and Dave Patterson. A Case for Networks of Workstations (NOW). *IEEE Micro*, pages 54–64, Feb 1995.

[2] ATM Forum. *ATM User-Network Interface Specification, Version 3.1*, September 1994.

[3] Donald J. Becker et al. Beowulf: A Parallel Workstation for Scientific Computation. In *International Parallel Processing Symposium*, 1996.

[4] M. A. Blumrich, C. Dubnicki, E. W. Felten, and K. Li. Protected, User-level DMA for the SHRIMP Network Interface. In *IEEE International Symposium on High Performance Computer Architecture (HPCA-2)*, pages 154–165, 1996.

[5] N. J. Boden, D. Cohen, et al. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, pages 29–35, Feb 1995.

[6] T. Bohman. Shared-Memory Computing Architectures for Real-Time Simulation-Simplicity and Elegance. Technical report, Systran Corporation, 1994.

[7] William M. Cardoza, Frederic S. Glover, and William E. Snaman Jr. Design of the TruCluster Multicomputer System for the Digital UNIX Environment. *Digital Technical Journal*, 8(1):154–165, 1996.

[8] J. Duato, S. Yalamanchili, and L. Ni. *Interconnection Networks: An Engineering Approach*. The IEEE Computer Society Press, 1997.

[9] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI, Message Passing Interface Standard. Technical report, Argonne National Laboratory and Mississippi State University.

[10] David Gustavson. The Scalable Coherent Interface and related Standards Projects. *IEEE Micro*, pages 10–12, February 1992.

[11] R. Hoare, H. Dietz, T. Mattox, and S. Kim. Bitwise Aggregate Networks. In *Symposium on Parallel and Distributed Processing*, 1996.

[12] C. Holt, J. P. Singh, and J. Hennessy. Application and Architectural Bottlenecks in Large Scale Distributed Shared Memory Machines. In *Proceedings of the International Symposium on Computer Architecture*, pages 134–145, May 1996.

[13] James V. Lawton, John J. Brosnan, et al. Building a High performance Message-passing system for MEMORY CHANNEL Clusters. *Digital Technical Journal*, 8(2):96–116, 1996.

[14] P. K. McKinley, H. Xu, A.-H. Esfahanian, and L. M. Ni. Unicast-based Multicast Communication in Wormhole-routed Networks. *IEEE Transactions on Parallel and Distributed Systems*, 5(12):1252–1265, Dec 1994.

[15] Lee Melatti. Fast Ethernet: 100 Mbit/s Made Easy. *Data Communications on the Web*, Nov.

[16] Stephen Menke, Mark Moir, and Srikanth Ramamurthy. Synchronization Mechanisms for SCRAMNet+ Systems. In *Proceedings of the 16th Annual Symposium on the Principlies of Distributed Computing*, July 1998.

[17] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, Mar 1994.

[18] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM). In *Proceedings of the Supercomputing*, 1995.

[19] G. F. Pfister. *In Search of Clusters*. Prentice Hall, 1995.

[20] M. W. Sachs and A. Varma. Fibre Channel. *IEEE Communications*, pages 40–49, Aug 1996.

[21] V. S. Sunderam. PVM: A Framework for Parallel and Distributed Computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[22] Systran Corporation. *SCRAMNET+ Network*, 1994.

[23] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-level Network Interface for Parallel and Distributed Computing. In *ACM Symposium on Operating Systems Principles*, 1995.