To the Graduate Council:

I am submitting herewith a thesis written by Robert J. Manchek entitled "Design and Implementation of PVM Version 3." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science with a major in Computer Science.

Dr. Jack Dongarra, Major Professor

We have read this thesis
and recommend its acceptance:

Accepted for the Council:

Associate Vice Chancellor
and Dean of The Graduate School

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a Master's degree at The University of Tennessee, Knoxville, I agree that the Library shall make it available to borrowers under rules of the Library. Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of the source is made.

Permission for extensive quotation from or reproduction of this thesis may be granted by my major professor, or in his absence, by the Head of Interlibrary Services when, in the opinion of either, the proposed use of the material is for scholarly purposes. Any copying or use of the material in this thesis for financial gain shall not be allowed without my written permission.

Signature ______________________

Date ______________________

DESIGN AND IMPLEMENTATION OF PVM VERSION 3

A Thesis
Presented for the
Master of Science Degree
The University of Tennessee, Knoxville

Robert J. Manchek
May, 1994

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Jack Dongarra, for the guidance he provided, and a job that was a lot of fun. I thank Dr. David Straight and Dr. Jim Plank for serving on my committee and some fine discussions about caves and computer-related stuff. Thanks to my friends for dealing with me as I wrote this evil thing: Chris Jepeway, Sharon Lewis, Reed Wade, Jan Jones, Al Geist, James Garnett, and especially Carolyn Aebischer. Thanks to my mom for all the stuff she taught me way a long time ago, without which I couldn't have done any of this. Thanks to Robert Benway, Ace PVM Debugger, for asking too many good questions.

The PVM project is the result of the efforts of several people. Involved in various ways are: Jack Dongarra, Al Geist, Vaidy Sunderam, Weicheng Jiang, Adam Beguelin, Jim Kohl, Keith Moore, Honbo Zhou and Carolyn Aebischer, distributed seemingly at random at: the University of Tennessee, Oak Ridge National Laboratory, Emory University, Carnegie Mellon University and the Pittsburgh Supercomputing Center.

Funding for this work was provided by the Office of Scientific Computing, U.S. Department of Energy, under Contract DE-AC05-84OR21400, the National Science Foundation Science and Technology Center Cooperative Agreement No. CCR-8809615, and the Science Alliance, a state-supported program at the University of Tennessee.

In addition, a number of computer vendors have encouraged and provided valuable input to the development of PVM. We thank Cray Research Inc., IBM, Convex Computer, Silicon Graphics, Sequent Computer, and Sun Microsystems.

ABSTRACT

There is a growing trend toward *distributed computing* – writing programs that run across multiple networked computers – to speed up computation, solve larger problems or withstand machine failures. A programming model commonly used to write distributed applications is *message-passing*, in which a program is decomposed into distinct subprograms that communicate and synchronize with one another by explicitly sending and receiving blocks of data.

PVM (Parallel Virtual Machine) is a generic message-passing system composed of a programming library and manager processes. It ties together separate physical machines (possibly of different types), providing communication and control between the subprograms and detection of machine failures. The resulting *virtual machine* appears as a single, manageable resource. PVM is portable to a wide variety of machine architectures and operating systems, including workstations, supercomputers, PCs and multiprocessors.

In this paper I describe the design, implementation and testing of version 3.3 of PVM, and survey related works.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 The Distributed Computing Scene

Distributed computing – running programs across several or many computers on a network – is fast becoming a popular way to build state-of-the-art applications. It is an evolutionary step in technology that has simple origins, but is fundamentally different than using a single computer. Distributed computing is attractive for several reasons and can be implemented in a variety of ways.

Performance is one motivation behind distributed computing. By connecting several machines together, we can access more compute power, memory and I/O bandwidth. Groups of workstation-class machines can provide very high performance more cheaply than traditional supercomputers. Workstations can be connected to supercomputers to provide a desk-top interface, and we can interconnect supercomputers to solve Grand Challenge problems.Scalar (single instruction stream) processors run only as fast as the technology used to fabricate the components. This limit is expensive to exceed, perhaps by picking abnormally fast components from production lots or by switching to a more esoteric technology. Vector processors, multiple instruction units and tightly-coupled multiprocessors give an order of magnitude more speed. Memory access is the bottleneck, because N processors contend for a single shared memory, unless some kind of (usually expensive) switching network is used. Distributed-memory multiprocessors increase memory bandwidth by giving each processor a private local memory and access to other processors through some sort of network. Aggregate processor power and memory size can grow very large. The challenge is to partition programs so as to minimize communication, since it's expensive compared to memory access, and maximize processor utilization (no one waiting). This is sometimes easy, for example when the computation is completely separable ("embarrassingly parallel") and expensive enough to balance the cost of sending the data. [1] We can connect complete machines (workstations or mainframes) together over a local- or wide-area network, to form yet another type of distributed memory machine. The division between traditional multiprocessors and networks of computers is not a sharp line. The main differences are: 1. The ratio of communication to computation costs is usually higher for network-based machines, because the processors are larger and faster while the network is a general-purpose one – noisier and slower. 2. Parts of the network machine can go down (due to power outage, hardware failure, reboot), leaving the rest still running.

A second motivation is the availability of machines to connect. At a typical university or industrial site, there is a workstation for every person, and some larger *server* machines. Much of the processor power goes unused; it's attractive to think of doing something with it. Modern operating systems (e.g. UNIX [2]) [LMKQ89] have networking capability, allowing us to do simple remote operations such as execute processes (the `rsh` command), or access files (NFS [Sun89]), but to build real distributed applications using these raw tools entails a lot of work. Naming systems for nodes and processes must be created. Communication protocols must be embedded in the application code. Implementation details at this level vary from machine to machine, even between different versions of an operating system; the same problems are solved repeatedly. We'd like to have a single programming system that encompasses single-processor machines and multiprocessors as well as networks of machines. It should allow one to write portable code, and hide the details of process control and communication from the programmer.

[1] A classic example is computing fractals.

[2] UNIX is a registered trademark of UNIX System Laboratories Inc.

A third reason is fault tolerance. If a program is distributed across several physically separate computers, it should be able to tolerate a few of them going down. When computing on workstations, with each under the control of a different owner, fault tolerance seems necessary just to have a reasonable probability that a program will run to completion.

Finally, shared resources (both hardware and software) are inherently distributed. Not every site can afford a supercomputer, but if several sites go in on one together, somebody has to take it home. Large databases should be shared instead of replicated. [3] The same programming model we use to connect machines together on a local network can allow us to access remote ones.

1.2 Message Passing Programming

Message passing is a programming model, generated by distributed-memory machines but applicable to others, for example multiprogrammed single-processors. A program is divided into *components* or subprograms, which may run on different *nodes* of the machine. The same subprogram might run on every node (SPMD) or they may all run completely different programs (MPMD). The components communicate with one another by explicitly sending and receiving messages, which are arrays of data copied from one node to another.

PVM is a message-passing system that can be implemented in different environments. It ties together separate physical machines into a *virtual machine*, and provides the means for components to communicate with and control one another, and the virtual machine itself. Applications written in PVM can be run on different virtual machine configurations without modification.

[3] The Information Superhighway, doncha know.

1.3 History of PVM

The first version of PVM was written during the summer of 1989 at Oak Ridge National Laboratory by Vaidy Sunderam and Al Geist. It was used to build several applications, and showed promise as a computing platform. The main troubles were lack of portability, and an incompletely-defined programming interface. It needed tuning on a per-application basis to get it to do the right thing. However, it did attempt to implement some interesting concepts such as simulated shared memory and majority-rule data conversion [Sun90, GS92].

I wrote Version 2 in March 1991 [BDG+91] in a fit of angst caused by trying to use Version 1 in an application – a parallel graphical language derived from Schedule [DS86] that later evolved into HeNCE [BDG+93]. Very little was redesigned in the rewrite; it was mainly an effort to clean up the code. The programming interface was simplified and some unsupportable features were removed.

I wrote Version 3 during the Summer and Fall of 1992, after we redesigned the system to be more portable and to allow fault tolerance. Since it was released a year ago, over 3000 copies have been snagged. It is now actively in use at hundreds of sites, and is offered as a supported product by several vendors.

1.4 Typographical Conventions

Literal identifiers such as file names or variables, and examples of input or output appear in typewriter font, for example `PvmDataDefault`. Function names are followed by parenthesis, for example `pvm_exit()`. Identifiers are always spelled correctly (case-sensitive), even when they occur at the beginnings of sentences. Non-literal terms are emphasized, for example *pvmd*, as are meta-identifiers or variable parts of real identifiers, for example `/tmp/pvmd.`*uid*.

CHAPTER 2

RELATED WORK

In this section I discuss related research: Alternative programming systems, and work that is used by or can build on PVM.

2.1 Virtual Shared Memory

Shared memory is an alternative model that can be used to program either tightly-coupled or distributed-memory multiprocessors. There is much discussion about which is the lower layer: Messages can be implemented in terms of shared memory, or the other way around.

Active Messages [vECGS92] are a portable form of asynchronous communication implemented on top of low-level message operations. When a message arrives at a processor, a handler function determined from the message header is automatically invoked to extract data from the message and integrate it into the program running in the foreground. Active messages are used to support Split-C, a modification to C that combines aspects from shared memory and message passing paradigms. It supports a global address space using special assignment operators to optimize remote access by overlapping communication with computation.

The Shrimp [BLA+94] system uses a virtual memory-mapped network interface to give applications protected but direct access to network hardware, eliminating the overhead of system calls. This allows high-bandwidth, low-latency data exchange and can be used to support either message-passing or shared memory.

2.2 Parallel Languages for Distributed Systems

Message passing can be viewed either as an end programming medium or a sort of assembly language, which would be the output from compilation of a parallel language. Much of the work and fun of creating a parallel application lies in building the communication and synchronization methods, and less of this would be less required if these menial tasks could be automated. The problems are then to design a language expressive enough so a programmer can get his work done without specifying all the details, and to build a compiler that can translate the language into something that runs efficiently.

Some parallel languages are intended to run on shared-memory machines and assume that access to all memory is inexpensive; those are not considered here. Most modern language designs take data distribution into account and, (for example) schedule communication with regard to network costs. As the central issues are how to divide up data and computation, many parallel languages just let the programmer deal with one or both of those directly, using constructs such as hints or declarations.

DINO [RSW90] is a language for writing parallel programs for distributed memory machines, with emphasis on data-parallel computations that occur in numerical programs. Based on standard C, it has extensions to let the programmer specify a virtual parallel computer, map data structures across the PEs, and specify concurrent procedures. Programs written in DINO are compiled into normal C code by the DINO compiler, which handles mapping of processes and data to real machines on a network, for example an Intel iPSC/2 hypercube.

HPF [KLS$^+$94] is a set of extensions to Fortran 90, designed to be able to be compiled to run efficiently on computers with non-uniform memory costs. It includes new directives, syntax and library routines, and also specifies restrictions on existing features of Fortran that lead to inefficiency in distributed memory environments.

Phred [BN91] is a parallel language in which the programmer specifies a program as a graph, where nodes hold computation (written in a standard serial language) or data. The graphical language is very expressive, and includes constructs for various types of parallelism and synchronization. The graph can be analyzed to see if the resulting program runs in a deterministic fashion.

Jade [RSL92] is a language for managing coarse-grained parallel applications. Concurrency is implicit; programs are written as sequential code, accessing a shared address space. Annotations to the program tell the system how data is accessed. The runtime system matches the parallelism to the real hardware and does dynamic load balancing. Data conversion is done between machines with different number representations. Jade is implemented as an extension to C, and runs on shared memory machines (SGI multiprocessors), distributed memory machines (Intel iPSC/860), as well as networks of workstations (using PVM Version 2).

2.3 Distributed Object Systems

Other parallel systems give the programmer the abstraction of distributed shared memory (DSM). Data structures or *objects* are shared across a set of processors, without specifying exactly how the sharing is done.

Midway [BZS93] is a distributed shared memory system that runs on Mach 3.0. Midway programs are written in C, with explicit associations between data and *synchronization objects*, which control access to the data. It has tunable memory consistency, allowing the programmer to select a strongly consistent model for quick development or porting, or weaker consistency for more carefully-designed parallel programs.

Dome [Beg94] is a system under development at Carnegie Mellon University. The goal of the Dome project is to build sets of distributed objects which can be used to program heterogeneous networks of computers. Dome addresses the problems of load balancing, heterogeneity, ease of programming, and fault tolerance.

DoPVM [HS93] is an object oriented distributed environment implemented on top of PVM Version 3. It is a portable shared-object toolkit written in C++, composed of macros, libraries and object servers. It defines shared object classes, using operator overloading to move data transparently between processes, and also provides process scheduling tools.

2.4 Parallel Programming Systems

Linda [1] [CG89] is a parallel programming model that can be added to a variety of ordinary sequential languages, for example C, Fortran or PostScript. It is based on a *tuple space*, which is a kind of shared memory. Processes share data by storing it in tuples, where it can be read or modified by other processes. Process control comes from the tuple space: To do concurrent computation, a process creates a *live* tuple, causing another process to be started. That process carries out its computation using the data in the tuple, which then becomes an ordinary (data carrying) tuple. Linda and Linda-like systems have been used on shared and distributed memory machines and networks, and incorporated into operating systems [Lel90]. A Public-domain implementation, POSYBL, [Sch91] runs on networks of UNIX machines.

Still other systems extract parallelism automatically from sequential programs. Forge [2] [LW89] is a commercial product which analyzes normal sequential code to find parallelism. It translates a sequential program into separate ones which can be run on shared or distributed memory machines.

2.5 Message-Passing Environments

Hundreds of message-passing systems have been built in the past several years, ranging in complexity from simple to baroque. Following is a survey of the several that are similar to PVM.

[1] Linda is a product of Scientific Computing Associates.

[2] Forge is a product of Applied Parallel Research.

p4 [BL92] is a portable parallel toolkit created at Argonne National Laboratory, which includes functions for explicitly programming shared memory machines (using *monitors* [Hoa74]) or distributed memory machines (using message passing). On shared memory machines, either or both of the programming models may be employed. Programs can be written in either C or Fortran. The process model can be either SPMD or MPMD, though all components of a program are listed in a *procgroup* file and started by the system. p4 is descended from other systems created at Argonne, notably the Argonne Macros or Monmacs, [LO83] and has been the basis for several other systems as well.

Express [3] [FKB91] is a collection of tools for programming distributed memory multiprocessors, both true multiprocessors and UNIX network clusters. It includes a message-passing interface and tools for automatic parallelization of serial code, debugging and visualization.

ISIS [BM89] is a message passing programming system for workstation clusters. The system uses a concept called *virtual synchrony* and knowledge of the level of synchronization necessary between application components to efficiently maintain distributed data structures. Isis applications can be programmed in C, Lisp or Fortran. The system can replicate data and processes in order to tolerate machine failures, and can also configure a new virtual machine and restart a whole application after a large crash.

PICL [GHPW91] is a portable message-passing library designed to standardize message functions available on machines such as the Intel iPSC/2, iPSC/860 and Ncube/3200. It also provides higher-level functions such as global broadcast and combine, and barrier synchronization. Execution of a PICL program can be traced at different levels of detail, and the trace output can be viewed on an X-Window display using the ParaGraph [Hea90] tool. In addition to the multiprocessors listed above, PICL has been ported to other platforms: The Cosmic Environment, Linda,

[3] Express is a product of ParaSoft, Inc.

and PVM Version 2.

CHIMP [CHI91] is a message-passing library callable from C or Fortran programs. CHIMP programs can run on networks of UNIX machines or on a Meiko Computing Surface machine.

LAM [Bur89] is an instance of Trollius, a portable kernel/operating system that runs on multicomputers, single-board computers, and other special-purpose machines, allowing them to be hosted over a network by, e.g. a workstation. LAM is a message-passing environment that runs across a network of UNIX machines. Applications can be written in C or Fortran, and data conversion can be done between different types of nodes. File access is provided via a custom version of the *stdio* library. In addition to its own programming interface, the LAM package provides libraries for compatibility with PVM and MPI.

TCGMSG [Har91] is a message passing toolkit, inspired by the Argonne Macros. It is designed to be small and efficient, and simplify the task of writing scientific programs for distributed memory environments. It is portable to true message passing machines and UNIX workstations, and can use sockets or shared memory as a transport layer. It supports Fortran or C applications and asynchronous send and receive primitives for efficiency. An SPMD process model is used.

APPL [QCB93] is a library of message passing subroutines callable from C or Fortran programs. It was created at NASA/Lewis in order to have a portable programming system for their shared memory (System-5), distributed memory and networked machines (TCP). It is based on TCGMSG and p4, but is more compact, including only the essential point-to-point message passing operations and scatter/gather operations such as broadcast and global sum. It provides an SPMD process model: N copies of a program are automatically started and named $0 \ldots N-1$.

Parform [CS92] is a parallel programming system that runs on a network of UNIX workstations. It incorporates a dynamic load-balancing system and sensor on each machine to adjust the size of jobs assigned to machines as workloads vary.

Processes are started and managed automatically by the system according to a hostfile. Communication is done in terms of *handles*, which are UNIX file descriptors connected through sockets between processes, and data can be converted between different machine types.

The MPI Forum [For93] is an effort to collect the knowledge gained in the last ten years of building message-passing systems into a single standard programming interface. The main concerns are that the resulting system is efficient, able to run on a heterogeneous system, is portable to many different types of machines, allows C and Fortran language bindings, and has a thread-safe library. However, MPI only defines communication functions. Process control, file access, and other facilities are needed to build a complete program.

2.6 Distributed Operating Systems

PVM is similar to a distributed operating system (DOS). The PVM daemons (*pvmds*, §3.4) provide a virtual machine, that runs user processes. Issues such as process control, naming and communication must be addressed, as with a real operating system.

Real DOSs are composed of many kernels or *microkernels* running on physical machines, connected together by some kind of network. They can be symmetrical, or have a central controlling part. The microkernels provide a *software backplane* (or virtual machine) on top of which servers and applications can be built. A survey of distributed operating systems and related topics is presented in [Mul93].

There are two main differences between a parallel programming system like PVM and a true DOS. First, a DOS normally is ported to the hardware of the target machines, whereas PVM runs on top of an existing operating system, (e.g. UNIX) though it doesn't have to) Second and more important, a DOS provides a more complete environment than PVM, for example a filesystem and management of memory and peripheral devices like teletypes and card punches. Because PVM stays above the operating system, it is more easily ported between machines.

Mach [ABB+86] is a microkernel system intended to provide a machine independent platform for different operating system environments, such as UNIX. Servers to manage filesystems and virtual memory are implemented outside the kernel, which is a message router and security door. Clients and servers in Mach communicate via messages, and (in later versions) can reside on separate physical machines.

The V Distributed System [Che88b] runs on clusters of workstations connected by a high-speed network. It is composed of a microkernel, a copy of which runs on each machine, servers for filesystem and external network, and the usual programming libraries and commands. Processes communicate by sending messages to one another, and system services are requested by processes sending messages to special kernel ports, much like in PVM. The VMTP [Che88a] transport protocol, a by-product of the V project, was considered for use in PVM (§4.4).

Sprite [OCD+88] is a distributed operating system that runs on networks of workstations. It uses a monolithic kernel and is oriented around a high-performance shared filesystem, in which files and devices, including memory, on remote machines are transparently accessible. Processes can migrate from one processor to another. Sprite supports UNIX-style applications that in general are not parallel (though are distributed).

The Amoeba [MvRT+90] operating system supports distributed computing on a network, with a central pool of processors, each running a monolithic kernel, and remote terminals for users. Hardware of different types and with different data representations can be mixed. The user environment is similar to UNIX, but focuses on parallel programming. RPC is used both in the kernel and by application programs. Programs can be written in several normal languages (C, Fortran, Modula 2, etc.), as well as Orca, a parallel language that provides virtually shared data objects, that can be migrated from machine to machine.

The Open Software Foundation DCE (Distributed Computing Environment)

[Har92] system is one of the few to address the problem of connecting machines distributed over a wide-area network, instead of a local-area or campus net. However, the system is mainly a collection of several popular network file service, name service and communication protocols. The utility for process-process interaction is RPC (Remote Procedure Call) [Sun88], in which operations are inherently nested, disallowing parallel operation, though there are RPC options to perform multicast/single-response handshaking. Parallel programming systems have been built on RPC, but they must cheat it in order to leave called processes running and reconnect to them later. DCE is a computing environment only in the sense that the Internet is a distributed computer.

CHAPTER 3

DESIGN

3.1 Goals

The most important goals for version 3 are fault tolerance, scalability, heterogeneity and portability. PVM is able to withstand host and network failures. It doesn't automatically recover an application after a crash, but it provides polling and notification primitives to allow fault-tolerant applications to be built. The virtual machine is dynamically reconfigurable. This goes hand-in-hand with fault tolerance – an application may need to acquire more resources in order to continue running once a host has failed. Management is as decentralized and localized as possible, so virtual machines should be able to scale to hundreds of hosts and run thousands of tasks. PVM can connect computers of different types together in a single session. It runs with minimal modification on any flavor of UNIX, or operating system with comparable facilities (multitasking, networkable). The programming interface is simple but complete, and any user can install the package without special privileges.

3.2 Assumptions

To allow PVM to be highly portable, I avoid the use of operating system and language features that would be be hard to retrofit if unavailable, such as multithreaded processes and asynchronous I/O. These exist in many versions of UNIX, but they vary enough from product to product that different versions of PVM might need to be maintained. The generic port is kept as simple as possible, though PVM can always be optimized for any particular machine.

I assume the use of *sockets* for interprocess communication and that each host in a virtual machine group can connect directly to every other host via TCP [Pos81a] and UDP [Pos81b] protocols. The requirement of full IP connectivity could be removed by specifying message routes and using the pvmds to forward messages. Some multiprocessor machines don't make sockets available on the processing nodes, but do have them on the front-end (where the pvmd runs).

3.3 Architecture Classes

PVM assigns an *architecture name* to each kind of machine on which it runs, to distinguish between machines that run different executables, due to hardware or operating system differences. Many standard names are defined and others can be added.

Some machines with incompatible executables use the same binary data representation. PVM takes advantage of this to avoid data conversion. Architecture names are mapped to *data encoding* numbers, and the encoding numbers are used to determine when it is necessary to convert.

3.4 PVM Daemon

One pvmd runs on each host of a virtual machine. Pvmds owned by (running as) one user do not interact with those owned by others, in order to reduce security risk, and minimize the impact of one PVM user on another.

The pvmd serves as a message router and controller. It provides a point of contact, authentication, process control and fault detection. An idle pvmd occasionally checks that its peers are still running. Pvmds continue to run even if application programs crash, to aid in debugging.

The first pvmd (started by hand) is designated the *master*, while the others (started by the master) are called *slaves*. During normal operation, all are considered equal. But only the master can start new slaves and add them to the configuration. Reconfiguration requests originating on a slave host are forwarded to the master. Likewise, only the master can forcibly delete hosts from the machine.

3.5 Programming Library

The libpvm library allows a task to interface with the pvmd and other tasks. It contains functions for packing (composing) and unpacking messages, and functions to perform PVM *syscalls* by using the message functions to send service requests to the pvmd. It is made as small and simple as possible. Since it shares an address space with unknown, possibly buggy, code, it can be broken or subverted. Minimal sanity-checking of parameters is performed, leaving further authentication to the pvmd.

The top level of the libpvm library, including most of the programming interface functions, is written in a machine-independent style. The bottom level is kept separate and can be modified or replaced with a new machine-specific file when porting PVM to a new environment.

3.6 Task Identifiers

PVM uses a *task identifier* (TID) to address pvmds, tasks, and groups of tasks within a virtual machine. The TID contains four fields as shown in figure 1. Since the TID is used so heavily, it is made to fit into the largest integer data type (32 bits) available on a wide range of machines.

The fields S, G and H have global meaning – each pvmd of a virtual machine interprets them in the same way. The H field contains a host number relative to the virtual machine. As it starts up, each pvmd is configured with a unique host number and therefore owns part of the TID address space. The maximum number

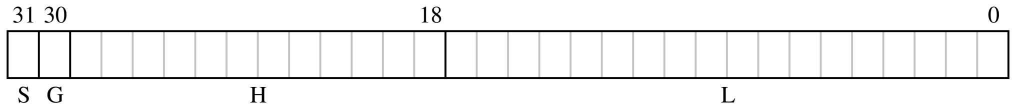

Figure 1: Generic Task ID

of hosts in a virtual machine is limited to $2^H - 1$ (4095). The mapping between host numbers and hosts is known to each pvmd, synchronized by a global *host table*. Host number zero is used, depending on context, to refer to the local pvmd or a *shadow pvmd*, called *pvmd'* (§4.2.7).

The S bit is used to address pvmds, with the H field set to the host number and the L field cleared. This bit is a historical leftover, and causes slightly schizoid naming; sometimes pvmds are addressed with the S bit cleared. It should someday be reclaimed to make the H or L space larger.

The G bit is set to form multicast addresses (GIDs), which refer to groups of tasks. Multicasting is described in §4.5.4.

Each pvmd is allowed to assign its own meaning to the L field (with the H field set to its own host number), except that all bits cleared is reserved to mean the pvmd itself. The L field is 18 bits wide, so up to $2^{18} - 1$ tasks can exist concurrently on each host.

In the generic UNIX port, L values are assigned by a counter, and the pvmd maintains a map between L values and UNIX process IDs. In multiprocessor ports the L field is subdivided as shown in figure 2. The P field specifies a machine partition, (physical group of processors), sometimes called a *process type* or *job*. The node number (N) determines a processor in a partition, and the W bit indicates whether a task runs on a compute node or host processor (service node).

The design of the TID enables the implementation to meet the design goals. Tasks can be assigned TIDs by their local pvmds without off-host communication. Messages can be routed from anywhere in a virtual machine to anywhere else, due to hierarchical naming. Portability is enhanced because the L field can be redefined.

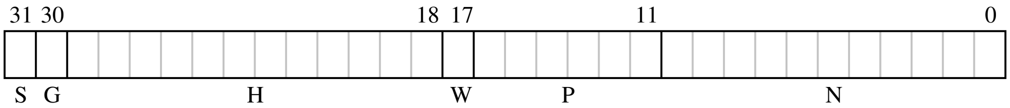

Figure 2: Multiprocessor Task ID

When sending a message, a task on a multiprocessor node can compare its own TID to the destination to determine whether to use native communication or send to the pvmd for routing. Finally, space is reserved for error codes. When a function can return a vector of TIDs mixed with error codes, it is useful if the error codes don't correspond to legal TIDs. The TID space is divided up as follows:

Use	S	G	H	L
Task identifier	0	0	$1..H_{max}$	$1..L_{max}$
Pvmd identifier	1	0	$1..H_{max}$	0
Local pvmd (from task)	1	0	0	0
Pvmd' from master pvmd	1	0	0	0
Multicast address	0	1	$1..H_{max}$	$0..L_{max}$
Error code	1	1	(small neg. number)	

Naturally, TIDs are intended to be opaque to the application and the programmer should not attempt to predict their values or modify them without using functions supplied in the programming library. More symbolic naming can be obtained by using a name server library layered on top of the raw PVM calls, if the convenience is deemed worth the cost of name lookup.

3.7 Message Model

PVM daemons and tasks can compose and send messages of arbitrary lengths containing typed data. The data can be converted using XDR [Sun87] when passing between hosts with incompatible data formats. Messages are tagged at send time with a user-defined integer code, and can be selected for receipt by source address or tag.

The sender of a message does not wait for an acknowledgement from the receiver, but continues as soon as the message has been handed to the network and the message buffer can be safely deleted or reused. Messages are buffered at the receiving end until received. PVM reliably delivers messages, provided the destination exists. Message order from each sender to each receiver in the system is preserved; if one entity sends several messages to another, they will be received in the same order.

Both blocking and non-blocking receive primitives are provided, so a task can wait for a message without (necessarily) consuming processor time by polling for it. Or, it can poll for a message without hanging. A receive with timeout is also provided, which returns after a specified time if no message has arrived.

No acknowledgements are used between sender and receiver. Messages are reliably delivered and buffered by the system. If we ignore fault recovery, then either an application will run to completion or, if some component goes down, it won't. In order to provide fault recovery, a task (T_A) must be prepared for another task (T_B, from which it wants a message) to crash, and be able to take corrective action. For example, it might re-schedule its request to a different server, or even start a new server. From the viewpoint of T_A, it doesn't matter specifically when T_B crashes relative to messages sent from T_A. While waiting for T_B, T_A will receive either a message from T_B or notification that T_B has crashed. For the purposes of flow control, a fully blocking send can easily be built using the semi-synchronous send primitive.

3.8 Asynchronous Notification

PVM provides *notification* messages as a means to implement fault recovery in an application. A task can request that the system send a message on one of the following three events:

Type	Meaning
`PvmTaskExit`	Task exits or crashes
`PvmHostDelete`	Host is deleted or crashes
`PvmHostAdd`	New hosts are added to the VM

Notify requests are stored in the pvmds, attached to objects they monitor. Requests for remote events (occurring on a different host than the requester) are kept on both hosts. The remote pvmd sends the message if the event occurs, while the local one sends the message if the remote host goes down. The assumption is that a local pvmd can be trusted; if it goes down, tasks running under it won't be able to do anything, so they don't need to be notified.

CHAPTER 4

IMPLEMENTATION

This section describes the implementation of the generic (UNIX) port, which was the first written. Details of ports to multiprocessors and other operating systems are described elsewhere.

4.1 Messages

4.1.1 Fragments and Databufs

The pvmd and libpvm manage message buffers, which potentially hold large amounts of dynamic data. Buffers need to be shared efficiently, for example to attach a multicast message to several send queues (§4.5.4). To avoid copying, all pointers are to a single instance of the data (a *databuf*), which is refcounted by allocating a few extra bytes for an integer at the head of the data. A pointer to the data itself is passed around, and routines subtract from it to access the refcount or free the block. When the refcount of a databuf decrements to zero, it is freed.

PVM messages are composed without declaring a maximum length ahead of time. The pack functions allocate memory in steps, using databufs to store the data, and frag descriptors to chain the databufs together.

A frag descriptor `struct frag` holds a pointer (`fr_dat`) to a block of data and its length (`fr_len`). It also keeps a pointer (`fr_buf`) to the databuf and its total length (`fr_max`); these reserve space to prepend or append data. Frags can also reference static (non-databuf) data. A frag has link pointers so it can be chained into a list. Each frag keeps a count of references to it, when the refcount decrements to zero, the frag is freed and the underlying databuf refcount decremented. In the case where a frag descriptor is the head of a list, its refcount applies to the entire list. When it reaches zero, every frag in the list is freed. Figure 3 shows a list of fragments storing a message.

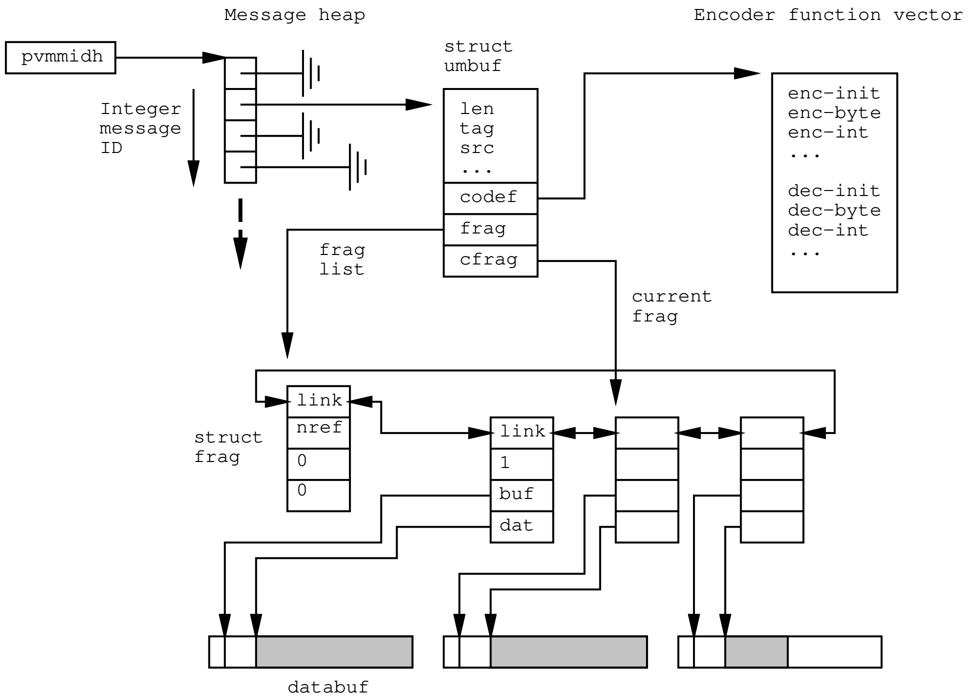

Figure 3: Message Storage in Libpvm

4.1.2 Messages in Libpvm

Libpvm provides functions to pack all primitive data types into a message, in one of five encoding formats. Each message buffer has an encoder and decoder set associated with it. When creating a new message, the encoder set is determined by the format parameter to `pvm_mkbuf()`. When receiving a message, the decoders are determined by the encoding field of the message header. The two most commonly used ones pack data in *raw* (host native) and *default* (XDR) formats. *Inplace* encoders pack descriptors of the data (the frags point to static data), so the message is sent without copying the data to a buffer. There are no inplace decoders. *Foo* encoders use a machine-independent format simpler than XDR, and are used when communicating with the pvmd. *Alien* decoders are installed when a received message can't be unpacked because its encoding doesn't match the data format of the host. A message in an alien data format can be held or forwarded, but any attempt to read data from it results in an error.

Figure 3 shows libpvm message management. To allow the PVM programmer to handle message buffers, they are labeled with integer message IDs (MIDs), which are simply indices into the message heap. When a message buffer is freed, its MID is recycled. The heap starts out small and is extended if it becomes full. Generally, only a few messages exist at any time, unless the an application explicitly stores them.

A vector of functions for encoding/decoding primitive types (`struct encvec`) is initialized when a message buffer is created. To pack a long integer, the generic pack function `pvm_pklong()` calls `(message_heap[mid].ub_codef->enc_long)()` of the buffer. Encoder vectors were used for speed (as opposed to having a case switch in each pack function). One drawback is that every encoder for every format is touched (by naming it in the code), so the linker must include them all in every executable, even when some are not used.

4.1.3 Messages in the Pvmd

By comparison to libpvm, message packing in the pvmd is very simple. Messages are handled using `struct mesg` (shown in figure 4). There are encoders for signed and unsigned integers and strings, which use in the libpvm *foo* format. Integers occupy four bytes each with bytes in network order (bits 31..24 followed by bits 23..16, ...). Byte strings are packed as an integer length (including the terminating null for ASCII strings), followed by the data and zero to three null bytes to round the total length to a multiple of four.

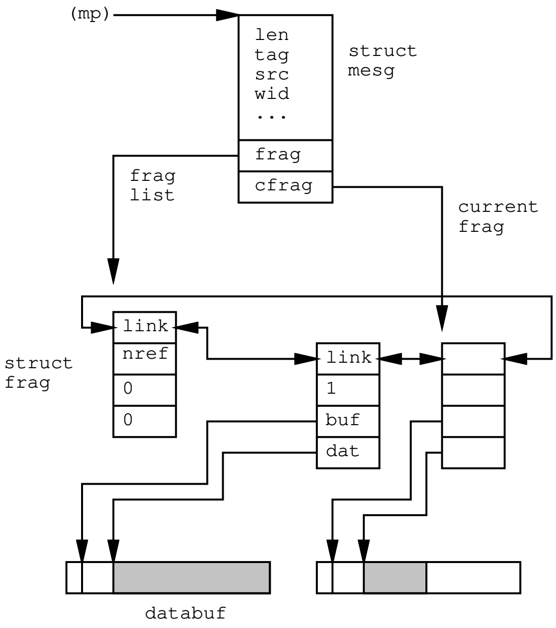

Figure 4: Message Storage in Pvmd

4.1.4 Pvmd Entry Points

Messages for the pvmd are reassembled from packets in **`loclinpkt()`** if from a local task, or in **`netinpkt()`** if from another pvmd or foreign task. Reassembled messages are passed to one of three *entry points*:

Function	Messages From
`loclentry()`	Local tasks
`netentry()`	Remote pvmds
`schedentry()`	Special tasks (Resource manager, Hoster, Tasker)

If the message tag and contents are valid, a new thread of action is started to handle the request. Invalid messages are discarded.

4.1.5 Control Messages

Control messages are sent to a task like regular messages, but have tags in a reserved space (between `TC_FIRST` and `TC_LAST`). Normally, when a task downloads a message, it queues it for receipt by the program. Control messages are instead passed to `pvmmctl()`, and then discarded. Like the entry points in the pvmd, `pvmmctl()` is an entry point in the task, causing it to take some asynchronous action. The main difference is that control messages can't be used to get the task's attention, since it must be in `mxfer()`, sending or receiving, in order to get them.

The following control message tags are defined. The first three are used by the direct routing mechanism (§4.5.3). `TC_OUTPUT` is used to implement `pvm_catchout()` (§4.6.2). User-definable control messages may be added in the future as a way of implementing PVM signal handlers.

Tag	Meaning
`TC_CONREQ`	Connection request
`TC_CONACK`	Connection ack
`TC_TASKEXIT`	Task exited/doesn't exist
`TC_NOOP`	Do nothing
`TC_OUTPUT`	Claim child stdout data
`TC_SETTMASK`	Change task trace mask

4.2 PVM Daemon

4.2.1 Startup

At startup, a pvmd configures itself as a master or slave, depending on its command line arguments. It creates and binds sockets to talk to tasks and other pvmds, opens an error log file `/tmp/pvml.`*uid*. A master pvmd reads the host file if supplied, otherwise it uses default parameters. A slave pvmd gets its parameters from the master pvmd via the command line and configuration messages.

After configuration, the pvmd enters a loop in function `work()`. At the core of the work loop is a call to `select()` that probes all sources of input for the pvmd (local tasks and the network). Packets are received and routed to send queues. Messages to the pvmd are reassembled and passed to the entry points.

4.2.2 Shutdown

A pvmd shuts down when it is deleted from the virtual machine, killed (signaled), loses contact with the master pvmd, or breaks (e.g. with a bus error). When a pvmd shuts down, it takes two final actions. First, it kills any tasks running under it, with signal `SIGTERM`. Second, it sends a final shutdown message (§4.4.2) to every other pvmd in its host table. The other pvmds would eventually discover the missing one by timing out trying to communicate with it, but the shutdown message speeds up the process.

4.2.3 Host Table and Machine Configuration

A host table describes the configuration of a virtual machine. It lists the name, address and communication state for each host. Figure 5 shows how a host table is built from `struct htab` and `struct hostd` structures.

Host tables are issued by the master pvmd, and kept synchronized across the virtual machine. The delete operation is simple: On receiving a `DM_HTDEL` message from the master, a pvmd calls `hostfailentry()` for each host listed in the message, as though the deleted pvmds crashed. Each pvmd can autonomously delete hosts from its own table on finding them unreachable (by timing out during communication). The add operation is done with a three-phase commit, in order to guarantee global availability of new hosts synchronously with completion of the add-host request. This is described in §4.2.8.

Each host descriptor has a refcount so it can be shared by multiple host tables.

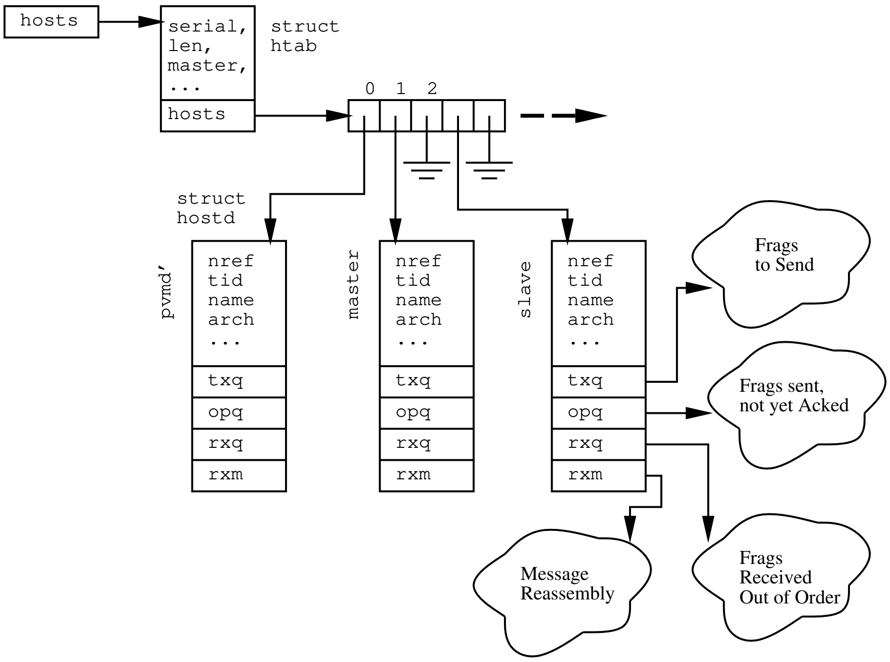

Figure 5: Host Table

As the configuration of the machine changes, the host descriptors (except those added and deleted of course) propagate from one host table to the next. This is necessary because they hold various state information.

Host tables also serve other uses: They allow the pvmd to manipulate host sets, for example when picking candidate hosts on which to spawn a task. Also, the advisory host file supplied to the master pvmd is parsed and stored in a host table, `filehosts`. If some hosts in the file are to be started automatically, the master sends a `DM_ADD` message to itself. The slave hosts are started just as though they had been added dynamically (§4.2.8).

4.2.4 Task Management

Each pvmd maintains a list of all tasks under its management (figure 6). Every task, regardless of state, is a member of a threaded list, sorted by task ID. Most tasks are also in a second list, sorted by process ID. The head of both lists is `locltasks`.

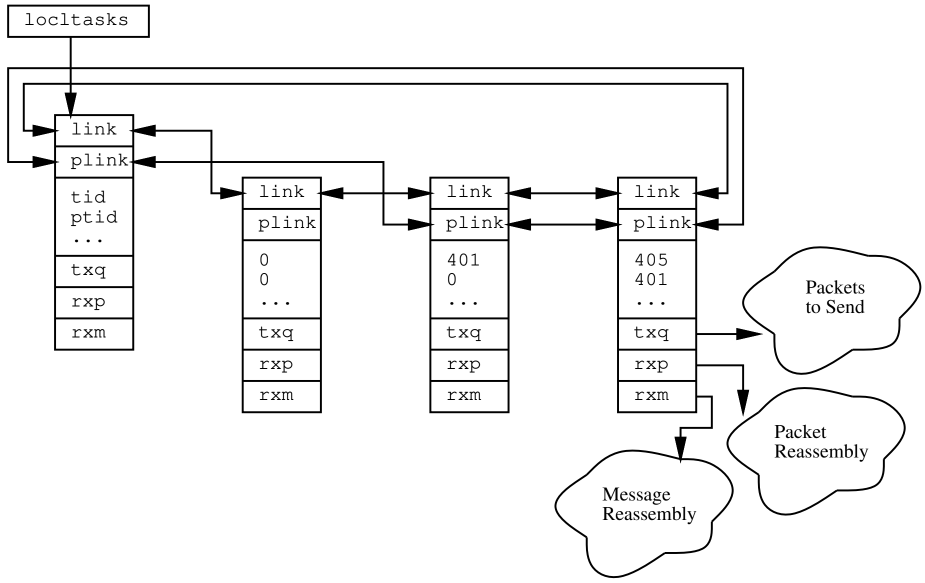

Figure 6: Task Table

4.2.5 Wait Contexts

The pvmd uses a wait context (waitc) to hold state when a thread of operation must be interrupted. The pvmd is not truly multi-threaded, but performs operations concurrently. For example, when a pvmd gets a syscall from a task and must interact with another pvmd, it doesn't block while waiting for the other pvmd to respond. It saves state in a waitc and returns immediately to the `work()` loop. When the reply arrives, the pvmd uses the information stashed in the waitc to complete the syscall and reply to the task. Waitcs are serial numbered, and the number is sent in the message header along with the request and returned with the reply.

For many operations, the TIDs and *kind* of wait are the only information saved. The `struct waitc` includes a few extra fields to handle most of the remaining cases, and a pointer, `wa_spec`, to a block of extra data for special cases – the spawn and host startup operations, which need to save `struct waitc_spawn` and `struct waitc_add`.

Sometimes more than one phase of waiting is necessary – in series, parallel, or nested. In the parallel case, a separate waitc is created for each foreign host. The waitcs are *peered* (linked in a list) together to indicate they pertain to the same operation. If a waitc has no peers, its peer links point to itself. Usually, peered waitcs share data, for example `wa_spec`. All existing parallel operations are conjunctions; a peer group is finished when every waitc in the group is finished. As replies arrive, finished waitcs are collapsed out of the list and deleted. When the finished waitc is the only one left, the operation is complete. Figure 7 shows single and peered waitcs stored in `waitlist` (the list of all active waitcs).

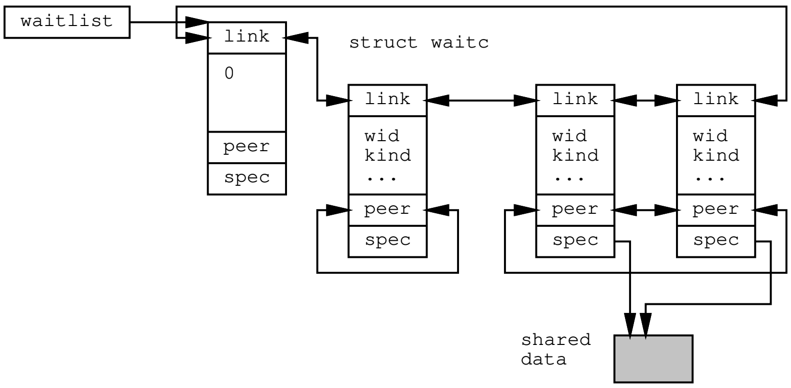

Figure 7: Host Table

When a host fails or a task exits, the pvmd searches `waitlist` for any blocked on this TID and terminates those operations. Waitcs from the dead host or task blocked on something else are not deleted, instead their `wa_tid` fields are zeroed. This is done to prevent the wait IDs from being recycled while replies are still pending. Once the defunct waitcs are satisfied, they are silently discarded.

4.2.6 Fault Detection and Recovery

Fault detection originates in the pvmd-pvmd protocol (§4.4.2). When the pvmd times out while communicating with another, it calls `hostfailentry()`, which scans `waitlist` and terminates any operations waiting on the down host.

A pvmd can recover from the loss of any foreign pvmd except the master. If a slave loses the master, the slave shuts itself down. This algorithm ensures that the virtual machine doesn't become partitioned and run as two partial machines. This decreases fault tolerance of the virtual machine because the master must never crash. There is currently no way for the master to hand off its status to another pvmd, so it always remains part of the configuration. This is still better than version 2, in which the failure of any pvmd would shut down the entire system.

4.2.7 Pvmd'

The shadow pvmd (pvmd') runs on the master host and is used by the master to start new slave pvmds. Any of several steps in the startup process (for example starting a shell on the remote machine) can block for seconds or minutes (or hang), and the master pvmd must be able to respond to other messages during this time. It's messy to save all the state involved, so a completely separate process is used.

The pvmd' has host number 0 and communicates with the master through the normal pvmd-pvmd interface, though it never talks to tasks or other pvmds. The normal host failure detection mechanism is used to recover in the event the pvmd' fails. The startup operation has a wait context in the master pvmd. If the pvmd' breaks, the master catches a `SIGCHLD` from it and calls `hostfailentry()`, which cleans up.

4.2.8 Starting Slave Pvmds

Getting a slave pvmd started is a messy task with no good solution. The goal is to get a process running on the new host, with enough identity to let it be fully configured and added as a peer.

Ideally, the mechanism used should be widely available, secure and fast, while leaving the system easy to install. We'd like to avoid having to type passwords all the time, but don't want to put them in a file from where they can be stolen. No one system meets all of these criteria. Using `inetd` or connecting to an already-running pvmd or pvmd server at a reserved port would allow fast, reliable startup, but would require that a system administrator install PVM on each host. Starting the pvmd via `rlogin` or `telnet` with a *chat* script would allow access even to IP-connected hosts behind firewall machines, and would require no special privilege to install. The main drawbacks are speed and the effort needed to get the chat program working reliably.

Two widely available systems are `rsh` and `rexec()`; both are used to cover the cases where a password does and does not need to be typed. A manual startup option allows the user to take the place of a chat program, starting the pvmd by hand and typing in the configuration. `rsh` is a privileged program which can be used to run commands on another host without a password, provided the destination host can be made to trust the source host. This can be done either by making it equivalent (requires a system administrator) [1] or by creating a `.rhosts` file on the destination host (this isn't a great idea). The alternative, `rexec()`, is a function compiled into the pvmd. Unlike `rsh`, which doesn't take a password, `rexec()` requires the user to supply one at run time, either by typing it in or placing it in a `.netrc` file (this is a really bad idea).

[1] Except on SunOS, which is still shipped with a "`+`" in `/etc/hosts.equiv`.

Figure 8 shows a host being added to the machine. A task calls `pvm_addhosts()`, to send a request to the its pvmd, which in turn sends a `DM_ADD` message to the master (possibly itself). The master pvmd creates a new host table entry for each host requested, looks up the IP addresses and sets the options from host file entries or defaults. The host descriptors are kept in a `waitc_add` structure (attached to a wait context), and not yet added to the host table. The master forks the pvmd' to do the dirty work, passing it a list of hosts and commands to execute (an `SM_STHOST` message). The pvmd' uses `rsh`, `rexec()` or manual startup to start each pvmd, pass it parameters and get a line of configuration data back. The configuration dialog between pvmd' and a new slave is as follows:

pvmd' → slave:	(exec) `$PVM_ROOT/lib/pvmd -s -d8 -nhonk 1 80a9ca95:0f5a 4096 3 80a95c43:0000`
slave → pvmd':	`ddpro<2312> arch<ALPHA> ip<80a95c43:0b3f> mtu<4096>`
pvmd' → slave:	`EOF`

The addresses of the master and slave pvmds are passed on the command line. The slave writes its configuration on standard output, then waits for an EOF from the pvmd' and disconnects. It runs in probationary status (`runstate = PVMDSTARTUP`) until it receives the rest of its configuration from the master pvmd. If it isn't configured within five minutes (parameter `DDBAILTIME`), it assumes there is some problem with the master and quits. The protocol revision (`DDPROTOCOL`) of the slave pvmd must match that of the master. This number is incremented whenever a change in the protocol makes it incompatible with the previous version. When several hosts are added at once, startup is done in parallel. The pvmd' sends the data (or errors) in a `DM_STARTACK` message to the master pvmd, which completes the host descriptors held in the wait context.

If a special task called a *hoster* is registered with the master pvmd when it receives the `DM_ADD` message, the pvmd' is not used. Instead, the `SM_STHOST` message is sent to the hoster, which starts the remote processes as described above

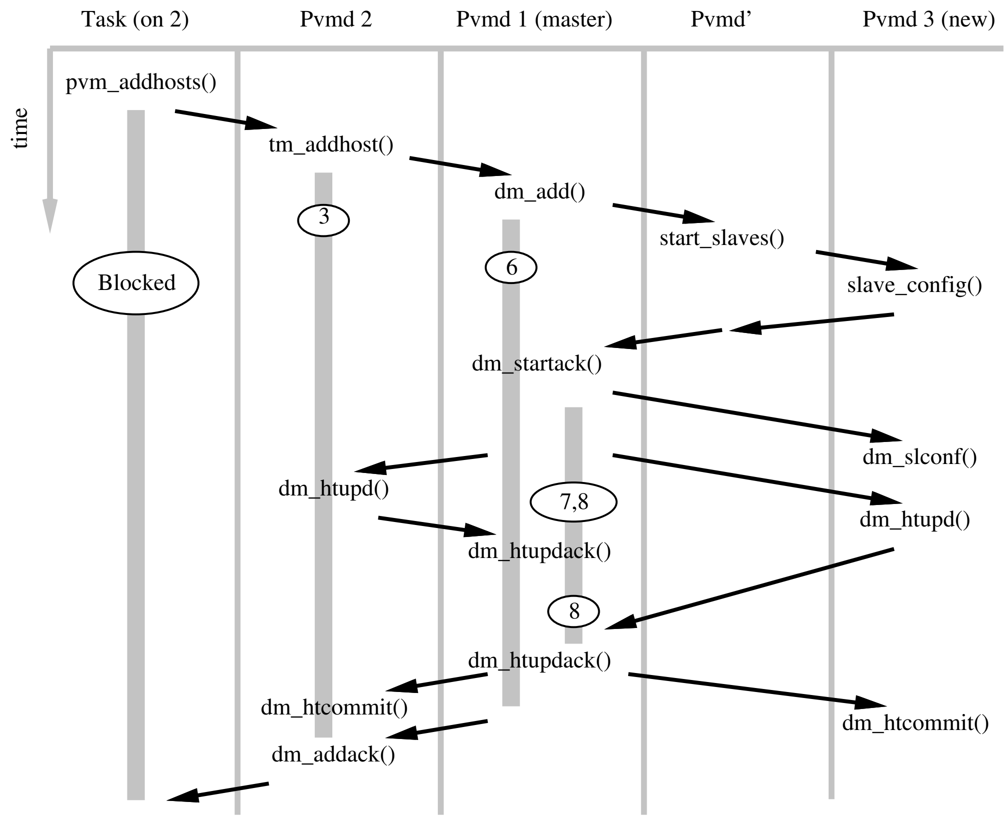

Figure 8: Timeline of Addhost Operation

using any mechanism it wants, then sends a `SM_STHOSTACK` message (same format as `DM_STARTACK`) back to the master pvmd. Thus, the method of starting slave pvmds is dynamically replaceable, but the hoster does not have to understand the configuration protocol. If the hoster task fails during an add operation, the pvmd uses the wait context to recover. It assumes none of the slaves were started and sends a `DM_ADDACK` message indicating a system error.

After the slaves are started, the master sends each a `DM_SLCONF` message to set parameters not included in the startup protocol. It then broadcasts a `DM_HTUPD` message to all new and existing slaves. Upon receiving this message, each slave knows the configuration of the new virtual machine. The master waits for an acknowledging `DM_HTUPDACK` message from every slave, then broadcasts an `HT_COMMIT` message, shifting all to the new host table. Two phases are needed so that new hosts are not advertized (e.g. by `pvm_config()`) until all pvmds know the new configuration. Finally, the master sends a `DM_ADDACK` reply to the original request, giving the new host IDs.

Experience suggests it would be cleaner to manage the pvmd' through the task interface instead of the host interface. This would allow multiple starters to run at once (parallel startup is implemented explicitly in a single pvmd' process).

4.3 Libpvm Library

4.3.1 Language Support

Libpvm is written in C and directly supports C and C++ applications. The Fortran library, `libfpvm3.a`, (also written in C) is a set of *wrapper* functions that conform to the Fortran calling conventions. The Fortran/C linking requirements are portably met by preprocessing the C source code for the Fortran library with `m4` before compilation.

4.3.2 Connecting to the Pvmd

On the first call to a libpvm function, `pvm_beatask()` is called to initialize the library state and connect the task to its pvmd. Connecting (for *anonymous* tasks) is slightly different from reconnecting (for spawned tasks).

The pvmd publishes the address of the socket on which it listens in `/tmp/pvmd.`*uid*, where *uid* is the numeric user ID under which the pvmd runs. This file contains a line of the form:

```
7f000001:06f7
```

This is the IP address and port number (in hexadecimal) of the socket. As a shortcut, spawned tasks inherit environment variable `PVMSOCK`, containing the same line.

To reconnect, a spawned task also needs its expected process ID. When a task is spawned by the pvmd, a task descriptor is created for it during the *exec* phase. The descriptor must exist so it can stash any messages that arrive for the task before it reconnects and can receive them. During reconnection, the task identifies itself to the pvmd by its PID. If the task is always the child of the pvmd, (i.e. the exact process *exec*'d by it) then it could use the value returned by `getpid()`. To allow for intervening processes, such as debuggers, the pvmd passes the expected PID in environment variable `PVMEPID`, and the task uses that value in preference to its real PID. The task also passes its real PID so it can be controlled normally by the pvmd.

`pvm_beatask()` creates a TCP socket and does a proper connection dance with the pvmd. Each must prove its identity to the other, to prevent a different user from spoofing the system. It does this by creating a file in `/tmp` writable only by the owner, and challenging the other to write in the file. If successful, the identity of the other is proven. Note this authentication is only as strong as the filesystem, and the authority of root on each machine.

A protocol serial number (`TDPROTOCOL`) is compared whenever a task connects to a pvmd or another task. This number is incremented whenever a change in the protocol makes it incompatible with the previous version.

Disconnecting is much simpler. It can be done forcibly by a *close* from either end, for example by exiting the task process. The function `pvm_exit()` performs a clean shutdown, such that the process can be connected again later (it would get a different TID).

4.4 Protocols

PVM communication is based on TCP and UDP. While other, more appropriate, protocols exist, they aren't as generally available. As originally specified, TCP can't take full advantage of the performance of modern high-speed, high-latency networks due to a window size limit of 64kB. Extensions have been defined to alleviate this problem and are becoming available [JBB92].

VMTP [Che88a] is one example of a protocol built for this purpose. Although intended for RPC-style interaction (request-response), it could support PVM messages. It is packet-oriented, and efficiently sends short blocks of data (such as most pvmd-pvmd management messages), but also handles streaming (necessary for task-task communication). It supports multicasting and priority data (something PVM doesn't need yet). Connections don't need to be established before use; the first communication initializes the protocol drivers at each end. VMTP was rejected because it is not widely available (using it requires modifying the kernel).

This section explains how TCP and UDP are employed and describes the PVM protocols built on them. There are three connections to consider: Between pvmds, between pvmd and task, and between tasks.

4.4.1 Messages

The pvmd and libpvm use the same message header, shown in figure 9. *Code* contains an integer tag (message type). Libpvm uses *Encoding* to pass the encoding style of the message, as it can pack in different formats. The pvmd always sets Encoding (and expects it to be set) to 1 (*foo*), Pvmds use the *Wait Context* field to pass the wait ID (if any, zero if none) of the *waitc* associated with the message. Certain tasks (resource manager, tasker, hoster) also use wait IDs. The *Checksum* field is reserved for future use. Messages are sent in one or more fragments, each with its own fragment header (described below). The message header is at the beginning of the first fragment.

	Byte 0	1	2	3
0	Code			
4	Encoding			
8	Wait Context ID			
12	(reserved for checksum)			

Figure 9: Message Header

4.4.2 Pvmd-Pvmd

PVM daemons communicate with one another through UDP sockets. UDP is an unreliable delivery service which can lose, duplicate or reorder packets, so an acknowledgement and retry mechanism is used. UDP also limits packet length, so PVM fragments long messages.

I considered TCP, but three factors make it inappropriate. First is scalability. In a virtual machine of N hosts, each pvmd must have connections to the other $N-1$. Each open TCP connection consumes a file descriptor in the pvmd, and some operating systems limit the number of open files to as few as 32, whereas a single UDP socket can communicate with any number of remote UDP sockets. Second is overhead. N pvmds need $N(N-1)/2$ TCP connections, which would be expensive to set up. The PVM/UDP protocol is initialized with no communication. Third is fault tolerance. The communication system detects when foreign pvmds have crashed or the network has gone down, so timeouts need to be set in the protocol layer. The TCP keepalive option might work, but it's not always possible to get adequate control over the parameters.

The packet header is shown in figure 10. Multi-byte values are sent in (Internet) *network byte order*, (most significant byte first).

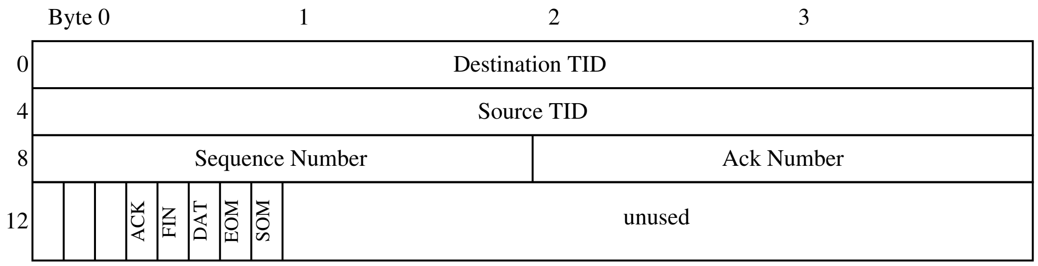

Figure 10: Pvmd-pvmd Packet Header

The source and destination fields hold the TIDs of the true source and final destination of the packet, regardless of the route it takes. Sequence and acknowledgement numbers start at 1 and increment to 65535, then wrap to zero.

SOM (EOM) – Set for the first (last) fragment of a message. Intervening fragments have both bits cleared. They are used by tasks and pvmds to delimit message boundaries.

DAT – If set, data is contained in the packet and the sequence number is valid. The packet, even if zero-length, must be delivered.

ACK – If set, the acknowledgement number field is valid. This bit may be combined with the DAT bit to piggyback an acknowledgement on a data packet [2].

FIN – The pvmd is closing down the connection. A packet with FIN bit set (and DAT cleared) begins an orderly shutdown. When an acknowledgement arrives (ACK bit set and ack number matching the sequence number from the FIN packet), a final packet is sent with both FIN and ACK set. If the pvmd panics, (for example on a trapped segment violation) it tries to send a packet with FIN and ACK set to

[2] Currently, the pvmd generates an acknowledgement packet for each data packet

every peer before it exits.

The state of a connection to another pvmd is kept in its host table entry. The protocol driver uses the following fields of `struct hostd`:

Field	Meaning
`hd_hostpart`	TID of pvmd
`hd_mtu`	Max UDP packet length to host
`hd_sad`	IP address and UDP port number
`hd_rxseq`	Expected next packet number from host
`hd_txseq`	Next packet number to send to host
`hd_txq`	Queue of packets to send
`hd_opq`	Queue of packets sent, awaiting ack
`hd_nop`	Number of packets in `hd_opq`
`hd_rxq`	List of out-of-order received packets
`hd_rxm`	Buffer for message reassembly
`hd_rtt`	Estimated smoothed round-trip time

Figure 11 shows the host send and outstanding-packet queues. Packets waiting to be sent to a host are queued in FIFO `hd_txq`. Packets are appended to this queue by the routing code (§4.5.1). No receive queues are used; incoming packets are passed immediately through to other send queues or reassembled into messages (or discarded). Incoming messages are delivered to a pvmd entry point (§4.1.4).

The protocol allows multiple outstanding packets to improve performance over high-latency networks, so two more queues are required. `hd_opq` holds a per-host list of unacknowledged packets and and global `opq` lists all unacknowledged packets, ordered by time-to-retransmit. `hd_rxq` holds packets received out of sequence until they can be accepted.

The difference in time between sending a packet and getting the acknowledgement is used to estimate the round-trip time to the foreign host. Each update is filtered into the estimate according to formula:

$$hd\_rtt_n = 0.75 * hd\_rtt_{n-1} + 0.25 * \Delta t$$

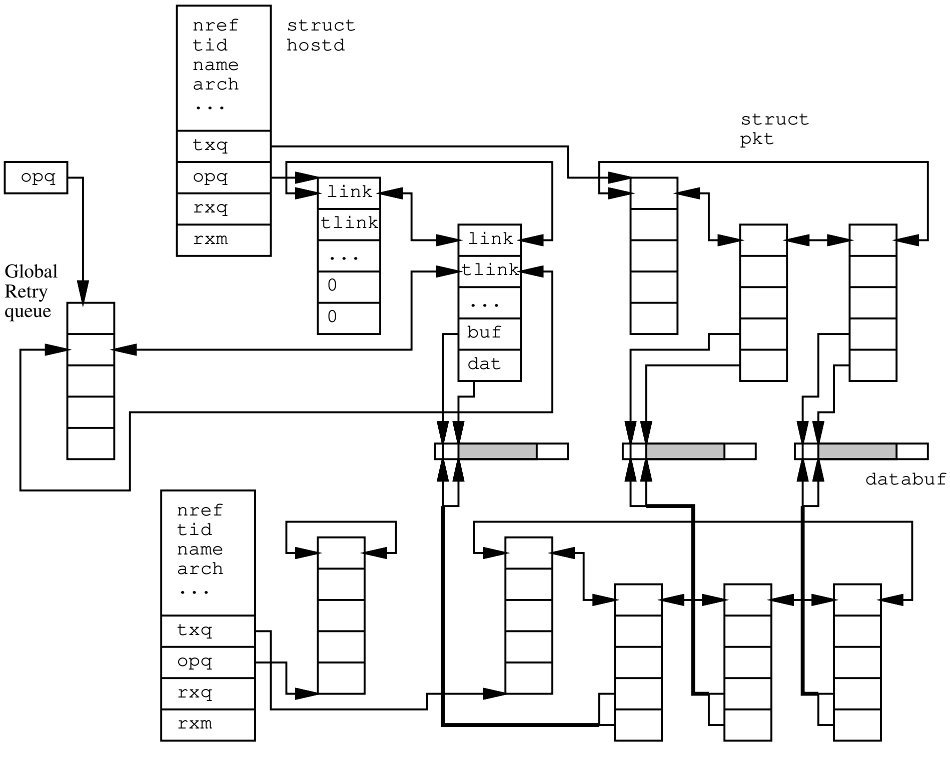

Figure 11: Host Descriptors with Send Queues

.

When the acknowledgement for a packet arrives, the packet is removed from `hd_opq` and `opq` and discarded. Each packet has a retry timer and count, and is resent until acknowledged by the foreign pvmd. The timer starts at $3 * hd\_rtt$, and doubles for each retry up to 18 seconds. `hd_rtt` is limited to nine seconds and backoff is bounded in order to allow at least 10 packets to be sent to a host before giving up. After three minutes of resending with no acknowledgement, a packet expires.

If a packet expires due to timeout, the foreign pvmd is assumed to be down or unreachable, and the local pvmd gives up on it, calling `hostfailentry()`

4.4.3 Pvmd-Task and Task-Task

A task talks to its pvmd and other tasks through TCP sockets. TCP is used because it delivers data reliably; UDP can lose packets even within a host. Unreliable delivery requires retry (with timers) at both ends, and tasks can't be interrupted while computing to perform I/O, so we can't use UDP.

Implementing a packet service over TCP is simple due to reliable delivery. The packet header is shown in figure 12. No sequence numbers are needed, and only flags *SOM* and *EOM* (these have the same meaning as in §4.4.2). Since TCP provides no record marks to distinguish back-to-back packets from one another, the length is sent in the header. Each side maintains a FIFO of packets to send, and switches between reading the socket when data is available and writing when there is space.

The main drawback to TCP (as opposed to UDP) is that more system calls are needed to transfer each packet. With UDP, a single `sendto()` and single `recvfrom()` are required. With TCP, a packet can be sent by a single `write()` call, but must be received by two `read()` calls, the first to get the header and the second to get the data.

When traffic on the connection is heavy, a simple optimization reduces the average number of reads back to about one per packet. If, when reading the packet

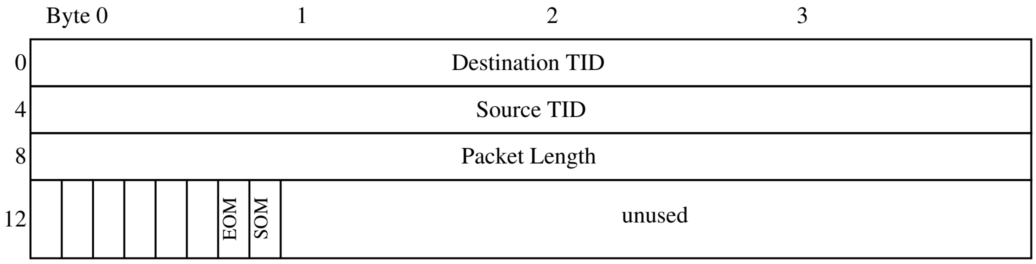

Figure 12: Pvmd-Task Packet Header

body, the requested length is increased by the size of a packet header, the read may succeed in getting both the packet body and header of the next packet at once. We have the header for the next packet for free and can repeat this process. [3]

4.5 Message Routing

4.5.1 Pvmd

Packet Buffers

Packet descriptors (`struct pkt`) track message fragments through the pvmd. Fields `pk_buf`, `pk_max`, `pk_dat` and `pk_len` are used in the same ways as similarly named fields of a frag, described in §4.1.1. Besides data, pkts contain state to operate the pvmd-pvmd protocol.

Message Routing

Messages are sent by calling `sendmessage()`, which routes by destination address. Messages for other pvmds or tasks are linked to packet descriptors and attached to a send queue. If the pvmd addresses a message to itself, `sendmessage()` passes the whole message descriptor to `netentry()`, avoiding the packet layer entirely.

[3] This was once implemented, but removed while updating the code and hasn't been reintroduced.

This loopback interface is used often by the pvmd. During a complex operation, `netentry()` may be reentered several times as the pvmd sends itself messages.

Messages to the pvmd are reassembled from packets in message reassembly buffers, one for each local task and remote pvmd. Completed messages are passed to entry points (§4.1.4).

Packet Routing

A graph of packet and message routing inside the pvmd is shown in figure 13. Packets are received from the network by `netinput()` directly into buffers long enough to hold the largest packet the pvmd will receive (its MTU in the host table). Packets from local tasks are read by `loclinput()`, which creates a buffer large enough for each packet after it reads the header. To route a packet, the pvmd chains it onto the queue for its destination. If a packet is multicast (§4.5.4), the descriptor is replicated, counting extra references on the underlying databuf. One copy is placed in each send queue. After the last copy of the packet is sent, the databuf is freed.

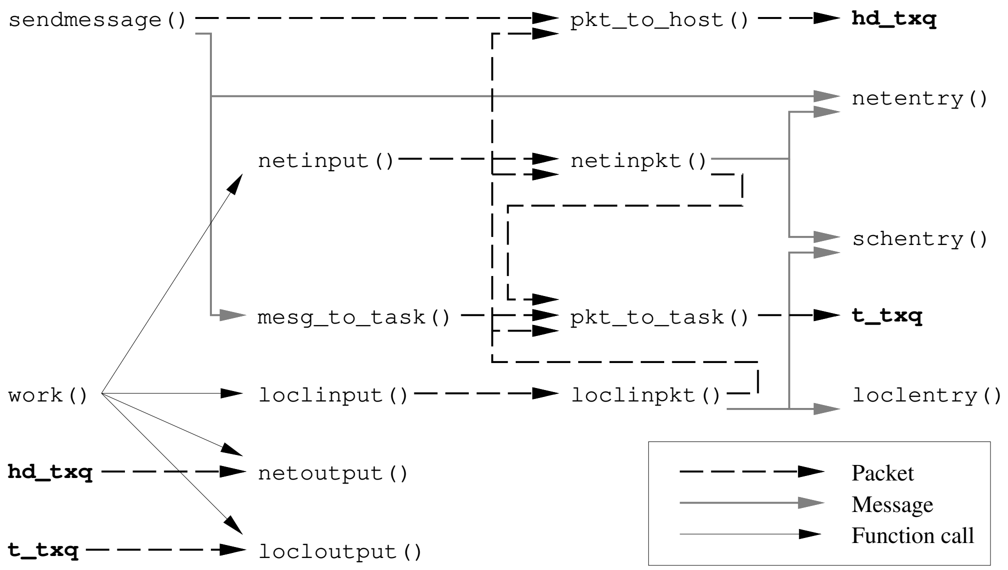

Figure 13: Packet and Message Routing in pvmd

Refragmentation

Messages are generally built with fragment length equal to the MTU of the host's pvmd, allowing them to be forwarded without refragmentation. In some cases, the pvmd can receive a packet (from a task) too long to be sent to another pvmd. The pvmd refragments the packet by replicating its descriptor as many times as necessary. A single databuf is shared between the descriptors. The `pk_dat` and `pk_len` fields of the descriptors cover successive chunks of the original packet, each chunk small enough to send. The `SOM` and `EOM` flags are adjusted (if the original packet is the start or end of a message). At send time, `netoutput()` saves the data under where it writes the packet header, sends the packet, then restores the data.

4.5.2 Pvmd and Foreign Tasks

Pvmds usually don't communicate with foreign tasks (those on other hosts). The pvmd has message reassembly buffers for each foreign pvmd and each task it manages. What it doesn't want is to have reassembly buffers for foreign tasks. To free up the reassembly buffer for a foreign task (if the task dies), the pvmd would have to request notification from the task's pvmd, causing extra communication.

For the sake of simplicity the pvmd local to the sending task serves as a message repeater. The message is reassembled by the task's local pvmd as if it were the receiver, then forwarded all at once to the destination pvmd, which reassembles the message again. The source address is preserved, so the sender can be identified.

Libpvm maintains dynamic reassembly buffers, so messages from pvmd to task do not cause a problem.

4.5.3 Libpvm

Four functions handle all packet traffic into and out of libpvm. `mroute()` is called by higher-level functions such as `pvm_send()` and `pvm_recv()` to copy messages into and out of the task. It establishes any necessary routes before calling `mxfer()`. `mxfer()` polls for messages, optionally blocking until one is received or until a specified timeout. It calls `mxinput()` to copy fragments into the task and reassemble messages. In the generic version of PVM, `mxfer()` uses `select()` to poll all routes (sockets) in order to find those ready for input or output. `pvmmctl()` is called by `mxinput()` when a control message is received (§4.1.5).

Direct Message Routing

Direct routing allows one task to send messages to another via TCP, avoiding the overhead of forwarding through the pvmds. It is implemented entirely in libpvm, using the notify and control message facilities. By default, a task routes messages to its pvmd, which forwards them on. If direct routing is enabled (`PvmRouteDirect`) when a message (addressed to a task) is passed to `mroute()`, it attempts to create a direct route if one doesn't already exist. The route may be granted or refused by the destination task, or fail (if the task doesn't exist). The message is then passed to `mxfer()`.

Libpvm maintains a protocol control block (`struct ttpcb`) for each active or denied connection, in list `ttlist`. The state diagram for a ttpcb is shown in figure 14. To request a connection, `mroute()` makes a ttpcb and socket, then sends a `TC_CONREQ` control message to the destination via the default route. At the same time, it sends a `TM_NOTIFY` message to the pvmd, to be notified if the destination task exits, with closure (message tag) `TC_TASKEXIT`. Then it puts the ttpcb in state `TTCONWAIT`, and calls `mxfer()` in blocking mode repeatedly until the state changes.

When the destination task enters `mxfer()` (for example to receive a message),

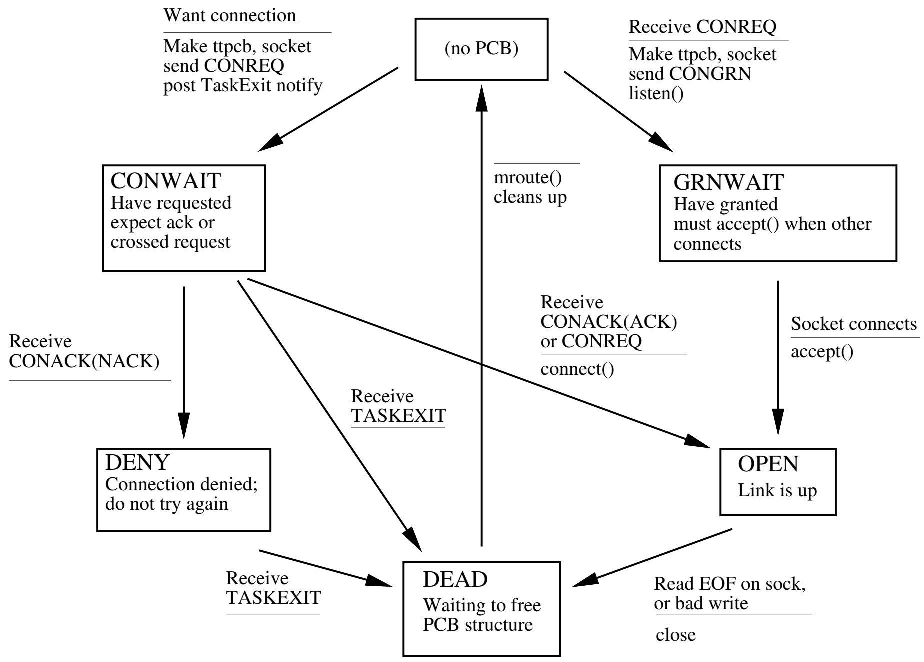

Figure 14: Task-Task Connection State Diagram

it receives the TC_CONREQ message. The request is granted if its routing policy (`pvmrouteopt != PvmDontRoute`) and implementation allow a direct connection, it has resources available, and the protocol version (TDPROTOCOL) in the request matches its own. It makes a ttpcb with state TTGRNWAIT, creates and listens on a socket, then replies with a TC_CONACK message. If the destination denies the connection, it nacks, also with a TC_CONACK message. The originator receives the TC_CONACK message, and either opens the connection (`state = TTOPEN`) or marks the route denied (`state = TTDENY`). Then, `mroute()` passes the original message to `mxfer()`, which sends it. Denied connections are cached in order to prevent repeated negotiation.

If the destination doesn't exist, the TC_CONACK message never arrives because the TC_CONREQ message is silently dropped. However, the TC_TASKEXIT message generated by the notify system arrives in its place, and the ttpcb state is set to TTDENY.

This connect scheme also works if both ends try to establish a connection at the same time. They both enter `TTCONWAIT`, and when they receive each others' `TC_CONREQ` messages, they go directly to the `TTOPEN` state.

4.5.4 Multicasting

Libpvm function `pvm_mcast()`, sends a message to multiple destinations simultaneously. The current implementation only routes multicast messages through the pvmds. It uses a 1:N fanout to ensure that failure of a host doesn't cause the loss of any messages (other than ones to that host). The packet routing layer of the pvmd cooperates with the libpvm to multicast a message.

To form a multicast address TID (GID), the G bit is set (refer to figure 1). The L field is assigned by a counter that is incremented for each multicast, so a new multicast address is used for each message, then recycled.

To initiate a multicast, the task sends a `TM_MCA` message to its pvmd, containing a list of recipient TIDs. The pvmd creates a multicast descriptor (`struct mca`) and GID. It sorts the addresses, removes bogus ones and duplicates and caches them in the mca. To each destination pvmd (ones with destination tasks), it sends a `DM_MCA` message with the GID and destinations on that host. The GID is sent back to the task in the `TM_MCA` reply message.

The task sends the multicast message to the pvmd, addressed to the GID. As each packet arrives, the routing layer copies it to each local task and foreign pvmd. When a multicast packet arrives at a destination pvmd, it is copied to each destination task. Packet order is preserved, so the multicast address and data packets arrive in order at each destination. As it forwards multicast packets, each pvmd eavesdrops on the header flags. When it sees a packet with `EOM` flag set, the flushes the mca.

4.6 Task Environment

4.6.1 Environment Variables

Experience indicates that inherited environment (UNIX `environ`) is useful to an application. For example, environment variables can be used to distinguish a group of related tasks or set debugging parameters. PVM makes increasing use of environment, and may eventually support it even on machines where the concept is not native. For now, it allows a task to export any part of `environ` to tasks spawned by it. Setting variable `PVM_EXPORT` to the names of other variables causes them to be exported through spawn. For example, setting:

```
PVM_EXPORT = DISPLAY:SHELL
```

exports the variables `DISPLAY` and `SHELL` to children tasks (and `PVM_EXPORT` too). The following environment variables are used by PVM:

Variable	Use
The user may set these:	
`PVM_ROOT`	Root installation directory
`PVM_EXPORT`	Names of environment variables to inherit through spawn
`PVM_DEBUGGER`	Path of debugger script used by spawn
These are set by PVM and should not be modified:	
`PVM_ARCH`	PVM architecture name
`PVMSOCK`	Address of the pvmd local socket (§4.3.2)
`PVMEPID`	Expected PID of a spawned task
`PVMTMASK`	Libpvm Trace mask

4.6.2 Standard Input and Output

Each task spawned through PVM has `/dev/null` opened for *stdin*. It inherits from its parent a *stdout sink*, which is a *(TID, code)* pair. Output on *stdout* or *stderr* is read by the pvmd through a pipe, packed into PVM messages and sent to the TID, with message tag equal to the code. If the output TID is set to zero (the default for a task with no parent), the messages go to the master pvmd, where they are written on its error log. Four types of messages are sent to a stdout sink. The message body formats for each type are:

Class	Message Body		
	`(code) {`		
		`int tid,`	Task ID
Spawn:		`int -1,`	Signals spawn
		`int ptid`	TID of parent
	`}`		
	`(code) {`		
		`int tid,`	Task ID
Begin:		`int -2,`	Signals task creation
		`int ptid`	TID of parent
	`}`		
	`(code) {`		
		`int tid,`	Task ID
Output:		`int count,`	Length of output fragment
		`char data[count]`	Output fragment
	`}`		
	`(code) {`		
End:		`int tid,`	Task ID
		`int 0`	Signals EOF
	`}`		

The first two items in the message body are always the task ID and output count, which allow the receiver to distinguish between different tasks and the four message types. For each task, one message each of types *Spawn*, *Begin* and *End* is sent, along with zero or more messages of class *Output* (`count > 0`). Classes *Begin*, *Output* and *End* will be received in order, as they originate from the same source (the pvmd of the target task). Class *Spawn* originates at the (possibly different) pvmd of the parent task, so it can be received in any order relative to the others. The output sink is expected to understand the different types of messages and use them to know when to stop listening for output from a task (EOF) or group of tasks (global EOF).

The messages are designed so as to prevent race conditions when a task spawns another task, then immediately exits. The output sink might get the *End* message from the parent task and decide the group is finished, only to receive more output later from the child task. According to these rules, the *Spawn* message for the second task must arrive before the *End* message from the first task. The *Begin* message itself is necessary because the *Spawn* message for a task may arrive after the *End* message for the same task. The state transitions of a task as observed by the receiver of output messages are shown in figure 15.

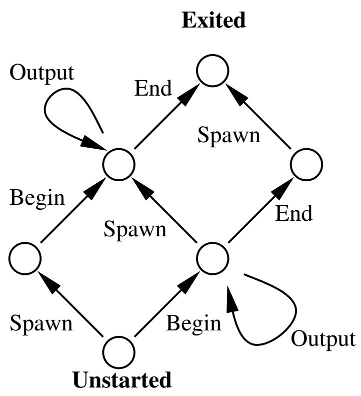

Figure 15: Output States of a Task

Libpvm function `pvm_catchout()` uses output collection to put the output from children of a task into a file (for example its own stdout). It sets output TID to its own task ID, and the output code to control message `TC_OUTPUT`. Output from children and grandchildren tasks is collected by the pvmds and sent to the task, where it is received by `pvmmctl()` and printed by `pvmclaimo()`.

4.6.3 Tracing

Libpvm has a tracing system which can record the parameters and results of all calls to interface functions. Trace data is sent as messages to a trace sink task just as output is sent to a stdout sink (§4.6.2). If the trace output TID is set to zero (the default), tracing is disabled.

Besides the trace sink, tasks inherit a trace mask, used to enable tracing per-function. The mask is passed as a (printable) string in environment variable `PVMTMASK`. A task can manipulate its own trace mask or the one to be inherited from it. A task's trace mask can also be set asynchronously with a `TC_SETTMASK` control message.

Trace data from a task is collected in a manner similar to the output redirection discussed above. Like the type *Spawn*, *Begin* and *End* messages which bracket output from a task, `TEV_SPNTASK`, `TEV_NEWTASK` and `TEV_ENDTASK` trace messages are generated by the pvmds to bracket trace messages.

The tracing system is very new at this time, and may evolve further. Details of the trace messages are not documented here.

4.6.4 Debugging

PVM provides a simple but extensible debugging facility. Tasks started by hand could just as easily be run under a debugger, but this is cumbersome for those spawned by an application, since it requires the user to comment out the calls to `pvm_spawn()` and start tasks manually. If `PvmTaskDebug` is added to the flags passed to `pvm_spawn()`, the task is started through a debugger script (a normal shell script), `$PVM_ROOT/lib/debugger`.

The pvmd passes the name and parameters of the task to the debugger script, which is free to start any sort of debugger. The script provided is very simple. In an *xterm* window, it runs the correct debugger according to the architecture type of the host. The script can be customized or replaced by the user. The pvmd can be made to execute a different debugger via the `bx=` host file option or the `PVM_DEBUGGER` environment variable.

4.7 Console Program

The PVM *console* program is like a command shell with a small set of built-in commands. It can be used to configure the virtual machine, start, kill and check status of processes. It can collect output or trace data from spawned tasks, using the redirection mechanisms described in §4.6.2 and §4.6.3, and write them to the screen or a file. It uses the *begin* and *end* messages from child tasks to maintain groups of tasks (*jobs*), related by common ancestors. Using host notify events (§3.8), it informs the user when the virtual machine is reconfigured. The console can be connected to any pvmd in the system, and any number of consoles can be connected at the same time. If no pvmd is running when the console is started, it attempts to start one automatically, then connect to it.

The console connects to PVM as a normal task, requiring no special privileges. Implementing it this way avoids creating a separate "management" interface to the pvmd (it uses the existing libpvm protocol), and ensures that the libpvm interface is powerful enough to manage the system.

4.8 Resource Limitations

Resource limits imposed by the operating system and available hardware are passed on to PVM applications. Whenever possible, PVM avoids setting explicit limits, instead it returns an error when resources are exhausted. Competition between users on the same host or network affects some limits dynamically.

4.8.1 In the PVM Daemon

How many tasks each pvmd can manage is limited by two factors: The number of processes allowed a user by the operating system, and the number of file descriptors available to the pvmd. The limit on processes is generally not an issue. Each task consumes one file descriptor in the pvmd, for the pvmd-task TCP stream. Each spawned task consumes a second descriptor, for the pipe to read its output (closing stdout and stderr in the task would reclaim this slot). A few more are always in use by the pvmd for the local and network sockets and error log file. With a limit of (for example) 64 open files, a user should be able to have up to 30 tasks running per host.

The pvmd uses dynamically allocated memory to store message packets en route between tasks. Until the receiving task accepts the packets, they accumulate in the pvmd in a FIFO. No flow control is imposed by the pvmd – it will happily store all the packets given to it, until it can't get any more memory. If an application is designed so that tasks can keep sending even when the receiving end is off doing something else and not receiving, the system will eventually run out of memory.

4.8.2 In the Task

As with the pvmd, a task may have a limit on the number of others it can connect to directly. Each direct route to a task has a separate TCP connection (which is bidirectional), and so consumes a file descriptor. Thus with a limit of 64 open files, a task can establish direct routes to about 60 other tasks. Note this limit is only in effect when using task-task direct routing. Messages routed via the pvmds only use the default pvmd-task connection.

The maximum size of a PVM message is limited by the amount of memory available to the task. Because messages are generally packed using data existing elsewhere in memory, and they must be reside in memory between being packed and sent, the largest possible message a task can send should be somewhat less than half the available memory. Note that as a message is sent, memory for packet buffers is allocated by the pvmd, aggravating the situation. Inplace message encoding alleviates this problem somewhat, because the data is not copied into message buffers in the sender. However, on the receiving end, the entire message is downloaded into the task before the receive call accepts it, possibly leaving no room to unpack it.

In a similar vein, if many tasks send to a single destination all at once, the destination task or pvmd may be overloaded as it tries to store the messages. Keeping messages from being freed when new ones are received by using `pvm_setrbuf()` also uses up memory.

These problems can sometimes be avoided by rearranging the application code, for example to use smaller messages, eliminate bottlenecks, and process messages in the order in which they are generated.

4.9 Debugging the System

4.9.1 Sane Heap

PVM uses dynamically allocated memory for things such as host tables and message buffers. To help catch bugs in the system code, the pvmd and libpvm use a sanity-checking library called *imalloc*. imalloc functions are wrappers for the regular *libc* functions `malloc()`, `realloc()` and `free()`. Upon detecting an error, the imalloc functions abort the program so the fault can be traced.

The following checks and functions are performed by imalloc:

1. The length argument to `i_malloc()` and `i_realloc()` is checked for insane values.

2. All allocated blocks are tracked in a hash table to detect when a block is freed more than once or `i_free()` is called with a block not obtained by `i_malloc()`.

3. `i_malloc()` and `i_realloc()` fill pads around each block with a pseudo-random pattern. The pattern is checked by `i_free()` to detect writing past the end of a block.

4. `i_free()` zeros each block before it frees it so further references may fail and make themselves known.

5. Each block has a serial number and is tagged to indicate its use. The heap space can be dumped or sanity-checked by calling `i_dump()`. This helps find memory leaks.

Since the overhead of this checking is quite severe, it is disabled at compile time by default. Defining `USE_PVM_ALLOC` in the source Makefile(s) switches it on.

4.9.2 Runtime Debug Masks

The pvmd and libpvm each have a debugging mask that can be set to enable logging of various information. Logging information is divided up into classes, each of which is enabled separately by a bit in the debug mask. The pvmd command line option `-d` sets the debug mask of the pvmd to the (hexadecimal) value specified; the default is zero. Slave pvmds inherit the debug mask of the master at the time they are started. The debug mask of a pvmd can be set at any time using the console `tickle` command on that host. The debug mask in libpvm can be set in the task with `pvm_setopt()`.

Note: The debug mask is not intended for debugging application programs.

The pvmd debug mask bits are defined in ddpro.h, and the libpvm bits in lpvm.c. The meanings of the bits are not well defined and are subject to change, as they're intended to be used when fixing or modifying the pvmd or libpvm. Presently, the bits in the debug mask correspond to:

Name	Bit	Debug messages about
`pkt`	1	Packet routing
`msg`	2	Message routing
`tsk`	4	Task management
`slv`	8	Slave pvmd startup
`hst`	10	Host table updates
`sel`	20	Select loop (below packet routing layer)
`net`	40	Network twiddling
`mpp`	80	MPP port specific
`sch`	100	Resource manager interface

4.9.3 Statistics

The pvmd includes registers and counters to sample certain events, for example the number of packets refragmented. These values can be computed from a debug log, but the counters have less adverse impact on the performance of the pvmd than would generating a huge log file. The counters can be dumped or reset using the `pvm_tickle()` function or the console tickle command. The code to gather statistics is normally switched out at compile-time. To enable it, edit the makefile and add `-DSTATISTICS` to the compile options.

CHAPTER 5

WATER TEST

PVM version 3.3 was used in all tests. I used the compiler shipped with each machine, and enabled optimization with "-O". The Sparc IPX machines used for most of the tests were running SunOS 4.1.3 and had 16MB main memory. They were up and running normally.

5.1 Performance Measurements

I ran experiments to determine the message-passing performance of PVM, relative to raw sockets. I measured message latency and throughput and calculated the overhead due to PVM. Ethernet is a 10Mbit/S broadcast medium. The network used for these tests is a moderately busy and messy wire (the Computer Science department backbone). All machines were connected to the same segment, possibly through a bridge; packets were routed from host to host in a single hop and not through any gateways or routers.

5.1.1 Message Latency

Communication latency (delay) between tasks is important to fine-grain parallel algorithms. Tasks synchronize by sending small messages back and forth, so with larger latency more time is spent idle, waiting for a reply.

Transfer time for a message is at best $D + L/W$, where D is the network latency, L the length of the message and W the bandwidth of the network. Latencies were measured for small messages because as length increases, delay becomes a smaller part of the transfer time. For example, if the latency is 1mS and throughput is 1MB/S, messages longer than 1kB will be bound more by throughput, and those shorter than 1kB bound more by latency. Latencies were measured for message lengths up to 8kB. Both UDP and TCP sockets were tested.

To avoid having to synchronize the clocks of two hosts or approximate the offset, messages are sent round-trip, and the total time difference (measured on a single host) is divided in half. Messages should take roughly the same amount of time to go in each direction.

Figure 16 shows latencies over raw UDP sockets for several UNIX systems, between two processes on the same machine and over a local Ethernet. They all perform more or less the same. On-host latency is determined by CPU speed and network code in the kernel, while off-host latency is determined mainly by network code and the physical network. The latency that should be attainable over Ethernet is also plotted. No UNIX machine comes anywhere near this line, especially at the low end (500μS real vs. 70μS theoretical).

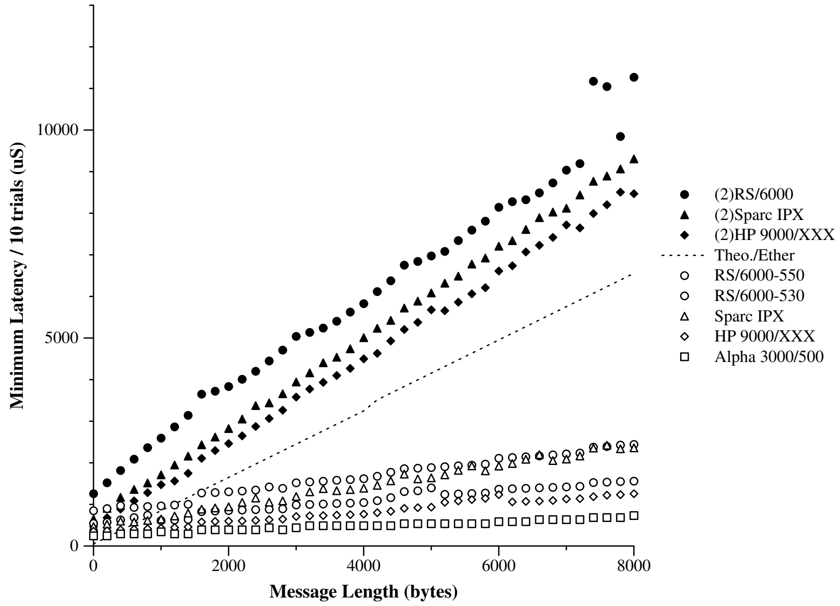

Figure 16: Typical On-host and Inter-host Latencies

5.1.2 Message Throughput

Tests of throughput were run with message lengths varying from 10 bytes to 10M bytes. Very large messages show where and how the system begins to lose. When messages are megabytes long, problems show up that aren't evident with small sizes. Behavior of memory-allocation systems becomes important, because message buffers are dynamically allocated. Effects of operating system paging and scheduling, and protocol implementation show up. Sparse samples can hide chaotic performance (for example when message length is close to a power of two) – an order of magnitude is way too much.

PVM message throughput is compared to a TCP connection. Both *fair* and *forwarding* message speeds were measured between tasks. Fair speed is measured as the time needed to pack a message, send it to another task, and unpack it; this is how long it would take to get real data from one task to another. Packing and unpacking is done to a static 10k byte array. Forwarding speed is the speed with which a task can receive a message and send it again. Comparison of the two against raw sockets shows where overhead comes from in PVM. Messages were passed between tasks on the same physical host and on different hosts, using both *default* and *direct* routing. As with latency measurements, messages were sent round-trip to simplify timing.

5.2 Analyzing the Performance

5.2.1 Latency

Figure 17 shows latencies between two processes (or tasks) running on the same physical host when communicating through UDP or TCP sockets (or PVM messages).

Raw sockets are of course faster than PVM messages, but are also more consistent. The variance with PVM messages is much larger than the average value.

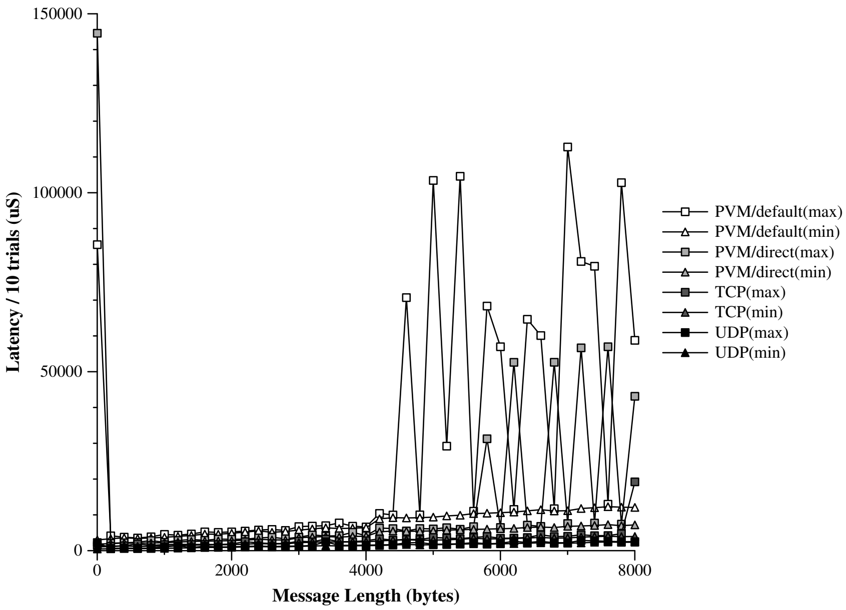

Figure 17: Minimum, Maximum On-host Latency, Sparc IPX

This is likely due to process scheduling (the machines were running other jobs at the time). In each case, one or two of ten data points were far higher than the rest. Maximum values for default routing are approximately twice as large as for direct routing, because the message travels through two processes and is more likely to be delayed by scheduling. The first data point on the PVM direct route curve corresponds to the first message sent, which is when the direct route is established, causing a large delay.

Median latencies, shown in figure 18, are close to minimum values. Latency is highest when the default message route is used, because the message is routed through a pvmd instead of directly from task to task, resulting in an extra copy, and an extra context switch to wake up the pvmd. Enabling direct routing improves the performance almost by 50%. Still, some time is lost in the PVM message buffer management code. The default PVM fragment size is 4k bytes, so we see a bump in latency at multiples of 4kB, when an additional fragment becomes necessary.

Dashed lines show predicted latencies of PVM messages, as calculated from the

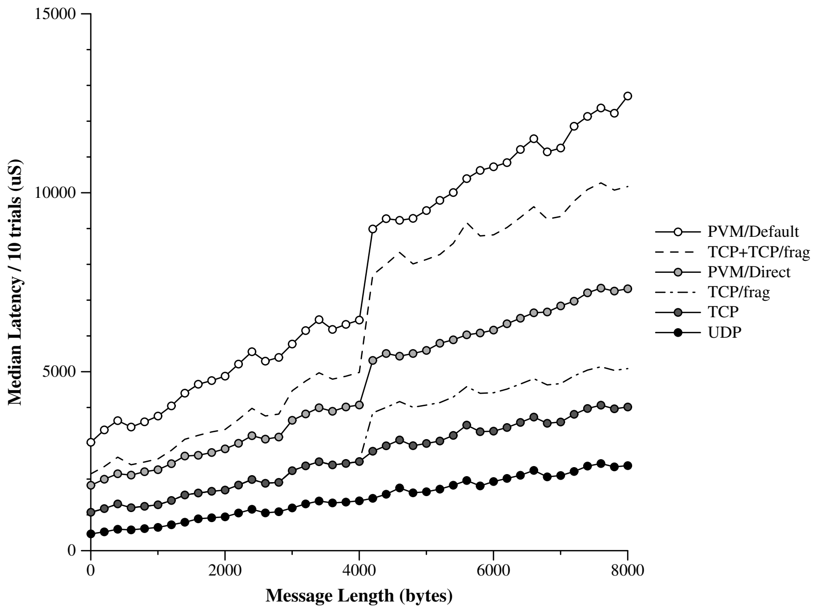

Figure 18: Median On-host Latency, Sparc IPX

measured socket latencies. Direct route is compared to a single TCP connection, with overhead (another call to `write()`) added in at multiples of 4kB. Default route is compared to two TCP connections, from each task to the pvmd. PVM buffer management overhead is approximately 900μS. It increases with message length because data must be copied to the message buffer.

Figure 19 shows the previous experiment repeated with processes (tasks) running on two different physical hosts, connected by Ethernet. Median latency is used, to filter out effects of process scheduling and network traffic.

It's interesting to note that TCP is slightly slower than UDP for messages around zero length, and slightly faster for messages around 8kB long, even though all messages fit into a single datagram. Perhaps this is because the UDP datagrams must be fragmented by the IP layer (Ethernet MTU is 1500 bytes) while TCP generates correct-length segments (slightly less than 1500 bytes).

Performance of PVM is again predicted by composing the measured performance of raw sockets. Direct route is compared to a single TCP connection, while default

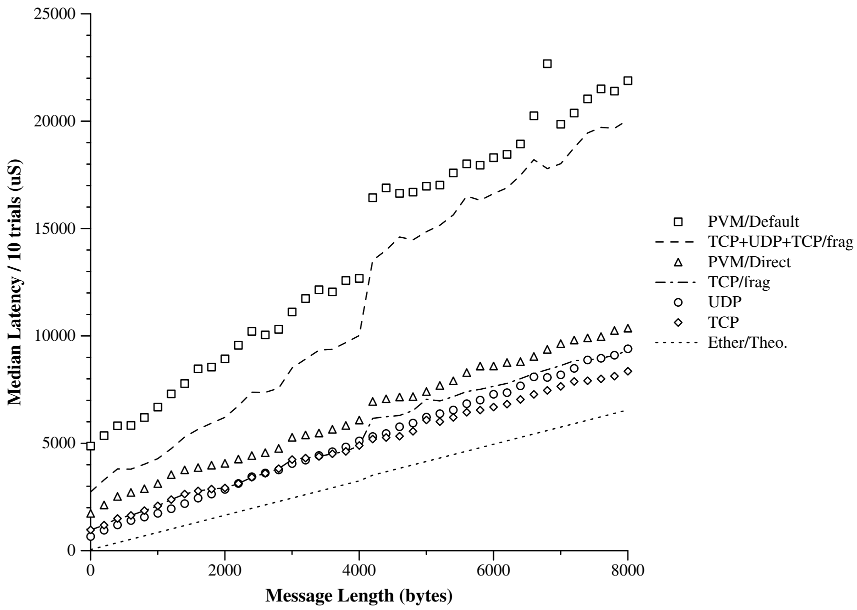

Figure 19: Median Inter-host Latency, Sparc IPX

route is compared to two (local) TCP connections in series with a UDP connection. Between hosts, default route latency is much worse than for direct route for two reasons: First, another pvmd is in the message path. Second, the pvmd-pvmd protocol uses stop-and-wait (a single outstanding packet), so the bump at 4kB is more noticeable.

5.2.2 Throughput

Figure 20 compares PVM throughput over Ethernet to a raw TCP socket. Throughput over UDP was not tested (though the pvmds use it) because it cannot handle long messages. [1]

Perfect performance on Ethernet is calculated by modeling PVM messages, IP and Ethernet protocols, assuming no contention for the network or CPUs. UNIX

[1] A message service built on UDP can achieve better performance than some implementations of TCP.

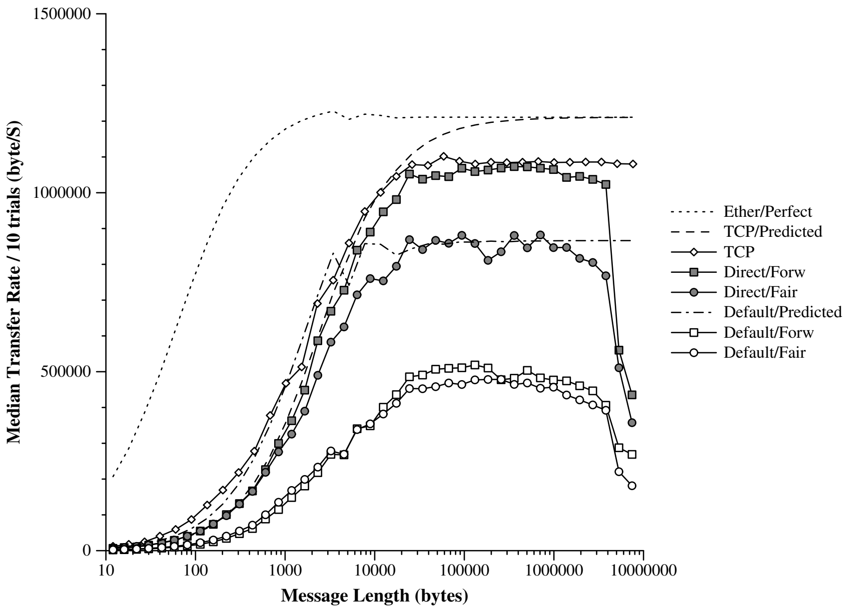

Figure 20: Median Inter-host Throughput, Sparc IPX

machines perform poorly for small messages, due to latency in the kernel. The predicted curve for direct routing (and TCP) was calculated by adding the measured TCP latency of one byte to the total message time from the ideal model. The predicted curve for default routing was calculated by assuming the UDP connection to be the limiting factor. Twice the measured UDP latency of one byte was added to the ideal model for each PVM fragment sent.

Performance drops off sharply for messages around 5MB, because the kernel is paging fragments of the message buffers. The workstations used had 16MB core, and the kernel and other programs running used about 6MB. The test program is written such that two message buffers exist in memory at the same time (one from the previous message), so paging begins with messages longer than 5MB. The TCP curve shows no rolloff because data sent through the socket wasn't written anywhere. Throughput is summarized in the following table:

Mode	Throughput	
	MB/S	nS/B
TCP	1.09	917
Direct, Forward	1.06	943
Direct, Fair	0.88	1140
Default, Forward	0.51	1960
Default, Fair	0.47	2130

Figure 21 shows the same tests repeated, with both tasks running on the same host. On a single machine, performance is only limited by memory-memory copy speed. Throughput is summarized in the following table:

Mode	Throughput	
	MB/S	nS/B
TCP	1.88	532
Direct, Forward	1.50	667
Direct, Fair	1.15	870
Default, Forward	0.75	1330
Default, Fair	0.66	1520

The difference between fair and forwarding speeds shows the cost of packing and unpacking a message to be approximately 200nS/B, (100nS/B each). On the same machine, `bcopy()` costs 80nS/B. Using PVM messages with direct routing is only slightly more expensive than raw TCP (no packing/unpacking was done in the TCP test), but the cost of default routing can be prohibitive (more than 100% overhead).

As shown in the previous experiment, performance drops sharply when message buffers must be paged. This happens at half the previous size (2.5 MB), because both tasks are running on the same workstation, competing for memory.

The drop in throughput around message length 100kB is due to another process stealing cycles from the two tasks. The drop occurs at the point where the time needed to transfer a message is equal to the kernel's scheduler time quantum, which on these machines is 100mS. The dashed line shows the points where

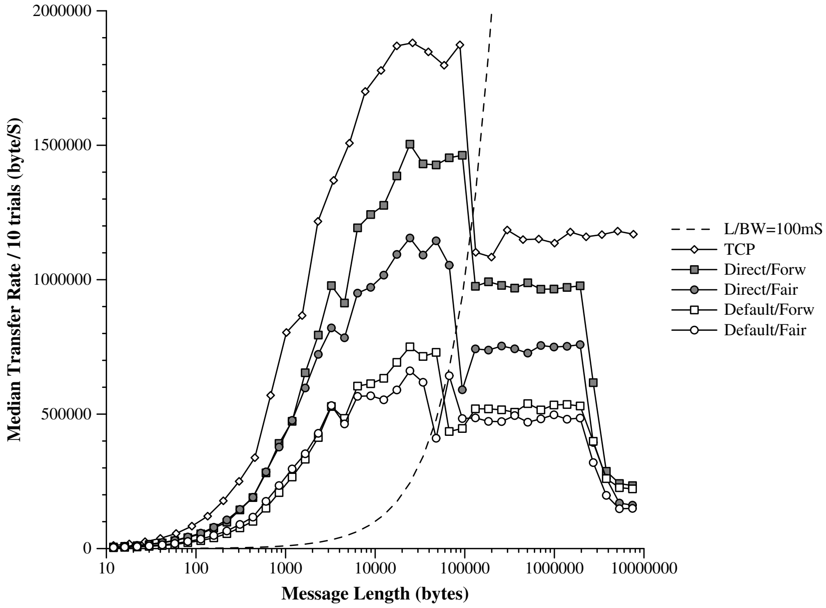

Figure 21: Median On-host Throughput, Sparc IPX

$message\ length/bandwidth = 100mS$. Short messages can be sent from one task to another completely in a single time slot. The system calls to read and write the underlying socket serve to synchronize the tasks with the scheduler. Longer messages are transferred during several time slots, with the tasks scheduled in a round-robin fashion. The external process gets one slot in every three, or one third of the CPU, so one third of the throughput is lost. This effect isn't seen as dramatically when sending messages between hosts (though it can be observed). The network moves data in the background and is somewhat slower than the processor, so the tasks have time to catch up when they are scheduled back in.

CHAPTER 6

CONCLUSIONS

6.1 Results

PVM is a message-passing system that can be ported to different machine environments, and used to combine them into a single meta-computer. It provides a standardized programming interface, callable from a normal sequential language (C, C++ and Fortran interfaces are provided). Fault-tolerant applications can be written by exploiting the ability of the system to detect and recover from hardware and software failures. It is easy to use, yet expressive enough to allow real applications to be built.

6.1.1 Fault Tolerance

The first goal of Version 3 is fault tolerance. The PVM system itself can withstand the loss of any host in the virtual machine, except for the master, and any task. The master host is currently not replaceable and must always remain in the configuration. PVM saves enough information for its own purposes so that if a host fails, the system can reschedule, or at least terminate, any operations pending on it.

This means that even a naive application will not hang waiting for an operation involving a down host to complete. More enlightened applications can receive a notify message from the PVM system and take their own measures to recover. Admittedly, writing code to take advantage of this facility is a messy task. It usually needs to be embedded into the application, because the actions needed to recover are very specific to each program. Hopefully, libraries can be built to make this somewhat easier. A *queue manager* system was written to work with PVM Version 2 [Sep93], which simplifies the job of writing bag-of-tasks applications by managing data dependencies and task creation. This system could possibly be extended to run with Version 3 and include automatic fault tolerance. For programmers who don't want to bite off the complexity of using notify events, the time-limited receive function, `pvm_trecv()`, provides a simple way out. Generally, when an application component fails, the rest of the application will hang waiting on results from it. Programs can instead time out and poll to make sure the component is still running.

6.1.2 Portability

Another important goal was for PVM to be portable to a variety of different architectures and operating systems. Thus far, it has been ported to more than 30 popular versions of UNIX, and several distributed and shared memory machines. Ports were done to VMS by Dan Clark at the Oregon Graduate Institute and to OS/2 by Jan Ftacnik at Brookhaven National Laboratory. To reduce maintenance work, the source code is intended to be sharable as much as possible between the different ports. The "generic" release UNIX ports differ by about a hundred lines of code (out of about 25000), conditionally compiled in by architecture type. About 2000 lines are replaced in the distributed memory ports (CM-5, iPSC/860 and Paragon), with about 1000 of those shared between the three ports. Results aren't in yet on a recently completed System-5 shared-memory port. PVM can run over any type of network, Ethernet, FDDI, HiPPI, ATM, even slip (really), as long as TCP and UDP are available.

6.1.3 Performance

The communication performance of PVM leaves a little to be desired. It is affected by three main factors. First, protocol drivers run in user space (in the pvmd and tasks). The pvmd-pvmd protocol suffers most, because it manages timers and resend queues. It's expensive to read timers from user space because it must be done via system calls (one profiling of the pvmd showed 10% of its time spent in `gettimeofday()`). Performance might improve if UDP could be replaced with a protocol with reliable delivery, eliminating the need to resend packets. The time-of-day clock could be mapped to memory, to eliminate the system call [Jep94]. The pvmd-task (also task-task) protocol is based on TCP and so doesn't require any timers. Also, both pvmd and task maintain a number of connections. A large fraction of time is spent in `select()`, multiplexing different inputs.

Second, the pvmd-pvmd protocol allows multiple outstanding (unacknowledged) packets on a connection, but the pvmd only sends them one-by-one. The number of outstanding packets can be set manually, but a control system is needed to set it dynamically. On a high-bandwidth, high-latency network, a single packet is not enough to keep the pipe full. Pvmd-pvmd communication speed is therefore limited by network latency and bandwidth, instead of just bandwidth. Performance on workstations over Ethernet is reasonable, because the network latency is low.

Third, message data is copied a number of times. Default message routing (through the pvmd) incurs five copies: the data must be packed, routed through four processes (three more copies), and finally unpacked. Direct routing improves over that (three copies total), since the message is sent directly between tasks, but the cost to establish a route is high because the request and acknowledgement messages travel via the default route. Inplace encoding eliminates one more copy by leaving data in place until send time. It would be nice to have some sort of inplace receive, but it's not clear how to define it.

6.1.4 Scalability

Virtual machines haven't yet been run at the sizes (hundreds of hosts) we were trying to achieve; the largest reported so far have included approximately 100 hosts.

Networks with higher performance than Ethernet aren't very common yet, neither are multiprocessors with more than a few hundred PEs. We haven't found many algorithms inherently parallel enough to make use of 100 relatively fast processors connected by a slow network, except for ray-tracing and cracking passwords. A few features of PVM will need redesign to scale better: Slave pvmd startup takes a long time (average one second per host) though is still much improved over Version 2, because several hosts are processed in parallel. Some scatter/gather operations, such as spawning tasks and multicasting, don't scale well because the communication uses a direct 1:N fanout. Acknowledgements tend to come back all at once, swamping the central host and causing it to drop packets, which then have to be retransmitted.

6.1.5 Heterogeneity

PVM supports clusters of mixed machine architectures. Data is converted automatically, provided that the `PvmDataDefault` message encoding is used. For simplicity, each library can only decode its native format and XDR. This results in some unnecessary conversions; it would be nice to use a format determined by majority rule or receiver-makes-right conversion. However, that would require the library to have several sets of decoders, resulting in additional maintenance work and larger executables.

6.1.6 How Real is It?

PVM is not a distributed operating system. One feature it lacks is a filesystem, making due with whatever is provided by machines on which it runs. The filesystem was left out not only because of the complexity of making a portable I/O library, but because programmers would have to use custom functions to access PVM files. This is less of a problem in languages like C, where I/O functions are like any others, than in Fortran, where I/O is built into the language in some non-portable fashion.

File I/O has been retrofitted onto PVM. The *pvmfs* [Phi93] system works for C programs by intercepting system calls like `open()` and `read()`, and generating its own file handles. It does normal I/O for local files, otherwise it uses PVM messages to communicate with a file server task. Since it replaces the system calls, libraries such as *stdio* can now work with PVM files or normal files. Programs run with only relinking. The main drawback to pvmfs is that is uses `syscall()` to do its dirty work, so it's not very portable. Getting it to a new machine can be fairly messy, and OS revisions can break it later on.

PVM lets several disjoint applications run on the same virtual machine, but it's more like a single-user system. No protection is provided between applications: any task can send a message to any other task or swamp the machine by flooding it with messages. [1] It's cumbersome to use in an everyday fashion – to start editors and compilers on machines around the house. If a host goes down, it must be manually added back to the machine when it comes up. Normal UNIX processes aren't managed as PVM tasks very well, as they live in a probationary status until they use libpvm functions to enroll.

6.2 Recommendations

6.2.1 Richer Programming Interface

In retrospect, the PVM programming interface should have been made richer. For simplicity and portability, features such as asynchronous message transfer were left out. These may not be supportable on all machines, but when available from the underlying system, they are critical to good performance.

[1] Different PVM users *are* protected from each other, though.

6.2.2 Losing the Master Pvmd

One key limitation to fault tolerance in PVM is the master pvmd. It must never crash and cannot be configured out of the system. It seems necessary to at least devise a way for master to hand off to another pvmd. This would allow an application to migrate if it knows in advance that the master host must go down. Even better would be to designate one or more backup masters to which slaves would automatically defer if the master crashed.

An interesting point made by the authors of the V kernel [Che88b] is that in their first design they used object identifiers with a *logical host* field, much like the H field in a PVM TID. They realized that this inhibits process migration (except if all processes on a host migrate together), and suggest that identifiers not be designed this way, even though it seems like a performance win. In a similar vein, TIDs are assigned arbitrarily by pvmds and can't be requested, making it difficult to restart a checkpointed task. These problems have been rediscovered in a few efforts to add checkpointing to PVM [Beg94, Pla94].

6.2.3 After-Market Options

PVM attempts to be "virtual hardware" and as such should include as little policy as possible (such as where to place tasks and how to manage hosts). Toward this end, parts of the system have recently been made replaceable. Tasks can be enrolled to take over some functions of the pvmd. Work with the Condor [LLM88] group led to the interface specification for a replaceable *resource manager*, a task that filters requests from libpvm functions such as `pvm_spawn()` and `pvm_addhosts()`. The same project defined a replaceable slave pvmd starter (or *hoster*) task that can take over for the built-in `rsh/rexec()` mechanism. In cooperation with the Paradyn [HMC94] group, the *tasker* interface was specified, which allows a task to manage (be the parent process of) other tasks. This is helpful when writing distributed debuggers and similar systems.

6.2.4 Distributed Debuggers

How to debug distributed applications is area of active research. Adding print statements to one's code still qualifies as a state-of-the-art technique with PVM (among others). The built-in ability to start tasks under a debugger works well for small applications, especially those without much distributed state to chase down. But with one debugger window open per task, it quickly loses beyond that. Tracing is helpful in some circumstances, but the level of tracing adequate to search for some bugs can make an application run so slow as to be unusable.

There is a need for a debugger that can manage a large number of processes, spread over many separate hosts, and deal with heteregeneous architectures. One such system is TotalView. [2] To be really useful, the debugger needs to understand concepts used in a message-passing program. For example, it should be able to trigger breakpoints when certain messages are received. Control of the debugger must be distributed to avoid a bottleneck at a central server.

A related topic is *program replay.* Parallel programs often have non-deterministic behavior internally, even though the output produced is the same for the same input. Serial programs can behave non-deterministically too, for example by basing decisions on a time-of-day clock. In message-passing systems such as PVM, one cause of non-determinacy is race conditions between messages. For example, in a program where a master task assigns jobs to some number of workers, the assignments may vary from run to run. Workers may take slightly different times on different runs, and as a result get different assignments. Under these conditions, it can be difficult to reproduce an error on demand, and so be able to debug the program. Program replay allows a run to be reproduced, so it can be studied. The program is run

[2] TotalView is trademark of Bolt Beranek and Newman, Inc.

once, and a log is made of all events that can cause non-determinism (receiving messages). The program is then rerun, and the data in the log file is used to resolve race conditions: Messages are held back by the system until they can be received in the proper order. An experimental version of PVM (based on Version 3) has been instrumented with message replay [Mac93] and can replay PVM programs.

6.3 Availability of the Code

This paper describes the most current version of PVM (3.3, nearing release at this time). The source code and Users' guide are published electronically on the Internet via Netlib (e-mail), Xnetlib (a whizzy interactive file grabber), and *ftp*:

Netlib:	`echo "send index from pvm3" | mail netlib@ORNL.GOV`	
Xnetlib:	*host*:	`netlib.ORNL.GOV`
	package:	`pvm3`
ftp:	*host*:	`netlib2.CS.UTK.EDU`
	directory:	`/pvm3`

BIBLIOGRAPHY

BIBLIOGRAPHY

[ABB+86] M. J. Accetta, R. V. Baron, W. Bolosky, D. B. Golub, R. F. Rashid, A. Tevanian, and M. W. Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of Summer Usenix*, July 1986.

[BDG+91] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. A users' guide to PVM (parallel virtual machine). Technical Report ORNL/TM-11826, Oak Ridge National Laboratory, Oak Ridge, TN, July 1991.

[BDG+93] Adam Beguelin, Jack Dongarra, Al Geist, Robert Manchek, Keith Moore, and Vaidy Sunderam. Tools for heterogeneous network computing. In *Proceedings of The Sixth SIAM Conference on Parallel Processing*, pages 854–861. SIAM, March 1993.

[Beg94] Adam Beguelin, 1994. Personal communication.

[BL92] R. Butler and E. Lusk. User's guide to the p4 programming system. Technical Report ANL–92/17, Argonne National Laboratory, Argonne, IL, 1992.

[BLA+94] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg. A virtual memory mapped network interface for the Shrimp multicomputer. In *Proceedings of The 21st Annual International Symposium on Computer Architecture*, April 1994.

[BM89] Kenneth Birman and Keith Marzullo. ISIS and the meta project. *SunTechnology*, Summer 1989.

[BN91] A. Beguelin and G. Nutt. Examples in Phred. In *Proceedings of Fifth SIAM Conference on Parallel Processing*, Philadelphia, 1991. SIAM.

[Bur89] Gregory D. Burns. A local area multicomputer. In *Proceedings of Fourth Conference on Hypercubes, Concurrent Computers, and Applications*, 1989.

[BZS93] Brian N. Bershad, Matthew J. Zekauskas, and Wayne A. Sawdon. The Midway distributed shared memory system. Technical Report CMU-CS 93-119, Carnegie Mellon University, Pittsburgh, PA, 1993.

[CG89] Nicholas Carriero and David Gelernter. LINDA in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[Che88a] David Cheriton. VMTP: Versatile Message Transaction Protocol. RFC 1045, Stanford University, February 1988.

[Che88b] David R. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, March 1988.

[CHI91] CHIMP concepts. Technical Report EPCC-KTP-CHIMP-CONC 1.2, Edinburgh Parallel Computing Centre, University of Edinburgh, June 1991.

[CS92] Clemens H. Cap and Volker Strumpen. The PARFORM – a high performance platform for parallel computing in a distributed workstation environment. Technical Report 92.07, University of Zurich Information Institute, June 1992.

[DS86] J. J. Dongarra and D. Sorensen. SCHEDULE: Tools for developing and analyzing parallel fortran programs. Technical Report ANL/MCS-TM-86, Argonne National Laboratory, Argonne, IL, November 1986.

[FKB91] J. Flower, A. Kolawa, and S. Bharadwaj. The Express way to distributed processing. *Supercomputing Review*, pages 54–55, May 1991.

[For93] MPI Forum. MPI: A message passing interface. In *Proceedings of Supercomputing '93*, pages 878–885, Los Alamitos, CA, 1993. IEEE Computer Society Press.

[GHPW91] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. A users' guide to PICL: A portable instrumented communication library. Technical Report ORNL/TM-11616, Oak Ridge National Laboratory, Oak Ridge, TN, January 1991.

[GS92] G. A. Geist and V. S. Sunderam. Network based concurrent computing on the PVM system. *Concurrency: Practice and Experience*, 4(4):293–311, June 1992.

[Har91] R. J. Harrison. Portable tools and applications for parallel computers. *Intern. J. Quantum Chem.*, 40:847–863, 1991.

[Har92] Douglas Hartman. Unclogging distributed computing. *IEEE Spectrum*, pages 36–39, May 1992.

[Hea90] M. T. Heath. Visual animation of parallel algorithms for matrix computations. In *Proc. Fifth Distributed Memory Comput. Conf.*, pages 1213–1222, Los Alamitos, CA, 1990. IEEE Computer Soc. Press.

[HMC94] J. Hollingsworth, B. Miller, and J. Cargille. Dynamic program instrumentation for scalable performace tools. In *Proceedings of 1994 Scalable High Performance Computing Conference*, Knoxville, TN, May 1994. to appear.

[Hoa74] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, pages 549–557, October 1974.

[HS93] C. Hartley and V. S. Sunderam. Concurrent programming with shared objects in networked environments. In *Proceedings of 7th Intl. Parallel Processing Symposium*, pages 471–478, Los Angeles, April 1993.

[JBB92] V. Jacobson, R. Braden, and D. Borman. TCP extensions for high performance. RFC 1323, LBL, ISI and Cray Research, May 1992.

[Jep94] Chris M. Jepeway. The design, implementation, and evaluation of RCalc. Master's thesis, The University Of Tennessee, Knoxville, 1994.

[KLS+94] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy K. Steel Jr., and Mary E. Zosel. *The High Performance Fortran Handbook*. MIT Press, Cambridge, MA, 1994.

[Lel90] Wm Leler. Linda meets Unix. *IEEE Computer*, pages 43–54, February 1990.

[LLM88] M. Litzkow, M. Livny, and M. Mutka. Condor — A hunder of idle workstations. In *Proceedings of the Eighth Conference on Distributed Computing Systems*, San Jose, California, June 1988.

[LMKQ89] S. Leffler, M. McKusick, M. Karels, and J. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, Reading, MA, 1989.

[LO83] Ewing A. Lusk and Ross A. Overbeek. Implementation of monitors with macros: A programming aid for the HEP and other parallel processors. Technical Report ANL–83–97, Argonne National Laboratory, Argonne, IL, December 1983.

[LW89] J. Levesque and J. Williamson. *A Guidebook to Fortran on Supercomputers*. Academic Press, San Diego, CA, 1989.

[Mac93] Milon Mackey. Program replay in PVM, Presented at the First PVM user group meeting, May 1993.

[Mul93] Sape Mullender, editor. *Distributed Systems*. ACM Press, New York, 1993.

[MvRT+90] Sape Mullender, Guido van Rossum, Andrew Tanenbaum, Robbert van Renesse, and Hans van Staveren. Amoeba: A distributed operating system for the 1990s. *IEEE Computer*, 23(5):44–53, May 1990.

[OCD+88] John K. Ousterhout, Andrew R. Cherenson, Frederick Douglis, Michael N. Nelson, and Brent B. Welch. The Sprite network operating system. *IEEE Computer*, 21(2):23–36, February 1988.

[Phi93] Christopher Phillips, 1993. Personal communication.

[Pla94] Jim Plank, 1994. Personal communication.

[Pos81a] J. Postel. Transmission control protocol. RFC 793, Information Sciences Institute, September 1981.

[Pos81b] J. Postel. User datagram protocol. RFC 768, Information Sciences Institute, September 1981.

[QCB93] Angela Quealy, Gary L. Cole, and Richard A. Blech. Portable programming on parallel/networked computers using the Application Portable Parallel Library (APPL). Technical Report NASA/TM-106238, NASA Lewis Research Center, Cleveland, OH, July 1993.

[RSL92] Martin C. Rinard, Daniel J. Scales, and Monica S. Lam. Heterogeneous parallel programming in Jade. In *Proceedings of Supercomputing 92*, November 1992.

[RSW90] Matthew Rosing, Robert B. Schnabel, and Robert P. Weaver. The DINO parallel programming language. Technical Report CU-CS-457-90, University of Colorado, Boulder, CO, April 1990.

[Sch91] G. Schoinas. Issues of the implementation of PrOgramming SYstem for distriButed appLications, 1991. Draft Paper.

[Sep93] Douglas J. Sept. The design, implementation and performance of a queue manager for PVM. Master's thesis, The University Of Tennessee, Knoxville, 1993.

[Sun87] Sun Microsystems, Inc. XDR: External Data Representation Standard. RFC 1014, Sun Microsystems, Inc., June 1987.

[Sun88] Sun Microsystems, Inc. RPC: Remote Procedure Call. RFC 1057, Sun Microsystems, Inc., June 1988.

[Sun89] Sun Microsystems, Inc. NFS: Network File System protocol specification. RFC 1094, Sun Microsystems, Inc., March 1989.

[Sun90] V. S. Sunderam. PVM : A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[vECGS92] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: a mechanism for integrated communication and computation. In *Proceedings of The 19st Annual International Symposium on Computer Architecture*, May 1992.

VITA

Robert Manchek was born in Cleveland, Ohio on October 17, 1964. He lived there and other places, like Detroit, before finally settling on Ann Arbor, Michigan, where he graduated from high school in June 1981. He immediately got a job at a local terminal manufacturer, gluing magnets to CRTs, and ended up building neat graphics hardware. Not wanting to stop with being happy and well fed, he bailed via Kalamazoo to Boulder, Colorado to find out what was there. In June 1988 he graduated with a B.S. in Electrical Engineering from the University of Colorado. He is presently employed as a Research Associate at the University of Tennessee, Knoxville and wondering what to do about that.