An Efficient Transport Independent Active Messaging Implementation for PVM

Philip J. Mucci
`mucci@cs.utk.edu`

August, 1998

Contents

List of Tables

List of Figures

1 Introduction

Over the last ten years, the popularity of networks of workstations for distributed computing has continued to rise. Rapid advances in processor and interconnect technology are bridging the performance gap between workstation clusters and traditional supercomputers The demand to solve larger, more sophisticated problems in the shortest amount of time has served as the driving factor.

There exist two major communication models that are used to write distributed or MIMD applications, shared memory and message-passing. With shared memory, all applications share the same virtual address space and all communication is implicit upon a memory copy from one task to another. Depending on the implementation, the application/compiler gives hints to the system regarding page ownership and the propagation of changes in an attempt to minimize the cost of communication. In distributed shared memory, the address space of each process are physically separate, but shared regions can be negotiated.

In the message-passing model, data is exchanged between threads of execution via explicit functions for sending and receiving. The two most popular application programming interfaces (APIs) are the Message Passing Interface (MPI)[SOHL+96] and Parallel Virtual Machine (PVM)[GBD+94].

PVM was developed at Oak Ridge National Laboratory and was one of the first freely available software packages for practical distributed computing. PVM Version 3 developed at The University of Tennessee included major improvements in functionality and performance and subsequently gained a tremendous following. Over the past few years, PVM has experienced widespread use in both academia and industry, so much so that the PVM API was adopted by many vendors and shipped with their machines as an alternate message passing layer. Although primarily a research effort, PVM paved the way for a more advanced message-passing API, namely MPI.

MPI (Message Passing Interface) is a rich standard that was finalized in 1994 as a collaboration between commercial and academic research institutions, including the developers of PVM. A variety of commercial and public domain implementations of the MPI standard are now available for a wide range of architectures. Version 2 of the MPI standard (MPI-2) was recently finished, although complete implementations have yet to appear. MPI-2 adds some functionality and addresses several shortcomings of the first version, specifically the lack fault tolerance and dynamic process creation. Recently, MPI's popularity has continued to increase due to its rich functionality, good performance, and serious commitments from many high performance computing (HPC) vendors.

1.1 PVM Performance

Faster, more affordable networks such as 100Mb/s Switched Ethernet and ATM have become a common occurrence in today's computing lab. As a result, many bottlenecks have surfaced in the PVM code, operations that are now of significant cost relative to the service time of the network. User's of PVM have been frustrated by the lack of increased network performance when switching to new hardware. Most users are moving from 10Mb/s Ethernet to a medium that is an order of magnitude faster, yet experiencing sub-linear application speedups. In addition, developer's have been frustrated by the lack of modularity in PVM's communication layer. PVM uses the socket interface to do all of its communication, the most efficient being PVM's `RouteDirect` option that allows tasks to communicate directly over TCP sockets. However, TCP implementations have been shown to be inefficient on many platforms[KP92][KP93][CFF$^+$93]. PVM's use sockets means that in order for one to port PVM to a new network interface, a stream-socket compatibility layer would need to be written. The very nature of byte-stream based communication can cause performance problems when used in a message passing environment. Byte-streams are reliable, undelimited data feeds, originally intended for use by data-flow type applications. Multiplexing of the message buffers must be performed twice, once at the transport level for delivery to the application and again by the application's communication library to maintain ordering and consistency of the user's sends and receives.

1.2 Protocol Overheads

These performance problems are not unique to PVM. Kay and Pasquale, Stevens, Lazowska, Culler[PP93] [KP93][KP92][CFF$^+$93] and others have authored numerous papers analyzing communication bottlenecks in software layers and how to address them. This research is largely responsible for the revolution in networking performance both at the software and hardware levels. A variety of optimizations were targeted specifically at the TCP/UDP/IP protocol suite. While these changes were well received, these protocols and the socket API still fell well of delivering the underlying hardware's capabilities for local area networks with link speeds beyond 100Mb/s. Multiple layers of protocol processing and additional memory copies still dominated the cost of message transmission producing prohibitively high latencies and reducing bandwidth. This had the undesirable effect of delaying the acceptance of cluster computing as an alternative to using MPPs.

David Culler at the University of California at Berkeley (UCB) recognized the potential of clusters lay hidden within the interconnect technology. Culler and his graduate students sought out to design a lightweight, reliable communication layer unhindered by the BSD socket API that would allow the user to make full use of the underlying hardware's performance. In addition, this layer was to facilitate very simple and elegant implementations of higher level protocols and communication layers. Out of this effort, the concept of Active Messaging was born[vEGS92].

1.3 Active Messages

At the lowest level, an Active Message is a packet of data sized to match the underlying transmission unit of the network. As an Active Message, there are a set of specifications as to what type of information it contains and how the message is to be handled. Every Active Message contains the address of a *handler*, a fixed number of arguments to that handler and any associated "bulk data". The handler is a user-defined function that is called implicitly at the time of reception. Its role is to integrate the information from the Active Message directly into the application. This differs greatly from the usual method of event-driven programming, where data is received, processed and then handed to the appropriate function *by the application.* To the uninitiated, an AM may appear to be an remote procedure call (RPC), but there are some important differences. Active Messages are designed to be as fast and as efficient as possible, and are exceedingly lightweight operations. Handler execution occurs almost simultaneously with reception and in the same execution context. Handlers must run to completion. They are not allowed to block or do any sort of I/O except for sending a reply. No sophisticated buffering or scheduling is required. Usually, buffering only occurs is on the sender side when the message is aligned and formatted for dispatch to the communication substrate. The only scheduling required is the detection of available data at the substrate so that the handler can be invoked. When communicating with this model, the application must explicitly schedule service of the network. This is called *polling.* On the other hand, RPC's make no assumptions about when the network is to be serviced, and rely on the operating system to buffer the network to avoid dropping messages.

By using AMs, protocols can be tailored to suit the demands of an application or family of applications. Thorsten von Eicken summarizes the AM model nicely in [vE94].

> This [the architecture of AMs] minimalist approach avoids paying a performance penalty for unneeded functionality. The power of Active Messages comes from the ability to customize the message formats and the handlers, and from the simplicity (hence efficiency) of the implementation.

Active Messages are not intended to be used directly by application engineers, but rather by higher level communication abstractions providing services like distributed shared memory or message passing, in this case, PVM. The user's application will be issuing PVM send and receive calls which will map to the the exchange of Active Messages. By doing so, we are streamlining PVM's communication path in the hopes of a more robust and flexible message-passing environment.

2 Previous Work

The earliest known predecessor to Active Messages was the VMTP[Che98] protocol developed in 1989. The goal of the authors was to provide a general protocol optimized for small packets and request-response traffic. For a variety of reasons VMTP never became widely accepted, primarily due to the timing of its release. It represented a solution to a problem that had not been widely exposed. At that time, research was just beginning on communications bottlenecks. Later studies done by Kay and Pasquale[KP93] effectively pinned down the major points of performance degradation in the IP protocol suite. Key contributions of this work was a cost analysis of the various protocol processing operations like memory movement, message demultiplexing, buffering, timer management, interrupts, kernel traps and shared device access. This work provided a quantitative basis for the conclusion that the IP based protocols are not well suited to local-area parallel and distributed computing environments. More precisely, IP based protocols cannot make full use of the highly-reliable, low-latency, high-bandwidth interconnection networks used in cluster computing.

Active messages were initially developed by David Culler and a PhD student of his, Thorsten von Eicken at UCB. Their work hinged on previous studies done by E. D. Lazowska[BALL90] and A. Thekkath[TL91] regarding light-weight remote procedure calls and their related performance bottlenecks. This, in concert with Culler's own knowledge and experience in the design of message-driven multiprocessors, led to the development of the Active Message.

UCB's first implementations of Active Messages were highly specialized and ran on a very limited subset of hardware. The very first implementation ran on the Thinking Machines CM5 and the nCUBE/2 as a means of speeding up the vendors message passing layer. Their effort proved remarkably successful and overwhelmingly outperformed the vendors message passing layer. The new passage passing layer provided exactly the same API to the user, but the implementation was based on Active Messages. The result was an eighty percent decrease in startup cost and a sixty percent decrease in per byte cost. Thinking Machines quickly realized the importance of this effort and subsequently purchased the software and shipped it as part of their run-time system. This code was highly specialized for the CM5. Many routines were hand coded in assembler for optimal performance, and indeed it paid off. The nCUBE implementation was very similarly designed. It achieved slightly less impressive numbers due primarily to the latency of initiating bus transfers (DMA) and protected access to the networking hardware.

The effort at TMC both justified and paved the technical pathway for other successful implementations of Active Messaging. The next logical step was to determine if the same approach would benefit networks of workstations (NOWs) with high performance networking cards. The test-bed for this implementation was a network of HP workstations with a high performance FDDI card.[Mar94] This setup was unique in that the FDDI interface card was directly accessible from user space. Another key difference in this implementation was the inclusion reliability and buffering. As with any LAN, this network had the capability to corrupt or drop packets, albeit at a very low rate. In addition, the packet size was much larger than that of the CM5 or the nCUBE/2, introducing cache and congestion effects. Rich Martin[Mar94] of UCB did the implementation and it proved remarkably successful, reaching 12MB/sec out of a possible 12.5MB/sec, with half-power point[1] of 176 bytes. That platforms native TCP stack had $n_{\frac{1}{2}}$equal to 1352 bytes. The problem of process/memory space protection was solved with user-level locks and a separate scheduler that arbitrated user processes among the device. It was at this point that Dr. Culler and his colleagues decided to standardize the Active Message API. In every implementation, their network was a static entity with N processors. Additional AM packages appeared out of UCB, all following the same model of a static network with a fixed number of preallocated message buffers per host.

After graduating Dr. Culler's group, Thorsten von Eicken moved to Cornell University where he continued his work. Von Eicken's group did an implementation of Active Messages using Sparc 20's connected by Fore's SBA-100 ATM adapters.[vEABB94] These were early OC-3 (155Mb/sec) cards that provided no hardware support for segmentation and reassembly of the small (48 bytes) ATM cells. This meant that each of the cells had to be directly manipulated by the AM library. Like the other AM packages, this implementation required modifications to the kernel. In this case, the kernel was modified with a simplified device driver and highly optimized kernel traps to read and write raw cells. This implementation did not perform nearly as well due to the overhead of accessing the device for each individual cell. The reason this was not a problem for the CM5 was due to the integration of the communication engine with the CPU and the operating system. The CM5's network interface was memory mapped by the processor and timeshared by the operating system, thus each process could read and write the network registers without interfering with other jobs. With the ATM card, cells had to be delivered across the bus, in this case, the Sparcstation's SBus. Von Eicken found that the dominating cost in his implementation was the trap to the kernel, which had to be performed for every 48 bytes of data. Nevertheless, the numbers proved superior to the protocols available at that time. The best API at that time was Fore's AAL 5 interface, which achieved an $r_\infty$of 4MB/sec., Von Eicken's package achieved 5.5MB/sec.

Von Eicken later developed the U-Net package as a followup to this work.[ABvE95] The goals of U-Net were to solve the performance problems of the previous implementation on more advanced ATM hardware. The U-Net project aimed to remove the kernel completely from the communication pathway through the implementation of virtual *endpoints*. The paper argues that all protocol stacks should be executing at the user level and that the operating system should provide direct and protected access to the network hardware. The actual implementation of the Active Message protocol was quite similar to his previous work, the difference being in the way in which the ATM card was accessed. The U-Net system allowed one process at a time to directly access the network interface buffers. Any other running processes that tried to access the device accessed a "shadow" device that existed in kernel memory. The platform for this work was again SPARCstation 20's, this time with Fore's new SBA-200 OC-3 ATM adaptor. The difference between this adaptor and the SBA-100 is that the SBA-200 has a full RISC CPU (an Intel 960) capable of performing direct memory access (DMA) in addition to some local memory. This allowed much more flexibility

[1]The half-power point ($n_{\frac{1}{2}}$) is defined as the message size at which half the peak (r_∞) measurement is obtained. Normally $n_{\frac{1}{2}}$is referred to in terms of bandwidth in megabytes per second.

in terms of implementation and outstanding results were achieved. The performance was only 200KB/sec off of the peak fiber speed for large packets. Like the previous implementation, the U-NET package required modification of the operating system, and in this case, required modification of the firmware existing on the communication adaptor.

Von Eicken's work has formed the basis for a large industry sponsored effort called the Virtual Interface Architecture or VIA.[ICC97] The goal of VIA is to standardize an API very similar to that of U-Net. That is, to offer direct, protected access to the network interfaces buffers in a timeshared environment. There is no reference implementation and there is no particular device or substrate targeted. VIA is a standard intended for implementation by the vendors. The evolution of VIA from U-Net closely mirrors that of MPI from PVM. It represents a collaborative effort to transfer research to the commercial marketplace.

Perhaps the most comprehensive work with Active Messages recently appeared in a work from MIT. D. Wallach's paper *ASHs: Application Specific Handlers for High-Performance Messaging.*[DAWK97] His system facilitates the importation of provenly correct code into the networking layers of the operating system on an application by application basis. These code fragments are executed on message arrival at the device and run in the address space of the user process associated with that message. ASHs function to direct message transfers using copies or DMA, as well as respond to asynchronous events such as retransmissions and interrupts. The ASH system also integrates the concept of Dynamic Integrated Layer Processing (DILP) to allow portability among a wide variety of devices and packet formats. The ASHs that are installed by an application are analyzed at run-time and integrated dynamically with the systems packet processing and demultiplexing system. For example, this means that *at most* only one pass is ever made over data received on a link. Consider the case where data must be copied from a network interfaces buffers, a checksum computation must be performed and then header information stripped from the packet which is then assembled and delivered to the appropriate application. With DILP, this entire process runs in user space with direct device access and only makes one pass over the data. This work is relatively new and requires extensive support from the OS. Currently it only runs on a DECstation 5000 running Aegis, an exo-kernel[DREJO95] based operating system. Nevertheless, the performance and flexibility allowed by the ASH system with Aegis is nothing short of amazing and provides us with a glimpse of what future production operating systems might have to offer.

3 Goals

The primary goal of this work is to accelerate PVM via Active Messages. Our work aims to lower PVM's protocol processing overhead and allow PVM to take advantage of more advanced communication API's. This will be facilitated by a generic Active Messaging API integrated into the PVM system. The software, including the Active Messaging library and the revamped PVM code, will be called PVMAM.

3.1 Restrictions

As in other implementations, end-to-end performance is of the utmost concern. However, here we place restrictions on what parts of the system are open to development. This is where this implementation of Active Messaging is so different from that of the others. Most other packages introduce modifications at all levels of the software hierarchy, especially the kernel of the operating systems. This is unacceptable for inclusion in the reference PVM implementation. For all practical purposes, PVM is a production application in widespread use. Or more appropriately stated, it is run on machines for general purpose use, in government, industry and most importantly educational institutions. These locations are highly suspect of any modifications to privileged code not shipped by the vendor. In a sense, modifications of system software can only be trusted when originating from the vendor or a commercial operation.[2] As PVM and public domain implementations of MPI are used heavily as research tools, it lowers the possibility for commercial success of any company looking to market third party PVM and MPI software based on custom communication protocols. So, we are left with a dilemma, a problem that is directly addressed by this thesis. Can we benefit from better application protocols by using Active Messaging over the relatively poor communication API's shipped by the vendors?

3.2 Code Structure

PVMAM layer will be very portable, maintainable and extensible. To that end, we will use an object-oriented approach to encapsulate different concepts in our implementation. By doing so, parts of the system can easily be modified or changed and communicate behind a well defined interface. We will shy away from using C++ for two reasons. First, many PVM users may not have access to a C++ compiler. Second, C provides us with more explicit control over memory referencing and allocation; translating to better performance.

3.3 Communication Layers

A key issue in advancing PVM's performance is to design our Active Message library to provide a uniform interface to faster, more advanced communication layers. This requirement stems from the extreme difficulty in porting PVM to a new communication API. These API's may be drastically different from one another.[3] Not only might the API be different, but the network may have very different characteristics. These differences may require modifications to the higher level communication layers in order to facilitate efficient usage of the medium. Our system will define *hints* that the substrate-dependent portion of our code will export to the higher layers of our active messaging library. In addition, this substrate-dependent portion of our Active Messaging package should be easily replaced. The methodology must be straightforward enough such that any developer should be able to write this code for a new transport and use our package for increased PVM performance.

[2] It is common practice to purchase and install device drivers.

[3] Consider the difference between BSD Sockets and System V shared memory.

Initially, we will target three communication substrates, TCP, UDP and the Myrinet BPI. Our implementation of AM over TCP and UDP will provide a strong point of comparison with the reference PVM implementation. Given suitable options, PVM uses TCP sockets between pairs of processes to communicate. In this case, PVM can deliver most of TCP's bandwidth to the application. However, PVM greatly affects the latency of these transfers with its protocol processing. We wish to greatly reduce this by using Active Messages. Our UDP-based implementation should also benefit from lower latencies due to the nature of datagram communication and the high reliability of most LANs. Myrinet currently has no native support in PVM. This means that the virtual machine must be running TCP/IP over these devices in order for PVM to take advantage of this sophisticated hardware. Again, the problem is that most IP stacks are not nearly capable of delivering the performance of the underlying hardware. PVMAM will exchange data with Active Messages, which makes use of the lowest level API provided by the vendor. These API's are usually highly tuned and frequently obtain bandwidth and latency figures within five percent of the hardware. Thus, optimizing the protocol processing portion of PVM will become that much more important, as performance will be limited by software, not the speed of the network.

3.4 Providing a Dynamic Network

PVMAM needs to be able to support a dynamic network. By dynamic we mean that during the course of an application, processing nodes could be added or deleted from the virtual machine. The very nature of PVM is based on this concept, unlike MPI-1 in which the virtual machine is deemed to be static. This greatly reduces code complexity and facilitates many optimizations. In fact, most current Active Messaging systems are based on a static network making them unsuitable for our purposes. PVM's flexibility in this regard, introduces significant data structure and buffering complexity into our implementation of Active Messages.

3.5 Reduction of Overhead

Here will provide a brief outline of the most severe sources of overhead. Of particular interest, is the work done on profiling the overheads of TCP/IP in [KP96]. While most of their work is in regard to the performance of the in-kernel protocol stack, the conclusions made apply to any software messaging layer. Generally, protocol processing is segregated on the basis of data-touching and non-data-touching operations. For further information the reader is referred to [KP93][KP96][KP92][PP93][Gus90][CFF$^+$93][KC94][Ous90].

3.5.1 Memory Movement

Data copies can be the single most influential factor in a messaging layers end-to-end performance. For TCP/IP under most operating systems, as many as four copies may occur from application to application. This involves the copy from user-space to kernel space, followed by a copy to the device and then back again on the remote machine. Additional copies may be required for staging a message or for the purposes of reliability. With respect to the speed of communication, memory copies used to be relatively fast. However this is quickly changing as gigabit per second technologies like SCI and Myrinet come to market. Works like U-Net, ASHs and HPAM go through great lengths to avoid copies as the are responsible for significant performance degradation. In PVMAM, we will avoid copies as much as possible, given proper support from the OS and the communication substrate. The request-reply nature of Active Messages will allow us to use any advance knowledge of the destination address to amortize the cost of data movement.

3.5.2 Kernel Traps

Active Messaging has brought to light many bottlenecks related to crossing protection boundaries. With faster substrates, the overhead entering the kernel and enforcing the address space becomes quite important. Even the timing routines used for stamping messages can have significant effect on performance. For further detail on the effects of operating systems on application performance, the reader is referred to [Ous90]. Since most PVM users do not have access to any specialized network interfaces, they will be using PVMAM on top of either TCP or UDP. Thus we must carefully evaluate our design decisions in the hopes of amortizing the cost of kernel and network access.

3.5.3 Buffering

Due to the loose synchronization found in most parallel applications, a member process may have not issued a receive before another process has executed the corresponding send. In addition, other data from previous transmissions may be still in flight over the network. In order for the application not to deadlock waiting for a particular communication request to complete, it must provide some buffering. For that reason buffering is of key importance to messaging layers. Buffering can greatly affect overhead due to the usage and performance of the memory allocator and associated data structures. PVM buffers all of it's incoming messages and provides no facility to receive them *in-place*. It allocates room for the incoming message at the time of reception possibly resulting in numerous calls to the memory allocator and prohibiting use of any a-priori knowledge of the messages destination. Our implementation will provide preallocated, fixed sized packet buffers, sized to match the maximum transmission unit[4] of the network. In addition, the number of these buffers will be tuned such that multiple packets can be pipelined in-transit without congestion in the network or protocol stack.

3.5.4 Reliability

Any messaging layer must be able to reliably and predictably deliver its data. Physics dictates that software must compensate for potential loss or corruption of data no matter how small the

[4]This can be interpreted as the maximum packet length of the substrate, not the physical medium over which it is being transferred. UDP has an MTU of 8192 bytes, while 10BT has an MTU of 1500 bytes.

error rate. Data copies are also part of reliability because if a message is lost or corrupted beyond recovery, it must be retransmitted. Andrew Chien et al in [KC94] finds that even in the highly efficient CMAM layer, up to twenty percent of instructions involved in a multi-packet transfer are for reliability. PVM solves the reliability problem by using a sophisticated buffering/retransmit mechanism in the daemon, and by using TCP/IP among direct-routed clients. Active Messages attempt to reduce the cost of reliability by making some assumptions about when and how they are used. When Active Messages were developed, they were targeted at MPPs and workstation clusters. Both networks exhibited very low packet loss and corruption rates. As a result, speed is gained by greatly simplifying the implementation. Rich Martin[Mar94] says in his paper "In exchange for high-performance most of the time, we occasionally pay a high penalty for the rare occurrence that a packet is corrupted." This is a valid assumption, as many networks are highly reliable and support error correction at the link level.

3.5.5 Protocol Handling

Most messaging systems encapsulate the concept of the state of the network. More precisely, only certain events can happen under certain conditions and after a certain amount of time. Demultiplexing, fragmentation, handshaking, connection management, timeouts and address resolution are all part of the protocol handling present in TCP/IP, and to some extent UDP. For small packets, these operations can take a large percentage of the time for a message exchange.[KP96] Note that this kind of processing cannot happen off-line or after a message has been transmitted. This kind of processing stands directly in the way of lowering communication latencies. Implementations of UDP often do not much perform any better, weakening the argument that UDP is a truly lightweight protocol.

Active Messages directly targets this area of communication bottlenecks. Data movement must always occur, only the number of passes over the data can hoped to be reduced. However, not so is the case with protocol processing. Active Messages essentially minimizes this cost. There is no state machine, no connection management logic, no demultiplexing operation and no chained data-structures. All protocol-like operations are to be performed *by the application.* Thus, the decision as to what processing is necessary for the application to execute correctly is left to the developer.

4 AM Implementation

Our implementation of Active Messages is focused around two data structures, *interfaces* and *endpoints*. Both are object-oriented structures that attempt to simplify the roles of various parts of our Active Messaging subsystem. Active Messages are sent to through an *interface* to a specific *endpoint*. Applications may use more than one interface at a time, but multi-threaded applications using the same interface are not supported.

The interface represents the communication substrate over which messages are sent. An interface contains:

- Information about the underlying transport layer. This primarily consists of pointers to the different transport functions.
- A list of tasks or processes (endpoints) the interface is connected to.
- Parameters controlling the AM protocol such as the maximum number of messages in flight and timeout interval.
- Buffers for incoming messages. This implementation cannot receive messages *in-place*. In other words, the non-header information contained in the AM must be copied from the underlying substrate into a staging area. The AM's handler is then free to copy this data into the application's (PVM) buffers.
- Statistics for the interface. These are used to short-cut redundant reliability computations for all connected tasks.

An endpoint represents a remote process with which Active Messages can be exchanged. An endpoint contains:

- A string-based network address of the connected task. This is used during connection establishment.
- A native address of the connected task. This address is handed directly to the transport layer in the message exchange functions.
- Outgoing request and reply buffers. These buffers are where headers are built and the user's data is copied to ensure reliability. If the underlying transport is reliable, then these buffers are never allocated.
- Reliability information. For unreliable transports, each endpoint keeps track of the number unacknowledged packets. This data is used, as in the interface, to shortcut redundant reliability computations for this particular task.

- Task state information.
- Protocol hints. The AM library keeps *hints* as to the most recently freed buffers. For intensive communication patterns, buffers are constantly allocated and freed. These *hints* improve the performance by reducing the lookup time and the thrashing of the TLBs and data cache.

The handling of these two structures is separated by the use of the API. The interface structure is allocated and filled upon initialization of the AM library. Endpoints can be added or removed dynamically. By implementing the library to be dynamic, we complicate the task addressing issues, but gain the ability to do implement client-server and fault-tolerant type applications.

5 The Active Message Protocol

This implementation uses the protocol presented in [Mar94]. This protocol is designed around the concept that every network has a *depth*, or the maximum number of packets that can be held in-flight or buffered by the network. In practice, this number is variable depending on the size of the packets being transmitted. However, in-order to simplify the implementation our Active Messaging layer assumes that the packets are full. The reason for this is that the majority of messages exchanged will be larger than the MTU of our AM layer. The depth parameter is tunable as either a run-time parameter passed at intialization time or via an environment variable.

This protocol requires that every Active Message generate and transmit an explicit acknowledgment. Before we describe the protocol, we must establish some assumptions and definitions.

- An Active Message has a maximum size, 8192 bytes.
- An Active Message has a request and reply, each can have a different handler.
- Active Messages are allowed to arrive out-of-order.
- Every endpoint contains separate request and reply buffers for every other endpoint.
- Each request and reply buffer is numbered as to it's depth in the buffering structure. This is referred to as the *instance* number.
- Every request and reply buffer has a binary *sequence* number used to detect dropped or duplicate data within that instance.
- No timer-lists are maintained and only requests are allowed to timeout.

The protocol is best explained with an example. On the sender's side:

1. Process A sends an Active Message to Process B over interface X.

2. B is looked up in interface's task table.

3. B's request table is examined for a free buffer. If none is found, keep servicing the network until a buffer becomes available.

4. When a buffer is becomes available (instance I), mark the buffer as in-use, build the AM's header and timestamp the request.

5. The message is sent to B using the interface's underlying transport. If the transport indicates a blocking condition, then the network is serviced and transmission is retried.

6. If the transport is unreliable, then A copies the "bulk data" of the message into the request buffer. By copying the data after transmission, we hope to hide the latency of the network, especially if our machine is an SMP and the second processor can do the network I/O.

At the receiver, B:

1. Either the AM library or higher layer explicitly services the network.

2. The AM layer obtains an incoming buffer. When data becomes available, the transport receives the message into that buffer.

3. If no data is available, check the request buffers for timeouts. *Recall that there are only a fixed number of these buffers, the network depth.* If some requests have expired, double their timeout interval and retransmit them. Then try to service the network again. If the requests timeout interval goes beyond a certain limit, then the request is dropped. This limit is tunable at initialization time.

4. The AM library determines that the message is a request and it examines the header for reliability.

    - Lookup the reply buffer for task A, instance I.
    - If the reply buffer is not in use, then continue processing.
    - Else if the sequence number of the incoming request is different from that in the reply buffer, then the previous reply was successfully transmitted so continue processing.
    - If these checks fail, then the previous reply was dropped and it is retransmitted.
    - Mark the reply buffer for task A, instance I as free.
    - Lookup the address of the user's handler and invoke it.

5. If message's handler was called and it did not send an explicit reply, send a blank reply. *Remember that all requests must have a corresponding reply in order for the protocol state information to be properly updated.*

Back at the sender, A:

1. Same as 1 through 3 as above.
2. The AM library determines that the message is a reply and it examines the header for reliability.
    - Lookup the request buffer for B, instance I.
    - If the request buffer is in use and the sequence number of the incoming reply matches that of the request buffer, then this reply is valid.
    - Otherwise, a duplicate reply was received and it is dropped.
    - Mark request buffer for task B, instance I as free and toggle the request buffer's sequence number.
    - Lookup the address of the user's handler and invoke it.

The reader will notice that the protocol does not guarantee ordered reception of data. Thus higher layers must be able to detect this condition and handle it appropriately.

5.1 Task Addressing, Naming and Management

Since our target environment is PVM, it was decided to base lookups upon PVM task identifiers or TIDs. PVM assigns TIDs dynamically when processes are spawned. Our AM library uses these TIDs to lookup remote endpoints in a circular linked list. Frequently, we expect numerous AMs to be transmitted to a specific endpoint as part of a longer PVM message. Therefore, when a task is successfully found in the list, that task is moved to the head of the list to reduce the time of future lookups.

This AM layer is designed to be connection-oriented. The reason for this was that it is far simpler to emulate connection-oriented protocols with connectionless protocols than the other way around. Specifically, the connection routines simply become no-ops. However, there are two problems with this arrangement. The first is that it imposes a knowledge of both endpoints of the communication path at the time of connection. The second is that no more than two processes should ever try to connect to each other. Otherwise, race conditions could develop between the arbitration and the messaging functions. This cannot when using TCP. When connecting two processes with TCP sockets, one side calls `connect()` and one side calls `accept()`. Only the task calling `connect()` needs to have information about the remote process. Once the connection is established, the side calling `accept()` gets returned the address of the connecting entity. In addition, this scheme allows multiple clients to connect to a master in any order. This behavior is highly unportable between communication layers and would result in a lot of extra code. The solution is to simple impose an exclusive, rigid, ordering scheme on the connection process that must be handled by the higher level software layers. To ease this implementation, this AM implementation separates the preallocation, connection and addressing of the endpoints. In PVMAM, a remote task is not able to be communicated with until it's endpoint has been inserted into the local processes task table. However, it can be allocated and connected to, ahead of time.

In order for a connection to be established, we must provide the address of the remote process to the AM library. Unfortunately, network addresses are always binary arrays in host-byte order. In order to solve this problem, each AM task computes it's own address at initialization time. This address is transformed to network-byte order and turned into a character string. This string can then be obtained from the AM library through an API call and passed to remote processes for connection.

5.2 Address Identifiers

As this layer is designed to be highly portable and used among heterogeneous clusters, our AM layer must contain some abstraction of virtual addresses. Internally, this abstraction is used only to lookup the text address of the AM's handler for execution. However, the user could use them externally to handle any necessary address-independent computations or data storage. Since we expect a small number of handlers and/or structures, our address identifiers are implemented as direct offsets into a dynamically sized array of addresses. The identifiers are managed with a small set of functions in the AM API.

5.3 Transports

In our AM library, we chose to implement three different communication substrates, TCP sockets, UDP sockets and the Myrinet BPI. Each of these protocols has different semantics, usage and characteristics. In this section we will discuss some of the implementation issues associated with using these protocols as substrates for our AM layer. Raw bandwidth and latency of the three protocols can be found in figures 8, 9, 10 and 11.

5.3.1 TCP

TCP is a connection-oriented, reliable, byte-stream based protocol that can send messages of any length. Message are undelimited; their boundaries are not preserved when communicated. Because TCP is reliable, we can avoid doing copies of the user data and processing most of the AM protocol. This will significantly increase the performance of our library. Since our implementation presents a connection-oriented model to the higher layers, the connection process maps naturally to the similarly named socket calls. However, this also implies that receives must be addressed to a specific connection[5], which can cause a severe performance loss. The only solution to this problem is to use the Unix system call, `select()`, on all the connections to obtain the one on which data is waiting. This operation can be also be expensive, although not quite as costly as executing a

[5]Otherwise known as a file descriptor.

non-blocking receive for each individual file descriptor. Another problem is that TCP uses discards message boundaries. This is a problem for AMs that are smaller than the maximum size allowed by the AM layer. If we execute a receive with the MTU as an argument, we might possibly read more than one Active Message into our receive buffer. This violates the semantics of our protocol. The solution is to execute a separate receive for the fixed-length header and the body of the message. Although this incurs performance hit, it is the only way to guarantee reception of one message at a time. PVM handles incoming TCP messages in much the same manner.

5.3.2 UDP

UDP is connectionless, unreliable protocol that preserves message boundaries. UDP messages have a maximum length, usually 8192 bytes. This protocol is much more suitable to an efficient AM implementation. The connection process is trivial, the AM connection routines simply map to name resolution functions to verify the destination address. UDP provides us with a single point of contact for all incoming data, thus we can avoid calling `select()`. Lastly, it preserves message boundaries, so only one receive need be executed per message. The only drawback is that it is unreliable. As a result, timeouts must be checked and the network must be serviced in a timely manner. Copies of each outgoing request and reply must be saved for retransmission in the case of an error. Nevertheless, the reduced number of system calls can possibly outweigh the performance impact of this data movement if we have a fast processor. If the machine is an SMP, then we can hope to hide much of the latency by copying the data after transmission. Ideally, the second processor could continue to output the packet on the network interface while the first processor is copying the data into the library's buffers.

5.3.3 The Myrinet BPI

Myrinet is a highly-reliable, switched interface with a link speed of 1Gb/s. In practice, the overhead of the Myrinet software consumes much of this bandwidth as can be seen in figure 10. Nevertheless, it still provides much better performance when used with the Myrinet IP protocol stack. There are two APIs shipped with the Myrinet hardware. The first, simply called the Myrinet API, is a DMA-based library with a FIFO like command interface. The semantics of this library are quite complex, albeit it performs quite well. The Myrinet BPI is an additional programming interface that hides much of the complexity of the API at the cost of some performance. Like UDP, the BPI provides an unreliable, connectionless communication layer that preserves message boundaries. It also has a maximum message size of 8192 bytes. Implementation issues for the BPI are identical to that of UDP with the exception of the latency hiding mechanism. With the BPI, the operating system is not involved, therefore the second processor will simply lie idle. However, we can still gain some performance by understanding how the BPI works internally. When a message is sent using the BPI, the BPI must copy that message to an area of kernel memory that the Myrinet device has direct access to. Once that occurs, the Myrinet card itself copies the data from the kernel memory to the network. This process involves a cache flush after the data has been transmitted to the network. By copying the data before we call the BPI function, we can take advantage of the temporal locality of this process. Since most if not all data caches are larger than 8K, we guarantee the data will come directly from the cache providing a much higher bandwidth.

6 PVM and AM Integration

PVM presents us with numerous difficult issues to solve when integrating Active Messages. The first and foremost being that PVM is designed around non-blocking TCP sockets and `select()` for polling and multiplexing among connections. Internally, the flow of control in PVM depends highly on the status of its incoming and outgoing message buffers and the arguments in the call stack. In addition, the message routing and handling functions suffer from recursion and multiple points of entry and return. Furthermore, the buffering structures are complex with multiple levels of indirection and redundancy. This makes integration of any new messaging layer very difficult, especially in regards performance. In order to address these issues, some background on the internals of PVM is necessary.

6.1 PVM Message Buffers

Outgoing messages in PVM are explicitly or implicitly *packed* before transmission by calling one of the `pvm_pk()` functions. This means that the application's data is encapsulated by internal storage structures before being transmitted. This process normally involves a copy of the user's data, unless the user explicitly requests that the data is packed *in-place*. In this case, the data is not actually copied but recorded with a pointer. Packing also provides a mechanism to encode and/or encrypt the data in a machine-independent fashion. By default, PVM uses a mechanism called XDR, a set of portable encoding routines that handle all of the base C and Fortran datatypes.

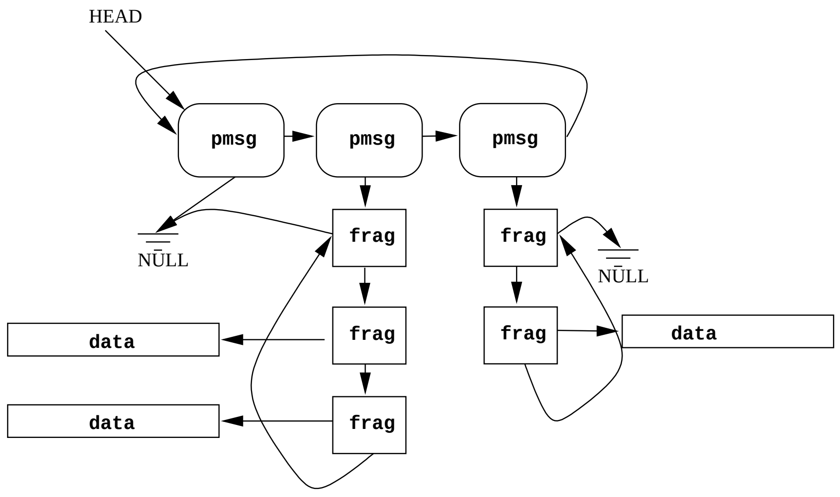

Figure 1: Internal representation of a packed message in PVM

Messages in PVM are stored internally as a list of a list of pointers as shown in the above figure. The top-level list elements are called `pmsgs`. These structures are chained together, each pmsg representing a single call to one of the packing functions for a specific datatype. Hanging off the pmsg structure is a list of structures called message fragments or frags. Fragments contain pointers to buffers that contain the packed application data. These buffers are of fixed size, usually 8K bytes, and are the fundamental unit of transmission across the network. Unlike the psmgs, there may be multiple fragments created by a call to one of the packing functions. In the case of *in-place* packing, the fragments point directly to the application's data. *In-place* fragments may be of arbitrary length.

6.2 PVM Message Handling and Routing

Before tackling the understanding of the messaging functions, we need to understand how messages and fragments are handled at the socket level. As mentioned above, messages are sent and received in fragments. Both the entire message and each fragment are preceded by headers that are sent separately from the application data. When a message is to be sent, a header is built for the entire message that contains the length, the tag and the encoding of the message as well as some other information. After that header is sent, a header is built for each fragment, which is sent followed by that fragments data. This process repeats until the entire message is sent.

Both internally and externally, PVM has multiple points of entry into the message routing and handling functions. The PVM API is built on top of the exported pvm_send() and pvm_recv() functions as well as the internal functions msendrecv(), mroute() and mxfer(). msendrecv() handles a singular message exchange usually to the PVM daemon. It calls mroute(). mroute() handles the routing of outgoing messages and receives incoming messages. It calls mxfer(). mxfer() polls and/or writes to the output socket. It might repeatedly call mxinput() to read and handle incoming fragments or headers. mxinput() reads either a message header, fragment header or fragment data and returns. If the header indicates that the data is for an internal control routine, that routine is executed, which might cause recursion by calling pvm_send().

6.3 PVM Task Management

There are two paths by which PVM messages can reach their destination. The first is to route the message through the daemons. The second is to set up a *direct-routed* connection, a TCP socket between the sender and receiver. For applications that require performance, *direct-routing* offers the best chance of success. These connections are managed by data structures called the task-to-task control blocks or ttpcbs. These structures are connected into a list containing the name, tid and socket address of each of the peer tasks. Connections are not static, they are made implicitly by mroute() if the user has enabled the PvmRouteDirect option. After a connection is attempted, a flag in the ttpcb indicates whether or not it was successful and if data can be sent using this route. Hosts that die or are killed are detected at the remote end as a broken socket and the route is disabled. The dynamic nature of this scheme allows a lot of flexibility from the application developers standpoint. The problem is that it is highly TCP specific and relies heavily on the control flow and re-entrancy of the internal message handling routines. This prevents us from using the internal mechanisms to establish AM connections. In addition, no guarantee is made against multiple tasks simultaneously connecting to the same socket, because it relies upon the semantics of the `accept()` call as described earlier.

6.4 PVMAM Message Handling

The first problem to address is how to make a PVM message look like a series of Active Messages. The natural answer is that every PVM fragment becomes an Active Message in PVMAM. As Active Messages have a small maximum size and the maximum fragment length is tunable in the PVM library, the two sizes can be matched. By doing this, a majority of the PVM message handling code can remain unchanged, in addition to the buffering, packing and encoding routines.

Because of the interested in performance, it was decided to shortcut execution of PVM code as soon as possible in favor of our AM layer. However, we need to make sure we guarantee fair communication with the PVM daemon as well as the other tasks in the virtual machine. In PVM, the socket to the daemon is polled upon every call to `mxfer()`. This is a significant source of overhead, as the `select()` call on multiple file descriptors can be very costly in terms of latency. The solution to this is to have the process be notified asynchronously when the socket to the daemon needs servicing. This is accomplished by setting up the socket to deliver the `SIGIO` signal to the process when data arrives. The handler for this signal merely toggles a boolean flag indicating that the daemon has data. This flag is then checked opportunistically within the new PVMAM code.

6.4.1 Message Transmission

Integration into `pvm_send()` seemed logical as this would remove a lot of of the PVM code paths from execution during normal operation. In addition, integration into `pvm_send()` allows higher level communication layers built upon that primitive to take advantage of any increase in performance. This holds for the combined *in-place* pack and send function, `pvm_psend()` as well.

`pvm_send()` works as follows in PVMAM:

1. Check to see if PVMAM is enabled.
2. Check to see if the destination *is not* the PVM daemon.
3. Check to see if this function has not been re-entered.
4. Check to see if the destination has a task control block (`ttpcb`).
5. Check to see if the destination is open for communication.
6. Check to see if the tag is not an PVM control message.

7. If any of the above checks fail, call `mroute()` and send the message through the daemon.
8. Else, build a message header.
9. Build a frag header.
10. Send the Active Message.
11. Move to the next fragment. Goto 9 until entire message has been sent.
12. Check to see if the `SIGIO` handler has been called. If so, call `mroute()` which will service the socket to the daemon.

One of the primary differences of this protocol is that the network need only be accessed once per fragment for communication substrates that preserve message boundaries. Previously, PVM would read fragment's header, process it, and then read the fragment's data. In PVMAM, the Active Messages contain both header(s) and fragment data. All data necessary for the processing of the fragment is contained within that AM. The fewer number of calls to the transport API directly translates to a reduction in processing overhead for PVMAM.

6.4.2 Message Reception

When `pvm_recv()` or `pvm_precv()` is called and the message has not already been received, control is eventually transferred to the `mxfer()` routine. `mxfer()` waits on possibly multiple sockets until data arrives. When data does arrive, `mxinput()` is called to read and handle the fragment. We choose this routine to be the basis of our handler for PVMAM fragments. However, some modifications to `mxfer()` are necessary to ensure fair servicing of the daemon socket and the PVMAM connections. Now, instead of waiting in `mxfer()` using the rather slow `select()` system call, `mxfer()` enters a loop, alternately polling the AM interface and checking the flag set by the `SIGIO` handler. If the `SIGIO` handler has been called, then data is available at the daemon socket and the original version of `mxinput()` is invoked. If fragments arrive on the AM interface, then the modified `mxinput()` routine is called as the handler of the Active Message. This modified version of `mxinput()` extracts the entire fragment and processes it. If this fragment is the last in a series, signifying the reception of a complete message, then a flag is set causing `mxfer()` to exit the loop and put the message on the receive queue.

`pvm_recv()` now works as follows:

1. If the message desired is not on the receive queue, repeatedly call `mroute()` until it appears.
2. `mroute()` checks the route and calls `mxfer()`.
3. `mxfer()` repeatedly calls the AM network service routine until:

- The `SIGIO` handler has been executed indicating data waiting on the socket to the daemon.
- An entire message has been received over Active Messages.

4. If the `SIGIO` handler has been called, invoke `mxinput()` on the daemon socket.
5. Otherwise, return control to `mxfer()` which returns to `mroute()` which returns to the user.
6. If this is the message we asked for, return, otherwise goto 1.

One of the difficulties in integrating this AM layer is guarding against data received out of order. Fortunately, there are three bytes of unused space in the fragment header. We reserve one byte of this to hold the sequence number. Upon the transmission of every AM, the sequence number in the header is incremented. This number is checked upon reception against the last known sequence number from the sending host. If these numbers do not match, the packet is dropped and we rely on the AM library to handle the retransmission.

6.5 PVMAM Task Management

The AM library is initialized as soon as the PVM client sets up the socket to the PVM daemon. The communications interface used is changed by setting an environment variable. By default, PVMAM uses UDP sockets.

It was decided for simplicity of the implementation to remove the *direct-routing* code, as our layer will provide native TCP and UDP sockets in addition to accelerated transports. This involved cutting out the message routing code from `mroute()` and disallowing the `PvmRouteDirect` option. However, the `ttpcb`s were maintained because they were an integral part of the messaging functions. Whether receiving or sending a message, PVM always attempts a lookup of the destination `ttpcb` in the task list. Thus, the `ttpcb`s make a logical choice to store task-related information specific to the AM library. In fact, by recording the destination's endpoint pointer in the `ttpcb`, a TID based lookup in the AM library can be eliminated.

As mentioned in the previous section, our AM library requires exclusive and explicitly-ordered connections to work properly with all transports. This problem was temporarily solved with the introduction of an additional API call called `pvmam_init()`. This function takes as an argument the total number of tasks participating in the job and is called by every member process. The role of `pvmam_init()` is to fully connect all the participating processes so that they can send Active Messages to one other. The order problem was solved with the following algorithm.

1. All processes obtain their "stringified" AM task address.
2. All processes that have been spawned, send this address to the master process, the process that did the spawning.

3. The master then instructs all the tasks, a pair at a time to connect to each other. The task with the larger TID does the accepting, the other does the connecting.

While exceedingly primitive, this algorithm satisfies our ordering and exclusivity requirement. In addition, it performs quite well for connectionless transports as no arbitration is actually involved.

6.6 Problems

Integration of Active Messages into PVM has proven to be difficult without suffering some loss in PVM functionality. The internal message handling functions of PVM have been a major stumbling block to a fully functional prototype. This implementation has broken communication with the group server mean collective operations no longer function properly. This problems could likely be solved by moving the AM-specific code from `mxfer()` into the API functions `pvm_recv()` and `pvm_precv()`. In addition, this would have the desirable side effect of slightly lowering the latency and increasing the bandwidth.

7 PVMAM Performance

In this section, we report on the performance of two benchmarks, *MPBench* a message passing benchmark designed by the author and a parallel FFT designed by Vasilios Georgitsis[Geo] and modified by the author. *MPbench* measures both bandwidth and latency of the `pvm_psend()` and `pvm_precv()` calls. Care has been taken to eliminate cache effects and interference from other processes. Here the latencies are computed by taking the roundtrip time and dividing it by two. For further information, the reader is referred to [Muc97]. Here we report *MPBench* data for all platforms.

The FFT is a master-slave model that computes a two-dimensional, slice-wise FFT on an array of dimension 1152. A constant size was used so as to increasingly stress the latency of the communication layer. The benchmark works as follows. The master partitions the data and the slaves compute and communicate amongst themselves until the FFT is finished. At that time, the master gathers the data from the slaves an exits. The time reported by this benchmark is that ignoring initial I/O and setup and begins before the master broadcasts the data to the slaves. The timer is stopped and printed after the data from the last slave has been received. This code is fairly communication intensive, sending numerous small messages. It is reported only for the Solaris platform as there weren't enough Linux nodes available to generate a meaningful plot.

Our goal in reporting these numbers is not only to show an increase in performance, but also to demonstrate little or no loss in performance for the situations where no advanced networking hardware is available. For these cases, PVMAM succeeds simply by providing an machine and transport independent abstraction of the network, delivering true portability and future extensibility to existing PVM implementations.

7.1 Measurement Platforms

The following table summarizes the measurement platforms. All source code was compiled with *gcc* version 2.7.2.1 with the flags `-g -O -Wall` except for the AM library which was compiled with `-O3`. For our experiments with TCP and UDP, a depth of 8 demonstrated the best performance. For the Myrinet platform, depths greater than 12 showed no significant performance improvement.

CPU	*OS*	*Network*	*Link Speed*	*Protocols*
2 Pentium II 300Mhz	Linux 2.0.34 SMP	Myrinet	1Gb/s	BPI, UDP, TCP
UltraSparc I 167Mhz	Solaris 2.5.1	Fore ATM	155Mb/s	UDP, TCP
UltraSparc I 167Mhz	Solaris 2.5.1	Switched Ethernet	100Mb/s	UDP, TCP

7.2 Sun/Solaris

7.2.1 Bandwidth

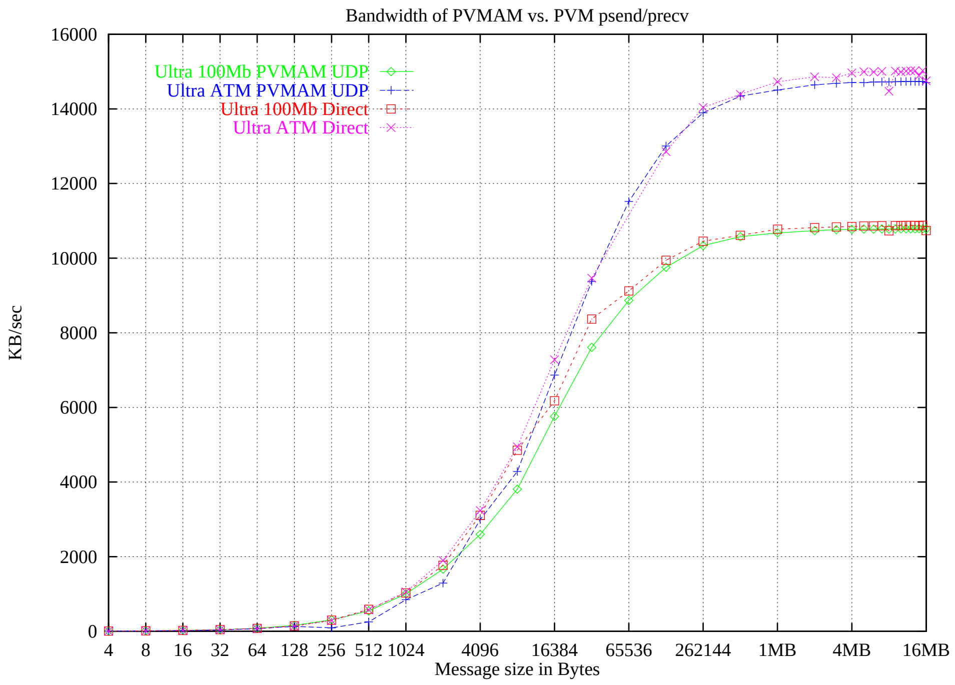

Figure 2: Solaris 2.5.1, Ultra 1, PVM 3.4 vs. PVMAM Bandwidth

Here we see almost identical performance of stock PVM and the PVMAM implementation. When running over the ATM switch, we find that the AM layer runs about about one percent slower than the *direct-routed* sockets that PVM uses. This difference is hardly significant in terms of application performance. The bandwidth curve for TCP has been intentionally left off this graph as UDP was slightly faster for this platform. TCP bandwidth can be found in figure 16 in the appendix. Here the performance of both PVM and PVMAM is bound by the speed of accessing the operating system and the underlying socket implementation. In this case, the cost is primarily dominated by the implicit copy of the user's data to kernel data structures when the socket is written to.

7.2.2 Latency

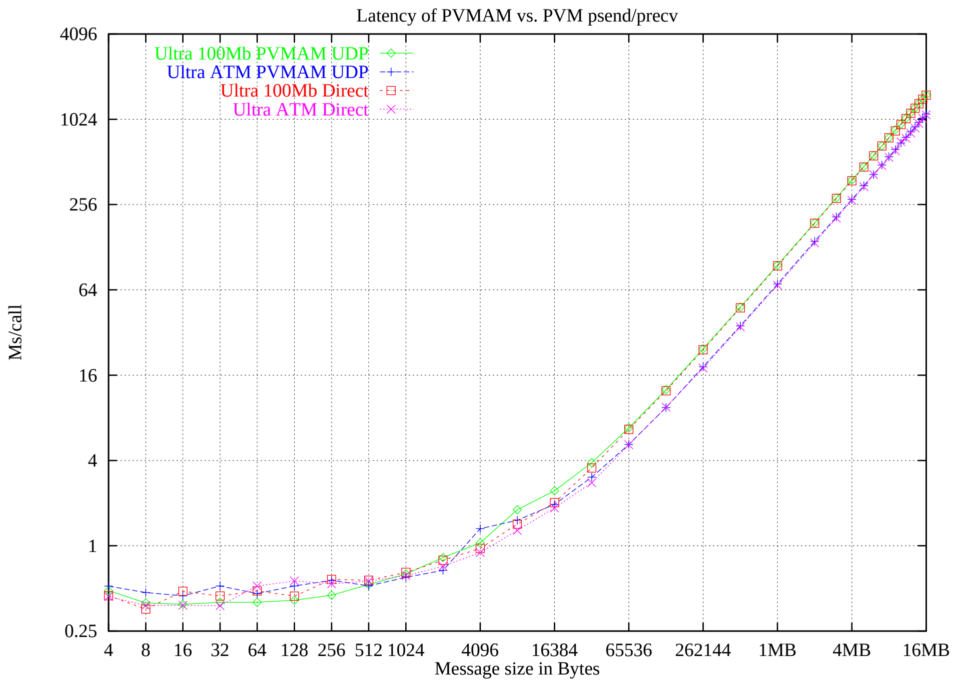

Figure 3: Solaris 2.5.1, Ultra 1, PVM 3.4 vs. PVMAM Latency

Again, we see almost identical performance of PVM and PVMAM for this platform due to the socket and kernel overhead.

7.3 Intel/Linux

7.3.1 Bandwidth

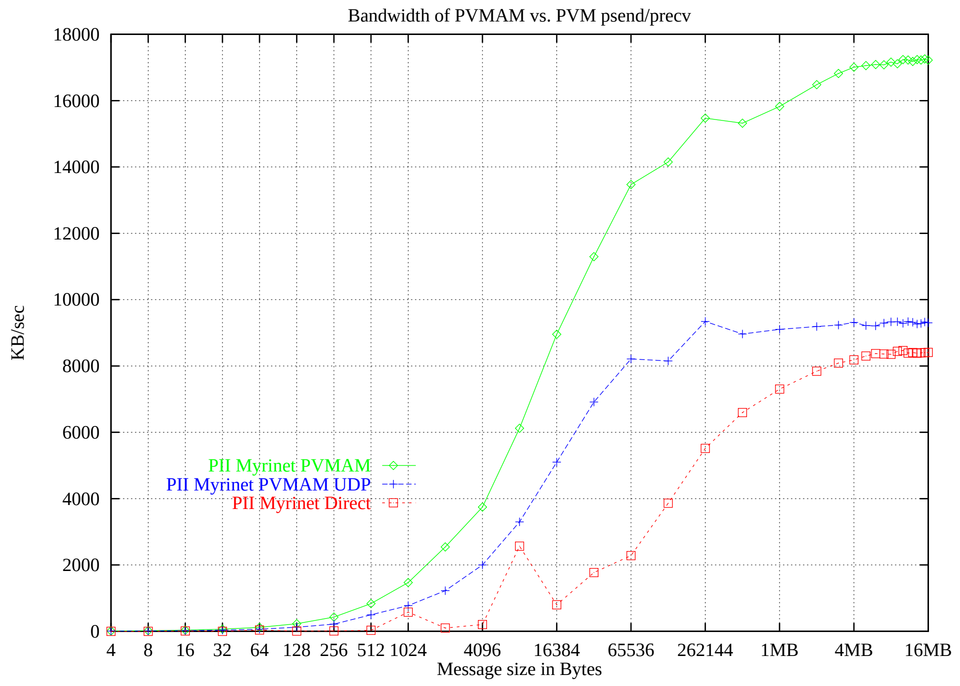

Figure 4: Linux 2.0.34 SMP, Pentium II, PVM 3.4 vs. PVMAM Bandwidth

In this graph, we greatly exceed the performance of standard PVM both with UDP sockets and with the Myrinet BPI. This performance difference from UDP is largely due to the lack of maturity of the TCP stack for the Myrinet hardware. PVM's performance loss after at 8K seems to be a repeatable anomaly in this stack. This also appears in the raw network performance graphs, figure 10 and figure 11. It is not clear whether this is a Myrinet specific problem or whether or not it is related to some SMP issues in the 2.0.34 kernel. As for PVMAM over the Myrinet BPI, the performance is more than double that of *direct-routed* PVM. With the Myrinet BPI, the AM layer can bypass the operating system and directly deal with the network interfaces DMA buffers. Comparing this curve with the peak speed of the API in figure 10, we find PVMAM to be nearly four megabytes short of the BPI's peak bandwidth. This can be attributed to the extra copy incurred at receive time by the AM library, which can hopefully be eliminated in the future.

7.3.2 Latency

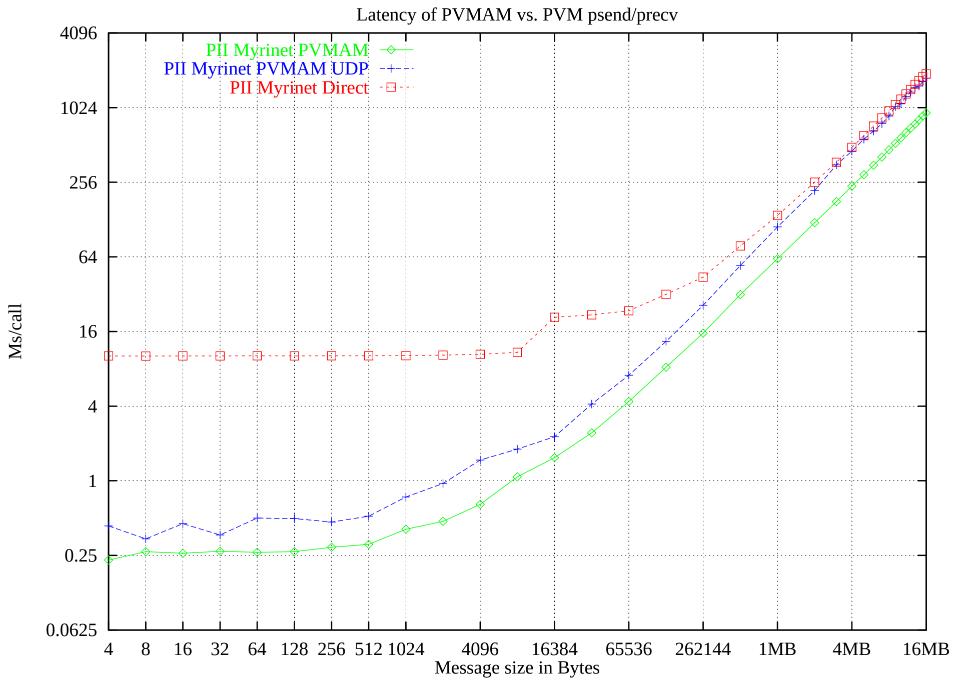

Figure 5: Linux 2.0.34 SMP, Pentium II(2), PVM 3.4 vs. PVMAM Latency

Again, we note the large performance difference from the PVMAM implementations versus stock PVM. Here, the latency for PVMAM over UDP asymptotically approaches that for TCP near message sizes of one megabyte. Below that, the latency is many orders of magnitude lower than PVM. For programs that exchange many small messages, we can expect noticeable performance increase even when just using UDP as our PVMAM transport. When using Myrinet and BIP, we find that the latency of transfers is consistently lower than that of stock PVM. The similarity in latency of UDP and the BIP transport at the low end of the message sizes can be attributed to the fact that the `gettimeofday()` system call is called upon every transmission. This cost could possibly be eliminated by using the Pentium II's hardware cycle counters instead.

8 Parallel FFT Performance

8.1 Solaris

8.1.1 Execution Time

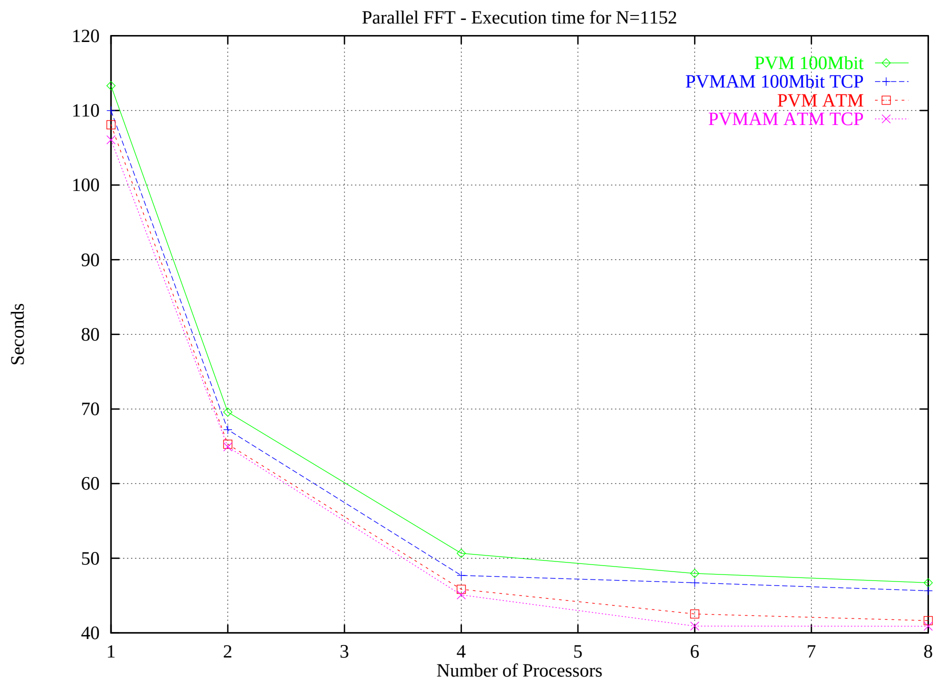

Figure 6: Solaris 2.5.1, Ultra 1, Parallel FFT Execution Time

In this graph, we examine the execution time of the parallel FFT while varying the number of processors. For this benchmark, we chose PVMAM over TCP as it provided better performance than did the Solaris implementation of UDP. Here we find a slight decrease in run time for those runs using PVMAM. For the most part, we find the difference to be slight between the PVM and PVMAM curve when run on each substrate. The reader will note that although the ATM network has a fifty percent faster link speed, the performance of the FFT over the two networks is nearly identical. This is an indication that the application is severely latency-bound. In figure 8, we find that the additional bandwidth is only utilized for transfers larger than 64K. For messages below that size, figure 9 shows that the latency is almost identical.

8.1.2 Scalability

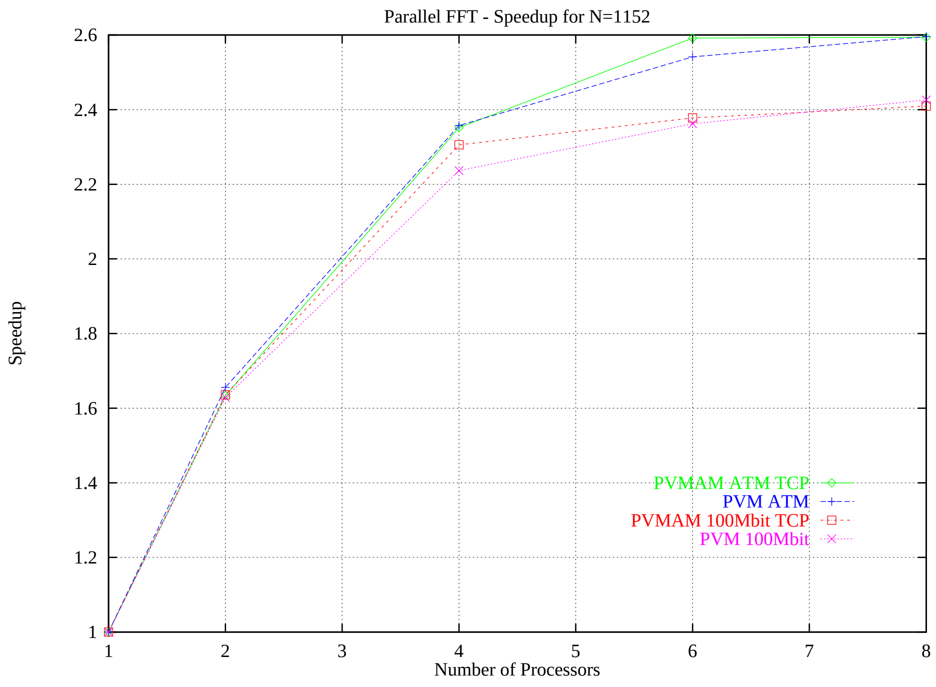

Figure 7: Solaris 2.5.1, Ultra 1, Parallel FFT Scalability

This graph very closely mirrors the previous. Here we find no significant increase in speedup for PVMAM when compared to stock PVM. The ATM substrate gives a little increase in scalability, but still no where near linear.

9 Conclusion

Upon the incarnation of this work, Active Messages were largely a fledgling research effort by a very small number of institutions. MPI was in it's infancy and PVM was the standard for parallel and distributed computing. Since that time, the community has changed significantly. Exhaustive research has been done on the performance of communication protocols and designing an efficient communication substrate suitable for cluster computing. MPI is now the standard distributed communications API and shared memory operations are rapidly gaining popularity. Furthermore, Active Messages have been widely embraced by the research and commercial community as an efficient mechanism to provide application specific, low-latency, high-bandwidth networking to distributed computing environments. Unfortunately, a majority of the AM implementations have been explicitly coded and tuned for a specific network-interface, processor and operating system and are highly unportable. This problem largely prevents the development of high performance communication in heterogeneous environments. This implementation of Active Messages directly addresses this issue by ensuring portability in addition to good performance and flexibility. This work also demonstrates that the integration of Active Messages and PVM can be successful for the many of the same reasons. With PVMAM, PVM is no longer bound by the performance of the underlying TCP socket implementation and the overhead of its message handling routines. No changes to the API are made except for the addition of `pvmam_init()` which could be eliminated in favor of a connection-on-demand scheme. PVM's internals remain largely unchanged except for the modifications mentioned earlier in this paper. Thus, through a small amount of changes, PVM has gained complete protocol independence providing the developer with increased portability and the user with increased performance.

For situations where an advanced network interface is available, PVMAM clearly outperforms stock PVM. The greatest benefit coming from the bypass of the operating system and its buffers. New transport layers for the AM API are not particularly difficult to write, so the library can grow as networking technology progresses. For the *normal* case where no advanced network interface is available, our AM implementation performs on par with stock PVM communicating over *direct-routed* TCP connections. On the Linux cluster, the UDP socket-based implementation of Active Messages actually performed much better than PVM, both in terms of latency and bandwidth. These characteristics can be attributed to the lower number of system calls, automatic flow control and the tuning of advanced socket options in the AM library. It is thought that with some minor changes to the transport layer and the message dispatch routines, the extra memory copy on the receive side could be eliminated. This modification would likely increase the performance well beyond *direct-routed* PVM over TCP for most platforms.

10 Future Directions

PVMAM could certainly benefit from some further performance tuning. There is still a large amount of redundancy in the code as well as a number of sanity checks for debugging purposes. After suitable testing, these checks could be eliminated in favor of reducing the latency. Another important issue is the additional memory copy incurred when the data is moved from the network interface into a staging area by `am_poll()`. This has a significant effect on bandwidth, especially when using the Myrinet interface. This copy could be eliminating by passing some additional information to `am_init()`. An Active Message in PVMAM contains both AM header information as well as PVM header information in addition to fragment data. The request handler for PVMAM needs the PVM header information so it will know how to handle the incoming message. Other applications using this AM layer will most likely function in a similar manner. The solution to removing this copy is provide the AM library with a hint as to how much "user"' header information will be carried by each Active Message. Then, when the network is serviced, the AM library only reads the entire header and let's the message handler itself receive the body of the message from the network. This modification should be fairly simple to implement and require little or no change to the transport layers. Other, more severe changes include:

- Use the VIA interface as a transport layer. This change would guarantee that the AM library will have high-bandwidth, low latency access to the next generation of network interfaces. For non-VIA transports, a simple, efficient VIA emulation layer could be written.
- Add in-place receive capability to PVM so that PVM message buffers can be preallocated instead of inside the message handlers.
- Remove the chained *in-place* packing modification and fragment the data at the time the Active Message is sent.
- Move the AM network service routines to `pvm_recv` and `pvm_precv()`. This would allow lower latency and remove the race conditions from communication with the group server.
- Remove `pvmam_init()` and make PVMAM jobs dynamic. Processes that wish to use Active Messages would connect on demand, just like *direct-routed* tasks do.
- Use more efficient timestamps where available.
- Eliminate header and timestamp processing for reliable transports.
- Eliminate the ordering and exclusivity requirements of the connection process.

A PVMAM Message Formats

A.1 Format of an Active Message in PVMAM

Bytes	*What*	*Description*
4	from_task	TID of the requesting task
2	total_length	Total length of this AM
1	instance	Instance number of this buffer
1	sequence	Binary sequence number of this buffer
1	pad	*Unused*
1	type	Type of AM (request or reply)
2	handler	Address identifier of handler
4	checksum	*Unused*

A.2 Format of a PVM Message Header

Bytes	*What*	*Description*
4	data signature	Encoding of the data
4	tag	PVM Message Tag
4	context	PVM Message Context
4	sequence	*Unused for non-MPPs*
4	wait id	*Unused for client*
4	checksum	*Unused*
4	reserved	*Unusued*
4	reserved	*Unusued*

A.3 Format of a PVM Fragment Header

Bytes	*What*	*Description*
4	destination	Destination TID
4	source	Source TID
4	length	Length of fragment
3	-	*Unused*
1	Control bits	Start and End-Of-Message Indicators

B The PVMAM API

`int am_global_init()` Initialize the address identifier tables and timestamp routines.

`void *am_init(transport_init_fn, info)` Initialize the transport with the provided information and return a pointer to the *interface*.

`void *am_shutdown(interface)` Make sure all messages are acknowledged, close all connections and free all buffers and associated memory.

`int am_global_shutdown()` Free the address identifier tables and invalidate further timestamps.

`void *am_id_to_address(id)` - Translates integer identifier `id` into an address.

`int am_bind_address_with_id(addr,id)` - Replaces the current address in `id` with `addr`.

`int am_address_to_id(addr)` - Inserts address `addr` into the table and returns an integer identifier.

`char *am_task_address(endpoint, tid)` - Returns the transport address of task tid.

`void *am_allocate_task(interface)` - Allocates the internal structures and buffers necessary to make a connection with a new task.

`int am_connect_task(interface, endpoint, name)` - Connect the remote task to our interface. `name` is that returned by `am_task_address()` on the remote machine.

`int am_accept_task(interface, endpoint, name)` - Accept the remote task at our interface. `name` is that returned by `am_task_address()` on the remote machine.

`int am_insert_task(interface, endpoint, tid)` - Insert `endpoint` with `tid` into our interface's task list.

`int am_remove_task(interface, tid)` - Remove the task with `tid` from our interface's task list.

`int am_deallocate_task(interface, endpoint)` - Free all memory associated with this `endpoint`.

`char *am_hostname(interface, tid)` - Return the actual name of `tid`.

`int am_request_ep(interface, endpoint, handler_id, data, length)` - Send a request to `endpoint` with data.

`int am_request(interface, tid, handler_id, data, length)` - Send a request to `tid` with data.

`int am_request_vdata_ep(interface, endpoint, handler_id, data, length)` - Send a request to `endpoint` with vectorized data.

`int am_request_vdata(interface, tid, handler_id, data, length)` - Send a request to `tid` with vectorized data.

`int am_reply(interface, header, handler_id, data, length)` - Send a reply to the requesting host with data.

`int am_request_vdata(interface, tid, handler_id, data, length)` - Send a reply to the requesting host with vectorized data.

`int am_poll(interface)` - Service the network.

C The AM Transport Interface Functions

`int (*network_init)` Allocates initializes the transport and allocates the our hosts endpoint structure. It also readies the localhost to accept connections. In addition, this routine reads the information structure that exists in the interface and modifies it to reflect the characteristics of the transport in use. *Required*

`int (*network_pkaddr)` Translates a machine independent representation of a endpoint address and packs it into the native representation as used by the transport. *Required*

`int (*network_upkaddr)` Translates a binary transport address into a machine independent character representation. *Required*

`int (*network_resolve)` Translates a character hostname and binary port number into a binary transport address. This function is used to check validity of an address with a name service. *Optional*

`int (*network_desolve)` Translates a binary transport address into a character hostname and binary port number. *Optional*

`int (*network_connect)` Connects to a remote endpoint specified by the binary transport address. *Optional*

`int (*network_accept)` Accepts the a remote endpoint specified by the binary transport address. *Optional*

`int (*network_send)` Sends a contiguous check of data to the endpoint specified. This function fully completes or returns 0. *Required*

`int (*network_gather)` Gathers a vector of data and sends it to the endpoint specified. This function fully completes or returns 0. *Required*

`int (*network_wait)` Waits until data is available at the interface. This function is currently only used internally by the TCP transport. *Optional*

`int (*network_recv)` Receives a complete Active Message or returns 0. *Required*

`void *(*network_get_buf)` Get message buffers to be used for transmission. This functions currently just calls `malloc`. It is intended to be used for DMA-like transport interfaces.*Required.*

`int (*network_free_buf)` Free message buffers to be used for transmission. *Required. See above.*

`char *(*network_error)` Return a string corresponding to the specified error code returned by one of the above function. *Optional*

`int (*network_shutdown)` Frees the localhost's endpoint returns all resources to the operating system. *Required*

D Raw Network Performance

The following results were collected using `netperf`[Inf94] for the TCP and UDP protocols and with `bpi_latency`[Myr] for the Myrinet cluster and plotted with `GNUplot`.

D.1 Raw TCP and UDP on Sun/Solaris

D.1.1 Bandwidth

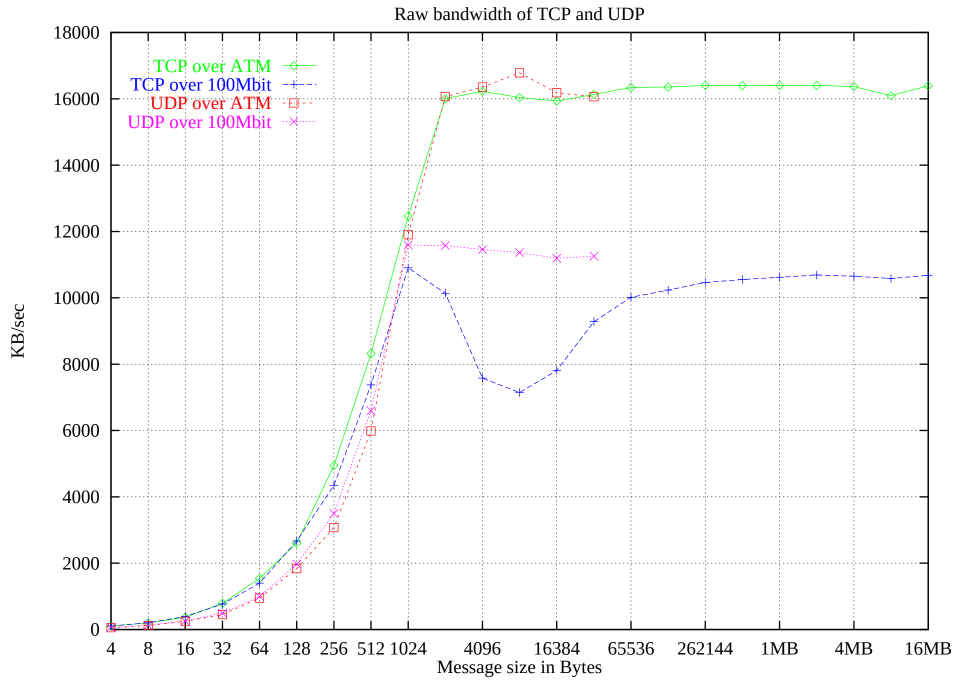

Figure 8: Solaris 2.5.1, Ultra 1, Raw Socket Bandwidth

D.1.2 Latency

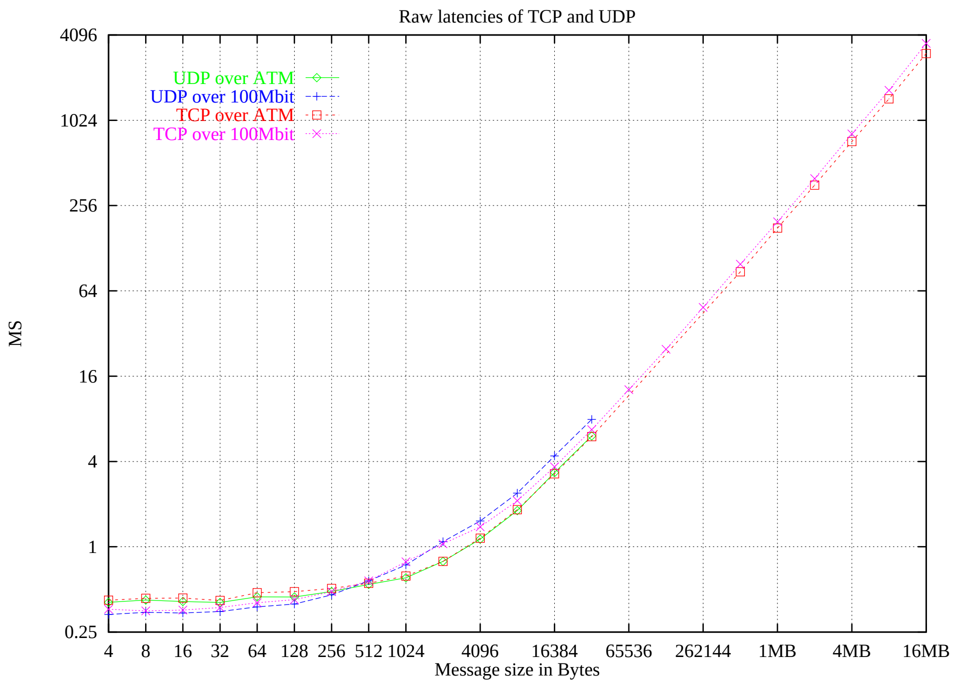

Figure 9: Solaris 2.5.1, Ultra 1, Raw Socket Latency

D.2 Raw TCP, UDP and the Myrinet BPI on Intel/Linux

D.2.1 Bandwidth

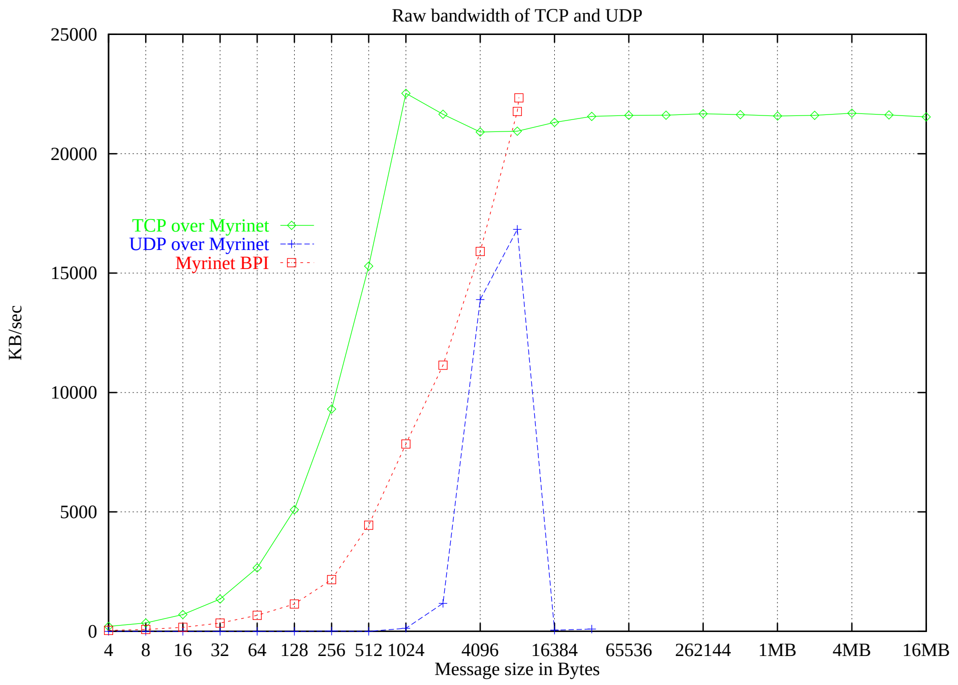

Figure 10: Linux 2.0.34 SMP, Pentium II(2), Raw Socket Bandwidth

D.2.2 Latency

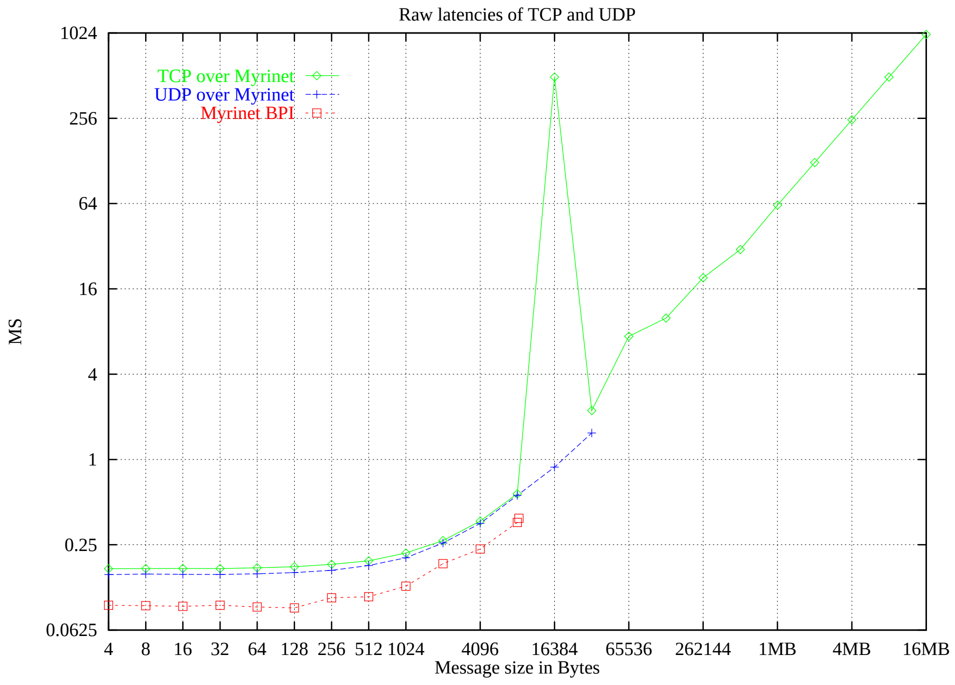

Figure 11: Linux 2.0.34 SMP, Pentium II(2), Raw Socket Latency

E PVM and PVMAM Performance

E.1 Direct Routed PVM(TCP) on Sun/Solaris

E.1.1 Bandwidth

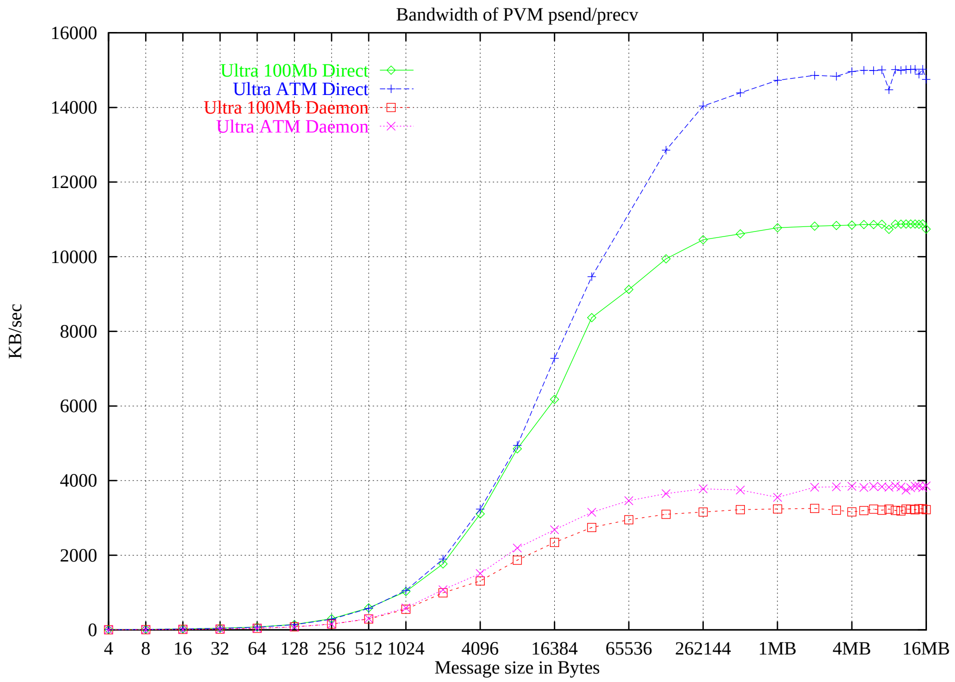

Figure 12: Solaris 2.5.1, Ultra 1, PVM 3.4 Bandwidth

E.1.2 Latency

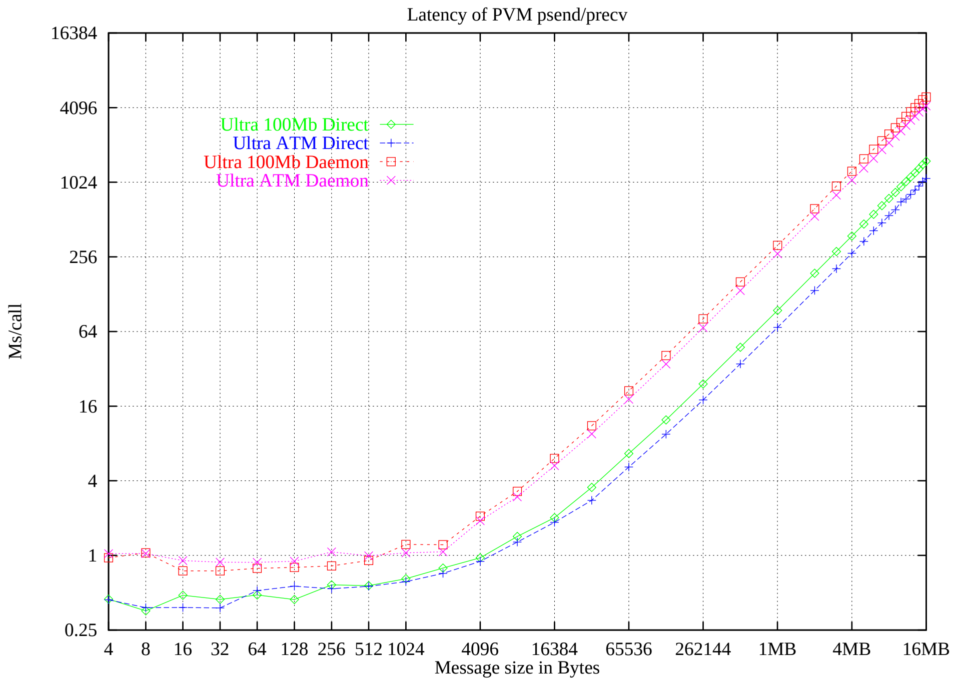

Figure 13: Solaris 2.5.1, Ultra 1, PVM 3.4 Latency

E.2 Direct Routed PVM(TCP) on Intel/Linux

E.2.1 Bandwidth

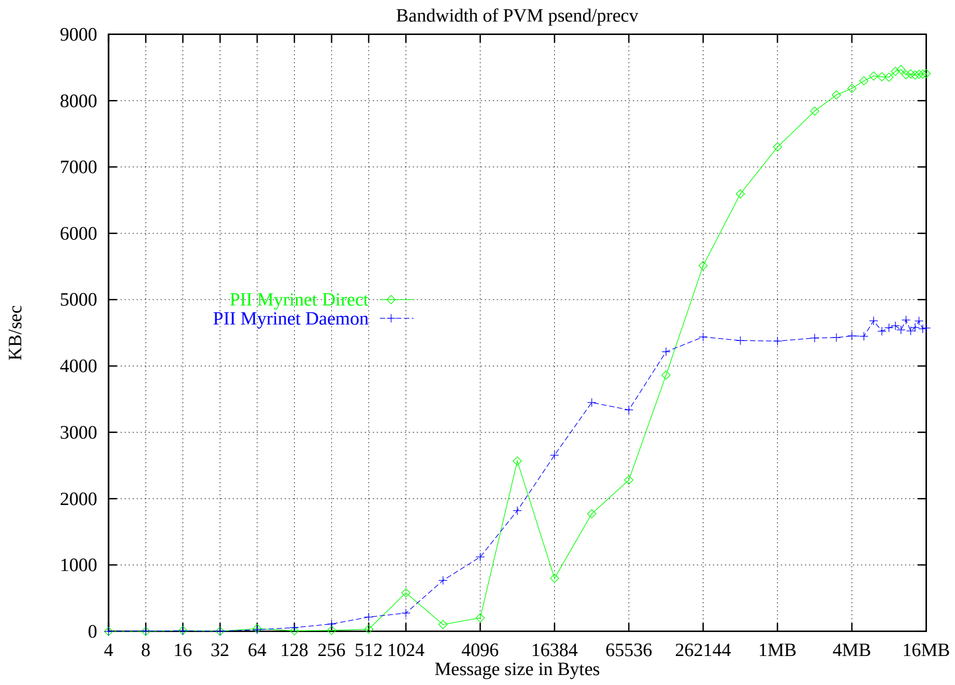

Figure 14: Linux 2.0.34 SMP, Pentium II(2), PVM 3.4 Bandwidth

E.2.2 Latency

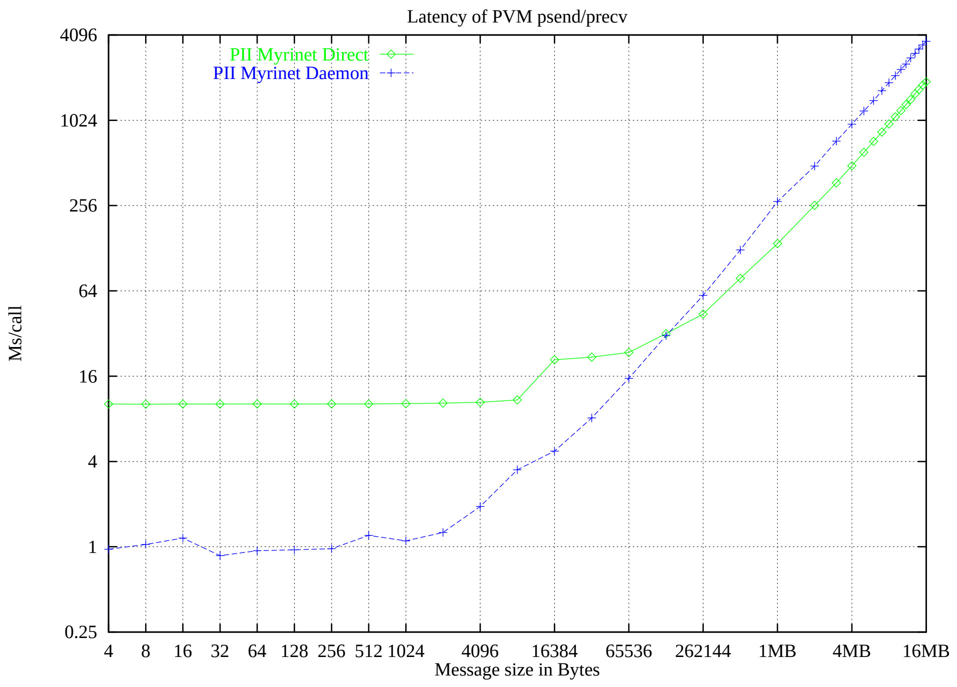

Figure 15: Linux 2.0.34 SMP, Pentium II(2), PVM 3.4 Latency

E.3 PVMAM on Sun/Solaris

E.3.1 Bandwidth

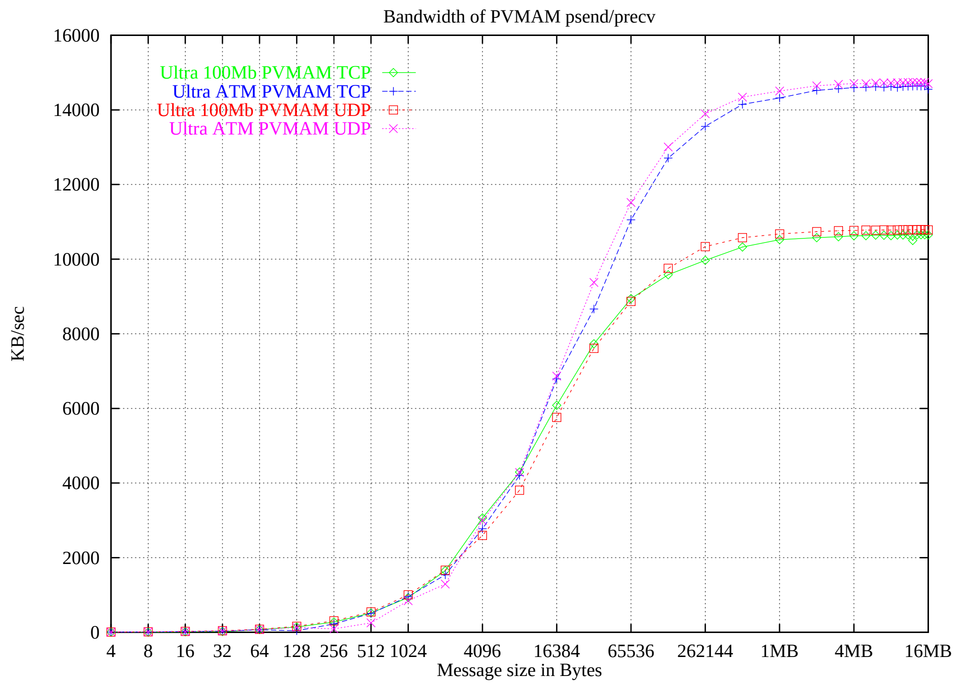

Figure 16: Solaris 2.5.1, Ultra 1, PVMAM Bandwidth

E.3.2 Latency

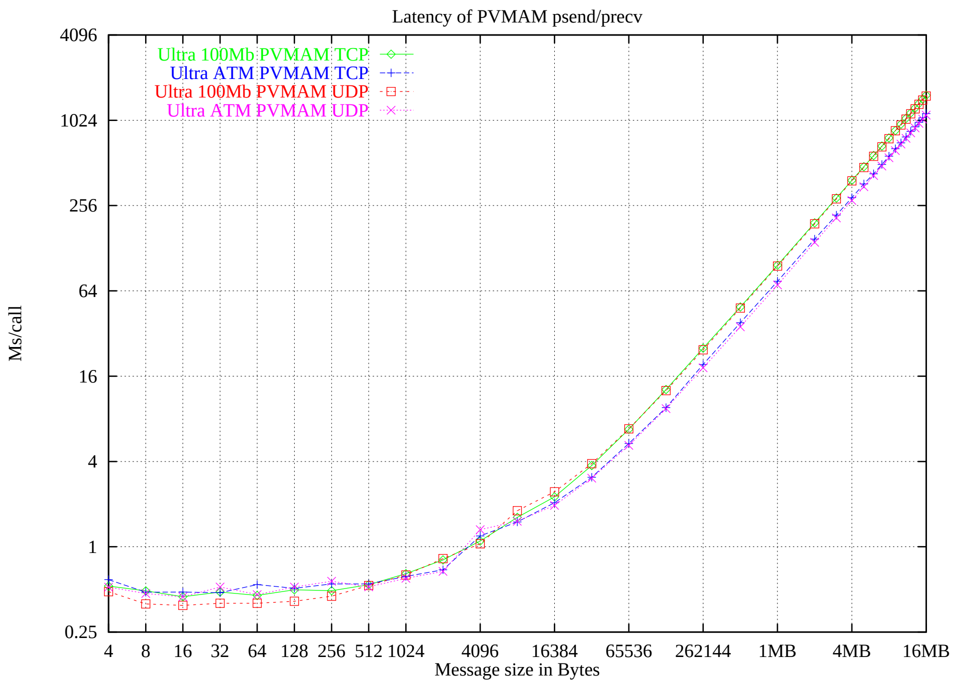

Figure 17: Solaris 2.5.1, Ultra 1, PVMAM Latency

E.4 PVMAM on Intel/Linux

E.4.1 Bandwidth

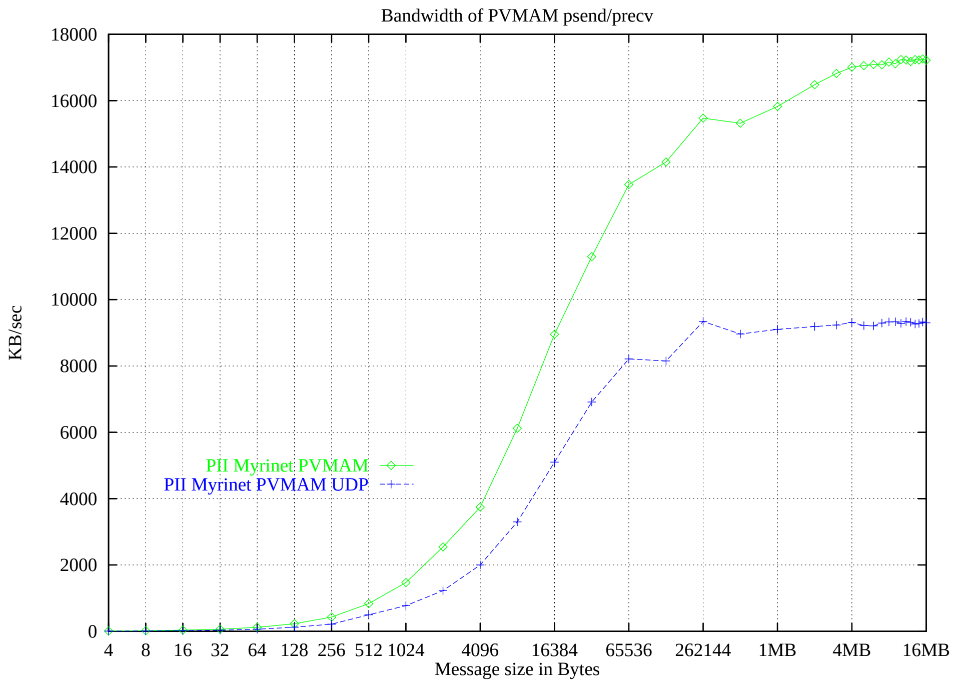

Figure 18: Linux 2.0.34 SMP, Pentium II(2), PVMAM Bandwidth

E.4.2 Latency

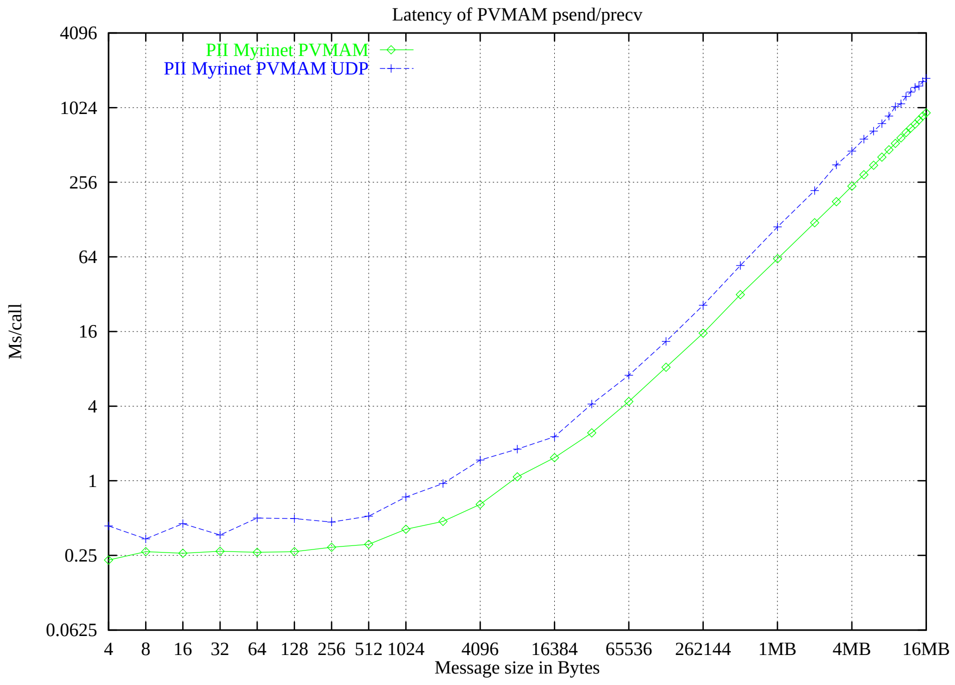

Figure 19: Linux 2.0.34 SMP, Pentium II(2), PVMAM Latency

References

[ABvE95] Werner Vogels Aninyda Basu, Vineet Buch and Thorsten von Eicken. U-NET: A User-Level Network Interface for Parallel and Distributed Computing. *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, Dec 1995.

[ABvE97] Matt Welsh Anindya Basu and Thorsten von Eicken. Incorporating memory management into user-level network interfaces. Technical report, Cornell University, 1997.

[BALL90] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. Lightweight remote procedure calls. *ACM Transaction on Computer Systems*, 8(1), February 1990.

[CFF+93] Chran-Ham Chang, Richard Flower, John Forecast, Heather Grey, William R. Hawe, K. K. Ramakrishnan, Ashok P. Nadharni, Uttam N. Shikarpur, and Kathleen M. Wilde. High Performance TCP/IP and UDP/IP Networking in DEC OSF/1 for Alpha AXP. *Digital Technical Journal*, 5(1), Winter 1993.

[Che98] D. Cheriton. Vmtp: Versatile message transaction protocol specification, January 1998.

[CKK+94] David E. Culler, Kim Keeton, Cedric Krumbein, Lok T. Liu, Alan Mainwaring, Richard P. Martin, Kristin Wright, and Chad Yoshikawa. Generic Active Message Interface Specification. Technical report, Department of Computer Science, University of California, Berkeley, November 1994.

[CKP+93] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: Towards a Realistic Model of Parallel Computation. *Proceedings of the Fourth ACM Symposium on Principles and Practice of Parallel Programming*, May 1993.

[CS94] Douglas E. Comer and David L. Stevens. *Internetworking with TCP/IP*, volume II. Prentice Hall, Englewood Cliffs, NJ 07632, 2nd edition, 1994.

[DAWK97] D. R. Engler D. A. Wallach and M. F. Kaashoek. Ashs: Application specific handlers for high-performance messaging. *IEEE/ACM Transactions on Networking*, 5, Number 4:460–474, August 1997.

[DBS93] T. Lasinski D. Bailey, J. Barton and H. Simon. The nas parallel benchmarks. Technical report, NASA Ames Research Center, July 1993.

[Dig93] Digital Equipment Corporation. *OSF/1 Guide to the Data Link Interface*, revision 1.2 edition, 1993.

[DM95] Jack Dongarra and Philip J. Mucci. Possibilities for active messaging in pvm. Technical Report ut-cs-95-277, University of Tennessee, Knoxville, February 1995.

[DREJO95] M. F. Kaashoek D. R. Engler and Jr. J. O'Toole. Exokernel: An operating system architecture for application-specific resource management. In *Proceedings of the 15th Symposium on Operating System Principles*, pages 251–266, December 1995.

[GBD$^+$94] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, Cambridge, Massachusetts, 1994.

[Geo] Vasilios Georgitsis. Parallel pvm fft benchmark. http://www.arc.unm.edu/~bennett/fft/srcs/vasilis/mscft2_c.html. University of New Mexico.

[Gus90] Riccardo Gusella. A measurement study of diskless workstation traffic on an ethernet. *IEEE/ACM Transactions on Communications*, 38, Number 9:1557–1568, September 1990.

[ICC97] Compaq Computer Corp. Intel Corporation and Microsoft Corporation. Virtual interface architecture specification. http://www.viarch.org, December 1997.

[Inf94] Information Networks Division, Hewlett Packard Company. *Netperf: A Network Performance Benchmark*, revision 1.9alpha edition, August 1994.

[KC94] Vijay Karamcheti and Andrew A. Chien. Software overhead in messaging layers: Where does the time go? In *Proceedings of ASPLOS*, October 1994.

[KP92] Jonathan Kay and Joseph Pasquale. A Performance Analysis of TCP/IP and UDP/IP Networking Software for the DECstation 5000. Technical report, University of California, San Diego, 1992.

[KP93] J. Kay and J. Pasquale. The importance of non-data-touching overheads in tcp/ip. In *Proceedings of the 1993 SIGCOMM*, pages 259–269, September 1993.

[KP96] Jonathan Kay and Joseph Pasquale. Profiling and reducing processing overheads in tcp/ip. *IEEE/ACM Transactions on Networking*, December 1996.

[LMY94] Lok T. Liu, Alan Mainwaring, and Chad Yoshikawa. White Paper on Building TCP/IP Active Messages. Technical report, Department of Computer Science, University of California, Berkeley, November 1994.

[Mar94] Richard P. Martin. Hpam: An active message layer for a network of hp workstations. In *Hot Interconnects II*, pages 40–58, August 1994.

[Muc97] Philip J. Mucci. Mpbench home page, 1997. URL: http://www.cs.utk.edu/~mucci/mpbench.

[MWvE96] Anindya Basu Matt Welsh and Thorsten von Eicken. Low latency communications over fast ethernet. *Proceedings of Euro-Par '96*, August 1996.

[MWvE97] Anindya Basu Matt Welsh and Thorsten von Eicken. Atm and fast ethernet network interfaces for user-level communication. *Proceedings of the Third International Symposium on High Performance Computer Architecure (HPCA)*, February 1997.

[Myr] Inc. Myricom. Myrinet users guide. http://www.myri.com:80/scs/documentation/mug.

[Myr94] Inc. Myricom. Atomic: A high speed local communication architecture. *Journal of High Speed Networks*, 3(1):1–30, 1994.

[Myr95] Inc. Myricom. Myrinet: A gigabit per second local area network. *IEEE Micro*, 15(1):29–36, February 1995.

[Nuc92] Neal Nuckolls. *How to Use DLPI.* Internet Engineering, June 1992.

[Ous90] John K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Proceedings of USENIX*, June 1990.

[pep96] *Proceedings of Euro-Par '96*, August 1996.

[pic92] *Proceedings of the 19th International Symposium of Computer Architecture*, May 1992.

[PP93] Craig Partridge and Stephen Pink. A faster udp. *IEEE/ACM Transactions on Networking*, 1, Number 4:429–439, August 1993.

[SOHL$^+$96] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference.* MIT Press, Cambridge, Massachusetts, 1996.

[Ste90] W. Richard Stevens. *Unix Network Programming.* Prentice Hall, Englewood Cliffs, NJ 07632, 1990.

[SWS94] A. Alund S. White and V. S. Sunderam. Performance of the nas parallel benchmarks on pvm based networks. Technical report, Department of Mathematics and Computer Science, Emory University, Atlanta, GA, May 1994.

[TL91] A. Thekkath and H. M. Levy. Limits to low-latency rpc. Technical Report 91-06-01, University of Washington, Seattle, Department of Computer Science, 1991.

[TvEB95] Anindya Basu T. von Eicken and Vineet Buch. Low latency communication over atm networks using active messages. *IEEE Micro*, pages 46–53, February 1995.

[UNI91] UNIX International, OSI Work Group. *Data Link Provider Interface Specification*, revision 2.0.0 edition, August 1991.

[vE94] Thorsten von Eicken. Building Parallel Programming Models using Active Messages. Technical report, Department of Computer Science, Cornell University, 1994.

[vEABB94] Thorsten von Eicken, Veena Avula, Anindya Basu, and Vineet Buch. Low-Latency Communication over ATM Networks using Active Messages. *Proceedings of Hot Interconnects II*, August 1994.

[vEC92] Thorsten von Eicken and David E. Culler. Building Communication Paradigms with the CM-5 Active Message layer (CMAM). Technical report, Department of Computer Science, University of California, Berkeley, July 1992.

[vEGS92] Thorsten von Eicken, David E. Cullerand Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In pica [pic92].

[VJB98] R. Braden V. Jacobson and D. Borman. Tcp extensions for high performance, January 1998.