Communication Characterization of a Cray T3D

Stevan Vlaović

February 13, 1997

Abstract

In order to develop efficient applications for large scale multiprocessors, the communication patterns of the target architecture should be well understood. By understanding these communication patterns, a faster running application can be developed. Some applications require the smallest latency possible, whereas others might benefit from higher throughput.

The Cray T3D is a multiprocessor that has a high speed 3D torus interconnect, and special communications hardware. By utilizing this hardware properly, a program has the potential to minimize its communications overhead. This study investigates three different libraries that provide communication on the T3D: PVM, MPI, and SMA. Both PVM and MPI are portable, whereas SMA is native. The latency, bandwidth, and collective communication costs of the communication primitives implemented in these libraries are characterized and compared. These results are also briefly contrasted with the results of related studies on the IBM SP2 and the Convex SPP-1000.

Contents

1 Introduction

In the scientific community there has been widespread interest in high performance parallel computers, as these machines tend to have the capability to solve compute-intensive problems in reasonable time. In order to get a speedup that is close to linear, the communication subsystem must be well characterized. This characterization can then be used to develop programs that utilize the full capability of the machine. This data can also be used to gain insight into problematic, or seemingly unscalable code.

In this study, we analyze the communication costs for the Cray T3D not so much at the hardware level, but more as seen by the application programmer who desires to get maximum efficiency out of the machine.

2 Cray T3D

The basis of the T3D is a scalable macroarchitecture that combines DEC Alpha microprocessors with a low latency interconnect network. This system is designed to work with a Cray Y-MP or C-90 as a front end. As a MIMD machine, it can be dynamically partitioned to emulate SIMD or multicomputer MIMD modes of operation. The system uses a logically shared, physically distributed memory with a Mach based microkernel resident on each processing element. Each of these micro kernels manages communications with other processing elements, and the Cray C-90 front end.

2.1 Processing elements

The T3D incorporates the Alpha architecture DECchip 21064 which is a dual issue machine and supports IEEE standard 64 bit floating point arithmetic. This particular configuration is capable of 150 MFLOPS peak performance. The 21064 has on-chip 8 KB instruction and data caches, each with 32 byte lines. Stores are nonblocking, and loads and stores may be reordered, so a memory barrier is required whenever it is necessary to serialize references. Each processing element (PE) on the Cray T3D has two of these processors with its own DRAM memory with a capacity of either 16 or 64 MBytes, along with associated support logic. A *node* consists of two PEs sharing a switch and network support logic, as depicted in Figure 1. With this network support logic, interprocessor communications can be handled without interrupting the PE's attached to the nodes.

2.2 Interconnect

The processing elements on this machine are connected by a bidirectional 3-D torus system interconnect network. The 3-D torus uses a 150 MHz clock, matching that of the processors. The configurations can vary from 32 to 2048 processors, but the work for this experiment was carried out on a 128 (8x4x4) processor machine. Although this was the total number of processors in the system, the most that was available interactively was 16 (4x2x2).

The interface between the processor and the communication system on the Cray T3D includes the annex, a memory mapped communication port which maps some free range of address space to the physical memory of another node. This annex, called the *DTB Annex*, contains 32 registers and is indexed by five bits of the physical address. Additional fields in each Annex entry control the mode of remote operation. The Alpha load-locked and store-conditional instructions are used to read and write these registers. This node, or communication partner, can be changed by modifying the appropriate annex entry, incurring a fixed overhead. Once a store operation is issued to the communication port, the communication subsystem takes over the specified address and data, and sends a message to the receiving node.

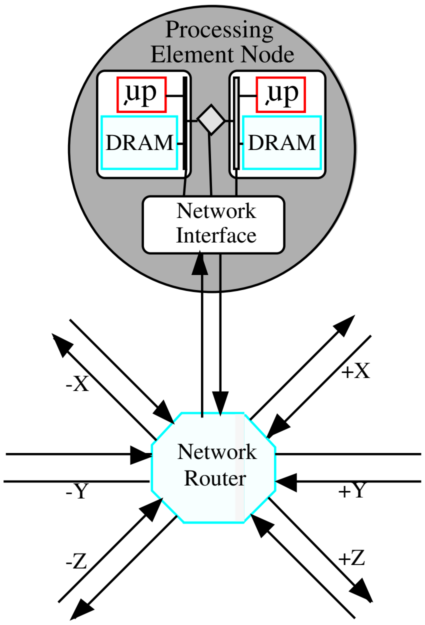


Figure 1: Node architecture

The nodes also have fetch/deposit circuitry that handle incoming remote operations on behalf of the communication system. These accesses can happen without interruption of the processor at the receiver node. This circuitry can store incoming data words directly into the user space of the processing element, since both address and data are sent over the network. The on-chip cache can be invalidated line by line as the data is stored into local memory, or can be flushed when the program reaches a synchronization point.

This hardware communications shell also supports an atomic-swap between a shell register and memory, two fetch-and-increment registers per processor, as well as global-OR and global-AND barriers. The Alpha fetch hint instruction is interpreted by the shell as a binding prefetch into a prefetch FIFO, which is popped by the processor through loads to a memory mapped address. The Alpha memory barrier instruction ensures that writes and prefetchs have been delivered to the shell. There is also a user-level message send FIFO, which either hands the incoming message off to a user level message queue or invokes a specific user thread.

In addition to the high speed network, the Cray T3D supports three major latency hiding mechanisms that augment the high performance system interconnect. There is a prefetch queue (PFQ), a read ahead capability, and a Block Transfer Engine (BLT). Each prefetch returns a word to the 16 word PFQ. The PFQ and the read ahead features are designed to hide latency for word level transfers, while the BLT is intended to hide latency for kiloword level transfers. The BLT can redistribute up to 65,536 cache lines of data without processor intervention. There is also support for parallel I/O operations.

This exceptionally wide range of mechanisms for carrying out communication within the T3D makes the optimization job quite complex unless one has at hand an adequate characterization of the performance consequences among the available choices. Providing such a characterization is the focus of this study.

3 Message-passing Libraries

3.1 PVM

In the following sections, two application level communication libraries are compared to the T3D's explicit global shared memory calls. One library supported by Cray is the **P**arallel **V**irtual **M**achine, or PVM. PVM was first developed as a software system to allow a network of heterogeneous Unix computers to be used as a single computer [6]. In version 3.0, the dependence on Unix sockets and TCP/IP software is relaxed. This allows multiprocessor systems to exploit the native calls in order to increase performance. Convex, DEC, KSR, and IBM have also decided to supply PVM 3.0 with their multiprocessors. Thus it has become a fairly portable standard worthy of comparative evaluation.

The Cray MPP PVM-3 version 1.2 can be used in two different modes. In stand alone mode, it can be used like the message passing libraries supplied for other MPP systems to program an application on the T3D system. In distributed mode, it can be coupled with the network version to let Cray T3D applications use message passing to communicate with processes running on any other systems that run PVM. The Cray MPP version uses the hardware capabilities of the Cray T3D system to handle communications between processor elements, and uses TCP/IP and UDP data transfers outside the Cray MPP system. Between processing elements, communications are fast, allowing the application more latitude in determining how often to send messages and how much data to send per message.

Before any communication calls can be initiated, a buffer must be allocated by calling `pvm_initsend`. Following this call, the data must be packed (copied) into the buffer using the `pvm_pkbyte` routine. This is all done on the sender's processor. There are three different methods of packing. The simplest and most universal choice is **DataDefault**. This method stores message data in a machine independent format, in order to facilitate the communication between processors with different architectures. There is no encoding scheme for 64-bit integers, however. **DataRaw** is the next method of packing. Unlike

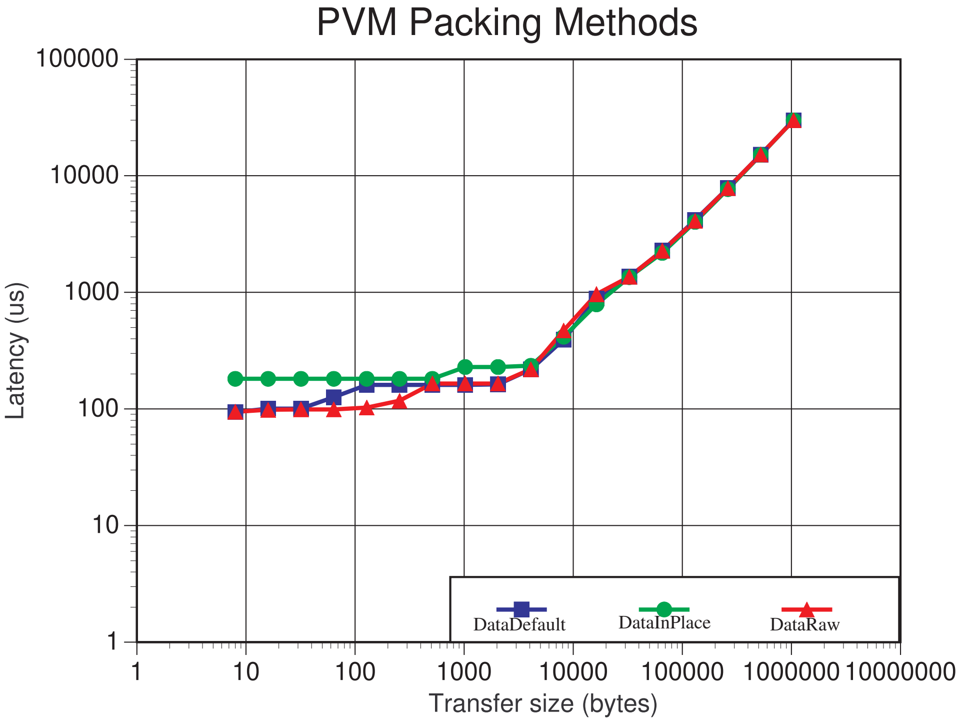


Figure 2: PVM Packing methods for floating point numbers

DataDefault, **DataRaw** avoids data conversion and is only appropriate for internal communication, or for communication between similar architectures. The final packing scheme is in fact not a packing scheme at all; **DataInPlace** provides for unbuffered message transmission. The user must ensure that the data to be transferred are not changed until the message has arrived at the destination processor. Figure 2 shows the latency difference among the packing schemes for point-to-point communication. Note that **DataRaw** outperforms or is comparable to the other two schemes. Because of its superior performance, in addition to the fact that all communication of interest in this study is internal, **DataRaw** is the packing method that we focus on in the remainder of this paper.

Since `pvm_send` is nonblocking, the end of the timing had to be measured at the receiver's end. In sends to multiple processors, the processor that last received the message was used as the completion point.

3.2 MPI

The University of Edinburgh in cooperation with Cray Corporation has developed the **M**essage **P**assing **I**nterface (MPI) library for use on the Cray T3D.

The goal of MPI was to develop a widely used standard for writing message passing programs. As such, the goals of the steering committee that defined MPI were as follows [5]:

- design an application programming interface.
- Allow efficient communication, i.e. avoid memory-to-memory copying, allow overlap of computation and communication, and offload to communication coprocessor, where available.
- Allow for implementations that can be used in a heterogeneous environment.
- Allow convenient C and Fortran bindings for the interface.
- Assume a reliable communication interface, i.e. the user should not need to cope with communication failures.
- Define an interface that is not too different from current practice.
- Define an interface that can be implemented on many vendors' platforms.
- Semantics of the interface should be language independent.
- The language should be designed for thread safety.

The MPI version that is available on the Cray T3D, was developed by Edinburgh Parallel Computing Centre at the University of Edinburgh. In [4], the architecture of this implementation is discussed. In particular, the point-to-point and collective communications are discussed. Point-to-point communication is facilitated by a speculative transfer of envelopes (packets), based on a shared queue for each process. Messages and queue slots correspond to data cache lines to facilitate cache coherence maintenance. Mutual exclusion on a process' queue is enforced by using the *atomic swap* functionality. A writer process obtains exclusive access to the remote queue when it acquires a valid queue indexed location as a result of the swap; it can then write into the index using a `shmem_put`, update the head index, and relinquish access using a second swap.

The implementation of collective communication within MPI for the T3D library uses a separate protocol mechanism from point-to-point, based on dedicated packet slots rather than the shared access queue. Each processor provides a slot for every other processor. This provides a lock free, bottleneck free approach as opposed to the shared queue implementation.

3.3 Shared Memory Access Library

The interconnect interface hardware of the T3D supports a single address space across the machine. This means that each processor has the capability to read from or write to any other processor's memory. Since these remote read/write calls must be explicitly enacted through the Shared Memory Access (SMA) library, the machine is typically viewed as a shared nothing architecture with remote read/write capability.

Remote memory reads are achieved by calling the function

```
void shmem_get(long *target, long *source, int len, int pe);
```

which copies `len` words from the source address, `source`, in the local memory of the processor specified by `pe` to the target address, `target`, of the calling processor. This function consists of a request followed by a remote write by the remote processor.

A remote write is accomplished by calling the function

```
void shmem_put(long *target, long *source, int len, int pe);
```

which copies `len` words from the source address, `source`, in the local memory of the calling processor to the target address, `target`, of the processor specified by `pe`. It is important to note that the `shmem_put` call does not maintain consistency of the cache. For instance, if processor A is working on data in its cache, and processor B then `shmem_put`s an updated version of this data in A's memory, then A will be working on a stale copy. The Cray manual suggests that applications may need to flush the data cache by using the suppress directives provided by the compiler.

Along with being able to read/write to individual remote processors, the SMA library provides the user with many collective operations as well. The `shmem_broadcast` operation is one that is investigated in this paper. It has the form

```
void shmem_broadcast(long *target, long *source, int nlong,
    int PE_root, int PE_start, int logPE_stride, int PE_size,
    long *pSync);
```

In this function, every processor must make the call. At `PE_root`, `len` words are copied from address `source` to the `target` of the processors specified by `PE_start`, `logPE_stride`, and `PE_size`.

In measuring the time for communication, it is necessary to know when these `shmem` routines return. For `shmem_get` routines, the function returns when the data has been loaded into the local data array. For `shmem_put` routines, the function returns when the data has been sent from the processor. Since the T3D has a separate communication processor, the function may return (and usually does) before the data actually arrives at the destination processor. This allows for many different ways of timing the `shmem_put` routines. In [9], four different measurement methods are mentioned.

Method 1:

Sender	Receiver
`start_time`	do nothing
`shmem_put`	
`_remote_write_barrier`	
`stop_time`	

Method 2:

Sender	Receiver
`start_time`	do nothing
`shmem_put`	
`stop_time`	

Method 3:

Sender	Receiver
`barrier`	`barrier`
`shmem_put`	`start_time`
	`while(!EOM)`
	`stop_time`

Method 4:

Sender	Receiver
`barrier`	`barrier`
`shmem_put`	`start_time`
	`shmem_wait`
	`stop_time`

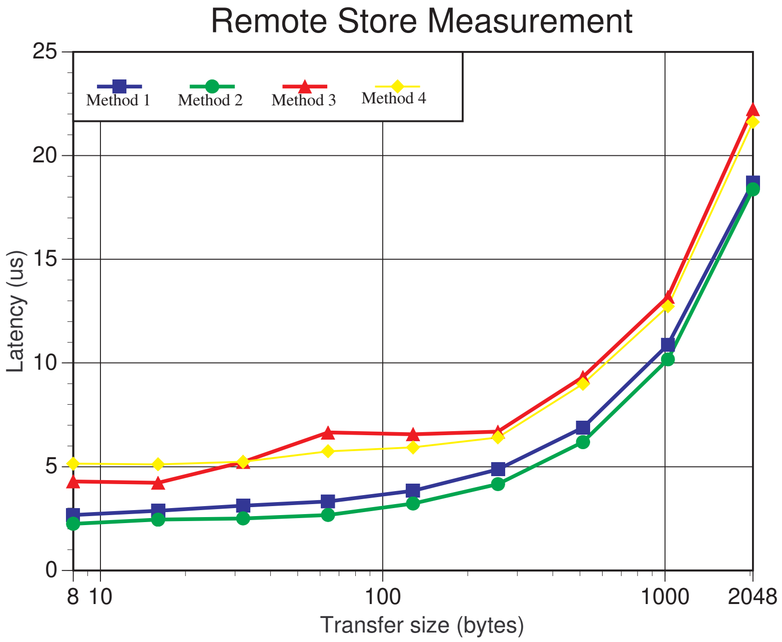


Figure 3: Four different measurement times of remote stores

In the first method, the intrinsic function `_remote_write_barrier` is used make sure that all outstanding remote write request packets are processed. Since this measurement method is the closest to the way that `MPI_Send` and `pvm_send` routines are measured, this was the technique chosen. The second method does not wait for the message to be processed (i.e. it is nonblocking) and would not be a fair comparison to the equivalent MPI and PVM calls. This method does, however, measure how long the sender is tied up, which is also useful information. The third method uses a manual polling method at the receiver's side to wait until the message arrives. As the manual polling method would add extra instructions, it was not used. In the fourth method, a `shmem_wait` call replaces the manual polling. Since the remote writes were to be sent to many processors and it is more straightforward to measure the time on one processor, this method was not employed. The graph in figure 3 shows these differences in the techniques for small data sizes.

4 Related Work

In Culler's work [3], the Cray T3D's raw hardware performance is investigated. By carefully crafting microbenchmarks, he is able to inspect each aspect of the communications hardware shell. The probes used in these experiments are written in assembly language, such that the measurements reflect actual hardware times, rather than overhead inserted by the compiler or message passing library. At the conclusion of the paper, a case study, EM3D, is used to demonstrate the techniques developed earlier.

The first experiment examines the latency of local memory operations. These results are then compared with a DEC Alpha workstation. The time to access local memory on the T3D is roughly 145 nanoseconds (22 cycles) while the DEC requires 300 nanoseconds (45 cycles). Culler explains this as the result of the fact that the T3D lacks an L2 cache.

As a remote access involves storing the destination node number into one of the Annex entries, it presents a new set of registers that the compiler must manage. Because the Annex registers specify a processor, if two different registers denote the same processor, synonyms can occur. If a run time table of Annex entries is kept, performing the lookup in software would likely cause greater delay than simply updating the Annex register (23 cycles). So it is their conclusion that only one Annex register is required. This is corroborated by work in [8, 10].

The performance of cached reads, uncached reads, and remote writes was also presented. An uncached read costs roughly 610 ns, while a cached read required 765 ns to complete. A remote write needs roughly 880 ns. All of these measurements were performed on accesses to an adjacent node, for farther nodes the added penalty is between 13 and 20 ns per hop.

The T3D's prefetch mechanism was broken down into four components: the prefetch issue, which took 4 cycles, the memory barrier (4 cycles), round-trip transfer (80 cycles) and the prefetch pop (23 cycles). They found that since about 75% of the cost of the remote fetch operation can be overlapped with other useful work, thereby masking most of the network latency.

The BLT, though only available through operating system invocation, achieves the highest transfer rate at roughly 140 MB/s for 512KB reads. From Culler's measurements, for reads of more than 16 KB, the BLT mechanism achieves better bandwidth than the other methods.

Another inspection of the Cray T3D hardware shell is done in [8]. In this work, Chien compares the T3D's communication hardware to that of the Thinking Machines CM-5. Because the CM-5 is focused on supporting the data parallel programming model, it requires processor intervention to process incoming messages. Therefore, if the processors are working asynchronously, output non-responsiveness can severely restrict the achievable communication performance. In Chien's experiments, when the standard deviation of the compute time is twice the mean, send overheads are 300% higher. This supports the notion of receiving the processor from the responsibility of processing each message.

This work also corroborates the work done in [3]. Their results also suggest that when output contention is a problem (fan-in), a pull based message system would work better than a push based one. By doing "lazy" communication, the processor that needs the data can grab it as fast as it can. Their scheme uses the prefetch queue to mask the latency on the receiver's side. This reduces the output contention at the sender. The prefetch scheme masks most of the latency, but since it is off chip it incurs 0.25 μs total time per fetch and extract pair. A simulation study is presented which explores the advantage of making this hardware available on chip.

In Stricker and Gross's paper they stress that the memory system performance is the key element in overall MPP performance [10]. They use the Cray T3D and the Intel Paragon as examples with which to demonstrate some proposed memory reference pattern models. The two models presented are buffer packing and chained transfers. They have some models that predict throughput; and for accesses other than contiguous (e.g stride n) chained transfers (supplying data directly to the network) seemed to perform better than buffer packing. This is simply because it avoids additional copying overhead.

Our work is most like [2]. In this paper, Abandah develops communication models for the IBM SP2. Like his paper, the goals of this research effort are to explain the achieved performance of message-passing applications, develop efficient message-passing applications and compare the performance of different multicomputers and application code implementations. Similar to the work in [2], our approach to developing communication performance models for a particular message passing library is to measure and model the performance of a set of common communication patterns. One difference between our work and [2], is that we look at three message-passing libraries (PVM, MPI, and SMA) and their characterization on the Cray T3D.

5 Message Passing

The time required to send messages processors to in a multiprocessor system is dependent on the processing nodes, the interconnection network, the message-passing library used, the communication pattern, the message size, and the number of processor nodes. Our approach to developing communication performance models for a particular message-passing library of a given multicomputer is to measure and model the performance of a set of common communication patterns. This set comprises the *basic* communication patterns that can be used as a basis for the construction of other more complex patterns. The basic communication patterns are point-to-point, one-to-many, many-to-one, and many-to-many. In addition to these basic patterns, we also inspect the performance of some collective communication subroutines found in the MPI, PVM and SMA libraries. The two most common collective routines are broadcast and combine.

For each experiment, we have prepared a C program that calls the appropriate subroutine. Each program performs one communication pattern many times, and reports the standard deviation and the 95% confidence interval. These programs are executed with varying numbers of processors and message lengths. In the following subsections, we describe the experiments used to gather the timing data, and develop a performance model for each pattern.

The point-to-point communication time can often be expressed by a few simple parameters as

$$\mathrm{T}(n) = t_0 + \frac{n}{r_\infty}$$

r_∞ is the asymptotic transfer rate in MB/s, and t_0 is the asymptotic zero length message latency in μs [7]. For small messages, t_0 is dominant, while for large messages the transfer time is dominant. Since n is measured in bytes, and is varied from 8 bytes to 1MB, the model or each library and each basic communication type was split into several regions where each region has its own t_0 and r_∞.

5.1 Point-to-point communication

In this section, the time to send a message from one processor to another is presented. The amount of data sent is varied exponentially from 8 bytes to 1 MByte. For the first graph, shown in Figure 4, total time is plotted for `shmem_put`, `shmem_get`, `pvm_send`, and `MPI_Send`. As was mentioned previously, the `shmem_get` requires sending a request packet followed by a remote write. This means that this function can never outperform a `shmem_put` for contention free communication, so it was not included in the Many-to-one or Many-to-many experiments.

An inflection point in the graph for PVM starts at the 4 KB message length. The reason for this, is that the default buffer size is set to be 4 Kb, however, 32 bytes of this is used for header information. So the maximum amount of data that can be sent using the default buffer is 4064 bytes. Any more data requires another buffer (implemented by the system). This was verified by increasing `PVM_DATA_MAX`, the environment variable responsible for setting initial buffer size. We have split the n domain into regions, in order to get more accurate models, the 4 KB boundary signifying the first region for the PVM library.

The second region is due to the `PVM_MAX_PACK` environment variable. This environment variable tells PVM to initially allocate some memory (4 KB default) for each send buffer, but, if more is needed to allocate a another buffer no smaller than whatever is specified by `PVM_MAX_PACK`. In another words, it tells PVM how large to make the second buffer, if there is more data than can accommodated by the default buffer. The format is `<initial> + <increment>`, and the default is 4096 + 12,288. This translates into a region from 4 KB to 16 KB. For transfers larger than 16 KB, system buffers are no longer used and the system copies directly from user space.

Using curve fitting, we have

`shmem_put`:

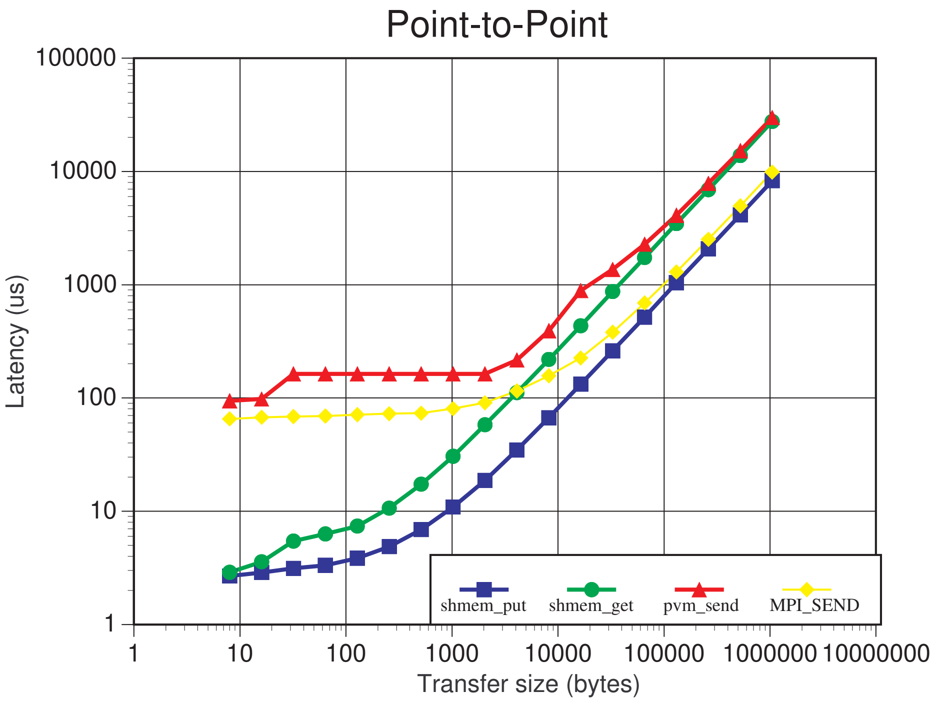


Figure 4: Point-to-point transfer latency

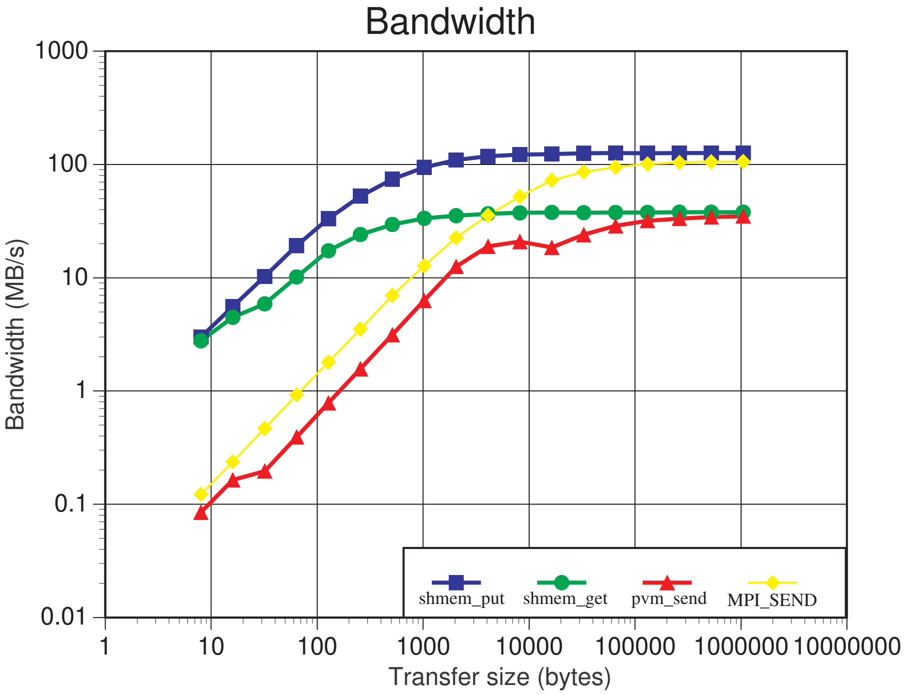


Figure 5: Effective bandwidth of different subroutines

$$T(n) = 2.75 + \frac{n}{126} \tag{1}$$

`shmem_get`:

$$T(n) = 4.64 + \frac{n}{37.9} \tag{2}$$

`pvm_send`:

$$T(n) = \begin{cases} 140 + \frac{n}{55.2} & 8\text{bytes} \le n \le 4\text{KB} \\ 18 + \frac{n}{19.37} & 4\text{KB} \le n \le 16\text{KB} \\ 444 + \frac{n}{35.4} & 16\text{KB} \le n \le 1\text{MB} \end{cases} \tag{3}$$

`MPI_SEND`:

$$T(n) = 71.6 + \frac{n}{107} \tag{4}$$

For `shmem_put`, the latency is 2.75 μs, and the effective bandwidth is around 126 MB/s. This is consistent with the data reported in [3, 9]. `shmem_get` has slightly worse latency at 4.64 μs, and the effective bandwidth is consistently worse than `shmem_put`, due to the additional packet required. `MPI_Send` initial latency is 71.6 μs, and a peak effective bandwidth of 107 MB/s. The reason for the additional latency is that MPI has added software overhead. For `pvm_send`, the latency is much worse, 140 μs, and its effective bandwidth peaks out at 55.2 MB/s. Note that this is after the unpacking operation, since PVM packs all of its data before transfer. The differences between the three regions for PVM are quite noticeable. The first region has the highest transfer rate at 55.2 MB/s, while having a initial latency of 140 μs. The third region has high initial latency, and a modest effective transfer rate of 35.4 MB/s, due to its direct copying method.

The transfer rate for each library can be found by dividing the message length by the communication time. These results can be seen in Figure 5. At 16 KB, the curve for `pvm_send` has a discontinuity. The local copy to the send buffer starts to affect the total time, so PVM avoids this copying and sending directly out of user space.

5.2 One-to-many

This experiment measures the outbound node performance of a multiprocessor. The sender processor sends a message to 15 other processors. The time taken to complete is the time taken by the sender processor finish sending. This is done by executing the wait subroutine call for MPI and PVM. For this communication pattern, `shmem_get` actually performs better than `shmem_put`. The reason for this is straightforward; the definition of time for this communication pattern is the time taken by the sender to finish sending. But in `shmem_get`, the receivers *get* their own data. The time to complete is the time taken for last receiver to complete. Intuitively, it takes less time for each receiver to be responsible for its own data, as opposed to having the sender send data all other processors. This is the reason for the negative t_0 in the curve fit for `shmem_get`.

The results can be seen in Figure 6. For large data sizes, all four library communication subroutines converge to a bandwidth of around 8 MB/s.

The curve fitting results for the One-to-many times are as follows:

`shmem_put`:

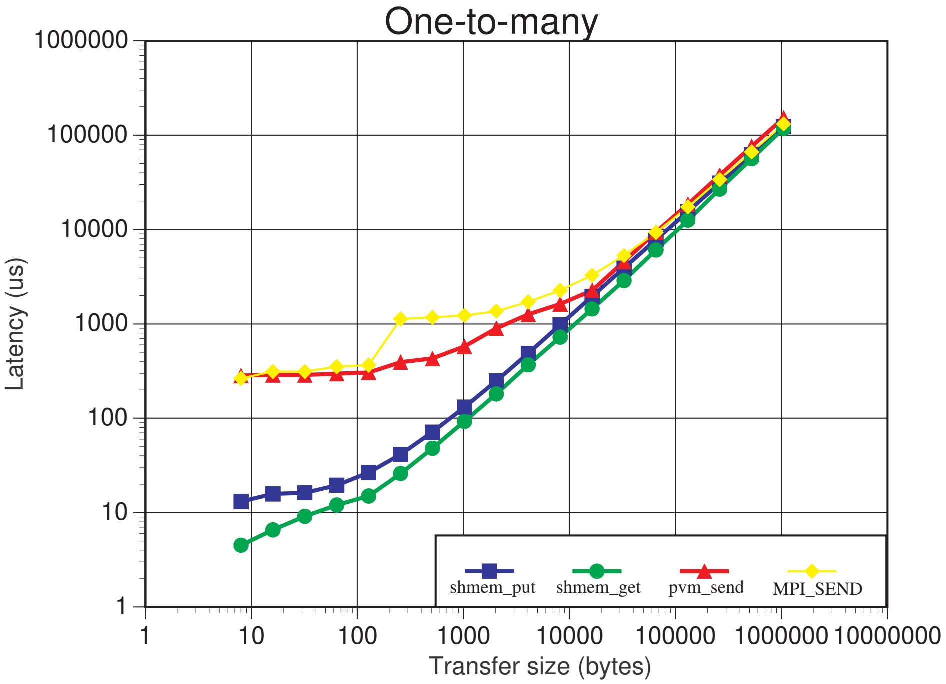


Figure 6: Time to send to fifteen processors

$$T(n) = 13.8 + \frac{n}{8.45} \tag{5}$$

`shmem_get`:

$$T(n) = -452 + \frac{n}{8.97} \tag{6}$$

`pvm_send`:

$$T(n) = \begin{cases} 299 + \frac{n}{4.03} & 8\text{bytes} \leq n \leq 4\text{KB} \\ 939 + \frac{n}{12.2} & 4\text{KB} \leq n \leq 16\text{KB} \\ 192 + \frac{n}{6.90} & 16\text{KB} \leq n \leq 1\text{MB} \end{cases} \tag{7}$$

`MPI_SEND`:

$$T(n) = \begin{cases} 286 + \frac{n}{1.41} & 8\text{bytes} \leq n \leq 128\text{bytes} \\ 1161 + \frac{n}{8.00} & 128\text{bytes} \leq n \leq 1\text{MB} \end{cases} \tag{8}$$

For this experiment, it becomes necessary to split up the regions for MPI. The discontinuity can be seen for `MPI_SEND` at 256 bytes. This is where the MPI implementation on the T3D switches from **T**ransfer to RTA. The T protocol specifies that when the message can be accommodated within the send envelope, it is transferred immediately and is available to the receiver following a successful match. In *Standard* mode, if the message can be accommodated within a protocol message, the T protocol is used; the system memory used is within the protocol message slot at the receiver [4]. Since messages and queue slots correspond to data cache lines to facilitate cache coherence [4], the amount of data that can be accommodated by the send envelope is 4 cache lines, or 128 bytes. For messages up to 128 bytes essential comes for free (fixed latency) and marks our first region. The second region is for messages of at least 128 bytes and messages of longer length.

Note that the bandwidth has dropped, and the latency has increased for most of the subroutine calls relative to Point-to-point communication. However, since the processors need to share the bandwidth of the interconnect now, and the above formulas are as seen by a single processor, the data seems more reasonable. This experiment tested the output contention and proved that it can indeed be a factor, even on only sixteen processors.

5.3 Many-to-one

The inbound node performance of a message-passing library is measured by this experiment. The time taken by the receiving processor is the Many-to-one communication time. The results are shown in Figure 7 for sending from fifteen processors to one.

MPI has rather poor initial performance due to the implementation of *Standard* mode on the Cray T3D. First, the message is copied into memory owned by the MPI implementation. The `MPI_SEND` call then returns, so it is the receiver's responsibility to fetch the message copy using get-based transfers. If there is no MPI buffer space for holding a copy of the message, the sender can advertise the actual message, but must block until the acknowledgment is received [4]. This corresponds to a **R**equest **T**ransfer **A**cknowledge (RTA) protocol. In this protocol, the receiver drives message data transfer, following a

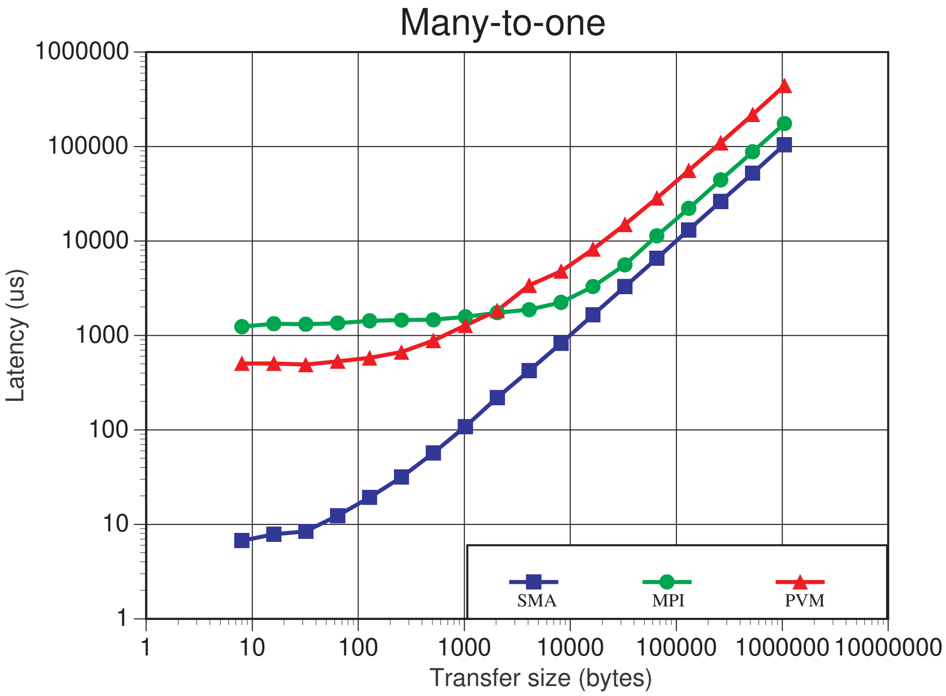


Figure 7: Time for 15 processors to send to one

successful send/receive match. An acknowledgment indicates completion of the get-based transfer to the sender.

The SMA library has the best overall performance, but MPI manages to remain competitive for large messages. For small messages up to 2 KB, PVM is better than MPI. At this point the higher effective bandwidth of MPI becomes more dominant over the lower initial latency of PVM. In this instance, since this experiment measures input contention, `shmem_put` outperforms the rest by a wide margin. MPI has a high initial latency due to contention for the shared access queue.

The Many-to-one characteristics are listed below.

SMA (`shmem_put`):

$$T(n) = 11.5 + \frac{n}{10.0} \tag{9}$$

MPI (`MPI_SEND`):

$$T(n) = \begin{cases} 1269 + \frac{n}{0.78} & 8\text{bytes} \leq n \leq 128\text{bytes} \\ 429 + \frac{n}{6.02} & 128\text{bytes} \leq n \leq 1\text{MB} \end{cases} \tag{10}$$

PVM (`pvm_send`):

$$T(n) = \begin{cases} 493 + \frac{n}{1.42} & 8\text{bytes} \leq n \leq 4\text{KB} \\ 1698 + \frac{n}{2.53} & 4\text{KB} \leq n \leq 16\text{KB} \\ 278 + \frac{n}{2.34} & 16\text{KB} \leq n \leq 1\text{MB} \end{cases} \tag{11}$$

5.4 Many-to-many

The total saturation bandwidth achievable by a message-passing library on a multiprocessor is measured by the many-to-many communication time. The time required is the maximum time of all the processors to finish sending and receiving. The results for 16 processors are shown in Figure 8.

The curve fitting plots for Many-to-many can be seen below.

SMA (`shmem_put`):

$$T(n) = 60.2 + \frac{n}{1.50} \tag{12}$$

MPI (`MPI_SEND`):

$$T(n) = \begin{cases} 2750 + \frac{n}{0.78} & 8\text{bytes} \leq n \leq 128\text{bytes} \\ 2850 + \frac{n}{1.30} & 128\text{bytes} \leq n \leq 1\text{MB} \end{cases} \tag{13}$$

PVM (`pvm_send`):

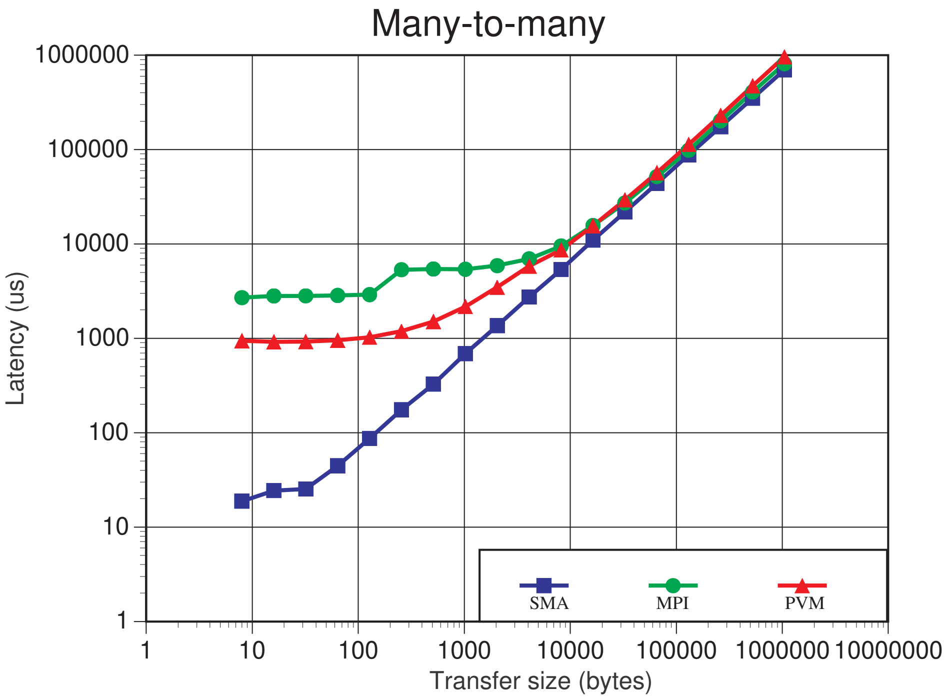


Figure 8: Time for 16 processors to send a size n messages to each other processor

$$T(n) = \begin{cases} 902 + \frac{n}{0.83} & \text{8bytes} \le n \le \text{4KB} \\ 2350 + \frac{n}{1.25} & \text{4KB} \le n \le \text{16KB} \\ 3030 + \frac{n}{1.08} & \text{16KB} \le n \le \text{1MB} \end{cases} \tag{14}$$

Although the initial latency varies substantially, around 60 μs for SMA to 2750 for MPI, the effective bandwidth rates for all three libraries approach more similar rate. PVM peaks at 1.25 MB/s, MPI at 1.30 MB/s, while SMA manages to achieve 1.50 MB/s. The effect of contention on the T3D does have an affect on the communication performance.

5.5 Broadcast Communication

The effectiveness of broadcast communication can play a crucial role in the performance of some applications. In this section the performance of the three libraries is examined. The total time for broadcasting to sixteen processors is shown in Figure 9.

The `shmem_broadcast` is the most efficient of the three. This might be expected, as the packing of data no longer is an issue. Since in broadcast algorithms the sending processor never has to send to all of the processors one by one, the overhead of packing dominates the `pvm_bcast` time.

The discontinuity associated with MPI at 256 bytes is associated with the T protocol described previously. The implementation of the `MPI_BCAST` uses a different protocol method to coordinate data transfer; however, the actual data transfer is done using the same module as the point-to-point communication [4]

Broadcast bandwidth and latency measures from the graph follow.

SMA (`shmem_broadcast`):

$$T(n) = 15.8 + \frac{n}{31.6} \tag{15}$$

MPI (`MPI_BCAST`):

$$T(n) = \begin{cases} 67.9 + \frac{n}{6.94} & \text{8bytes} \le n \le \text{128bytes} \\ 312 + \frac{n}{29.9} & \text{128bytes} \le n \le \text{1MB} \end{cases} \tag{16}$$

PVM (`pvm_bcast`):

$$T(n) = \begin{cases} 199 + \frac{n}{11.6} & \text{8bytes} \le n \le \text{4KB} \\ -259 + \frac{n}{5.2} & \text{4KB} \le n \le \text{16KB} \\ -718 + \frac{n}{6.67} & \text{16KB} \le n \le \text{1MB} \end{cases} \tag{17}$$

Once again, the SMA library outperforms the other two message passing libraries. The interesting point is that the 29.9 MB/s asymptotic bandwidth of MPI is reasonably close to the 31.6 of SMA. The cause for PVM's negative initial latency for the later two regions is due to the progressively worse bandwidth. The initial latency is actually around 199 μs, but the transfer time degrades significantly for large messages).

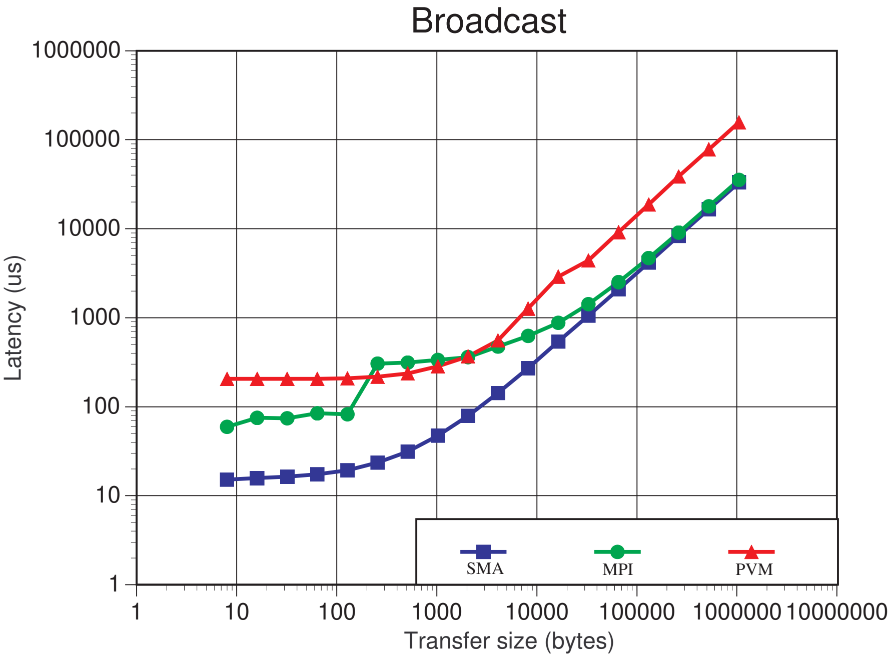


Figure 9: Time to broadcast to 15 processors

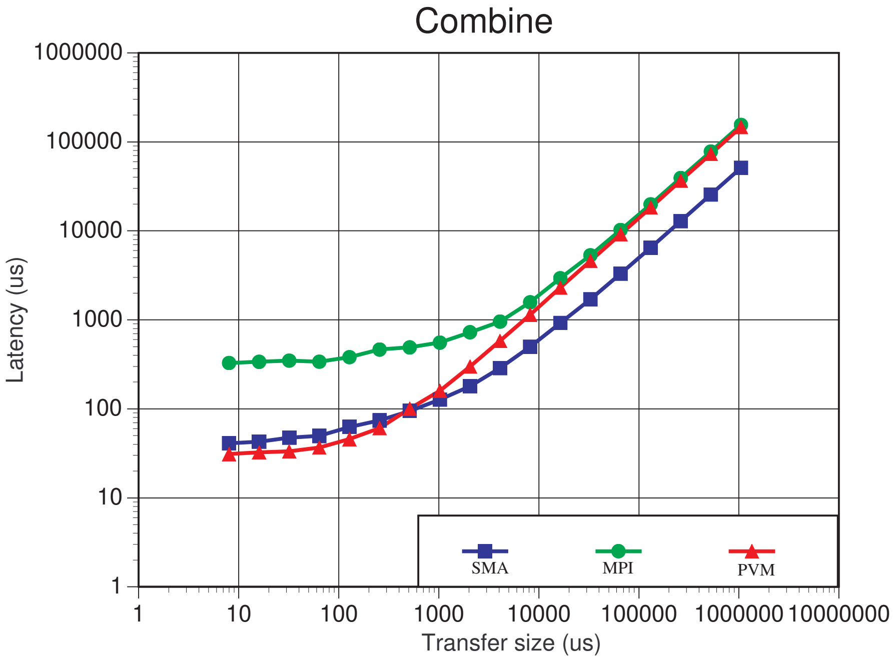


Figure 10: Time for reduction operation across 16 processors

5.6 Combine

This experiment measures the Combine performance of the three message passing libraries on the Cray T3D. The Combine communication involves reducing messages of length n from p different processors to one message of length n which is sent to all p processors. The reduction operation used in this particular experiment was double precision maximum. Figure 10 shows the Combine time for the three message passing libraries on the T3D.

The interesting thing to note in this mode of communication is that `pvm_reduce` actually performs better than `shmem_real4_max_to_all`. The `shmem_real4_max_to_all` requires as its parameters two work arrays. One of them, $pWrk$, requires $n/2+1$ elements, while the other one, pSync, requires 36 elements. Because the SMA library needs *two* extra arrays to do the work, for small messages this overhead dominates the combine time. For larger messages, the overhead of copying data into contiguous work arrays is relatively small compared to the time required to transfer.

The equations for Combine are the following:

SMA (`shmem_real4_max_to_all`):

$$T(n) = 77.8 + \frac{n}{20.6} \tag{18}$$

MPI (`MPI_REDUCE`):

$$T(n) = \begin{cases} 328 + \frac{n}{2.62} & 8\text{bytes} \le n \le 128\text{bytes} \\ 452 + \frac{n}{6.76} & 128\text{bytes} \le n \le 1\text{MB} \end{cases} \tag{19}$$

PVM (`pvm_reduce`):

$$T(n) = \begin{cases} 28.2 + \frac{n}{7.46} & 8\text{bytes} \le n \le 4\text{KB} \\ 1.41 + \frac{n}{7.14} & 4\text{KB} \le n \le 16\text{KB} \\ 19.3 + \frac{n}{7.14} & 16\text{KB} \le n \le 1\text{MB} \end{cases} \tag{20}$$

PVM manages to have the best results for small data sizes. Its subroutine call has the lowest asymptotic zero latency of 28.2 μ. For larger data sizes, the SMA library has the advantage. Note that for `pvm_reduce` the data is not packed.

6 Comparison to other Multiprocessors

This section presents of a simple comparison between the Cray T3D, the IBM SP2, and the Convex SPP-1000. The results for the SP2 are from [2], while the results from the Convex are from [1]. The data for Point-to-point communication is listed in the table below. The data represents a single region curve fit for Point-to-point communication. The value for the Cray T3D represents results using the SMA library, the data for the IBM SP2 is taken using the MPL library, and the data for the Convex SPP-1000 utilizes the MPI library.

	Latency (μs)	Bandwidth (MB/s)
Cray T3D	2.75	127
IBM SP2	47.0	36.2
Convex SPP-1000 near	104	13.5
Convex SPP-1000 far	115	4.5

This data is then plugged into the equation

$$\mathrm{T}(n) = t_0 + \frac{n}{r_\infty}$$

that was discussed previously. By using these equations, a rough estimate of the time required to transfer a message can be determined, as illustrated in Figure 11. Although the T3D outperforms the others by a wide margin, it is a far more expensive machine. The data shown for the Convex is for the MPI library; for *shared memory* accesses it performs better with an initial latency of 1.3 μs and transfer rate of 23.5 MB/s for uncached near accesses. The initial latency is even better than the results for the T3D, however since the performance degrades so severely for larger transfers, it was not included in Figure 12. In this figure, the communication performance of the Cray T3D and IBM SP2 are contrasted.

In these figures, the superiority of the T3D can be seen. It performs better than the IBM SP2 by a wide margin, except for Many-to-many communication with long messages where the SP2 comes close.

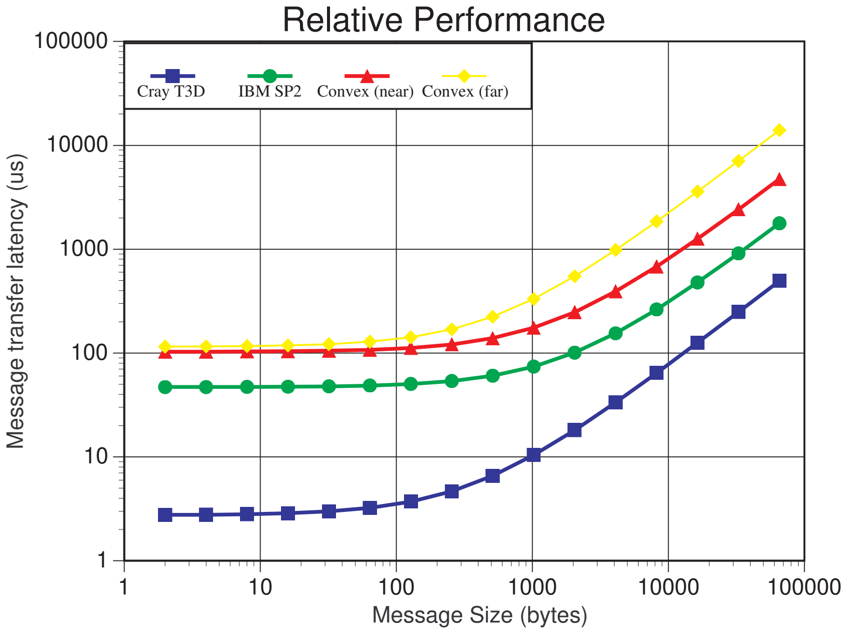
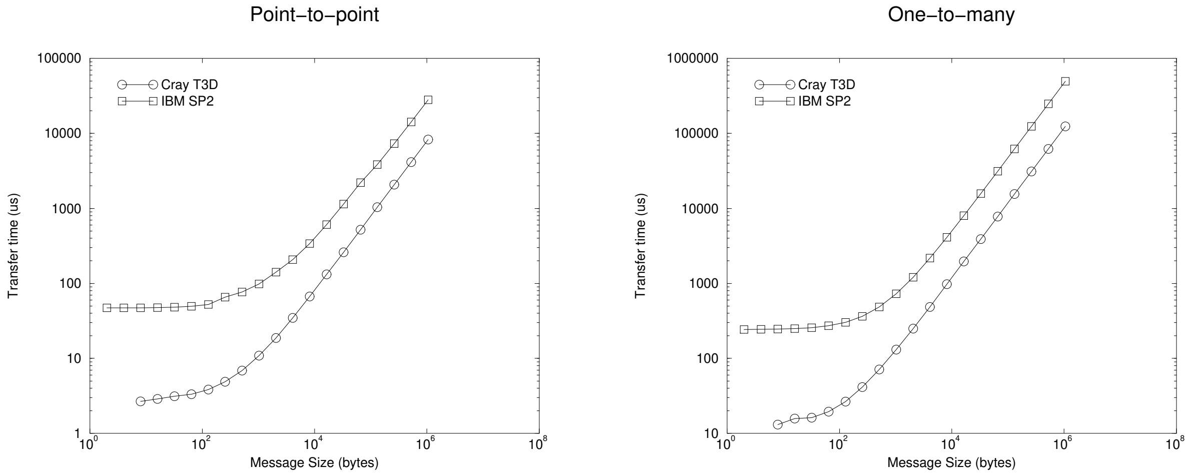
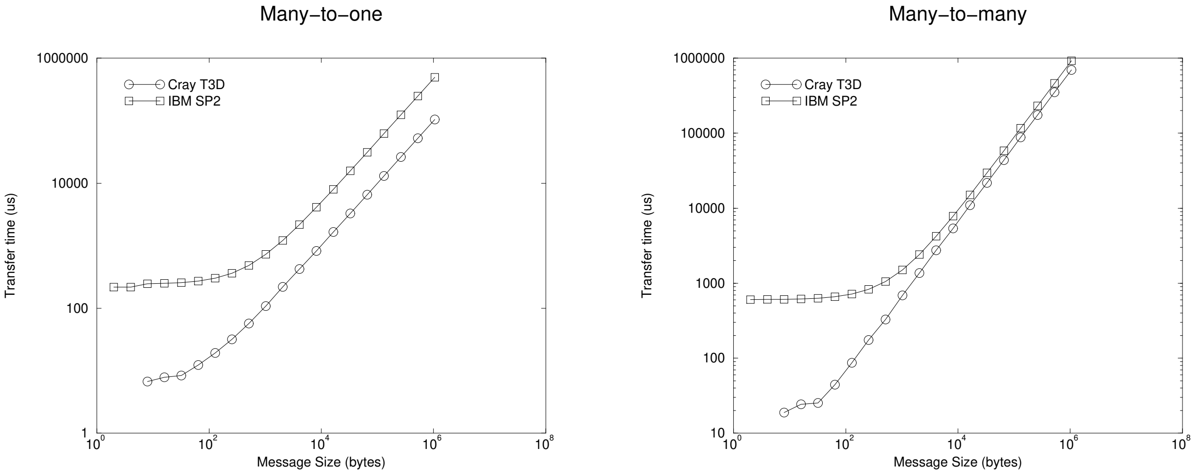


Figure 11: Point-to-point performance of the Cray T3D, IBM SP2, and Convex SPP-1000



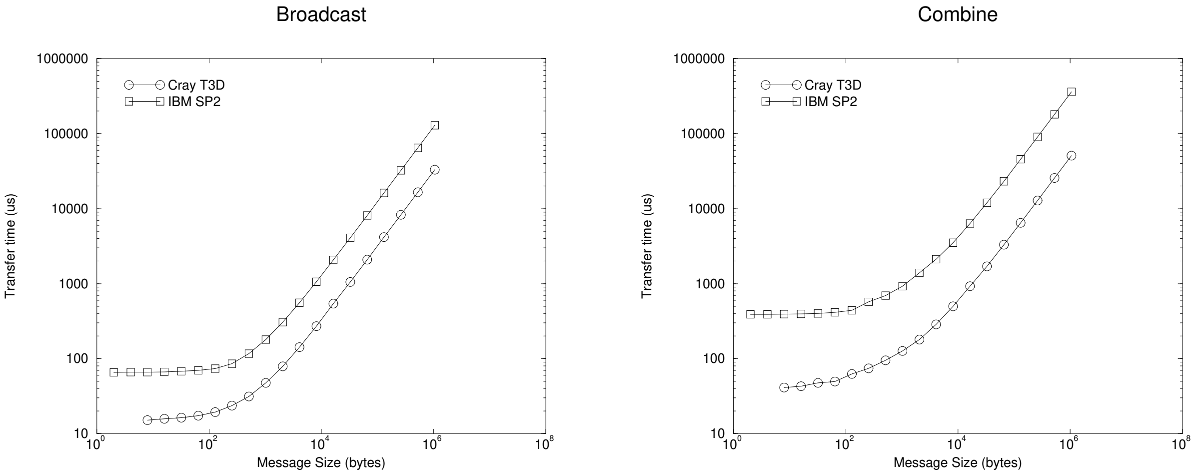
(a) Point-to-point comparison

(b) One-to-many comparison



(c) Many-to-one comparison

(d) Many-to-many comparison



(e) Broadcast Comparison

(f) Combine comparison

Figure 12: A comparison of communication times of the T3D and SP2

It seems that contention in the T3D plays a large role for larger messages. Breaking down Many-to-many communication into multiple broadcasts would appear to be a better solution.

What these graphs fail to measure is the overlap of communication with computation. If the communication can be effectively overlapped with useful computation, then communication latency might not be as important. Both of these machines have separate communication engines, so such overlap is possible in theory at least. One concern, however, is that the overlap of communication with computation on the IBM SP2 has been measured at only around 4 percent for one large application [11]. While the Cray T3D has more sophisticated communication support circuitry, the question is whether it can be effectively used concurrently with the processor. This will be one target of our future work.

7 Future Work

Currently, some of the NAS kernels are being examined for use in evaluating the usefulness of the data gathered by this paper. These NAS kernels will be instrumented in all three message-passing libraries, and their execution times profiled. By mixing in computation, the actual impact of communication performance can be seen on a "real" application. Once we have gained experience with these NAS kernels, an actual application will be profiled. By using the results of this study, we hope to be able to tune the performance of this application.

These communication kernels are also expected to be run on the Cray T3E, when it becomes available (winter 1997). It will be interesting to examine the impact of a more updated design will have on the communication performance.

8 Conclusions

One inherent problem is that ease of use, and portability requires an application to sacrifice performance. If minimizing latency is the prime concern, then the SMA library provides an order of magnitude superior performance over both MPI and PVM. The SMA library is available only on the T3D, and has the drawback of not maintaining cache consistency. If the application developer is well versed in the behavior of the intended application, then hand coding some time critical sections of code may be worthwhile.

However, for large applications that are not easily characterized, it may not be worth the effort to rewrite some portions of code. In fact, for most of the basic communication types, MPI manages to perform almost as well as the SMA library for larger messages. The Edinburgh implementation of MPI on the T3D seems to have carefully taken communication performance into consideration. For those applications where the latency of small messages is absolutely crucial, taking the time to convert to intrinsic SMA subroutine calls could provide a significant benefit; for other applications, MPI appears to be quite competitive in performance.

References

[1] G. Abandah. Ford group presentation, November 1995.

[2] G. Abandah and E. Davidson. Modeling the communication performance of the ibm sp2. In *10th International Parallel Processing Symposium*, page ??, 1996.

[3] R. Arpaci, D. Culler, A. Krishnamurthy, S. Steinberg, and K. Yelick. Empirical evaluation of the cray t3d: A compiler persective. In *International Conference on Computer Architecture*, pages 298–307, 1995.

[4] K. Cameron, L. Clarke, and A. Smith. Cri/epcc mpi for cray t3d. Technical report, The University of Edinburgh, 1995.

[5] Message Passing Interface Forum. *The MPI message passing interface standard.* University of Tennessee, Knoxville, 1994.

[6] G. Geist and V. Sunderam. The pvm system: Supercomputer level concurrent computation on a heterogeneous network of workstations. In *Proceedings of the Sixth Distributed Memory Computers Conference*, pages 111–122, 1992.

[7] R. Hockney. Performance parameters and benchmarking of supercomputers. *Parallel Computing*, 17(10):1111–1130, Dec. 1991.

[8] V. Karamcheti and A. Chien. A comparison of architectural support for messaging in the tmc cm-5 and cray t3d. In *International Conference on Computer Architecture*, pages 298–307, 1995.

[9] M. Pahud and T. Cornu. Measurement of the contention phenomena on the cray t3d architecture. 1995, submitted.

[10] T. Stricker and T. Gross. Optimizing memory system performance for communication in parallel computers. In *International Conference on Computer Architecture*, pages 308–319, 1995.

[11] T. Tabe. Ford group presentation, December 1995.