Efficient Message Passing Interface (MPI) for Parallel Computing on Clusters of Workstations

Jehoshua Bruck* Danny Dolev† Ching-Tien Ho‡ Marcel-Cătălin Roşu§ Ray Strong‡

*California Institute of Technology
Mail Code 116-81
Pasadena, CA 91125
bruck@systems.caltech.edu

†Institute of CS
Hebrew University
Jerusalem, Israel
dolev@cs.huji.ac.il

‡IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120
{ho,strong}@almaden.ibm.com

§Georgia Institute of Technology
College of Computing
Atlanta, GA 30332-0280
rosu@cc.gatech.edu

Abstract

Parallel computing on clusters of workstations and personal computers has very high potential, since it leverages existing hardware and software. Parallel programming environments offer the user a convenient way to express parallel computation and communication. In fact, recently, a Message Passing Interface (MPI) has been proposed as an industrial standard for writing "portable" message-passing parallel programs. The communication part of MPI consists of the usual point-to-point communication as well as collective communication. However, existing implementations of programming environments for clusters are built on top of a point-to-point communication layer (send and receive) over local area networks (LANs) and, as a result, suffer from poor performance in the collective communication part.

In this paper, we present an efficient design and implementation of the collective communication part in MPI that is optimized for clusters of workstations. Our system consists of two main components: the MPI-CCL layer that includes the collective communication functionality of MPI and a User-level Reliable Transport Protocol (URTP) that interfaces with the LAN Data-link layer and leverages the fact that the LAN is a broadcast medium. Our system is integrated with the operating system via an efficient kernel extension mechanism that we developed. The kernel extension significantly improves the performance of our implementation as it can handle part of the communication overhead without involving user space.

We have implemented our system on a collection of IBM RS/6000 workstations connected via a 10Mbit Ethernet LAN. Our performance measurements are taken from typical scientific programs that run in a parallel mode by means of the MPI. The hypothesis behind our design is that system's performance will be bounded by interactions between the kernel and user space rather than by the bandwidth delivered by the LAN Data-Link Layer. Our results indicate that the performance of our MPI Broadcast (on top of Ethernet) is about twice as fast as a recently published software implementation of broadcast on top of ATM.

1 Introduction

Recently, a Message Passing Interface (MPI) [16] has been proposed as an industrial standard for writing "portable" message-passing parallel programs. The MPI standardization effort involved about 60 people from 40 organizations including universities, national laboratories, and most MPP vendors. Version 1 of MPI was released in May 1994. MPI adopts most, if not all, common practices from existing communication libraries. One of the key components of MPI is the collective communication subset that allows users to conveniently call library routines for various "global" communication operations, like broadcast, scatter and gather. All MPI collective communication routines are implicitly defined with respect to a *process group* [3] which specifies an ordered set of processors within which the collective communication will be performed. For example, a multicast is specified as a broadcast to a particular process group. The performance of a parallel program depends on an efficient implementation of point-to-point as well as collective communication.

In existing parallel programming environments, such as PVM, EXPRESS and IBM's MPL [12, 20, 2], for Local Area Networks (LANs), collective communication routines are implemented on top of point-to-point communication. As a result, these environments suffer from poor collective communication performance. For example, a broadcast that is implemented using a TCP or point-to-point UDP over a LAN is obviously inefficient as it is not utilizing the fact that most LANs are based on a broadcast medium.

In this paper, we present an efficient design and implementation of the Collective Communication Library in MPI (MPI-CCL) that is optimized for clusters of workstations. In particular, we demonstrate the implementation on a traditional 10Mbit Ethernet-based LAN. We note here that the ideas presented in this paper can be easily extended to any Network of Workstations (NOW) [21] that provides an unreliable broadcast transport protocol (such as to an ATM network where the ATM switches have broadcast capability as provided by many vendors nowadays).

Our main contributions in this paper are:

- We have designed and implemented a system that consists of two main components: the MPI-CCL layer that delivers the collective communication functionality of MPI and a User-level Reliable Transport Protocol (URTP) that interfaces with the LAN Data-link layer.

- We provide a formal specification of the semantics of the user parallel program that is required for the correctness of our MPI implementation. This property is essentially that a parallel program can be described as a single *global program* running on multiple processors. The implementation of MPI-CCL and URTP makes use of the global program semantics.

- We have incorporated a number of novelties in our protocols. Our URTP is based on a sliding window protocol with efficient acknowledgement and also makes use of a novel (pessimistic) immediate request protocol. Our request protocol is the only place where we actively retry to provide reliable communication. The global program semantics makes this new approach sufficient.

- Our system is integrated with the operating system via an efficient kernel extension mechanism that we developed. The kernel extension significantly improves the performance of our implementation as it can discard unnecessary request messages without involving user space. It also discards broadcast messages for which the local processor is not in the intended multicast target.

- We provide performance measurements of our prototype implementation. The measurements are taken from typical scientific programs that run in a parallel mode by means of the MPI. Thus, they can be directly compared with performance measurements taken from other implementations of the MPI. Since our emphasis is on collective communication, we measured programs that make heavy use of collective operations, including, (1) straightforward matrix-matrix multiplication, and (2) iterated multiplication of a vector by a sparse matrix. The latter is part of the differential equation solver of a computer aided engineering application.

- The hypothesis behind our design is that performance will be bounded by interactions between kernel and user space rather than by the bandwidth delivered by the LAN Data-Link Layer. The validity of this hypothesis clearly depends on both the reliability of the LAN Data-Link Layer and the patterns of communication generated by parallel applications. Our results indicate that the performance of our MPI Broadcast (on top of Ethernet) is about twice as fast as the software implementation of broadcast on top of ATM that is presented in [14]. For example, a broadcast of a 4Kbyte message on 8 machines takes about 6 msecs in our implementation compared to 15 msecs in the implementation in [14].

We note there have been many existing systems, such as [4, 23, 1], that provide a reliable transport protocol and other services for distributed computing. Our URTP protocol distinguishes itself from previous ones because it is targeted for supporting parallel computing using MPI programs and can take advantage of the global program semantics derived from our MPI-CCL implementation.

Note also that there have been various works in improving communication latency and bandwidth by modifying message passing protocols to facilitate efficient system implementation. Examples are the *Active Message* project by Culler et al. [9], the *Fast Message* project by Chien et al. [19], and the *Shrimp* project by Li et al. [15]. In this paper, we address the same issue using a different orthorgonal approach. Thus, we do not claim that our approach should replace any work in this area. Rather, our approach can be integrated with many proposed system designs in improving message passing protocols on networks of workstations.

The rest of the paper is organized as follows. In Section 2 we describe our system architecture. In Section 3 we present the design, prototype implementation and performance measurement of our URTP protocol. In Section 4 we describe our design, implementation and performance measurement of our MPI-CCL layer. We then present performance measurements of two MPI programs in Section 5. Finally, Section 6 concludes the paper. We also include pseudocode for some key parts of the URTP protocol in the appendix.

2 The System Architecture

In this section we will overview our system architecture. We are interested in efficient MPI implementation on a LAN. The system consists of an MPI program, typically SPMD (Single Program Multiple Data) style, running on a cluster of processing elements, each with its own local memory. Processors communicate with each other via asynchronous and unreliable point-to-point or broadcast packets, as supported by the LAN Data-Link layer. Although we expect that some packets might be lost, we assume that the content of a received packet is not corrupted beyond the tolerance of standard error correction.

2.1 The Software Layers

A process has four logical layers of software (see Figure 1). The lowest layer is a LAN Data-Link Layer (typically Ethernet) that interfaces the LAN. The second layer (from the bottom) is our URTP protocol layer which guarantees reliable point-to-point and reliable multicast to the upper layer. The third layer is our MPI-CCL layer which maps all MPI-CCL routines into the interface supported by URTP. It also deals with packetization of user messages and scheduling of packets. The highest layer is the MPI application program layer. In this paper we provide efficient design, implementation, and performance measurement of the two middle layers (URTP and MPI-CCL). Note that in the MPI-CCL layer, we focus on the collective communication subset of MPI only. For MPI point-to-point communication routines, they can be easily mapped to reliable point-to-point communication provided by URTP or similar protocols (e.g. [11]).

For clarity, here we define a few terms that are used throughout this paper. A message, which has no upper bound on size, is the unit of communication at the MPI applications layer. A packet, which has an upper bound of about 1.5 KBytes, is the unit of communication at the MPI-CCL layer or URTP layer. Specifically, multicast, send, receive and multireceive calls, issued from MPI-CCL layer to URTP layer, are all with respect to packets. (Note that multireceive is defined as receiving the "next" packet from each processor in a specified set, in arbitrary order.) We sometimes refer to the packets at the URTP layer as *URTP packets* for clarity. A URTP packet is either a *data* packet or a *control* packet.

2.2 The Global Program

Here we provide a formal specification of the semantics of user parallel programs that is required for the correctness of our MPI implementation. This property is essentially that a parallel program can be described as a single *global program* running on multiple processors. The implementation of MPI-CCL and URTP makes use of the global program semantics.

There are two main properties guaranteed by the global program semantics. First, each node knows the ordering of expected packets from every other node. Second, it also guarantees freedom from deadlock due to lack of system buffer space. We will now describe the semantics of the global program that is generated by our MPI-CCL layer. The global program can be viewed as a function, taking a *pid* and an instruction counter as parameters, mapping to one of the 3 possible calls: *multicast*, *receive* and *skip*. A *multicast* call takes as argument a set of destination processors called the *target set*. (For simplicity, we treat a point-to-point *send* as a multicast with one destination in the target set. This is only for specification purposes and does not determine the implementation.) A *receive* takes as argument one explicit *source* processor.[1] In the most strict definition, we say a global program is correct if, for *each* instruction counter, there is at most one *multicast* and exactly the processors in the target set issue *receive* with the source matching the multicast source. (See [6] for a more formal and detailed definition.) Note that the skip call, the instruction counter and the restriction to one *multicast* call per instruction call are all introduced for the purpose of specification. In particular, each processor will execute its program in an asynchronous and greedy manner. Note that having a correct global program (and the fact that multicast/send and receive always match) reduces dependency on the number of buffers in the system, and avoids deadlock due to exhaustion of resources.

For performance reasons, we will relax and generalize our definition of correct global program as follows: A global program is *k-buffer correct* if (i) there exists a uniform grouping of instructions such that every group spans at most k instructions and (ii) within each group all *multicast* and *receive* calls are matched completely and consistently. Clearly, our original semantics of correct global program becomes *1-buffer correct* in the generalized semantics. If we assume (for the moment) that the LAN Data-Link Layer is completely reliable, then it is easy to provide a deadlock free implementation for any k-buffer correct program using system buffers for k packets per processor.

As an example, consider an implementation of all-to-all broadcast among p processors. With the 1-buffer correct semantics, *multicast* must be called by the p processors in a round-robin manner. With the p-buffer correct semantics, each of the p processors can issue a *multicast* followed by $p-1$ appropriate *receive* calls in arbitrary order. Clearly, p buffers for p packets at each processor are sufficient for this purpose.

It should be noted that the users do not need to worry about whether their MPI programs are k-buffer correct or not. Our MPI-CCL layer, which takes MPI programs with collective communication calls, is implemented to generate k-buffer correct global programs for some carefully chosen k. Note also that we do not change any semantics of MPI point-to-point communication calls by users. For instance, wildcard receives are still allowed by users. This is because implementation of our MPI-CCL layer can be made "disjoint" from that of MPI point-to-point communication routines.

3 The URTP Protocol

Present LANs have a very low packet loss rate due to media errors. The sender side provides a highly reliable data transfer path to the communication medium. The vast majority of packet loss is due to the lack of buffers at the receiver side(s). URTP was designed to provide a reliable transport protocol with point-to-point and multicast capabilities and with performance as close as possible to that of the hardware it is running on.

In this section we present our key approaches to this problem. Briefly, they are:

1. Build URTP on top of the lowest available layer (LAN Data-Link Layer in our case).

2. Take advantage of the multicast/broadcast capabilities offered by the lower layers.

[1] That is, the *wildcard* source is not allowed. See [3] for details.

3. Move the packets from kernel buffers into user level buffers and free kernel buffers as soon as possible. This minimizes the chances that packets are dropped due to the lack of free kernel buffers. There are two main differences between kernel and user buffers. First, kernel buffers occupy physical (non-pageable) memory and user buffers occupy virtual (pageable) memory. Second, kernel buffers are a shared resource for the communication needs of all processes on a machine; user buffers are managed by individual user's application.

4. Drop unwanted packets as soon as possible and with a minimum overhead.

As a result URTP is implemented as a combination of a kernel extension and a user-level library. Most of the protocol code is in the user-level library. This decision made the implementation easier without a significant performance degradation (see [17]). The kernel extension part enables fast processing (dropping) of multicast packets at processors that are not part of the target set[2] and also reduces the inter-processor communication overhead between processes.

3.1 Protocol Description

The issues and requirements that we consider while designing the protocol are as follows:

1. Packet ordering properties: we require reliable point-to-point and multicast FIFO. Point-to-point and multicast packets respect the same order. The fact that a packet was sent by a point-to-point send or a multicast is invisible to the layers above URTP[3]; the same call is used to receive it.

2. Buffer management on the sender side: since the broadcast domain in the LAN Data-Link layer is not reliable, the sender keeps a copy of each packet sent, until every processor in the target set has (implicitly) acknowledged its receipt.

3. Buffer management on the receiver side: since the broadcast domain is not reliable, some packets may be lost. For performance reasons we buffer all "useful" data packets at the receiving side, even when a gap is detected.

4. Detection of packet loss on the receiver side: given the semantics of the global program we can detect a packet loss (or delay) when a *receive* is issued from the MPI-CCL layer. In addition, packet loss can be detected by violation of FIFO ordering at the receiver.

URTP uses a modified version of the sliding-window protocol. Since we expect to have many groups of varying sizes it is important to have sequence number management associated with every pair of processors. Namely, every sender-receiver pair has a counter associated with it. Because every processor can act as a sender and receiver there will be two counters for every processor pair. Point-to-point packets will contain the current value of the counter associated with the $(sender, receiver)$ pair. Multicast packets will contain a counter value for

[2]URTP multicast is implemented using the underlying LAN broadcast and multicast capabilities. For reasons to be explained later, a processor not in a target group might also receive packets for the group.

[3]For instance, URTP uses a point-to-point send to resend a multicast packet upon getting REQ from some processor in the target set.

every processor in the target set. These numbers correspond to the pairs $(sender, receiver_i)$ for each $receiver_i$ in the target set. After a packet is sent all the counters whose values were used in the packet are incremented by one. As a result, the numbers used for point-to-point and multicast packets sent from processor i to processor j are generated by the same counter. All the necessary packetization is handled by the upper (MPI-CCL) layer; a URTP packet always translates into exactly one LAN packet.

The MPI-CCL layer manages the sending buffers. URTP is passed a pointer to the buffer containing the packet to be sent/multicast together with a call-back function. The function will be called (to free the buffer) when acknowledgements for the packet are received from all the destination processors in the target set (point-to-point packets have only one destination); the address of the buffer is passed as a parameter. For a good performance, the recommended size of the sending buffer pool in MPI-CCL is $(p-1)W$, where p is the number of machines in the configuration and W is the window size. This guarantees that the sender won't ever run out of sending buffers. All those buffers are in user space (pageable memory) and layers using URTP can implement their own buffer management policies. To prevent deadlock, each processor must be able to increase its sending buffer pool to the recommended size.

The receiving buffers are managed by URTP. For simplicity, current URTP implementation allocates $(p-1)W + c$ buffers, where p and W are defined as before and c is a small constant. Among the buffers, $(p-1)W$ are required for the URTP data packets and c are used for the URTP control packets (e.g. ACK, REQ). (Other implementations using fewer buffers are possible, such as having a "reserved buffer" for the critical packet from each processor.) A pointer to the received packet and a call-back function are passed to the MPI-CCL layer as the result of a receive call from MPI-CCL. The call-back function must be called by MPI-CCL layer after it finishes using the buffer (i.e., processing the packet). The buffers are all in user space (pageable memory) and are the size of the maximum LAN packet. A packet can be acknowledged as soon as it reaches a URTP buffer. If the number of received and unacknowledged packets reaches a threshold then an ACK packet will be sent to the sender (as in [5]). As already evident, the receive call has a rather unusual semantics: data is not returned in a buffer supplied by the MPI-CCL layer. A pointer to a buffer in the receiving buffer pool is returned instead. The packet is copied twice before it reaches the URTP buffer: first from the network adapter card memory into a kernel buffer and second from the kernel buffer to the URTP buffer. (The assembled message will be further copied from URTP buffers into the user's buffer in the MPI program by the MPI-CCL layer.) A protocol using an average of little more than one data copy per message is described in [7] but it works only for large data transfers. URTP is intended to be used for both small and large data transfers.

In addition, URTP has the following mechanisms:

1. A REQ packet is a point-to-point communication requesting a specific packet from a source. It implicitly acknowledges all previous packets.

2. When a processor receives a REQ packet for a data packet which has been sent earlier, it sends the requested data packet again using point-to-point send (even if the original data packet was sent through multicast). When the REQ refers to an unsent data packet (in which case the receiver is ahead of the sender) the REQ is simply ignored.

3. A timeout mechanism is used to ensure the delivery of a REQ packet and of the requested data packet. The timeout process stops when the requested packet arrives.

4. ACKs and REQs are sent using point-to-point communication.

5. Security is guaranteed (within the limits imposed by the LAN). URTP packets cannot be received or sent by processors that are not in the process group defined for the "MPI world". Because URTP is intended for parallel applications running over local (and relatively secure) networks no additional security enforcing mechanisms were implemented.

In what follows we include some of the details of the implementation and optimizations related to the URTP protocol.

3.2 Implementation and Optimizations

The system architecture of URTP is both simple and efficient. It is built on top of a LAN protocol (Ethernet) and it is entirely software based. It consists of two separate modules: a kernel extension and a library that needs to be linked to every application using the protocol (see Figure 2).

We consider that current LANs are *almost* completely reliable and that most instances of packet loss are due to a lack of free buffers at the receiving end. In the following subsections we present our implementation and optimization of URTP, a reliable transport layer with very little performance degradation.

3.2.1 The Kernel Extension

The kernel extension module contains a customized device driver (an extension to the existing one) and code that implements a number of new system calls. The device driver is modified in order to minimize the overhead of dropping unwanted packets. The new system calls are used by the user level library to transfer data between the kernel and the user process.

There are two sources of unwanted packets. One is that a URTP multicast is implemented using a LAN multicast to a superset of the target set so that any processor in this superset receive the packet whether it is a packet destination or not. The other is that when the user application issues a *receive* and the corresponding packet is not waiting in a user buffer, our pessimistic request protocol assumes that the packet may have been lost and sends a REQ packet to the source. If the source is behind the receiver or has just sent the desired packet, then the REQ packet is unwanted.

We implement multicasts as LAN multicast to a superset of the target set because of the limited number of multicast addresses a network interface can listen to (about a dozen). In an MPI program, a processor can be a member of many more groups. Moreover, there is significant overhead associated with the creation of a hardware group containing multicast target set. This operation involves selecting an acceptable multicast address, adding it to the set of addresses recognized by each member of the new group, and acknowledging the success of the operation. About the same overhead is necessary for group removal. While this overhead might be acceptable for heavily used groups (with a lot of packets targeted to that group) it is not acceptable for groups that are rarely used.

The header of a multicast packet contains a sequence number for every processor in the hardware group (which contains the target set). Entries corresponding to machines in the target set contain valid sequence numbers. Entries corresponding to machines in the hardware group but not in the target set contain an invalid sequence number. When a multicast packet is received, the processor checks the validity of the sequence number corresponding to its rank in the target group. If an invalid value is found the packet is dropped (and the kernel buffer is freed). Otherwise the packet is put into a kernel queue. Because the overhead is small the above action can be executed in the interrupt handler.

The other source of unwanted packets is the *receive* call. When a receive is posted from the upper layer at processor j for the next packet from processor i, and the packet has not been received by the URTP at processor j, a REQ is sent to processor i and a timeout mechanism is initiated. The response at processor i, upon receiving a REQ from processor j, has a few possible cases:

1. The requested packet was sent more than δ milliseconds ago. The sender i assumes that the packet was lost and sends it again. The REQ is not considered an unwanted packet. We selected a value of 1 for δ based on our experiment in optimizing all-to-all broadcast.

2. The packet was sent less than δ milliseconds ago. The sender assumes that the packet is on its way and considers the REQ an unwanted packet.

3. The packet has not been sent or multicast from the upper layer yet (the sender is behind the receiver). The REQ is considered an unwanted packet.

For case 3, the REQ is ignored and treated as a unwanted packet and dropped while still in a kernel buffer. The kernel extension has a local copy of the sequence number of the last packet sent to each destination. This copy is updated only when a new packet is sent. Using it the device driver can detect with very little overhead when a REQ packet refers to an unsent packet (case 3 above). When this case is detected the packet is dropped (and the kernel buffer is freed). This action is done in the interrupt handler.

Case 2 is rare. When it occurs the REQ packet is referred to user space where the time comparison is performed and the packet dropped.

In case 1, the REQ packet is also referred to user space where the requested packet resides. There the REQ is treated as an implicit ACK of all previous packets from the same source. Then the requested packet is resent.

The Ethernet implementation of the device driver is presented next. The device driver interfaces with the network interface card and it is extended to handle packets of a special type (that we defined) called URTP. Those URTP Ethernet packets are identified by the type field in the header of the Ethernet packet that is set to a specific value. This value is different from that used by IP, ARP, etc.

Every packet with an Ethernet address (unicast, multicast or broadcast) recognized by the interface hardware is copied from the network interface memory into a kernel buffer. We note here that URTP packets have the same structure as IP packets: data is preceded by the header.

The received packet is processed as follows:

1. Non-URTP packets (IP, ARP, etc.) are handled as usual.

2. For URTP type packets the driver checks the URTP header. ACK and point-to-point packets are always linked into a kernel queue. Based on the content of their header, multicast and broadcast packets are either considered unwanted and dropped (as explained

above) or linked into the same kernel queue. REQ packets that are not dropped are linked into the same queue.

3. Whenever a packet is linked into an empty queue, a signal is sent to the destination process. The signal handler, part of the user level library, uses one of the new system calls to transfer *all* the packets from the kernel address space into the destination process address space. All the packets are transferred in one system call (only one context switch to kernel mode and one back to user mode).

4. The queue is accessed by the device driver (to enqueue packets) and by the new system call (to dequeue packets). In order to minimize the overhead of the synchronization, two queues are used; one for queuing packets and one for dequeuing packets. When the first queue is non-empty and the second one is empty the queues are interchanged. The interrupt level associated with the Ethernet card is disabled *only* during the switch.

5. When a *receive* is posted the packet is either already in user space (and a pointer to the buffer is returned) or it was not received yet; in this case the user's process sends a REQ and blocks. It is not possible for the packet to be in kernel space for the following reason: after the interrupt handler returns, the control returns to a user process. If there are any packets in the kernel queue to be delivered to a process then a signal has already been sent to the process. The first code that will be executed when the process is scheduled is the signal handler that will transfer the packet into user buffers.

We do not implement any CRC checking mechanism. This relieves both the sender and receiver from an unnecessary effort. Because every URTP packet is transferred using exactly one Ethernet packet we silently use the Ethernet CRC check. The advantage of this approach is that both CRC computations are done by the network interface card.

The process of sending a packet is even simpler. The kernel extension contains new system calls for each packet type: new point-to-point packet, retransmission of a previous packet, multicast packet and control packet (ACK, REQ, etc.). Each of them allocate a kernel buffer, build the Ethernet header, copy the data from user space and place the buffer in the send queue of the driver. At this point it is evident that our protocol does much more work at packet reception than at packet transmission.

3.2.2 The User Level Library

The kernel extension presented above implements an unreliable transport layer. In addition, it efficiently drops unnecessary packets. The main goal of the library is to make it reliable by implementing the modified sliding-window protocol. In addition, we have the following mechanism related to REQ packets.

1. After sending a REQ packet the *receive* routine goes to sleep for a determined period. When it wakes up another REQ packet will be sent if the packet was not received. Signals generated by incoming packets wake up the receive routine. When the desired packet arrives, the routine returns.

2. REQ packets also act as implicit ACK packets. A request for packet n means that all unacknowledged packets up to $n-1$ were safely received.

3. The above scheme seems to use a lot of REQ packets. During the same *receive* call the time-out period is increased exponentially. This reduces the number of REQ packets and consequently, the load on the net.

4. The copy of the sequence number of the last packet sent to every destination maintained inside the kernel extension facilitates the dropping of almost all unnecessary REQ packets at the driver level. As a result, the overhead at the destination of an unnecessary REQ packet is very small.

Since we assume that packet loss is primarily due to lack of free receiver buffers, our kernel extension attempts to free kernel buffers as soon as possible. It also attempts to minimize the number of context switches required to transfer packets from the kernel to user space. We considered the alternative of changing the kernel buffers (mbufs) management or using special buffers that can be switched from the pageable to non-pageable state; but we believe that the architecture presented above is superior because the packets must be eventually transferred to user space in any case.

3.3 Performance of URTP

The main objectives of our design and implementation of URTP with respect to performance are low processor overhead, low latency and high effective bandwidth. To reduce the overhead at the sender side, we drop the too-early REQ packet in kernel space (case 3 described in Section 3.2.1). To reduce the overhead at the receiver side, we drop the unnecessary multicast packets (for which the receiver is not in the target group) in kernel space. Figure 3 shows the overhead in dropping an unnecessary packet at the receiving side as a function of packet size. The URTP header is accounted for in the packet size. Note that there is a jump in the figure for the following reason. For every Ethernet packet received, the kernel allocates an *mbuf* to store it. The size of an *mbuf* is 256 bytes but only about half of it is used to store incoming data. When packets are too large to fit into an *mbuf*, an extension (a page of 4K bytes) is allocated and linked to the *mbuf*. We also measured the overhead to receive a UDP packet up to the user level to be between 400 and 600 μsecs. The overhead to receive a URTP packet decreases if many packets are transferred from kernel buffers to user space in the same system call. Even for the worst case, where only one packet is transferred in a system call, the overhead (not including the processing associated with the sliding window protocol) is about 100 μsecs less than that of UDP (which is unreliable transport). Figure 4 shows the one-way latency of a packet between two processors, measured at the user level.

We aim to keep the number of control packets low, for instance, by using block ACKs and by using REQs for implicit ACKs of earlier packets. For a message of size 1MBytes or more, we have measured an effective URTP bandwidth of around 8.8 Mbits/sec, which is quite high for a reliable transport protocol, compared to the 10 Mbits/sec raw Ethernet bandwidth as an unreliable transport protocol. The bandwidth of the protocol was measured by transferring a large amount of data between two processors, say i and j, and back. We divide the total amount of data transferred (both ways) by the elapsed time between the time we send out the first packet from processor i and the time the last packet was received by processor i. We use URTP point-to-point transfer with maximum size packets. The data used to compute the effective bandwidth does not include any Ethernet or URTP overhead; it is the "effective" user data only.

The discrepancy of effective URTP bandwidth from raw Ethernet bandwidth (around 10 Mbits/sec) mainly comes from three sources: (i) the need of certain control packet (such as ACK and REQ) in order to build a reliable transport protocol; (ii) the additional header information of URTP packet carrying data (such as the URTP type, counters, etc.); and (iii) the overhead from Ethernet packet (such as the sync bits, Ethernet header and trailer, etc.).

4 MPI-CCL

In this section, we describe the algorithms and implementation of selected MPI-CCL routines on URTP. We also present performance results of our implementation.

We have implemented on URTP a selected subset of MPI-CCL routines: MPI_Bcast (one-to-all broadcast within a group), MPI_Allgather (each node broadcasts a message to all nodes within a group), MPI_Gather (gather generally distinct messages from all nodes to one node in a group), MPI_Scatter (scatter generally distinct messages from one node to all nodes in a group) and MPI_Barrier (barrier synchronization within a group). We also implemented MPI_Init and MPI_Finalize to initialize and terminate our URTP environment properly. The general goal of this MPI-CCL layer is to map MPI-CCL routines onto the URTP interface in an efficient way, and to break (or sometimes pack) messages into packets. For the interface to URTP, we use *multicast*, *send* and *receive*, as defined before. In addition, we also use *multireceive* in which a processor blocks until the next packet from each processor in a specified set is received.

Throughout this section, we let M be the message size of the send buffer or receive buffer, whichever is smaller. For instance, M is the size of the user *send* buffer for MPI_Allgather and MPI_Gather, and is the size of the user *receive* buffer for MPI_Scatter. We let m be the maximum MPI-CCL packet size (which is the data size a maximum URTP packet can hold) and p be the number of processors in the processor group.

4.1 The Environment

All timing measurements are averaged over all processors that call the MPI routine and over a few runs. For each run, we further take the average of at least 10 iterations of the considered MPI routine. For MPI_Bcast, MPI_Scatter and MPI_Gather, we round-robin the root for different iterations. The 8 workstations that we used for our experiments are IBM RS/6000 workstations: 3 with model 320 (20MHz clock), 2 with 530H (33 MHz clock), 1 with 375 (62.5 MHz), and 2 with 250 (66 MHz clock). They are all on the same sub-network with 10Mbit Ethernet. Whenever we use fewer than 8 workstations, choosing the *slow subset* (respectively, *fast subset*) means choosing the required number of workstations starting from the slowest (respectively, fastest) one.

4.2 MPI_Bcast

The mapping of MPI_Bcast to URTP interface is straightforward. The root calls *multicast* $\lceil M/m \rceil$ times, while all other processors in the processor group call *receive* $\lceil M/m \rceil$ times.

Figure 5 shows the measured times of MPI_Bcast as a function of the message size. Note that once the message size is greater than 1 Kbyte, doubling the message size roughly doubles the number of packets, and hence the broadcast time. From the figure, broadcasting one packet takes about 1 to 4 msecs.

The performance of our MPI Broadcast (on top of Ethernet) is about twice as fast as the software implementation of broadcast on top of ATM that is presented in [14]. For example, a broadcast of a 4Kbyte message on 8 machines takes about 6 msecs in our implementation compared to 15 msecs in the implementation in [14]. The hardware implementation of [14] is faster than ours as would be expected from the higher bandwidth of the ATM equipment. As another comparison, our broadcast of a 1 Kbyte message on 8 processors takes about 3.87 msecs as compared to 8.9 msecs based on TCP/IP measured in [6].

4.3 MPI_Allgather

Let $q = \lceil M/m \rceil$ be the number of packets, per processor, that need to be multicast. There are two possible algorithms for MPI_Allgather described as follows.

Algorithm 1: Round-robin multicast.

$i = mypid$;
srcs1 $= \{0, 1, \cdots, i-1\}$;
srcs2 $= \{i+1, i+2, \cdots, p-1\}$;
dests $= srcs1 \cup srcs2$;
for $(j = 0;\ j < q;\ j{+}{+})$ {
    multireceive (srcs1, recvbufs, j);
    multicast (dests, sendbuf, j);
    multireceive (srcs2, recvbufs, j);
}

Algorithm 2: k-concurrency multicast.

dests $= \{0, 1, \cdots, p-1\} - \{mypid\}$;
$j1 = 0$;
$j2 = 0$;
for $(j = 0;\ j < \lceil q/k \rceil;\ j{+}{+})$ {
    for $(l = 0;\ l < k;\ l{+}{+})$ {
        if $(j1 < q)$ {
            multicast (dests, sendbuf, $j1$);
            $j1{+}{+}$;
        }
    }
    for $(l = 0;\ l < k;\ l{+}{+})$ {
        if $(j2 < q)$ {
            multireceive (dests, recvbufs, $j2$);
            $j2{+}{+}$;
        }
    }

}

In Algorithm 1, each processor takes turn as a broadcaster for each packet while all other processors receive. Successive receives are combined into a multireceive call. In Algorithm 2, each processor issues k multicast calls followed by k multireceive calls. Here, k is a carefully chosen positive integer. Algorithm 2 is kp-buffer correct. Thus, at least a buffer of kp packets in the URTP is required to avoid possible deadlock.

We have implemented both algorithms and observed that Algorithm 2 (with $k = 1$) generally performs better than Algorithm 1 by a factor of around 1.5 to 2. This is mainly because Algorithm 2 has less synchronization points than Algorithm 1 and, therefore, has less chance of being in a processor idle state. Figure 6 shows the measured times of MPI_Allgather as a function of the message size based on Algorithm 2 with $k = 1$. Note that on 8 workstations, MPI_Allgather runs about 6.1 msecs and 12.7 msecs, for 32 bytes and 1K byte messages, respectively. As a comparison, a hand-coded all-to-all broadcast based on the PCODE protocol in [6] runs about 9.0 msecs and 16.7 msecs for 20 bytes and 1 Kbyte messages, respectively, on faster (100 MHz clock) workstations. Figure 7 shows the times of MPI_Allgather of a 64 Kbyte message as a function of the number of workstations using slow subset.

Figure 8 shows the times of MPI_Allgather of a 64 Kbyte message as a function of k (defined earlier) on 4 workstations. It can be seen from the figure that the optimal value of k is around 4, and there is a significant improvement by increasing k from 1 to 2. We also observed from our experiment that the optimal value of k generally decreases as the number of processors increases, as expected. We expect to further reduce MPI_Allgather time by fine tuning k.

4.4 MPI_Scatter and MPI_Gather

MPI_Scatter and MPI_Gather are the dual operations of each other. Assume p is the number of processors in the given processor group. In MPI_Scatter, the root has an array of p blocks of data of the same size initially and wishes to distribute the i-th block of data to the i-th processor in the group. In MPI_Gather, each node in the group has a block of data, all of the same size, initially. The goal is to collect (concatenate without reduction) all p blocks into the root. On most parallel systems, they are implemented with similar algorithms, running in reverse of each other. Since we have a *multicast* interface, MPI_Scatter for small messages is implemented by packing them into one packet and multicasting the packet to all related processors. MPI_Scatter for large messages is implemented by sending multiple point-to-point packets to each node. The exception is that the last point-to-point packet for each node can be packed together and be multicast to all related processors.

Figure 9 shows the measured times of MPI_Scatter on 4 and 8 workstations as a function of the message size. Note that for small message sizes, MPI_Scatter is implemented by packing them into one packet and multicasting the packet. Thus, the measured times stay pretty flat for small message sizes, and grow exponentially for large message sizes when they are doubled the size. Figure 10 shows the measured times of MPI_Gather on 4 workstations as a function of the message size.

Intuitively, one would expect MPI_Scatter to perform better than MPI_Gather for small message sizes. This is because for small message sizes, the total number of packets put on the LAN by MPI_Scatter is less than that by MPI_Gather. However, our experiments show that MPI_Gather performs better than MPI_Scatter for small message sizes. This is because a processor in repeated MPI_Gather calls does not need to wait for any other processors to complete when it is not the root. On the other hand, a processor in repeated MPI_Scatter calls needs to wait for the message from the root.

4.5 MPI_Barrier

MPI_Barrier is implemented by MPI_Gather with 0-byte message to processor 0 followed by MPI_Bcast with 0-byte message from processor 0. Figure 11 shows the measured times of MPI_Barrier as a function of the number of workstations. In the figure, the slow subset uses the slowest processor as processor 0, while the fast subset uses the fastest processor as processor 0. It is evident that, with our implementation, the processing speed of the root (processor 0) affects the timings mostly. The timing is very flat for up to 8 processors.

4.6 MPI Initialization and Termination

According to MPI specification, MPI_Init() must be called before any MPI routines and MPI_Finalize() must be called after any MPI routines. Also, a system-defined *communicator* (see [16] for details) called MPI_COMM_WORLD is defined after MPI_Init call. From MPI_COMM_WORLD, one can derive the *processor group* of the "MPI world", denoted MPI_GROUP_WORLD in the paper. Our MPI-CCL layer has to implement MPI_Init and MPI_Finalize to handle MPI initialization and termination properly.

MPI_Init is implemented in our MPI-CCL layer as follows. First, a fixed number of processors p, specified by the user, is chosen from a set of "pre-registered" workstations and each processor in the MPI_GROUP_WORLD agrees on their pids and the value p from a common file. Then, each processor sleeps for p seconds and calls MPI_Barrier within the MPI_GROUP_WORLD using the URTP protocol. Note that the MPI_Barrier call in MPI_Init is the first time we use the URTP protocol. The purpose of the sleep is to minimize the loss of initial URTP packets at receivers that are not yet ready.

To implement MPI_Finalize, we first issue an MPI_Barrier call within the MPI_GROUP_WORLD in the MPI-CCL layer. Then, each processor sleeps for p seconds and terminates itself. Note that as long as one processor finishes (i.e., returns from) the MPI_Barrier call in MPI-CCL, we can conclude that all other processors have at least reached (not necessarily finished) the same MPI_Barrier call. However, for all processors to terminate normally, each processor needs to "know" that all its URTP data packets (generated from MPI_Barrier) have reached their destinations safely before it terminates the process. This termination problem is a variant of the well-known Two Generals Problem, which is unsolvable in the presence of the possibility of message loss [13, 22]. In fact the problem is harder than the consensus problem because it cannot be solved, even when it is guaranteed that no process will fail (cf. [10]). Fortunately, we do not need to solve this problem because we are assured that all the useful work of the application has been performed if only one processor successfully returns from the MPI_Barrier call.

In practice, we have observed that having each processor sleep for p seconds after MPI_Barrier is sufficiently long to provide a very high probability of normal termination for all processors.

5 Performance of MPI Applications

There have been lots of activities in porting real sequential applications as well as porting existing parallel applications written on various message passing libraries onto MPI. Some such existing and non-standard message passing libraries include Express, PVM, p4, PARMACS, TCGMSG, Chamelion, Zipcode and various machine-specific ones, such as the Intel NX (for Paragon), TMC CMMD (for CM-5) and IBM MPL (for SP-2). We have ported two sequential programs into parallel programs written in MPI. The first one is PolyFEM, a simulation and modeling program that uses p-type finite-element-method algorithms for elasticity modeling [18]. We have isolated and parallelized the portion of the PolyFEM solver that takes most of the running time on large problems. This subapplication involves iteratively multiplying a vector by a large sparse matrix. A typical PolyFEM problem has a vector length about 110000 elements and the sparsity of the matrix is about 0.02%.

Since the minimal required memory to run PolyFEM is too large for our available environment, we have reduced the vector length by factors of 25 and 100, respectively, for our experiments. Figure 12 shows the running times of one iteration (taken from the average of 100 iterations) on 1, 2 and 4 processors, respectively, for two different problem sizes. Figure 13 shows the corresponding speedups. It is expected for this problem that as the problem size increases, the percentage of communication decreases and therefore the speedup increases. Note that the timing for one processor is for the original sequential code (not the parallel code running on one processor).

The second application is a dense matrix-matrix multiplication: $C \leftarrow A \times B$. We assume that the two input matrices A and B are partitioned into p column blocks (where p is the number of processors) and the i-th block is allocated to processor i initially. Also, the final matrix should be distributed in a similar manner. In order for processor i to compute its final submatrix, the i-th column block of C, it needs the entire matrix A. Thus, an MPI_Allgather is required among the p processors.

Figure 14 compares the measured times of the dense matrix-matrix multiplication using two implementations: one is written in MPI and runs on our MPI-CCL/URTP/LAN environment, and another is written in IBM MPL and runs on UDP/LAN environment. Due to some practical constraint, the former was measured on a set of slower and heterogeneous IBM RS/6000's while the latter was measured on a set of faster and homogeneous IBM RS/6000's. The difference in the raw processor speed is evident from the one processor case from the figure. However, even with the disadvantage of processor speeds and heterogeneity, the timings on 2 and 4 processors based on our new URTP protocol are still faster than their corresponding timings based on IBM MPL using UDP.

Figure 15 shows the running times (taken from the average of 100 runs) on 1, 2 and 4 processors, respectively, for three different matrix sizes. Figure 16 shows the corresponding speedups. As before, the one processor case is measured from the original sequential code. Note that for $N \times N$ matrices, the communication cost per processor grows as $O(N^2)$ and the total computation cost grows as $O(N^3)$. Clearly, the speedup increases as the matrix size increases. In fact, some data points in the speedup figure are better than linear speedup probably due to substantially fewer cache misses and page faults with more processors.

6 Conclusion

We have described a way to perform parallel computation using the MPI over a reliable software package that allows us to take advantage of an underlying broadcast medium. Our performance measurements demonstrate the feasibility of such an approach and show reasonably effective bandwidth utilization for patterns of communication required by representative parallel programs.

Our approach is not limited to clusters of workstations running AIX (IBM's version of UNIX). Most modern OS kernels, including commercial variants of UNIX like SunOS and Solaris, are made extensible (mainly to accommodate dynamically loadable device drivers). Most of these kernels can also be dynamically extended with other kind of modules, for instance, modules that implement additional system calls. Implementing URTP on such an operating system is relatively easy; if the networking code of the target system is based on *mbufs*, as most BSD-based UNIX systems are, only minor changes are necessary.

We will concentrate on improving URTP performance in the future. The first target is improving the latency for packets of all sizes. The new architecture will not be based on the signal mechanism we use now. Possible improvements include using DMA for transferring data from kernel to user space, using a better management for the sending buffer pool, and, finally, moving more of the protocol into the kernel. The URTP interface may experience some small extension. One possible improvement is to integrate user sending buffers into URTP. In order for URTP to be used as a stand alone tool, packetization and reassembling need to be included into URTP.

Other extensions of this work include dealing with processor failure, using all available Ethernet hardware multicast groups in an efficient way (we have used only one hardware group in our implementation), supporting multiple MPI applications (at the same time) in the implementation, and supporting the complete MPI in the implementation.

Acknowledgements

We thank Greg Joyce, Frank Schmuck, Jim Wyllie and Steve Wise for their advice on AIX. We thank Karen Brannon and Yoichi Tsuji for their help with PolyFEM. We thank Alex Ho for his comments on the paper. This research was supported in part by the NSF Young Investigator Award CCR-9457811, by the Sloan Research Fellowship, by a grant from the IBM Almaden Research Center, San Jose, California, and by GIF Research Grant No. I-0207-199.06/91.

References

[1] Y. Amir, D. Dolev, S. Kramer and D. Malki, "Transis: A communication subs-system for high availability," *Proceedings of the 22nd International Symposium on Fault-Tolerant Computing,* IEEE, pp. 76–84, 1992.

[2] V. Bala, J. Bruck, R. Bryant, R. Cypher, P. de Jong, P. Elustondo, D. Frye, A. Ho, C.T. Ho, G. Irwin, S. Kipnis, R. Lawrence, and M. Snir, "The IBM External User Interface for Scalable Parallel Systems", Parallel Computing, Vol. 20, No. 4, pp. 445–462, April 1994.

[3] V. Bala, J. Bruck, R. Cypher, P. Elustondo, A. Ho, C.T. Ho, S. Kipnis, and M. Snir, "CCL: A portable and tunable collective communication library for scalable parallel computers", International Parallel Processing Symposium, pp. 835–844, Cancun, Mexico, April 1994.

[4] K. Birman, R. Cooper, T. A. Joseph, K. Marzullo, M. Makpangou, K. Kane, F. Schmuck and M. Wood, *The ISIS System Manual,* Dept. of Computer Science, Cornell University, September 1990.

[5] G.M. Brown, M.G. Gouda, and R.E. Miller "Block Acknowledgement: Redesigning the Window Protocol", In Proc. ACM SIGCOMM'89, Austin, Texas.

[6] J. Bruck, D. Dolev, C.T. Ho, R. Orni, and R. Strong, "PCODE: An Efficient and Reliable Collective Communication Protocol for Unreliable Broadcast Domains", IBM Research Report, RJ 9895, September, 1994.

[7] J.B. Carter and W. Zwaenepoel, "Optimistic Implementation of Bulk Data Transfer Protocols" In *Performance Evaluation Review*, Vol. 17, No. 1, May 1989, Pages 61-69.

[8] D.R. Cheriton and W. Zwaenepoel, "Distributed Process Groups in the V Kernel", In *ACM Transactions on Computer Systems*, Vol. 3, No. 2, May 1985, Pages 77-107.

[9] D. Culler et al., the Active Message Project by U.C. Berkeley. See papers and related information from http://now.cs.berkeley.edu/AM/active_messages.html.

[10] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *JACM 32* (1985) 373-382.

[11] H. Franke, P. Hochschild, P. Pattnaik, and M. Snir, *MPI-F: An Efficient Implementation of MPI on IBM-SP1*, Proceedings of 1994 International Conference on Parallel Processing, Vol. III, pp. 197–201, August, 1994.

[12] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, B. Manchek, and V. Sunderam, *PVM: Parallel Virtual Machine—A User's Guide and Tutorial for Network Parallel Computing*, MIT Press, 1994.

[13] J. N. Gray, "Notes on Database Operating Systems," *Operating Systems: an advanced course, Lecture Notes in Computer Science 60*, Springer Verlag (1978) 393-481.

[14] C. Huang, E. P. Kasten, and P. K. McKinley, "Design and Implementation of Multicast Operations for ATM-Based High Performance Computing", Proceedings of Supercomputing 94 conference, pp. 164-173, Washington D.C., November 1994.

[15] K. Li et al., Princeton University, the Shrimp Project. See http://www.cs.princeton.edu/shrimp.

[16] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard*, May 1994.

[17] J.C. Mogul, R.F. Rashid, and M.J. Accetta, "The Packet Filter: An Efficient Mechanism for User-level Network Code", In *Proceedings of the 11th Symposium on Operating System Principles*, ACM SIGOPS, Austin, Texas, November 1987.

[18] R. B. Morris, Y. Tsuji, and P. Carnevali, "Adaptive Solution Strategy for Solving Large Systems of p-type Finite Element Equations", Int. J. Numer. Methods Eng., Vol 33, 2059-2071, 1992.

[19] S. Pakin, M. Lauria and A. Chien, "High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet", in Supercomputing '95, San Diego, CA. See also http://www-csag.cs.uiuc.edu/projects/comm/fm.html for related papers.

[20] Parasoft Corporation, Express version 1.0: A communication environment for parallel computers, 1988.

[21] D. Patterson et al., "A Case for Networks of Workstations (NOW)", Symposium Record of Hot Interconnects II, pp. 24–39, August 1994.

[22] R. Strong, D. Dolev, and F. Cristian, "A Unified Theory of Distributed Coordination with Communication Uncertainty," IBM Research Report RJ7727, 1990, in the *Proceedings of the 28th Allerton Conference*, Allerton, September, 1990.

[23] A. S. Tanenbaum, M. F. Kaashoek and H. E. Bal, "Parallel programming using shared objects and broadcasting," *IEEE Computer,* vol. 25, 1992.

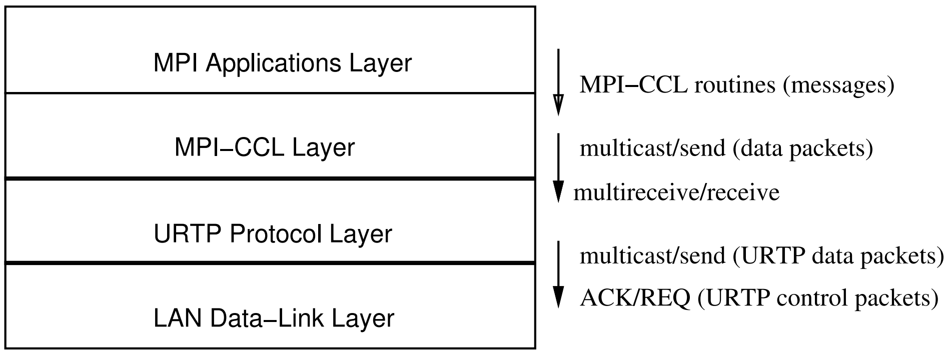

Figure 1: The four software layers of the system architectures. In this paper, we present design, implementation and performance of the two middle layers.

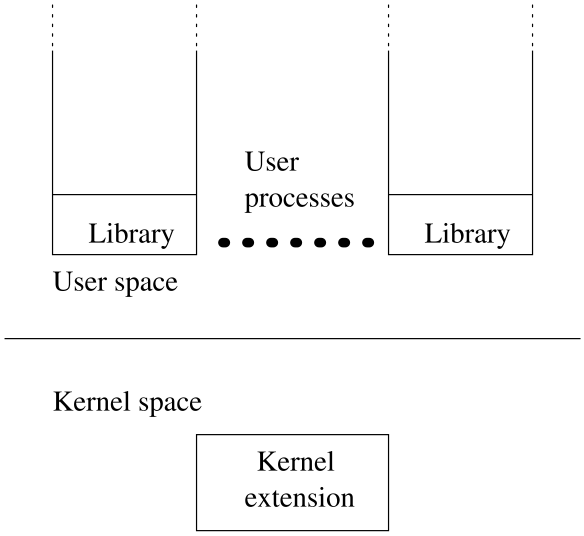

Figure 2: The system architectures of URTP.

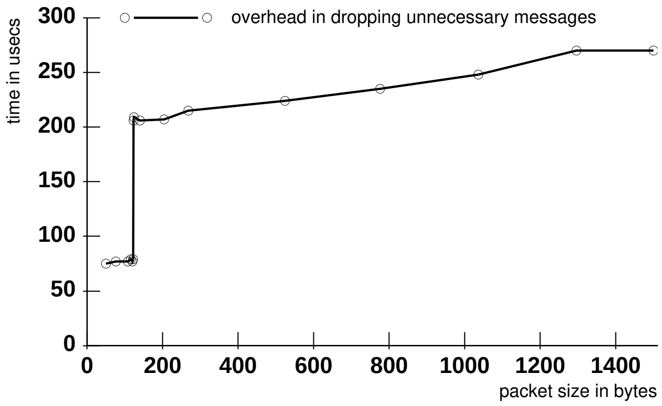

Figure 3: The overhead of dropping an unnecessary message.

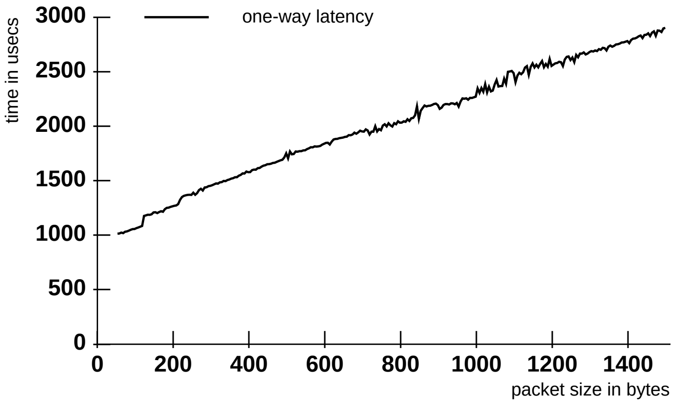

Figure 4: The one-way latency of URTP.

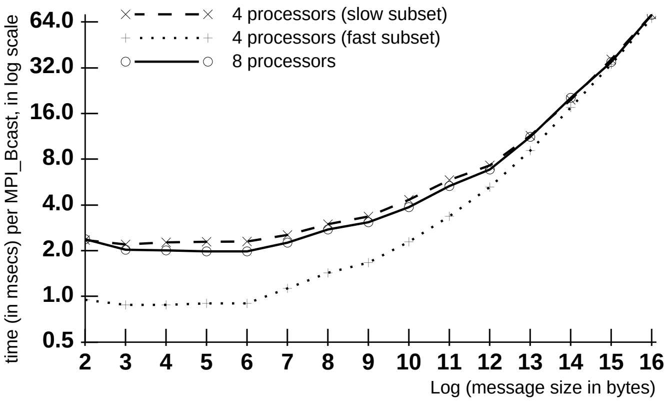

Figure 5: The time of MPI_Bcast as a function of the message size.

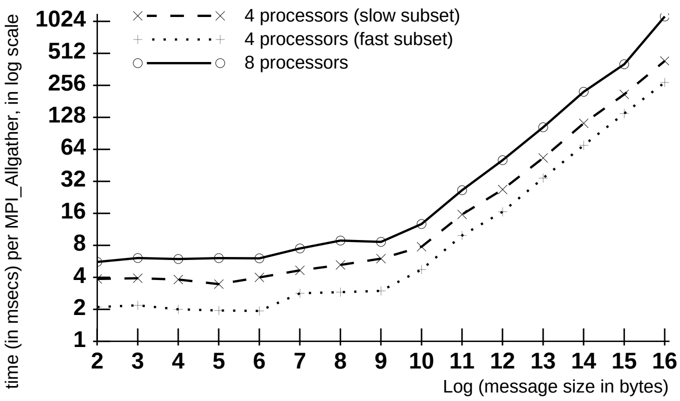

Figure 6: The time of MPI_Allgather (Algorithm 2, $k = 1$) as a function of the message size.

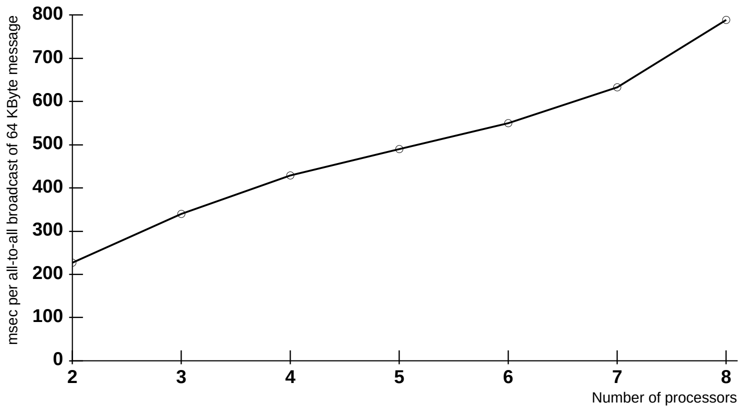

Figure 7: The time of MPI_Allgather (Algorithm 2, $k = 1$) as a function of the number of workstations using slow subset.

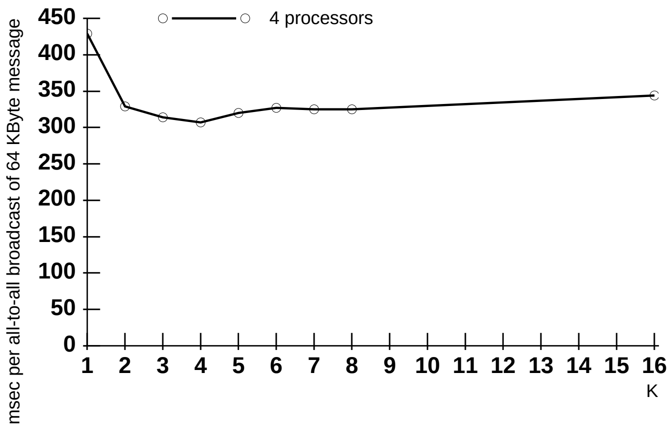

Figure 8: The time of MPI_Allgather as a function of k using slow subset.

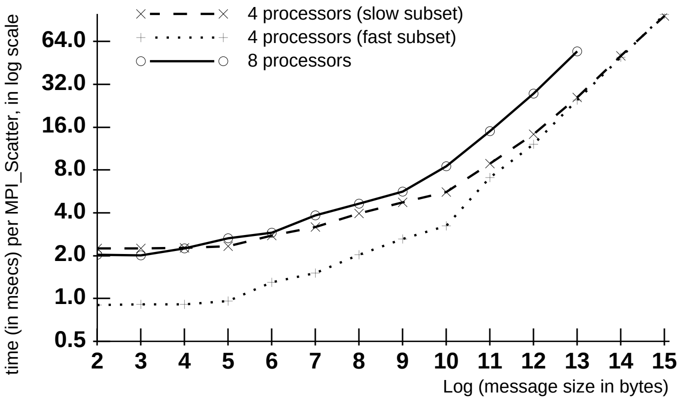

Figure 9: The time of MPI_Scatter as a function of the message size.

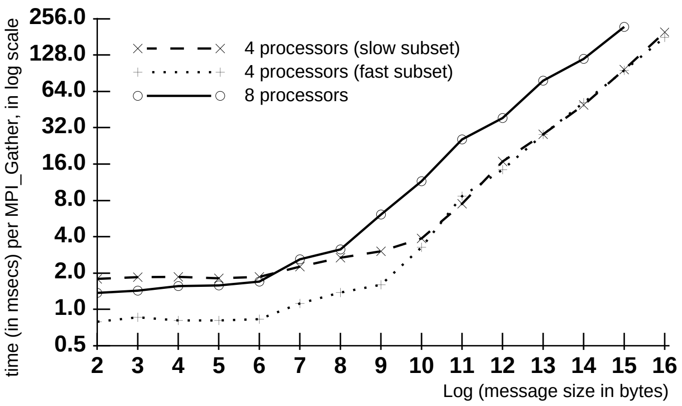

Figure 10: The time of MPI_Gather as a function of the message size.

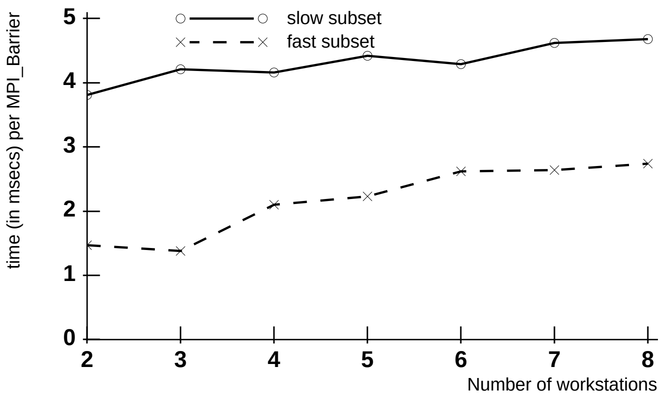

Figure 11: The time of MPI_Barrier as a function of the number of workstations.

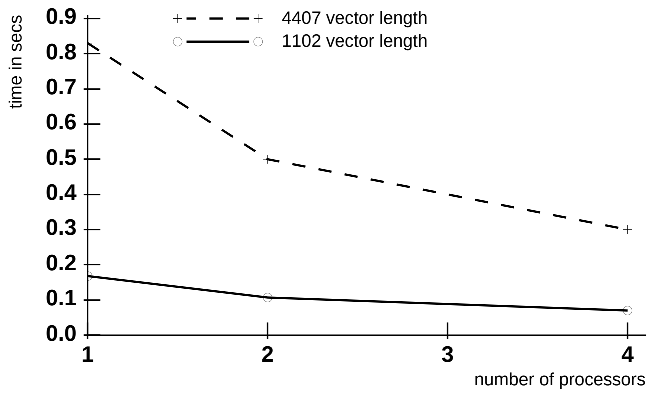

Figure 12: Performance of sparse matrix-vector multiplication as a function of number of processors for two vector lengths using our MPI-CCL on URTP protocol.

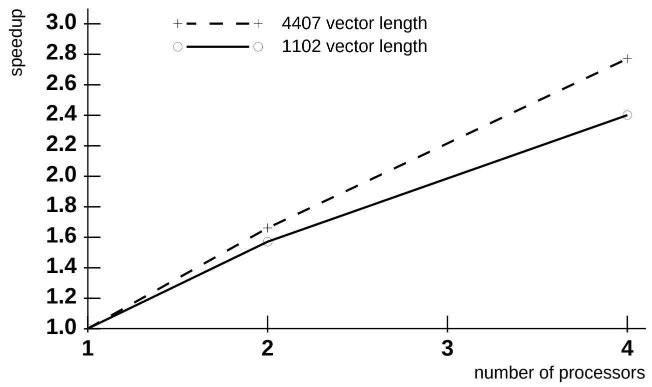

Figure 13: Speedup of sparse matrix-vector multiplication as a function of number of processors for two vector lengths using our MPI-CCL on URTP protocol.

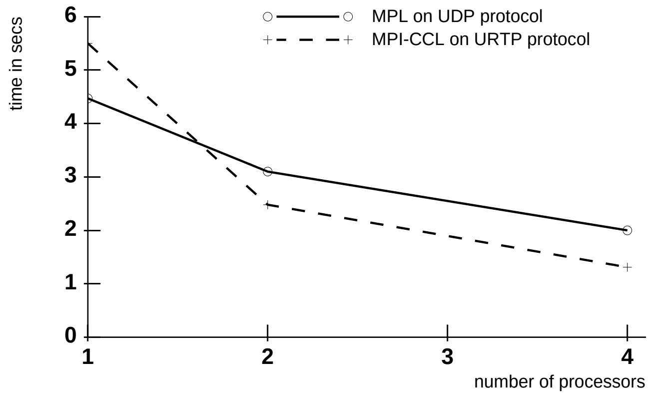

Figure 14: Performance of dense matrix-matrix multiplication using our MPI-CCL on URTP protocol versus that using IBM MPL product on LAN. The input matrices are 128×128.

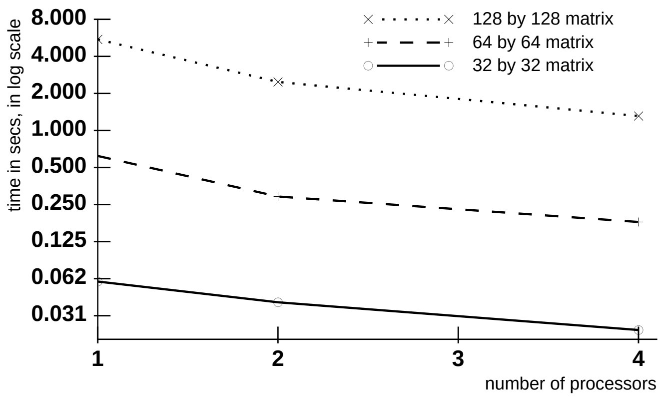

Figure 15: Performance of dense matrix-matrix multiplication as a function of number of processors for various matrix sizes using our MPI-CCL on URTP protocol.

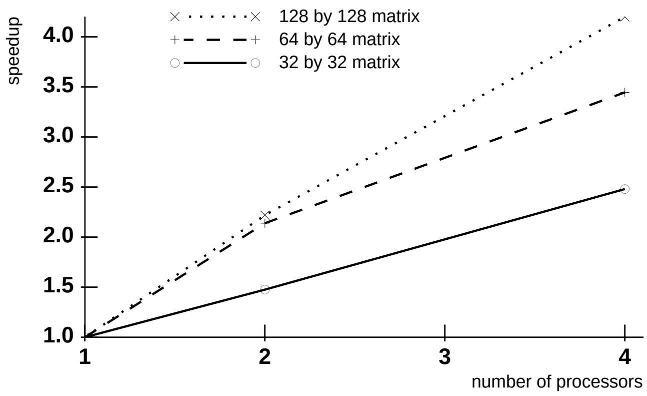

Figure 16: Speedup of dense matrix-matrix multiplication as a function of number of processors for various matrix sizes using our MPI-CCL on URTP protocol.

A Pseudocode of URTP

Define InWindow(i,j) as

$$(0 \leq i - j < \text{WINDOW}) \textbf{ or } (0 \leq i + \text{RANGE} - j < \text{WINDOW}).$$

Define $\text{Inc}(x, r) = (x + 1) \bmod r$. Define $\text{Dec}(x, r) = (x - 1) \bmod r$. We maintain a buffer pool Bufs that is organized as a subpool FreeBufs and, for each source m, an indexed subpool Received(m, i) where

$$\text{ReceivedWindowBase}(m) \leq i < (\text{ReceivedWindowBase}(m) + \text{WINDOW}) \bmod \text{RANGE}.$$

We also maintain a pool of minibuffers Unacks that is organized into a subpool FreeUnacks and a subpool with each element the target of some number of the indexed family of pointers Sent(m, i), where

$$\text{SentWindowBase}(m) \leq i < (\text{SentWindowBase}(m) + \text{WINDOW}) \bmod \text{RANGE}.$$

The system call SysRecv(FreeBufs,msg) moves one of the buffers from FreeBufs to msg, if successful. Free(Received(m, i)) returns the buffer in Received(m, i) to FreeBufs and sets Received(m, i) to NULL. Free(Sent(m, i)) decrements the count of required acknowledgements in the minibuffer to which Sent(m, i) points. If the count becomes zero, then the minibuffer is returned to the FreeUnacks subpool and the buffer to which it points is freed. In any case, Free(Sent(m, i)) sets Sent(m, i) to NULL.

For simplicity we have suppressed the locking code necessary to protect data structures in a reentrant environment. We have also suppressed the details of the routines we put in our kernel extension. Our code is responsive to signals as well as calls from the application. We have suppressed the methods by which we solved the problem of data structure sharing between application and signal handler without signal masking. Also we have suppressed some of the delays involved, except as applied to buffer management, where we illustrate exponentially growing delay (see Figure 17).

In Figure 21 we indicate the action of the signal handler that is called by the interrupt handler in the kernel that processes new packets as they arrive. Here the pseudocode is written as if exactly one packet were transferred from kernel space to user space. Actually SysRecv can transfer an unknown but bounded number of packets into user space, provided there are sufficiently many buffers in FreeBufs. Thus the number of context switches from user to kernel is minimized.

The variable MinReqGap of Figure 23 is the δ mentioned in Section 3.2.1.

```
WaitAndDouble(d, x)
{
    wait(d);
    d ← min(x, 2d);
    return;
}
```

Figure 17: WaitAndDouble (exponentially growing delay with upper bound).

```
mcastprep(destinations, bp)
{
    delay ← MinSendWait;
    do while (FreeUnacks subpool of Unacks is empty)
        WaitAndDouble(delay, MaxSendWait);
    delay ← MinSendWait;
    u ← dequeue(FreeUnacks);
    u.count ← 0;
    u.buf ← bp;
    for (0 ≤ m < MACHINES)
    {
        if (m is in destinations) then
        {
            do while
            (not InWindow(ToBeSent(m),SentWindowBase(m)))
                WaitAndDouble(delay, MaxSendWait);
            d(m) ← ToBeSent(m);
            Sent(m, ToBeSent(m)) ← address(u);
            u.count ← Inc(u.count);
        }
        else d(m) ← RANGE;
    }
    return(d);
}
```

Figure 18: mcastprep (called by Mcast).

```
Mcast(destinations, bp)
{
    d ← mcastprep(destinations, bp);
    timestamp(time);
    for (m in destinations)
        LastSendMoment(m, ToBeSent(m)) ← time;
    SysBsend(d, bp);
    for (m in destinations)
        ToBeSent(m) ← Inc(ToBeSent(m));
    return;
}
```

Figure 19: Application Call: Mcast.

```
recv(m)
{
    done ← 0;
    delay ← MinRecvWait;
    do while (done = 0)
    {
        i ← ReceivedWindowBase(m);
        if (Received(m, i) ≠ NULL) then
        {
            bp ← Received(m, i);
            Received(m, i) ← NULL;
            i ← Inc(i, RANGE);
            ReceivedWindowBase(m) ← i;
            done ← 1;
        }
        else
        {
            SysSendCtrl(REQ, m, i);
            WaitAndDouble(delay, MaxRecvWait);
        }
    }
    return(bp);
}
```

Figure 20: Application Call: Recv.

```
handle()
{
    do until (no more packets)
    {
        SysRecv(FreeBufs, msg);
        switch(msg.type)
        {
            case PT_TO_PT:
            case BCAST:
                HandleSND(msg); break;
            case REQ:
                HandleREQ(msg); break;
            case ACK:
                HandleACK(msg); break;
        }
    }
return;
}
```

Figure 21: Signal Handler: handle.

```
HandleACK(msg)
{
    m ← msg.source;
    i ← msg.count;
    j ← SentWindowBase(m);
    if (InWindow(i,j)) then
    {
        ii ← Inc(i, RANGE);
        FreeSendBuf(m, j, ii);
        SentWindowBase(m) ← Dec(ii, RANGE);
    }
    Free(msg); return;
}
```

Figure 22: HandleACK (called by handle).

```
HandleREQ(msg)
{
    m ← msg.src;
    i ← msg.cnt;
    j ← SentWindowBase(m);
    if (notInWindow(i,j)) then { Free(msg); return; }
    timestamp(time)
    if (time-LastSendMoment(m, i) < MinReqGap) then
        { Free(msg); return; }
    SysSendOld(m, i);
    FreeSendBuf(m, j, i);
    SentWindowBase(m) ← Dec(i, RANGE);
    Free(msg); return;
}
```

Figure 23: HandleREQ (called by handle).

```
HandleExpected(m, j, e)
{
    do while (InWindow(e,j) and (Received(m, e) ≠ NULL))
        e ← Inc(e, RANGE);
    ee ← Inc(e, RANGE);
    if (InWindow(e,j) and (Received(m, ee) ≠ NULL)) then
        SysSendCtrl(REQ, m, e);
    la ← LastAcked(m);
    if ((e − la > AckWindow) or (RANGE > RANGE + e − la > AckWindow)) then
    {
        ea ← Dec(e, RANGE);
        SysSendCtrl(ACK, m, ea);
        LastAcked(m) ← ea;
    }
    TooEarly(m) ← 0;
    Expected(m) ← e;
    return;
}
```

Figure 24: HandleExpected (called by HandleSND).

```
HandleSND(msg)
{
    m ← msg.source;
    i ← msg.count;
    j ← ReceivedWindowBase(m);
    if (InWindow(i,j) and (Received(m, i) = NULL)) then
    {
        Received(m, i) ← address(msg);
        e ← Expected(m);
        if (e=i) then HandleExpected(m, j, e);
        else
        {
            TooEarly(m) ← Inc(TooEarly(m), REQDelay);
            if (TooEarly(m) = 0) then
            {
                SysSendCtrl(REQ, m, e);
                LastAcked(m) ← Dec(e, RANGE);
            }
        }
    }
    else Free(msg);
    return;
}
```

Figure 25: HandleSND (called by handle).

```
FreeSendBuf(m, j, ii)
{
    do while (j ≠ ii)
    {
        Free(Sent(m, j));
        j ← Inc(j, RANGE);
    }
    return;
}
```

Figure 26: FreeSendBuf (frees buffers).