ANALYSIS OF INTERTASK COMMUNICATION IN PVM COMPUTATIONS ON WORKSTATION CLUSTERS

by

Sarah E. Zabel

THESIS

Presented to the Faculty of

The University of Texas at San Antonio

In Partial Fulfillment of

The Requirements for the Degree of

MASTER OF SCIENCE

April 15, 1996

Acknowledgements

I would like to thank Dr. Rajendra Boppana for the long hours and hard work he dedicated to this effort. I would also like to thank Dr. Bob Hiromoto and Dr. Samir Das for participating in my thesis committee, and for reviewing and providing comments on my thesis.

Abstract

PVM and other message-passing libraries facilitate parallel processing on workstation clusters. However, the intertask communications latency often limits the achievable speedup. This thesis studies the intertask communications latency in PVM computations. The PVM message library is instrumented to generate traces which are in turn used to drive a simulator. This approach provides a flexible method to evaluate the performance improvement which newer high-speed networks could provide to a workstation cluster. This thesis presents the results of this analysis for the NAS parallel benchmarks on workstations interconnected by 10 Mbps and 100 Mbps Ethernets.

Contents

1 Introduction **1**

2 The PVM Message Model **5**

2.1 Overview of PVM Communications 6

2.2 Communications Between PVM Tasks 8

2.2.1 Pvmd to Pvmd Communications 8

2.2.2 Pvmd to Task Communications 10

2.2.3 Task to Pvmd and Task to Task Communications 11

2.3 PVM Group Server 13

2.4 An Example of PVM Communications 13

2.5 Data Sending and Encoding Options 20

2.6 The PVM Trace Facility 22

3 Instrumentation of PVM Message Library **24**

3.1 Trace Generation 25

3.1.1 Instrumentation of Send Function 27

3.1.2 Instrumentation of Receive Function 29

3.1.3 Instrumentation of the Multicast Operation 29

3.1.4 Instrumentation of PVM Communications Overhead 29

3.2 Trace Output 30

3.3 Converting Traces to Simulation Input 30

4 Simulations of PVM Program Executions 35

4.1 CSIM 35

4.2 Ethernet, 10 Mbits/second 37

4.2.1 Send a Message to Another Task 37

4.2.2 Receive a Message From Another Task 39

4.2.3 An Example of Simulation Processing 39

4.3 Ethernet, 100 Mbits per Second 41

4.4 Assumptions and Approximations 41

5 PVM Benchmark Programs 44

5.1 Multigrid 45

5.2 Conjugate Gradient 46

5.3 Integer Sort 46

5.4 Fast Fourier Transform 53

5.5 Lower-Upper Diagonal 53

5.6 Scalar Pentadiagonal 58

5.7 Block Tridiagonal 58

5.8 Test Network Configuration 61

6 Performance Analysis **65**

6.1 Test and Simulation Results 66

6.2 Impact of Network Speed on Communications Time 68

6.2.1 Multigrid (MG) 68

6.2.2 Conjugate Gradient (CG) 70

6.2.3 Integer Sort (IS) 70

6.2.4 Fast Fourier Transform (FT) 71

6.2.5 Lower-Upper Diagonal (LU) 71

6.2.6 Scalar Pentadiagonal (SP) 72

6.2.7 Block Tridiagonal (BT) 73

6.3 Algorithmic Blocking Factors 73

6.4 Summary 76

7 Discussion and Conclusions **78**

A Simulation Results **80**

A.1 Benchmark: mg 80

A.2 Benchmark: cg 81

A.3 Benchmark: is 83

A.4 Benchmark: ft 84

A.5 Benchmark: lu 85

A.6 Benchmark: sp 87

A.7 Benchmark: bt 88

Bibliography **92**

Vita **93**

List of Figures

1.1 Network Simulation Analysis Process 3

2.1 PVM Communications and Communications Support Functions 7

2.2 Message Storage in Pvmd 9

2.3 Relationship of Pvmd Messaging Functions 10

2.4 Pvmd Messaging 11

2.5 Message Storage in libpvm 12

2.6 Direct and Default Message Routing 13

2.7 Libpvm Messaging Functions 14

2.8 Example PVM System 15

2.9 Pseudocode for Ring Application 15

2.10 Sample Application Trace Results 16

2.11 PVM Implementation of Communications Functions 17

2.12 Multi-Packet Message Receipt Scenarios 18

2.13 Pseudocode for mxfer and mxinput Functions 19

2.14 More Complicated Receive Operation 20

2.15 PVM Send and Receive Via TCP 23

3.1 Mxfer and mxinput, Instrumented 26

3.2 Trace Output Formats 28

3.3 Pvm_send and pvm_psend Instrumentation 31

3.4 PVM Overhead Trace Files Format 32

3.5 PVM Communications Functions Instrumented for Overhead 33

3.6 Format of Simulation Input 34

4.1 Simulation Timeline 40

5.1 Aggregate Communications Profile for MG 47

5.2 Dynamic Communication Profile for MG 48

5.3 Aggregate Communications Profile for CG 49

5.4 Dynamic Communication Profile for CG 50

5.5 Aggregate Communications Profile for IS 51

5.6 Dynamic Communication Profile for IS 52

5.7 Aggregate Communications Profile for FT 54

5.8 Dynamic Communication Profile for FT 55

5.9 Aggregate Communications Profile for LU 56

5.10 Dynamic Communication Profile for LU 57

5.11 Aggregate Communications Profile for SP 59

5.12 Dynamic Communication Profile for SP 60

5.13 Aggregate Communications Profile for BT 62

5.14 Dynamic Communication Profile for BT 63

5.15 Network Testbed Computer Configuration 64

6.1 Benchmark Test and Simulation Results 68

6.2 Comparison of Multigrid Simulation Results 69

6.3 Comparison of Conjugate Gradient Simulation Results 70

6.4 Comparison of Integer Sort Simulation Results 71

6.5 Comparison of Fast Fourier Transform Simulation Results 72

6.6 Comparison of Lower-Upper Diagonal Simulation Results 73

6.7 Comparison of Scalar Pentadiagonal Simulation Results 74

6.8 Comparison of Block Tridiagonal Simulation Results 75

6.9 Components of Blocking Time 77

6.10 Benchmark Blocking Times for 10 Mb/sec Ethernet Simulation 77

Chapter 1

Introduction

Scientific and technical research demands a great deal of computer power. In many cases, a single workstation cannot provide enough system resources to conduct a realistic computation, but a researcher's access to supercomputers is limited and such resources are expensive. A much less expensive alternative is the use of networks of less powerful computers, operating in cooperation on a single task.

Parallel Virtual Machine (PVM) provides such an environment [1]. PVM is a system that allows a programmer to treat a heterogeneous collection of computers as one "virtual" machine. PVM programs use the message-passing model to link resources together across a network so that components of a user's task can be processed in cooperation on several machines at once. In this manner, a computation takes advantage of the processing power of several machines rather than one. The performance of this virtual parallel computer on the task can be determined from three factors: the processing power of the machines participating in the computation, the efficiency of the division of tasks among the processors, and speed and efficiency of intertask communication. In this thesis, we address the less-widely studied intertask communication and its impact on the overall execution of PVM programs. Our work here is different from many previous performance studies [16, 17, 18, 19] in that we consider both aspects of intertask communications in our performance analysis: PVM message processing and network travel time.

Network efficiency is usually expressed in terms of speed, bandwidth, and latency. The resulting numbers provide a means for making comparisons among different types of networks, but do not directly translate into a measure of the performance improvement an application would experience on a different type of network. The speeds of local area networks (LANs) used to interconnect workstations have increased by orders of magnitude over the last 20 years; from Ethernet at a nominal 10 Mbps, through FDDI (Fiber Distributed Data Interface) at 100 Mbps, to HiPPI (High- Performance Parallel Interface) at up to 1.6 Gbps. If a user of a workstation cluster suspects that her parallel application would run faster with a higher speed network, there is no mechanism to confirm or deny this opinion based on the application's performance on a low speed network.

The purpose of this project is to provide a tool to help a PVM programmer predict the performance of his PVM application on a workstation cluster connected by a LAN other than the existing one. We do this by collecting traces of PVM applications running on a baseline network. We then run the traces through simulations of different network types so that the user may see what performance improvement, if any, can be expected on that type of network. To test this process, we traced the execution of seven sample applications and simulated their execution first on 10 Mb/sec Ethernet, then on a conceptual 100 Mb/sec network. We selected seven of the eight Numerical Aerodynamic Simulation (NAS) Parallel benchmarks to use in these tests [9]. These benchmarks incorporate a variety of challenging and complex communications profiles, and provide insight into the communication actions of a real environment.

A PVM programmer would use these tools to predict PVM application performance by recompiling his application and linking to our instrumented libpvm library. The resulting executable is then run on a network testbed, generating traces of the application's communications events. The trace files are converted into simulation input and applied to the Ethernet simulations (Figure 1.1).

The rest of this thesis is organized as follows. In Chapter 2, we provide

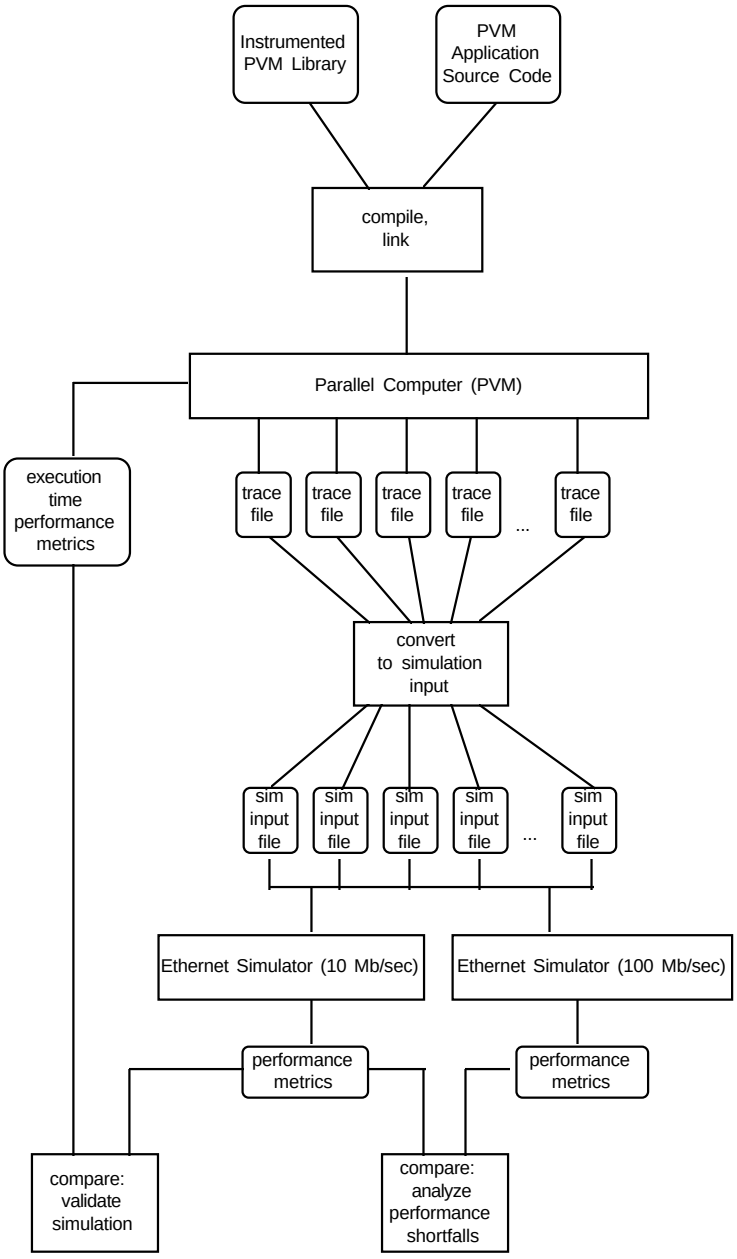


Figure 1.1: Network Simulation Analysis Process

background information on the PVM message model, including a discussion of its data structures and the functions called both internally and by an externally- interfacing user-level process. Chapter 3 describes how we instrumented PVM source code to generate traces of PVM applications. The simulations are presented in Chapter 4. In Chapter 5, we describe the test applications used in this report and the test network configuration. The results of testing, simulation, and analysis are described in Chapter 6. In Chapter 7, we present conclusions and ideas for future work.

Chapter 2

The PVM Message Model

Parallel Virtual Machine (PVM) was developed through a collaborative effort by the Oak Ridge National Laboratory, Emory University, the University of Tennessee, and Carnegie Mellon University as part of their Heterogeneous Network Computing Environment (HeNCE) research project. PVM and its salient features are well described in many articles and books [1, 2, 3]. In this chapter, we describe in detail the aspects of PVM that are used most in our experiments. PVM provides a parallel computing environment to the user through incorporation of a functionally complete message passing model. The PVM system was designed to be run on a heterogeneous network of computers and has been ported to a variety of computer types, such as the DEC Alpha, Sequent Balance, BBN Butterfly, 80386/486 machines running Unix, Thinking Machines CM-2 and -5 series, Crays, Silicon Graphics, and Sun.

PVM is being used for a variety of applications, including computer-aided tomography at Lawrence Livermore Laboratory, seismic migration applications at the Colorado School of Mines, parallel solvers for nonsymmetric partial differential equations at the University of Utah, and Computational Fluid Dynamics problems at NASA Ames Research Center [3].

The parallel machine created by PVM consists of one or more nodes. Each node is a real computer, such as a workstation, Massively Parallel Processor (MPP), or Symmetric Multiprocessor (SMP). Each computer in PVM hosts a PVM daemon,

pvmd, and zero or more PVM tasks. The number of tasks that can be supported by a PVM daemon depends on limits imposed by the operating system of the host. The first PVM daemon is designated the master, and starts slave pvmds on other machines in the network as they are added to the PVM virtual system. Tasks interface with the PVM daemon and other PVM tasks via the libpvm library. Libpvm in C and C++ and libfpvm in Fortran provide functions for packing, unpacking, sending, and receiving messages, as well as service requests for the PVM daemon.

2.1 Overview of PVM Communications

PVM communications mechanisms are described extensively in various books and articles [1]. For the remainder of this chapter, we describe in depth the communications aspects of PVM that are not readily available in the literature but are necessary for this project. Our descriptions are based on study of the PVM source code, experimental results, and bits and pieces of information collected from other sources.

In general, PVM daemons and tasks use a message-passing model in their communications. Figure 2.1 lists communications and supporting functions that are employed by PVM tasks in message exchange. These functions are included in libpvm and libfpvm, which are linked to user-level applications to provide message passing services.

PVM tasks communicate via sockets, using the mechanisms native to their host computers. Messages are of arbitrary length; if a message length is not supported by the communications protocol in use, PVM divides the message into smaller fragments. PVM uses asynchronous send: when sending a message, the sender does not wait for acknowledgment from the receiver, but continues as soon as the message buffer is ready for reuse. A variety of receive operations are supported. The receiver of a message may choose to block or not, or may probe the receive buffer to see if a message has arrived. PVM also provides a receive function with time-out, so a process can be prevented from waiting indefinitely.

Function	Action	Calls
pvm_barrier	Blocks the calling process until all processes in a group have called in.	pvm_send, pvm_recv
pvm_bcast	Broadcasts the data in the active message buffer.	pvm_mcast
pvm_mcast	Multicasts the data in the active message buffer to a set of tasks.	mroute
pvm_nrecv	Checks for non-blocking message with matching label.	mroute
pvm_precv	Receivs a message directly into a buffer.	pvm_recv
pvm_probe	Checks whether message has arrived.	mroute
pvm_psend	Packs and sends data in one call.	pvm_send
pvm_recv	Receives a message.	mroute
pvm_reduce	Performs a reduce operation over members of the specified group.	pvm_send
pvm_send	Sends the data in the active message buffer.	mroute
pvm_sendsig	Sends a signal to another PVM process.	msendrecv
pvm_trecv	Receives with timeout.	mroute
pvm_scatter	Performs a scatter of messages to each member of a group.	pvm_send, pvm_recv
pvm_gather	Performs a gather of messages from each member of a group.	pvm_send, pvm_recv
msendrecv	Internal function to send and receive a message to/from one task.	mroute

Table 2-1a: PVM Communications Functions

Function	Reciprocal Function	Data Type
pvm_pkbyte	pvm_upkbyte	byte
pvm_pkcplx	pvm_upkcplx	complex array
pvm_pkdcplx	pvm_upkdcplx	double precision complex array
pvm_pkdouble	pvm_upkdouble	double
pvm_pkfloat	pvm_upkfloat	float
pvm_pkint	pvm_upkint	integer
pvm_pkuint	pvm_upkuint	unsigned integer
pvm_pklong	pvm_upklong	long integer
pvm_pkulong	pvm_upkulong	unsigned long integer
pvm_pkshort	pvm_upkshort	short integer
pvm_pkushort	pvm_upkushort	unsigned short integer
pvm_pkstr	pvm_upkstr	string
pvm_packf	pvm_upackf	specified by printf()like format string

Table 2-1b: PVM Packing and Unpacking Functions

Figure 2.1: PVM Communications and Communications Support Functions

The process of sending a message involves several steps. The sender must first initialize a send buffer using pvm_initsend or pvm_mkbuf. The message is packed into the buffer using the pvm_pk* functions (see Figure 2.1b). The application then calls pvm_send, pvm_mcast, pvm_bcast, or another send function (see Figure 2.1a). The destination process receives the message through a call to pvm_recv, pvm_nrecv, pvm_trecv, or other receive operation. The receiving task must unpack the data in the same order it was packed, using the corresponding pvm_upk* functions.

Messages can be tagged with a user-specified, integer type designator. The receiver may choose to receive messages only from a particular source task, or with a particular message tag, or both, or may receive all messages regardless of source task or tag.

2.2 Communications Between PVM Tasks

The PVM system as a whole supports two dissimilar sets of communications utilities, one for PVM daemons and another for tasks. Both PVM daemons and tasks manage message buffers but use different messaging utilities in their communications. PVM daemons use the User Datagram Protocol (UDP) [4] to communicate with other PVM daemons, and Transmission Control Protocol (TCP) [5] to communicate with its supported tasks. PVM tasks use TCP to communicate with their daemon and in direct communications with other tasks.

2.2.1 Pvmd to Pvmd Communications

A PVM daemon communicates with PVM daemons on other machines and with tasks on its own machine. Whenever communication with a foreign task (hosted on another machine) is necessary, the PVM daemon send the message to the foreign task's home pvmd, which acts as a repeater. Pvmd to pvmd communications take place using UDP. UDP is an unreliable delivery service, so acknowledgments are required at the pvmd level to ensure the delivery of each message. UDP also limits packet

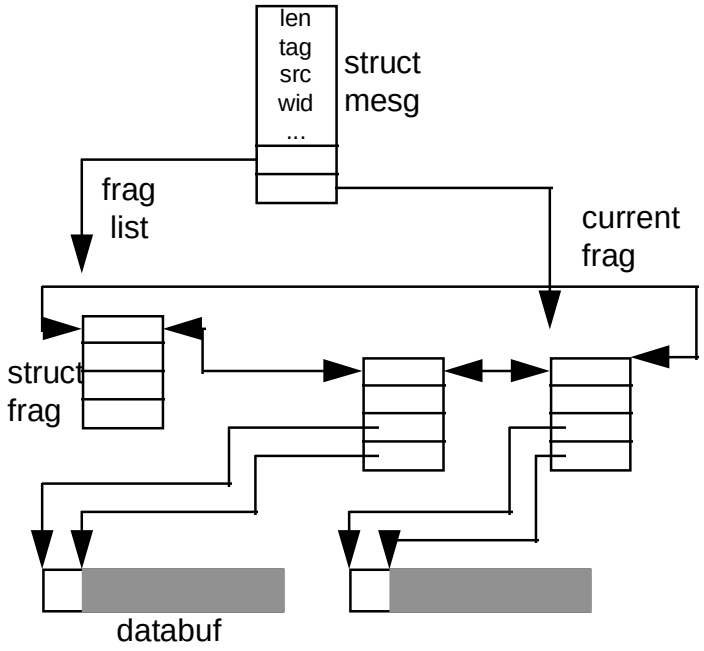


Figure 2.2: Message Storage in Pvmd [1]

length, so messages are fragmented. UDP was chosen over TCP for pvmd to pvmd communications for better scalability and fault tolerance, and lower overhead [1].

A pvmd message structure is shown in Figure 2.2. Each message structure contains the true source and destination task IDs (tids), message length, and other useful values. Pvmd supports a lot of messaging functions, most of which manage queues but do not participate in actually transferring a message over the network. These functions are shown in Figure 2.3 and explained below.

Messages to be sent by the pvmd are added to an outgoing message queue in the sendmessage function. This function is used for messages to local tasks, remote pvmds, and even for the local pvmd. If the message is directed to a local task, sendmessage calls mesg_to_task, which adds the message to the send queue for that task. If the message is directed to the local pvmd, sendmessage calls netentry, which calls the service function associated with the code attached to the message. If the message is directed to a remote pvmd, sendmessage packetizes the message frag-

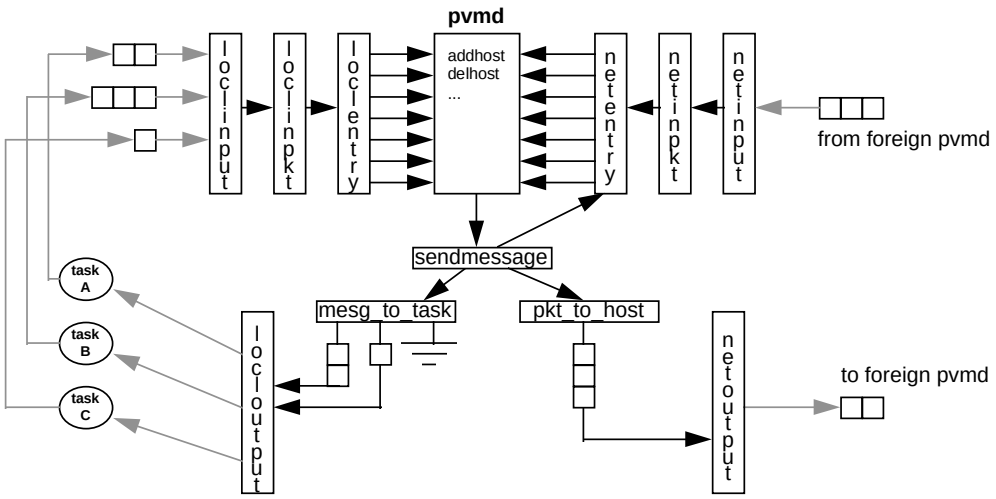


Figure 2.3: Relationship of Pvmd Messaging Functions

ments and calls pkt_to_host, which adds the packets to the send queue for a host.

The PVM daemon executes in a loop named work (Figure 2.4). Each time through work, pvmd calls netoutput to send out any queued messages, and netinput to receive any messages waiting on its socket. Netinput uses the UDP function recvfrom to read packets from its socket. If the packet is received without error, netinput places an acknowledgment on the outgoing message queue and passes the fragments to netinpkt. Netinpkt restores the fragments into a message, and calls netentry. Also each time through the work loop, the pvmd calls netoutput to send any queued messages. Netoutput uses the UDP function sendto to send the message packets on to the network. Each packet contains a sequence number, and messages are retained and retransmitted if not acknowledged within a certain time duration.

2.2.2 Pvmd to Task Communications

The PVM daemon handles messages to and from local tasks in a similar manner. Each time through work, the pvmd executes loclinput to see if there are messages from local tasks pending. Loclinput accepts packets from the local task's TCP con-

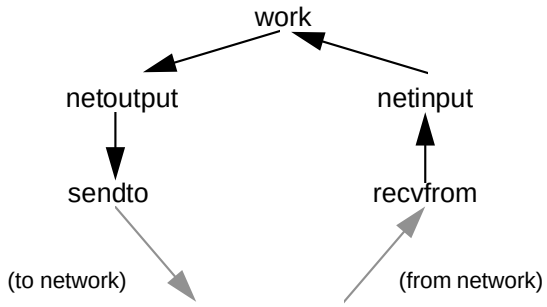


Figure 2.4: Pvmd Messaging

nection to its pvmd and passes it to loclinpkt. If the message is for the local pvmd, it is restructured into a complete message and passed to schentry or loclentry. If it is for a local task, the packets are added to the send queue for that task. If the message is for a remote task, pkt_to_host is called to attach the message to the send queue for that task. If the message is for a remote pvmd, the packets are reassembled into a message using sendmessage. Loclentry and schentry, like netentry, call the service function associated with the received message type. Locloutput is called each time through work to pass on messages received locally to other local tasks. Locloutput calls write directly for its TCP service.

2.2.3 Task to Pvmd and Task to Task Communications

PVM tasks (applications using libpvm functions) use TCP in their communications, regardless of whether those communications are directed to another task or their pvmd. Message structures in libpvm are similar to those used by pvmd, except that libpvm uses a message ID (mid) to index messages in the message heap (Figure 2.5). PVM preferentially passes pointers to messages rather than the messages themselves to reduce inter-task communications. PVM tasks may send messages through a default route, or by directly routing the message through TCP to the recipient. In the default route, the message is passed by the sending task to its local pvmd using a TCP protocol. The local pvmd passes the message to the pvmd on the receiver's host

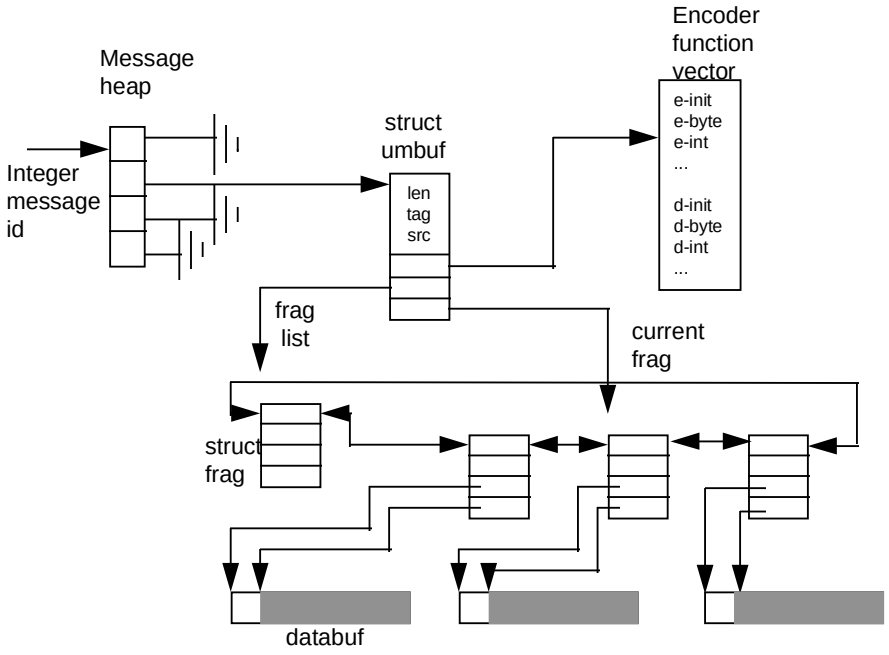


Figure 2.5: Message Storage in libpvm [1]

using UDP. The message is then sent via TCP from the receiver's pvmd to the receiving task. In direct routing, the sending task sends to the receiver through a TCP connection (Figure 2.6). The default route is used to negotiate the connection the first time a direct route is established between two tasks.

PVM tasks use the same functions for both the direct and default routes (Figure 2.7). The heart of the communications function is mroute, which passes communications requests to mxfer. If a direct route is needed, mroute will send a short message to the destination task requesting the direct route, and wait for a response. To send a message, mxfer packetizes the message and makes multiple calls to the Unix function write [6] to send out the packets. To receive a message, mxfer calls mxinput, which in turn calls the Unix function read [6] to accept incoming packets. TCP is a reliable protocol; it guarantees all packets are received and in the correct order.

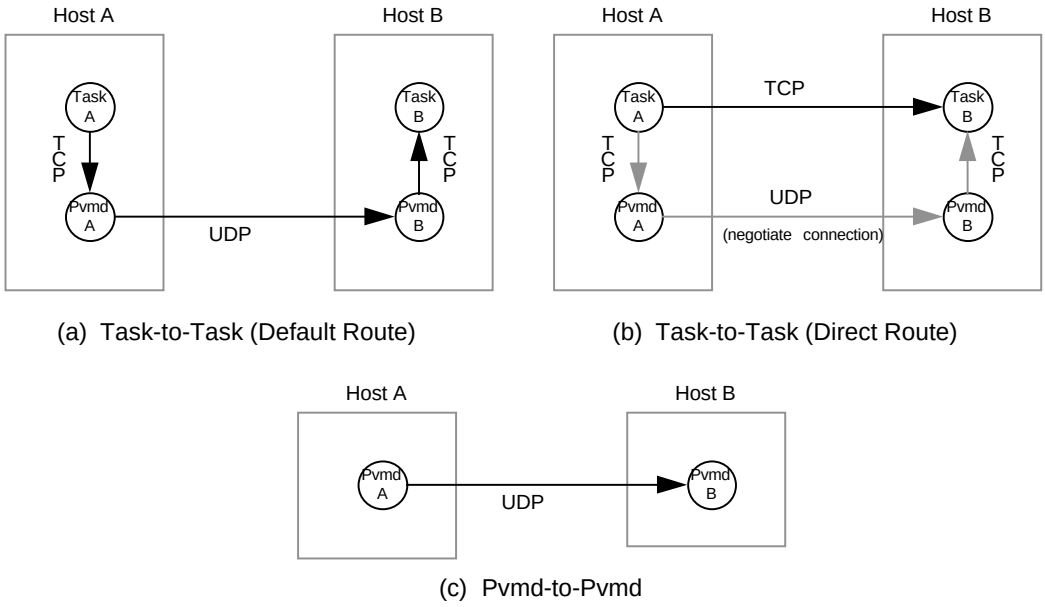


Figure 2.6: Direct and Default Message Routing

2.3 PVM Group Server

PVM supports a limited set of collective communications primitives using its group functions. The PVM group functions are enabled through the cooperation of a group server, named ”pvmgs”. The group server is started automatically when a PVM task requests to join a group. Groups are formed dynamically, and tasks may be members of several groups. Each task in a group has a unique instance number. The group functions supported by PVM include pvm_barrier, pvm_bcast, pvm_scatter, pvm_gather, and pvm_reduce.

2.4 An Example of PVM Communications

We illustrate the handling of communications in PVM using a simple example in which a token is passed around a ring of processes. Figure 2.8 shows a four-machine PVM configuration. Each machine hosts one PVM daemon. The ”ring” application is

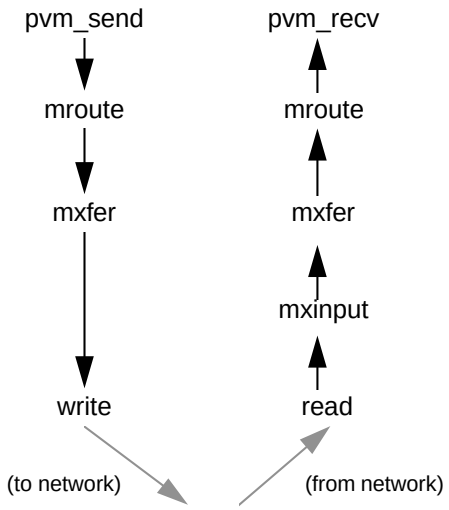
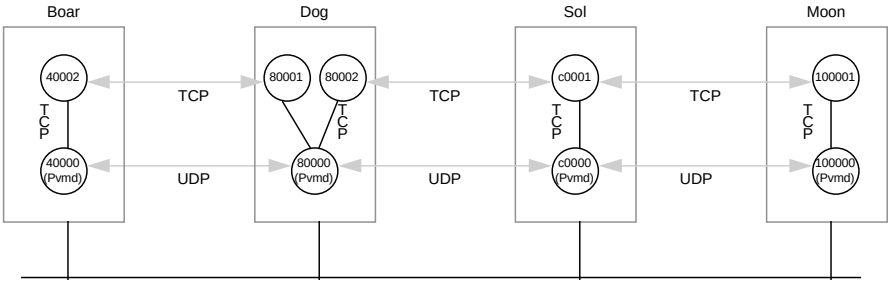


Figure 2.7: Libpvm Messaging Functions

started on one machine, and spawns copies of itself on three others. Each ring task uses pvm_joingroup to join the group named ”foo”. A PVM group server, ”pvmgs”, is started when the first ring task requests to join the group. Each ring task uses its number in the group to determine its neighbors. The first ring task creates a token of 4032 bytes and sends it to one neighbor. Each subsequent ring task waits for the token using pvm_recv, and passes it to a neighbor, using pvm_send, until the token has been passed the specified number of times. The activities of the four PVM tasks during the execution are shown in Figure 2.10. In Figure 2.8, processes with the PVM task ids 40000, 80000, c0000, and 100000 are PVM daemons, processes 40002, 80002, c0001, and 100001 are instances of ”ring”, and process 80001 is the group server. Pseudocode for the ring application is listed in Figure 2.9.

Figure 2.11 shows how the internal functions called by PVM to send or receive data actually interact. In this scenario, process 40002 sends a message to process 80002 on another machine, and process 80002 returns a message. In each case, the sender calls pvm_send, which calls the internal function mroute, which in turn calls mxfer. Mxfer queries the socket’s availability using the Unix function select [6]. When the socket is able to receive data, the process writes the message to



a) System configuration

conf
4 hosts, 1 data format

HOST	DTID	ARCH	SPEED
boar	40000	SUN4SOL2	1000
dog	80000	SUN4SOL2	1000
sol	c0000	SUN4SOL2	1000
moon	100000	SUN4SOL2	1000

b) PVM configuration reported by conf

Figure 2.8: Example PVM System

```
ring()
        {
        join group
        if I am the first ring process
                spawn more copies of myself
        wait at barrier for other processes
        +-----------------barrier-----------------+
        determine my neighbors in the ring
        initialize send buffer
        pack token into message buffer
        if I am the first ring process
                send token
                receive token
                send token
                receive token
        otherwise
                receive token
                send token
                receive token
                send token
        leave group
        exit
        }
```

Figure 2.9: Pseudocode for Ring Application

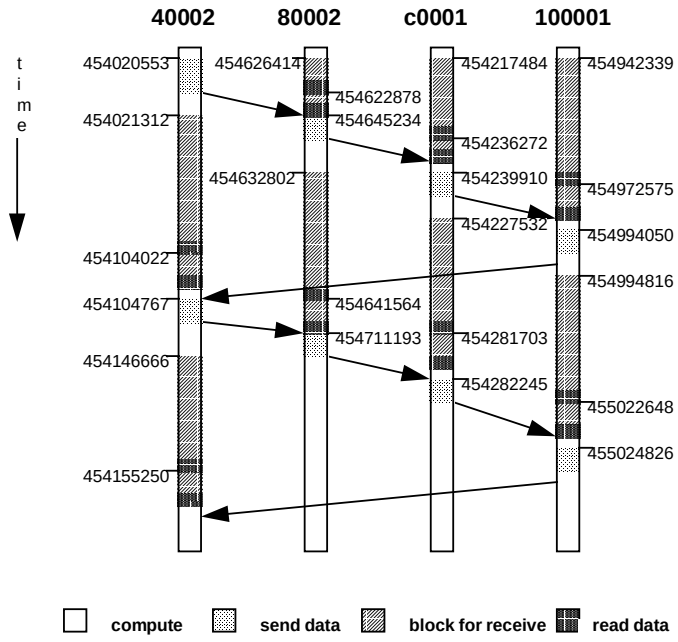


Figure 2.10: Sample Application Trace Results

the socket and returns. PVM function calls necessary to prepare the message and message buffer (e.g. pvm_initsend and pvm_pkint) are not pictured. These activities, along with the rest of the computation the machine is conducting, can be considered to proceed in the shaded areas of the timeline. The receive operations proceed in a similar manner, with Unix read retrieving the message in pieces when the socket reports the message has arrived.

In running these experiments, we observed a few implementation details of PVM communications that deserve note. When receiving data, mxfer always receives each message in at least two parts, as illustrated by the select-read-select-read sequence in Figure 2.11. The first action reads 16 bytes from the receiving socket; the second and subsequent actions read the remainder of the data as it becomes available in the TCP buffer. Because Ethernet has a maximum data packet size of 1500 bytes, TCP processing will break messages down to packets on the sending end, add its own header, and reconstruct the message on the receiving end. Thus, a PVM mes-

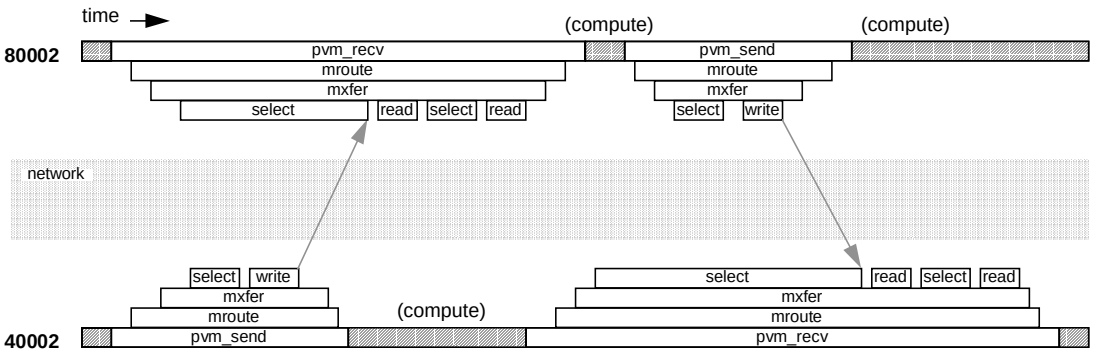


Figure 2.11: PVM Implementation of Communications Functions

sage of greater than 1460 bytes will actually flow through the network as multiple packets of up to 1500 bytes (1460 data bytes plus 20 TCP header plus 20 IP header bytes). We also observed that as the network and the TCP handling process become idle or busy, the PVM task may be able to collect the data from the TCP buffer all at once, or packet by packet, or by groups of packets. This means that a single write of 4032 bytes by the sender may result in two to four read actions by the receiver, as illustrated in Figure 2.12.

The specific pattern of the receive operation seen in the trace output appeared to depend primarily on the speed of the computer. For the slower computers, such as the Sun Sparcstation ELCs we used initially to instrument and test the trace generation, almost all receive operations resembled case A. For the Sun Sparc 5 computers used in the testbed, many receive traces resembled case C. On the send side, the number of write actions executed depends on the buffer capacity of the host machine.

The mxfer function handles this varying number of reads and writes necessary by executing in a loop each time called (Figure 2.13). In this loop, mxfer will check all active file descriptors, reading any data waiting to be received and writing to the appropriate socket any data waiting to be sent.

The loop enables mxfer to send and receive any number of packets in one

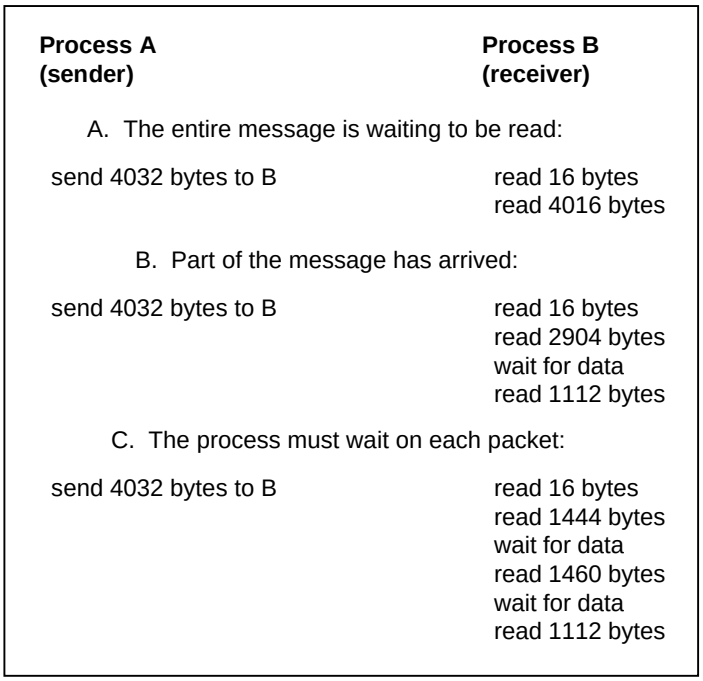


Figure 2.12: Multi-Packet Message Receipt Scenarios

function call, but also results in a confusing variety of actions. Because mxfer will always look to see if there is any data to be received, a call to pvm_send sometimes results in receives as well as sends, and a call to pvm_recv sometimes results in a send as well as one or more receives. This latter case can be observed when PVM tasks negotiate a direct connection. In that instance, the PVM task requesting the direct connection sends a message to the destination PVM task via the PVM daemons using the default route. Upon receiving the request, the destination task generates a response and sends it back to the requester in the same mxfer call that resulted in receiving the request. Such a scenario is depicted in Figure 2.14. Here both 80001 and c0001 send messages that arrive at 40002 almost simultaneously. In this example, the message from c0001 caused an automatic response from 40002 back to that process.

All network communications on this network configuration originate within a PVM task, and go through the TCP/IP protocol stack before using the network. We

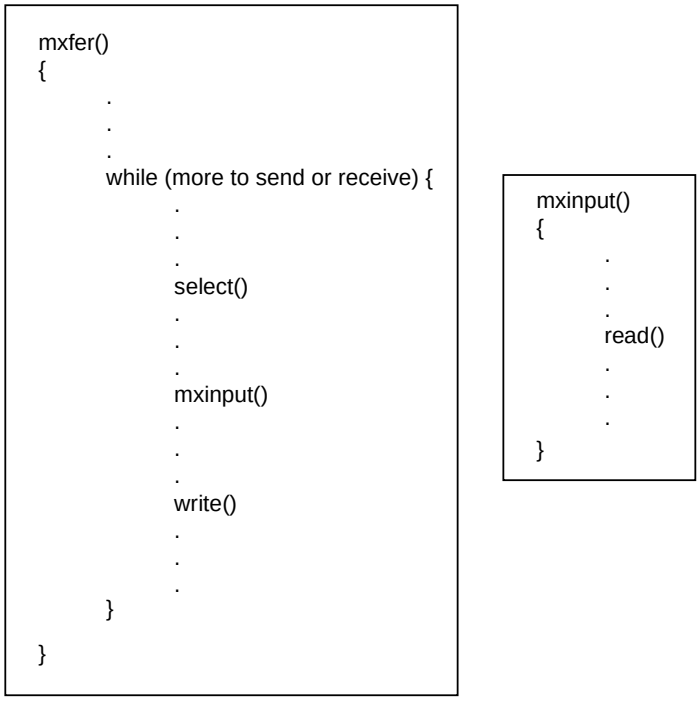


Figure 2.13: Pseudocode for mxfer and mxinput Functions

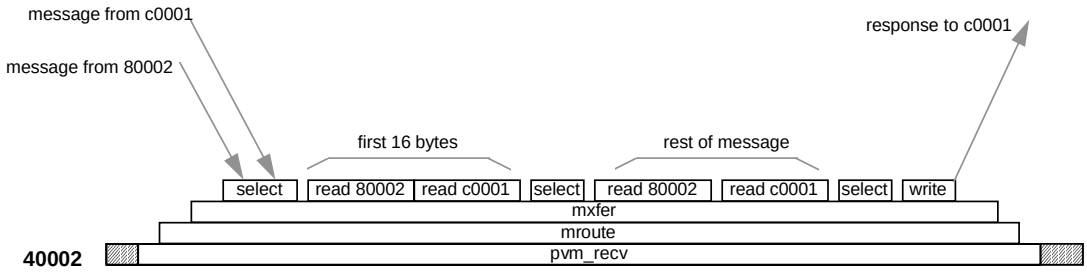


Figure 2.14: More Complicated Receive Operation

assume that this TCP processing is executed in by a kernel-level process on the same machine, and that at most one instance of TCP execution is active at any given time. Based on our observations, we modeled the TCP processing such that for writes of more than 1460 bytes, subpackets of 1460 bytes are created on-the-fly and sent through the network. This means that packets actually enter the network while the write operation is still going on, as shown in Figure 2.15.

2.5 Data Sending and Encoding Options

To facilitate communications between different types of machines, PVM provides an option to encode data in the message buffer in a machine-independent format (Sun's External Data Representation, or XDR). Alternatively, if the destination machine is the same type as the sending machine, the application can send data in its native format, using "PvmDataRaw". If "PvmDataInPlace" is selected, the message buffer contains sizes and pointers to the items to be sent. Then when pvm_send is called, those data items are copied directly out of the computer's memory. A PVM task specifies one of these encoding options when it initializes the send buffer with pvm_initsend.

To send a message, a PVM task must pack data into a buffer and call the function pvm_send. PVM offers a short-cut with the pvm_psend and pvm_precv functions. The pvm_psend function takes a destination task id, the data, and an indicator of the data type, packs the message and sends it in one operation. The re-

ciprocal operation, pvm_precv, receives and unpacks the data using one function call. Pvm_psend packs data into the send buffer by counting the number of bytes to be packed and calling pvm_pkbyte. As indicated in Figure 2.1a, pvm_psend and pvm_precv call pvm_send and pvm_recv respectively to actually send or receive the message.

The particular encoding option and send/receive scheme selected affect how the computer treats the message, both in PVM buffering and writing to the socket. While reviewing traces of various PVM applications, we observed that when pvm_send is used with either PvmDataDefault or PvmDataRaw encoding, the computer will write messages longer than 4080 bytes to the socket 4080 bytes at a time. If pvm_psend or PvmDataInPlace is selected, then long messages are written to the socket without being subdivided.

Casanova et al explain how PVM handles messages on MPPs for each of the possible encodings options and send/receive schemes [7]. When pvm_send is used with default encoding, the packing functions translate the data into XDR format while copying it to a send buffer in PVM space. If PvmDataRaw is selected as the encoding option, PVM copies the data in its native format rather than translating to XDR. When PvmDataInPlace is selected, the system merely keeps a pointer to the data in its current location. When the time comes to send the data, PVM first sends the 32-byte header so that the destination machine can reserve enough buffer space to hold the arriving data. Casanova et al report that when pvm_psend is used, the data is neither translated nor copied to PVM space, but sent directly to the next system. Through examination of our traces, we noticed that pvm_psend calls pvm_pkbyte, which suggests that the data is actually copied to another buffer. We also noticed that using pvm_psend also causes the sender to send the 32-byte header separately from the rest of the message, which suggests that in this case as well, the system must prepare the destination machine for the arriving data. These differences might result from the differences of the MPP and the SUN4SOL2 implementations of PVM.

2.6 The PVM Trace Facility

The PVM system offers a tracing capability for all libpvm functions [8]. Each libpvm function comprising the application interface begins and ends with code that gathers relevant information, packages it into a message, and sends it to the task designated to receive trace events. A user-controlled trace mask determines which events are monitored. Trace events are sent by calling mroute directly, so they do not contribute to other traceable events when pvm_send is being monitored.

The standard PVM trace facility is useful primarily for debugging and determining obvious performance bottlenecks in user programs. Because it provides only coarse-grained timing and performance information, it is not suitable for detailed analysis of computation and communication aspects of user programs and provides little insight into the messaging overhead and latencies of the network used to interconnect the machines that are part of the PVM environment.

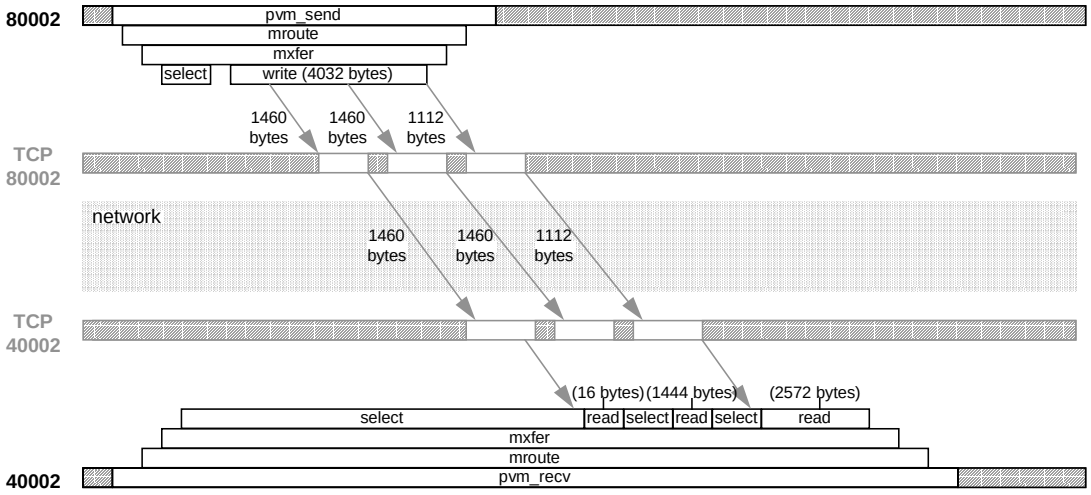


Figure 2.15: PVM Send and Receive Via TCP

Chapter 3

Instrumentation of PVM Message Library

In this chapter, we discuss how we instrumented PVM and convert the trace files to simulator input. Though we investigated other trace generation systems as a precursor to this effort, none of those systems were suitable for this project. The other trace generation environments we researched were NASA's Automated Instrumentation and Monitoring System (AIMS) [11], the University of Wisconsin at Madison's Paradyn [12], and the native PVM trace facility [8]. Each of these instrumentation environments was oriented towards locating performance bottlenecks in the computation. Though each environment gives enough information for the analyst to identify the communications processes as the source of a bottleneck, none of them produced detailed information on the underlying actions involved in the supporting communications. Therefore, we generated our own traces and used them in subsequent simulations and analyses.

We instrumented PVM communications and communications support functions in libpvm to produce trace output of executing applications. These traces report the time applications spend sending data, receiving or waiting to receive data, and executing other tasks required by PVM to support the communications activity, such as initializing, packing, and unpacking message buffers. As this tracing activity adds its own overhead to the communications process, pre-test runs were also

instrumented to report the time consumed by the tracing activity itself. After post-processing, the traces report time spent on the network, blocking time, and time spent in PVM communications and communications support activities.

By confining our instrumentation to libpvm, we created an environment where any PVM application can be instrumented and used in similar simulations and analyses without modification to the PVM application itself. Even applications written in Fortran make use of the same libpvm library. A PVM application must be recompiled in order to link it to our instrumented library, however, and there is also no way to attach this instrumented library to a process that is already in execution.

3.1 Trace Generation

In instrumenting PVM, we used the philosophy that all calls to communications functions should be recorded as close to the function call as possible. The actual communications activities take place in two functions, mxfer and mxinput, in the source code file lpvm.c. The general flow of mxfer and mxinput are shown in Figure 2.12. In our instrumentation, we bracketed the calls to select, read, and write with code to record the time durations of those operations and other data pertinent to the communications event, as shown in Figure 3.1. As indicated in Figure 2.1a, PVM communications functions either call mroute directly or call pvm_send, pvm_recv, or pvm_mcast, which in turn call mroute. Mroute, in turn, passes the communications request to mxfer.

For all traces, times are measured using clock_gettime, which reports the number of seconds and nanoseconds from Jan 1, 1980. Clock_gettime is a POSIX 4 function and has an accuracy of 1 μsec. [6]

There are four functions in the simulation: compute, send, receive, and multicast. Broadcasts are treated as multicasts. Time listed in each compute line reflects the time between send and receive functions, and is simulated by holding for that number of microseconds. Compute time is determined by subtracting the

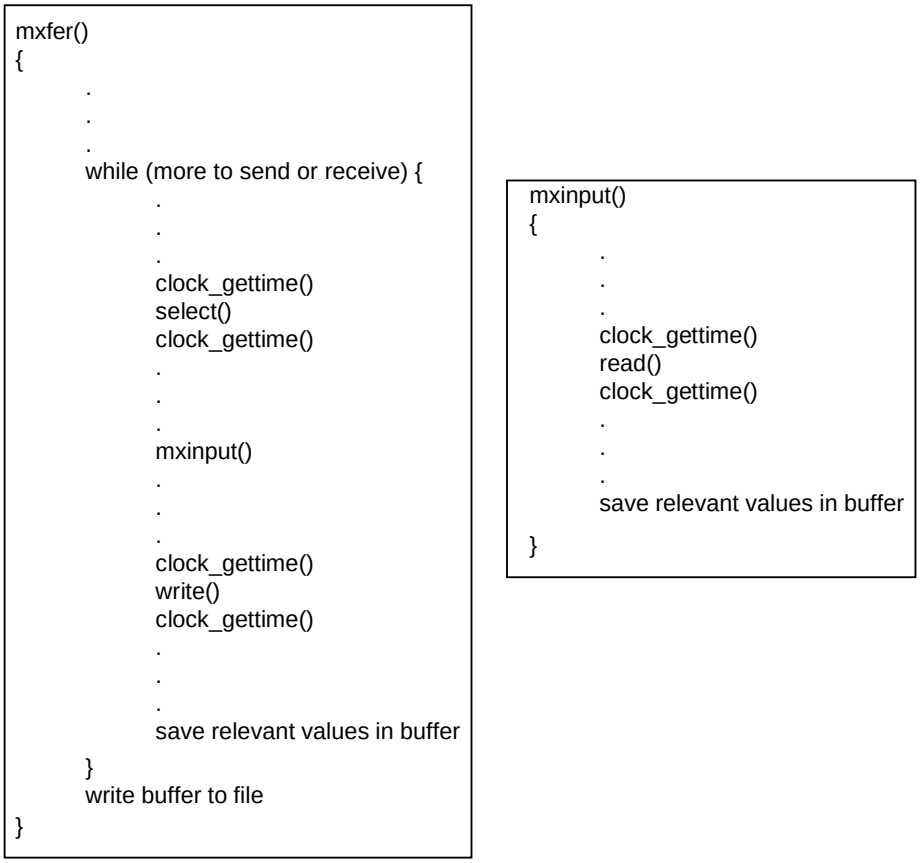


Figure 3.1: Mxfer and mxinput, Instrumented

previous action start time plus the previous action duration from the current action start time.

Send lines list the duration of the send activity, the number of bytes written to the socket, and the source and destination processor numbers. Source and destination processor numbers are assigned as a one-up number for each PVM task id in the traced system. Though communications with the PVM daemons are reflected in the traces, they are not included in the simulation input. This is because task to pvmd communications take place on the same machine, and so are not subject to change based on a changed network.

The multicast lines (those that start with ”m”) in the simulation input file are similar to send lines, except that they may have more than one destination.

Receive lines in the trace files contain the blocked time and blocking start time. This information is cut out of the simulation input as the simulation will determine how long a receiving process blocks. The blocked time in the traces from the select operation is useful in determining how much time is spent computing both before and after a receive, however. Also, to validate the simulation, these blocked times were added up and compared to simulated blocked time on several application runs.

3.1.1 Instrumentation of Send Function

Figure 2.11 shows the functions that underlie a call to pvm_send. Instrumentation was placed at the lowest level possible so that the traces generated would include very little other than the actual network communications time. For pvm_send, instrumentation is contained in mxfer, bracketing the write statement. The resulting trace lines report the start time and duration of the write function, and the number of bytes actually written to the socket (Figure 3.2). PVM adds a 32 byte header to each message sent.

Send Line:

s dur 239 start 845900277 len 1032 src 4000a dst 80008
a b c d e f

a - send flag
b - time required by process to write the data to the socket, in microseconds
c - send start time, in microseconds
d - number of bytes written to socket
e - source PVM task id
f - destination PVM task id

Receive Line:

r blk 6039 start 845913300 len 1016 src 4000a dst 80008 read 134
a b c d e f g

a - receive flag
b - time block for receive, in microseconds
c - start time for operation, in microseconds
d - number of bytes read from socket
e - source PVM task id
f - destination PVM task id
g - time required to read last data packet, in microseconds

Multicast Line:

m dur 322 start 845918992 len 1032 src 40012 dst 80008 100004 140003
a b c d e f

a - multicast flag
b - time required by process to write the data to the socket, in microseconds
c - multicast start time, in microseconds
d - number of bytes written to socket
e - source PVM task id
f - destination PVM task ids, separated by blanks

Figure 3.2: Trace Output Formats

3.1.2 Instrumentation of Receive Function

Figure 2.11 also shows the functions involved in a pvm_recv operation. Again, instrumentation was inserted at the lowest possible level so that actual network blocking time could be measured as accurately as possible. Pvm_recv makes use of the instrumentation bracketing the select function as well as that bracketing read. As called by pvm_recv, select blocks until data is available. The resulting traces determine the blocking time from this value, and report the duration of the subsequent read from the instrumentation bracketing that function call in mxinput (Figure 3.2). When called from pvm_trecv (receive with time-out) or pvm_nrecv (non-blocking receive), mxfer passes a time value to select. Select will block until the time has expired or a message has been received.

3.1.3 Instrumentation of the Multicast Operation

The pvm_mcast operation is instrumented in much the same way as pvm_send, with start and elapsed send times determined from bracketing the write function in mxfer. Unlike pvm_send, though, pvm_mcast may have as destination several PVM tasks. The multicast address used in the actual pvm_mcast operation does not correspond to any of the destination task ids, however, so this information must be made available from the pvm_mcast call in the source code file lpvmgen.c. This is accomplished by setting an externally accessible pointer to the list of tasks referred to in a multicast address. Instrumentation in mxfer includes this list in the trace output as destination task ids.

3.1.4 Instrumentation of PVM Communications Overhead

As mentioned above, traces that will be used in simulating network activities are measured as close as possible around the lowest level communications: socket read, write, and select. As PVM uses computer resources to set up these communications, the time required by PVM for communications support must also be mea-

sured. These functions include pvm_initsend, pvm_pkint, pvm_upkint, and others. The traces generated by these functions overlap in time with those generated to simulate communications, and can overlap with each other. An example of the latter is pvm_psend. This function packs data into the message buffer and sends it in one operation. Overhead traces for a call to pvm_psend therefore include one or more calls to a pvm_pk* function and one call to pvm_send. The overlapping portions of time are removed in the overhead accounting portion of the simulation (Figure 3.3) so as not to penalize PVM unfairly.

3.2 Trace Output

All trace output is written to files, with the output for each PVM task directed to a different file. Trace files generated fall into two general categories. Traces that report communications time and the type of communications operation (send, receive, or multicast) are written to files named results.X where X is the task id in hexadecimal.

Files that record the output of PVM communications overhead traces are named oh.X, oh2.X, pk.X, or pk2.X, depending on the PVM source code file where the traced function resides. We instrumented 39 PVM communications and communications support functions to report overhead; they are listed in Figure 3.5. We selected these PVM functions to trace because they were necessary for the actual communications activity. These overhead trace files have the format shown in Figure 3.4.

3.3 Converting Traces to Simulation Input

Because of the differences between the instrumentation output and the simulation input formats, some preprocessing is necessary before trace files can be read into the simulations. For example, PVM task ids must be converted into an integer counting up from zero so that each task can index data structures and communicate with

Function	Start time (usec)	Duration (usec)
.		
.		
.		
pvm_initsend	242643527	172
pvm_pkdouble	242643799	15
pvm_send	242643886	1731
pvm_recv	242645718	14068
pvm_upkdouble	242659882	15
pvm_initsend	242705149	178
pvm_pkdouble	242705436	1165
pvm_send	242706697	3899
.		
.		
.		

a) Sample output from communications event where pvm_send is used. Start times plus duration do not overlap.

Function	Start time (usec)	Duration (usec)
.		
.		
.		
pvm_psend	795294710	3087
pvm_pkbyte	795294798	28
pvm_send	795294918	2770
pvm_precv	795297905	23496
pvm_recv	795297915	23276
pvm_upkbyte	795321282	14
pvm_psend	795381544	3983
pvm_pkbyte	795381651	23
pvm_send	795381771	3642
.		
.		
.		

b) Sample output from communications events where pvm_psend is used. Pvm_pkbyte and pvm_send calls occur within pvm_psend start time plus duration. Durations of pvm_pkbyte and pvm_send are subtracted from duration of pvm_psend in post-processing.

Figure 3.3: Pvm_send and pvm_psend Instrumentation

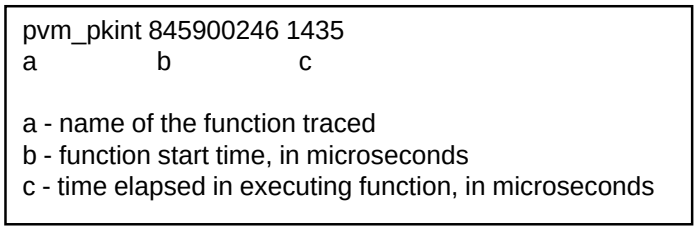


Figure 3.4: PVM Overhead Trace Files Format

other tasks. Preprocessing converts files structured as shown in Figure 3.2 to that shown in Figure 3.6.

The process of converting trace files to simulation input involves generating send, receive, multicast, and compute lines. The duration of the compute line is calculated from the time difference between the end of a send or receive operation and the start of the next send or receive operation. The time a process is blocked for each receive operation is reported in the traces and removed from the corresponding receive line in the simulation input. The simulation determines the blocking time associated with each receive operation. Each receive line in the simulation input contains the time that was used to read the data once it had arrived, as reported in the traces. The time value associated with the receive operation in the simulation input is the time used by the process to read the data when it finally arrived, taken from the traces. The blocking time associated with each receive operation is computed by the simulation. Send and multicast lines in the simulation input list the amount of time the process spent in Unix write for that operation, as reported in the traces. Accordingly, all time values used in the simulation, except receive blocking time, are taken directly from the traces.

PVM communications support overhead traces also undergo processing before being input into the simulation. In this case, trace file post-processing combines all overhead trace files from each PVM task into one file per PVM task, and sorts that file based on start times. In the simulation, the start time and the duration of

Function	PVM Source Code File	Trace Output File
pvm_barrier	pvm_gsulib.c	pk2.xxxxx
pvm_bcast	pvm_gsulib.c	pk2.xxxxx
pvm_mcast	lpvmgen.c	oh.xxxxx
pvm_nrecv	lpvmgen.c	oh.xxxxx
pvm_precv	lpvm.c	oh2.xxxxx
pvm_psend	lpvm.c	oh2.xxxxx
pvm_recv	lpvmgen.c	oh.xxxxx
pvm_reduce	pvm_gsulib.c	pk2.xxxxx
pvm_send	lpvmgen.c	oh.xxxxx
pvm_trecv	lpvmgen.c	oh.xxxxx
pvm_scatter	pvm_gsulib.c	pk2.xxxxx
pvm_gather	pvm_gsulib.c	pk2.xxxxx
pvm_pkbyte	lpvmpack.c	pk.xxxxx
pvm_pkcplx	lpvmpack.c	pk.xxxxx
pvm_pkdcplx	lpvmpack.c	pk.xxxxx
pvm_pkdouble	lpvmpack.c	pk.xxxxx
pvm_pkfloat	lpvmpack.c	pk.xxxxx
pvm_pkint	lpvmpack.c	pk.xxxxx
pvm_pkuint	lpvmpack.c	pk.xxxxx
pvm_pklong	lpvmpack.c	pk.xxxxx
pvm_pkulong	lpvmpack.c	pk.xxxxx
pvm_pkshort	lpvmpack.c	pk.xxxxx
pvm_pkushort	lpvmpack.c	pk.xxxxx
pvm_pkstr	lpvmpack.c	pk.xxxxx
pvm_vpackf	lpvmpack.c	pk.xxxxx
pvm_upkbyte	lpvmpack.c	pk.xxxxx
pvm_upkcplx	lpvmpack.c	pk.xxxxx
pvm_upkdcplx	lpvmpack.c	pk.xxxxx
pvm_upkdouble	lpvmpack.c	pk.xxxxx
pvm_upkfloat	lpvmpack.c	pk.xxxxx
pvm_upkint	lpvmpack.c	pk.xxxxx
pvm_upkuint	lpvmpack.c	pk.xxxxx
pvm_upklong	lpvmpack.c	pk.xxxxx
pvm_upkulong	lpvmpack.c	pk.xxxxx
pvm_upkshort	lpvmpack.c	pk.xxxxx
pvm_upkushort	lpvmpack.c	pk.xxxxx
pvm_upkstr	lpvmpack.c	pk.xxxxx
pvm_vupackf	lpvmpack.c	pk.xxxxx
pvm_initsend	lpvmgen.c	oh.xxxxx

Figure 3.5: PVM Communications Functions Instrumented for Overhead

a	b	c	d	e		
c	3356	0	0	0		
s	234	48	0	2		
c	499	0	0	0		
m	184	58	0	1	2	3
r	188	16	2	0		
c	64	0	0	0		
r	142	32	2	0		

a - function (c=compute, s=send, r=receive, m=multicast)
b - time, in microseconds, associated with the function
c - number of bytes of data
d - source processor
e - destination processor(s)

Figure 3.6: Format of Simulation Input

each PVM communications support function will be used to determine where one such function call is contained within another, and this overlap removed from the enveloping function call. For example, in Figure 3.3b, quantities of 28 and 2770 μsec (durations of pvm_pkbyte and pvm_send respectively) are subtracted from the duration of pvm_psend, leaving 289 μsec as the duration of that operation.

Chapter 4

Simulations of PVM Program Executions

In this chapter, we describe the Ethernet simulators we created to facilitate analysis of the PVM communications traces. The purpose of using the traces in simulations is to provide insight into the actions, dependencies, and constraints of the operational PVM environment. By using traces of the actual execution of a PVM application to drive the simulations, we hope to reconstruct the communications events of the application execution as closely as possible. The benefit of the simulation is that aspects of the communications process that could not be directly observed during the real application execution, such as the number of collisions on the network, are open to review and analysis. Also, such a simulation allows the analyst to predict the impact of changing some variable parameters on many aspects of the application execution. In our case, we changed the network speed from 10 Mb/sec to 100 Mb/sec to evaluate the effect of that change on the benchmark execution as a whole.

4.1 CSIM

We used CSIM, a process-oriented, discrete-event simulation package, to create these simulations [13]. This package is a library of routines that can be used with C or C++ programs. A CSIM program models a system as a collection of CSIM processes.

These processes interact with each other through other CSIM constructs, such as mailboxes, the setting and clearing of events, facilities, and storage blocks. CSIM maintains a clock holding the simulation time, and advances this clock under the control of the simulation.

In simulating a parallel system, each node of the system is modeled as a CSIM process. These processes communicate with each other and synchronize the simulation by accessing global and private variables and other CSIM constructs. In addition to CSIM processes, we used CSIM facilities, events, and mailboxes to pass information between processes. The interactions of the processes through these resources controls the execution of the simulation.

A CSIM facility is a process that controls a resource or set of resources. The resource or service is granted to requesting processes according to a service discipline. In our simulations, we used a simple first-come-first-served service discipline on a single resource per facility, essentially modeling an M/M/1 queue. A process requests to use a facility by calling reserve(facility_name). After the calling process has used the facility's resource, it exits the resource by calling release. In our model, the Ethernet channel and each station's TCP server were modeled as facilities.

CSIM uses events to control and synchronize interactions between processes. A process waits for an event to occur though the wait command, sets an event through a call to set, and clears the event though a call to clear. Though CSIM supports non-blocking and timed event waiting protocols, we used blocking waits exclusively on events in these simulations.

The mailbox construct allows processes to communicate with more information than the simple on/off state of an event. Processes use mailboxes to send and receive integers or pointers, using the CSIM function calls send and receive. Though the default action is to block on receive if the mailbox is empty, CSIM also offers non-blocking and timed varieties of receive.

Another important CSIM function is hold. A call to hold causes the calling

process to sleep for the specified number of clock ticks. This can be used to simulate the CPU processing in a PVM computation. This CPU processing can be either useful computation or PVM messaging overhead.

4.2 Ethernet, 10 Mbits/second

The first simulation is a 10 Mbits/second Ethernet [14] broadcast local area network (LAN). As this is the network used to connect the testbed computers, these tests were also used to validate the simulation. After initializing some values, the simulator spawns a task, or station in CSIM terminology, for each processor in the system. A TCP server for each task is simulated by another CSIM task, a facility, and called tcp_server. There is exactly one tcp_server associated with each station. The CSIM tasks station and tcp_server both execute a CSIM create statement, which acts as a Unix fork and spawns the task off as an independent function.

Each station task reads its input file and executes the action indicated by the function flag. A "c" causes the station to hold for the specified number of microseconds. Other function flags involve much more activity.

4.2.1 Send a Message to Another Task

For a function flag of "s", the station will simulate sending a message to another task. The station first simulates the PVM task handing off data to the TCP processor by holding for 375 μsec. We estimate 375 μsec to be the time required to write one block of 1460 bytes of data to the socket. We arrived on this number by measuring the time spent in write in a traced PVM program that sent 1428-byte packets to another process 100 times. (PVM added a 32 byte header to the data, so the writes operated on 1460 bytes each time.) This program was executed on one of the Sun Sparcstation 5s on the network testbed. The writes averaged 375 μsec.

After holding, the station task starts its TCP process by reserving that fa-

cility and passing it the message particulars (source, destination, and length). The station resumes processing immediately, even if the TCP facility for the simulated machine is in use, but the station's message will not go out until the TCP facility becomes available. The station task then holds for the remainder of the time listed in trace file as the duration of that write operation. This is based on the assumption that the time taken by the write operation depends on the speed of the machine, not network availability factors. We assume that the TCP facility has sufficient buffer space to hold the entire message so that a network slowdown does not result in TCP pausing and blocking the calling process in the middle of a write, though it will delay the beginning of a write.

The tcp_server facility breaks down the message into data packets of 1460 bytes, adds 40 bytes to each packet for TCP/IP overhead, and simulates passing them through Ethernet. After each data packet is successfully transmitted through Ethernet, the tcp_server facility sends a CSIM message to the destination tcp_server facility to tell it a data packet has arrived.

Ethernet is a Carrier Sense Multiple Access, Collision Detect (CSMA/CD) system [14]. Typically, Ethernet stations will test the network to sense a carrier. If the carrier exists, the station broadcasts its message through the physical media, and listens for that message. If the sending station receives its own message correctly, it assumes that the message was also received correctly at the intended destination station. If it does not receive its own message correctly, it assumes that another station on the Ethernet also broadcasted at the same time, causing a collision. When this occurs, each station attempting to broadcast a message will back off, waiting a random amount of time, and try again.

Multicast. The "m" flag indicates a message will be multicast to several other tasks. In this case, the simulation proceeds in the same manner as with send. After the data is transmitted, the tcp_server facility sends a CSIM message to every task specified as destination.

4.2.2 Receive a Message From Another Task

If the function flag is ”r”, the station process will reserve its tcp_server facility to receive data, and sleep until the tcp_server facility reports that a data packet has arrived. The tcp_server facility waits for a CSIM message from the source facility, buffering messages that arrive from other sources. When a message from the correct source arrives, the tcp_server sets an event to wake up the station process, and sets a global variable with the amount of data received. The station process decrements this amount from the number of bytes it wants to receive. If the number of bytes to receive is zero, the station process reserves the tcp_server facility to send an acknowledgment. If the number of bytes in the received message is more than the number of bytes for which the process is waiting, a stub for the received messages is left in its mailbox so that the remaining bytes can be received in subsequent actions.

4.2.3 An Example of Simulation Processing

Figure 4.1 shows an example of the interaction between the processes comprising our Ethernet model. At the top of this example, a station process has just begun an action to send a 4032- byte message to another process. The station first holds for 375 μsec to simulate handing the first 1460 bytes to the TCP handler, then reserves the tcp_server facility. Though the sending station will not block, the tcp_server facility will not process the request until the facility is free. The station continues on by holding for the remainder of the time listed in the trace file for that particular write action. It then acts on the next line from its simulation input file, which in this case is a receive.

Once freed from its previous task, the TCP process reserves the tcp_server facility, and holds for 300 μsec to simulate TCP send processing. It then passes the message on to the Ethernet by 1500- byte packets. When finished, it will send a CSIM mail message to the TCP process associated with the destination station to tell it the packet has arrived, and call release to make itself available for other task-

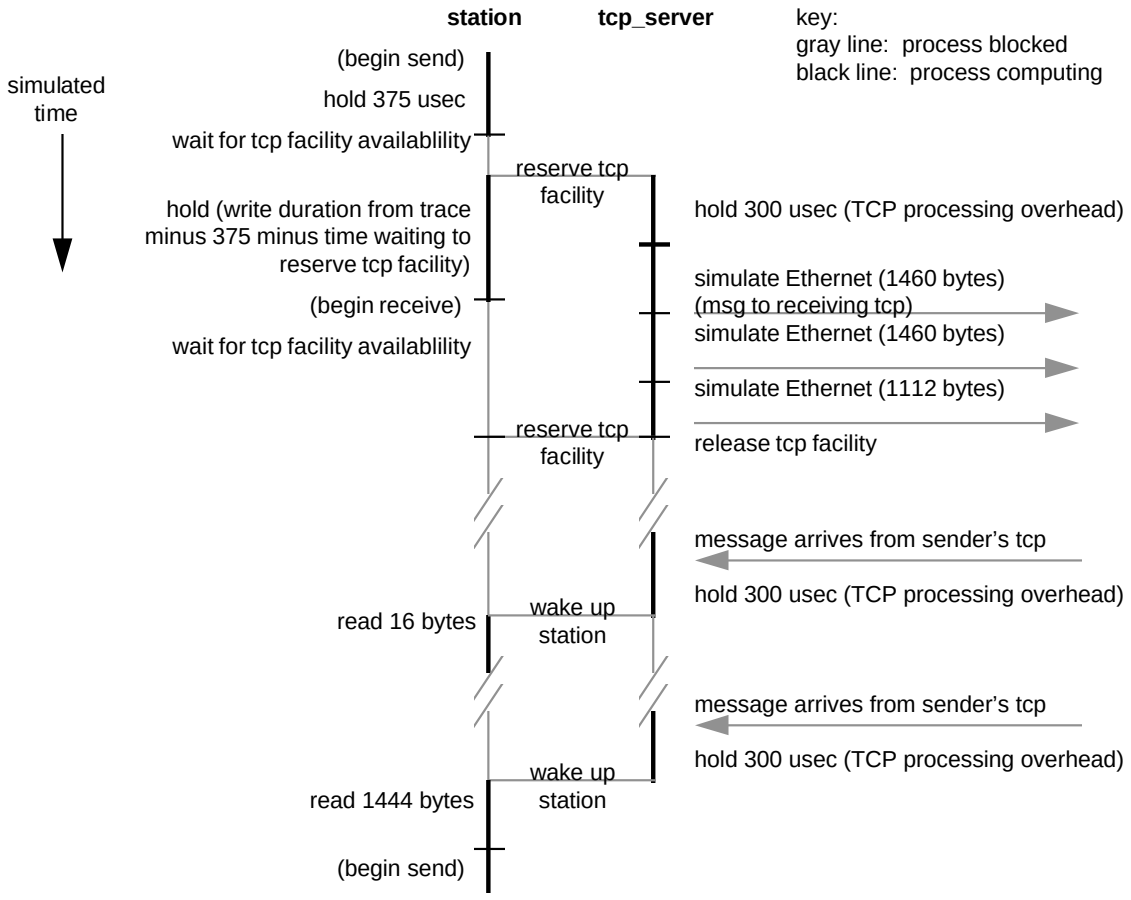


Figure 4.1: Simulation Timeline

ing. In the example in Figure 4.1, the station next issues a request for a receive, and blocks waiting for the completion of that action. The TCP process waits on its mailbox for a message from the source specified. Once that message arrives, the TCP process holds 300 μsec to simulate TCP receive processing, and then wakes up the blocked task. The station that was waiting for the message holds for the number of microseconds listed in the trace file as the duration of that read. If there are still more data packets to be received, the TCP process will be waiting for and responding to them at the same time the station is holding to read the data just delivered. After the complete message has been processed by the TCP process and read by the station, the station uses the TCP process to send an acknowledgment.

4.3 Ethernet, 100 Mbits per Second

The 100 Mbits/second Ethernet simulation works in exactly the same way as the 10 Mbits/second Ethernet simulation, except that data is transmitted at 0.08 μsec per byte rather than 0.8 μsec per byte. 100 Mb/sec Ethernet (a.k.a. Fast Ethernet) was standardized as 802.3u and approved by the IEEE in June 1995 [4]. In our simulation, as in the standard, the only difference between the 10 Mbps and 100 Mbps Ethernet is the speed of transmission. All the packet formats, interfaces, and procedural rules of the 10 Mb/sec Ethernet standard (IEEE 802.3) remain the same.

4.4 Assumptions and Approximations

We made several assumptions and approximations concerning PVM, TCP, and host machine operation:

A. The overhead incurred in generating traces of a PVM application does not alter its overall execution time substantially. The benchmarks we used are latency independent, so the additional overhead does not change the execution behavior of these programs. We have quantified the amount of time used to generate traces, but

did not remove that time from the results because these costs are extremely variable depending on how many times mxfer executes its loop on receiving messages. On the average, each send costs 182 μsec, each two-part receive costs 278 μsec, and each PVM communications support function call instrumentation costs 80 μsec. These costs show up in the compute time between communications activities, not as a component of the communication. Because each message is sent by one station and received by a different one (except in the case of BT and SP as noted in Chapter 5), the added time per message due to instrumentation overlaps on the send side with the added time on the receive side. On the send side, the instrumentation adds 342 μsec (80 μsec to report pvm_pk*, 80 μsec to report pvm_send, and 182 μsec to report write). On the receive side, it adds 438 μsec. Because these values overlap in time, we consider 438 μsec to be the cost, per message, of the instrumentation.

B. We estimate the overhead added by TCP to be 300 μsec. We derived this number from Clark et al [15], who measured Berkeley TCP overhead on a Sun 3/60, which has a 20 MHz CPU. On this system, processing a 1460 byte packet was measured at 1211 μsec on the average. The Sun Sparcstation 5s used in these PVM experiments are more advanced, 70 MHz machines, and we estimated 300 μsec, or roughly one-fourth the time required on the Sun 3/60. The part of the TCP protocol most commonly executed is common to both versions.

C. The simulation always uses direct routing, though the benchmarks were all executed using default routing. When tasks communicate through the default route, each communication must go through at least one PVM daemon. When the communication is between tasks on different machines, the default route includes two PVM daemons and UDP, an unreliable protocol. The PVM daemons do not block for messages; instead, they loop through work handling whatever messages arrive as they appear. These factors together add up to a system that is difficult to simulate. As noted in Chapter 5, the direct routing option was unstable in six of the seven benchmarks used. To be consistent, we used default routing in all trace generation runs. The only apparent effect of this change was that direct routing breaks

down messages of more than 64K bytes into several messages less than that size. Default routing allowed the larger message size to pass unaltered. This option selection reduces the correspondence between the actual execution of a benchmark and its simulated execution, but comparisons of the results of each showed little difference.

D. Communications between a PVM task and its PVM daemon are not included in these simulations. As these tasks are on the same machine, they do not involve network activity. Communications between tasks for other than data exchange (for example, to set up a direct route) are included.

E. We assume the local area network has no other traffic than that caused by these PVM tasks. For our testbed network, the amount of other traffic during the time of day that the applications were run is minimal.

F. PVM provides a Fortran library as well as a C library. The Fortran library is implemented as a Fortran function that calls the corresponding C library function. In calculating PVM overhead costs, the time required to process these Fortran ”wrappers” is considered to be minimal, and not included.

Chapter 5

PVM Benchmark Programs

In this chapter we discuss the PVM applications we selected for our tests. The PVM Numerical Aerodynamic Simulation (NAS) Parallel benchmarks are eight problems designed by the NASA Ames Research Center to provide a common basis for comparisons of parallel computers. The problems consist of five kernels and three simulated computational fluid dynamics (CFD) applications, all specified algorithmically [9]. These benchmarks were implemented as PVM applications at Emory University by V. S. Sunderam, S. White, A. Alund, and X. Lu [10].

We chose the NAS parallel benchmarks for our experiments for three reasons. First, the source code and documentation are widely available over the Internet. Second, the complexity and variety of the benchmarks provide a meaningful test environment. And finally, these and other implementations of the same benchmarks have been used in evaluations of many types of parallel systems, so our results may be compared to the results of other such evaluations. In these experiments, we do not analyze the benchmarks in terms of message size, complexity of communications, or effectiveness of implementation. Instead, we treat them simply as examples of PVM applications and observe how they behave as network conditions change. We used only seven of the eight benchmarks. The Embarrassingly Parallel (EP) benchmark has minimal communications. We obtained the PVM source code for the other seven from Sunderam's group at Emory University [10].

For most of the benchmarks, memory requirements constrained the problem size to the smallest size available. All benchmarks were run in a four-computer network configuration, with another computer added to the PVM system in three benchmarks as noted below. Finally, though three of the seven benchmarks offered an option for direct routing instead of default routing, we used default routing in all tests. Setting the option for direct routing caused six of the seven benchmarks to fail on execution.

We programmed the simulations to maintain a record of each PVM message's source, destination, length, and the (simulation) time it was passed to the TCP facility. After running the simulations on the four-node network testbed using the traces generated in the execution of each benchmark, we used these records to construct communications profiles for each benchmark. Figures 5.1 through 5.14 show the aggregate and dynamic communications profiles of each benchmark. The aggregate communications profile reflects the number of bytes and messages sent from each station to the other stations. The dynamic communications profile shows the number of bytes and messages sent by all processes as a function of time.

5.1 Multigrid

Multigrid (MG) is a kernel benchmark that solves a 3D Poisson partial differential equation with constant coefficients [9]. MG was implemented in C as pvmmg. In our tests, we used a problem size of 128x128x128.

MG is reported to test both short and long distance communications. Due to the limited number of nodes on our test network, we could not observe long distance communications directly. Figure 5.1 shows that each MG process communicates with two other processes consistently, but has very little communication with the third. Figure 5.2 shows the number of messages and bytes sent by all MG processes throughout the execution of the application. According to this graph, MG traffic can be characterized as sporadic, and the number of bytes sent per message

varies between messages.

5.2 Conjugate Gradient

The Conjugate Gradient (CG) benchmark was implemented in Fortran as pvmcg. CG is a kernel benchmark that uses a conjugate gradient method to compute an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix [9]. We ran pvmcg with a matrix size of 1400, in 15 iterations. The recommended problem size for CG is 14000, but memory constraints did not permit this problem size.

CG is reported to use irregular, long-distance communications. Again, the limited number of nodes on our test network prevented us from observing long distance communications. The communications profile of CG we constructed (see Figure 5.3) shows that a CG process sends more data to some fellow processes than others, and also sends this data in varying message sizes. The dynamic communications profile given in Figure 5.4 shows that the message traffic is consistently of a high level, with varying numbers of bytes per message.

5.3 Integer Sort

The Integer Sort (IS) benchmark was implemented in C as pvmis. This kernel benchmark tests a sorting operation used in particle method codes, which require reassignment of particles that have drifted from place back to the appropriate cells [9]. We selected options to run a problem size of 2^{20}, with a key range of 0 to 2^{20}. Load balancing was not used.

The aggregate and dynamic communications profiles in Figures 5.5 and 5.6 show that the amount of data sent between processors is extremely variable, but the number of bytes sent to each process is consistent with the number of messages sent to that process.

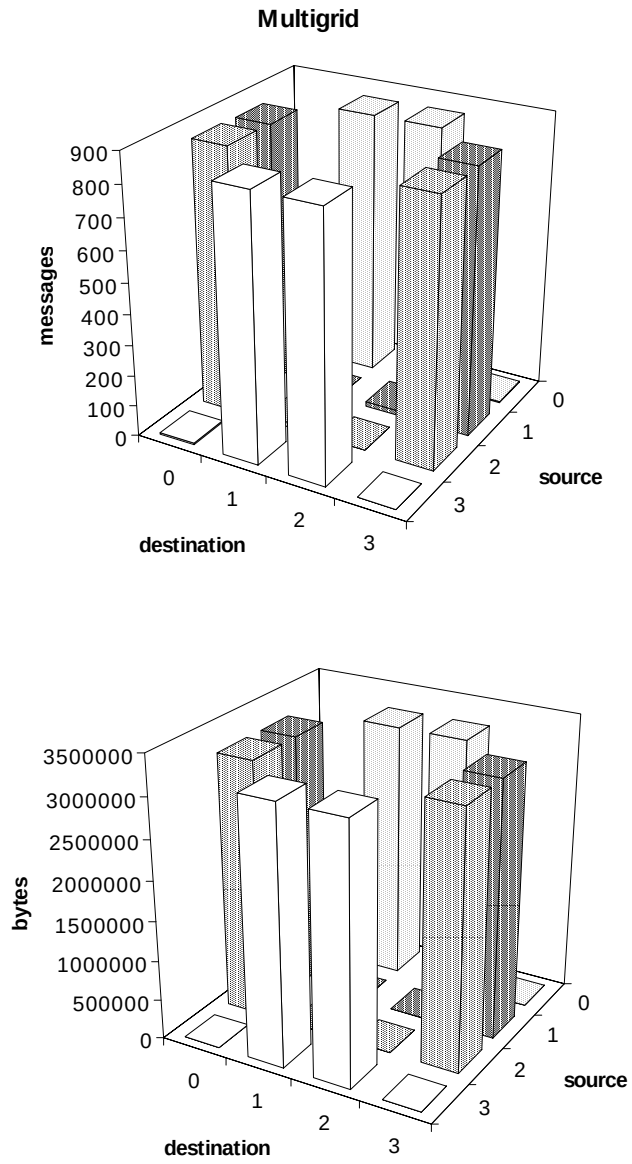


Figure 5.1: Aggregate Communications Profile for MG

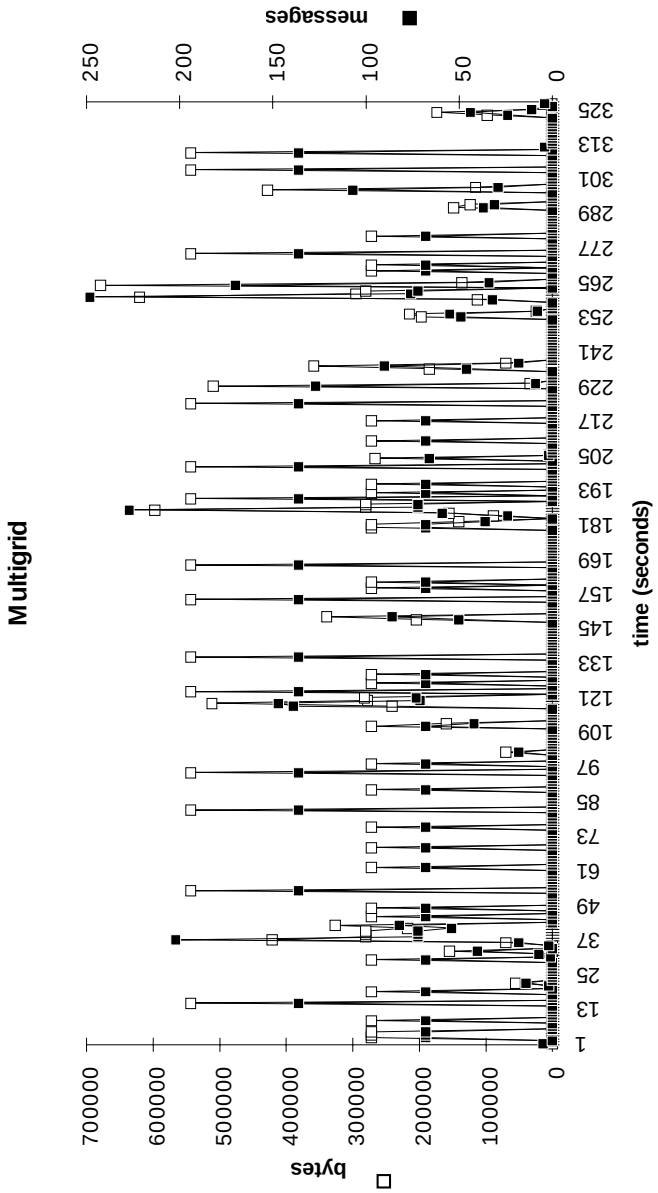


Figure 5.2: Dynamic Communication Profile for MG

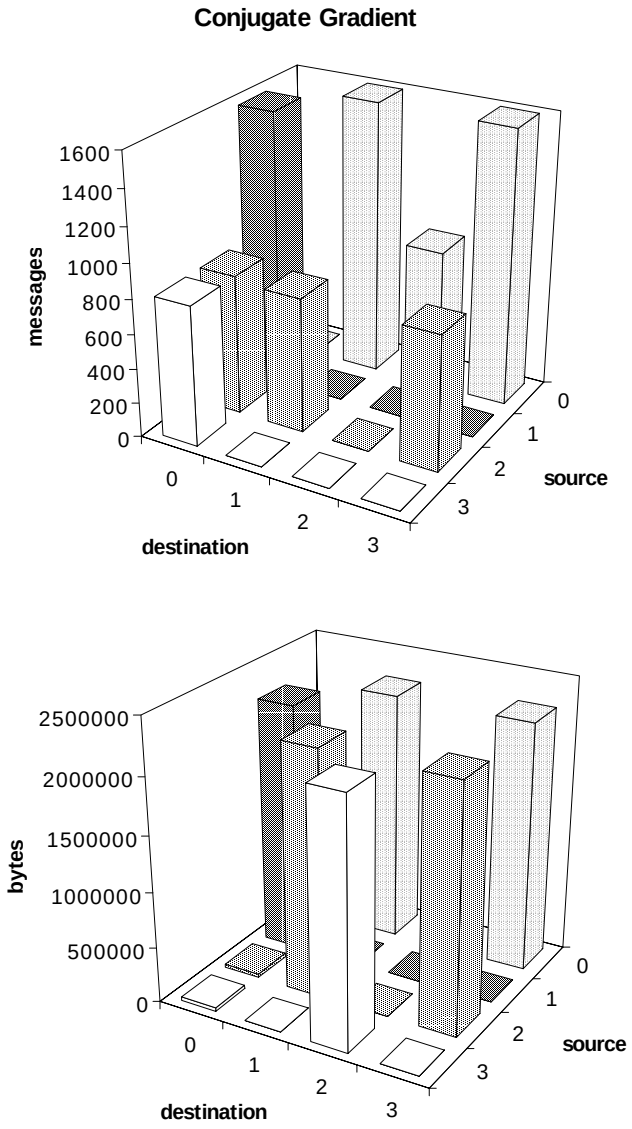


Figure 5.3: Aggregate Communications Profile for CG

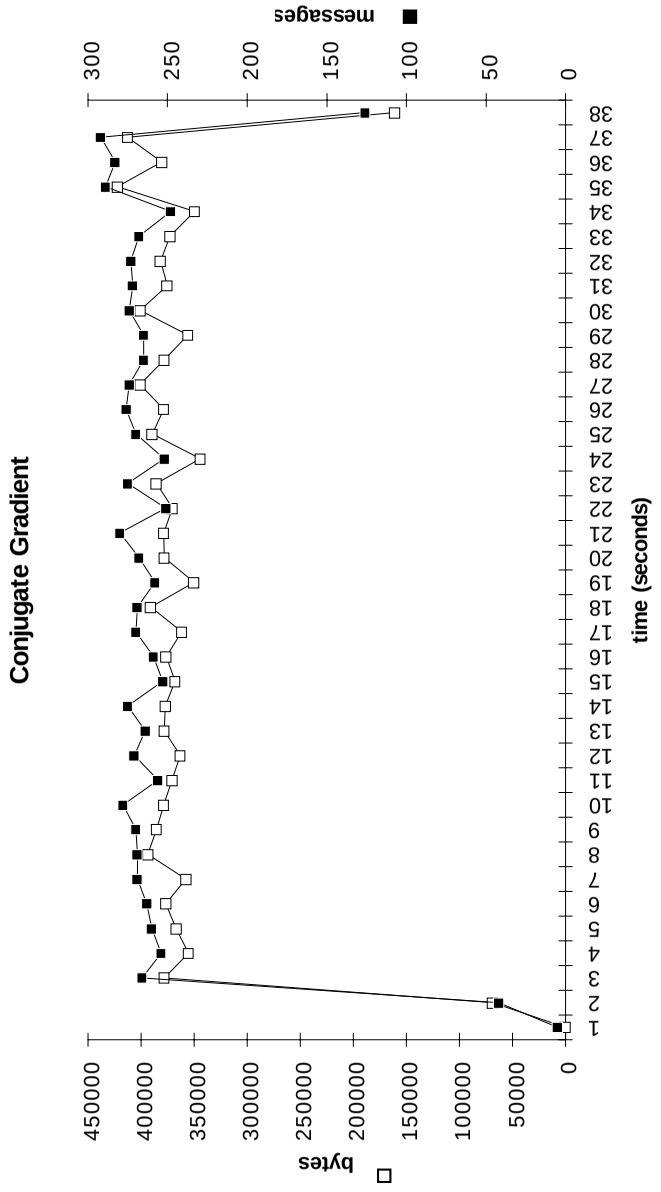


Figure 5.4: Dynamic Communication Profile for CG

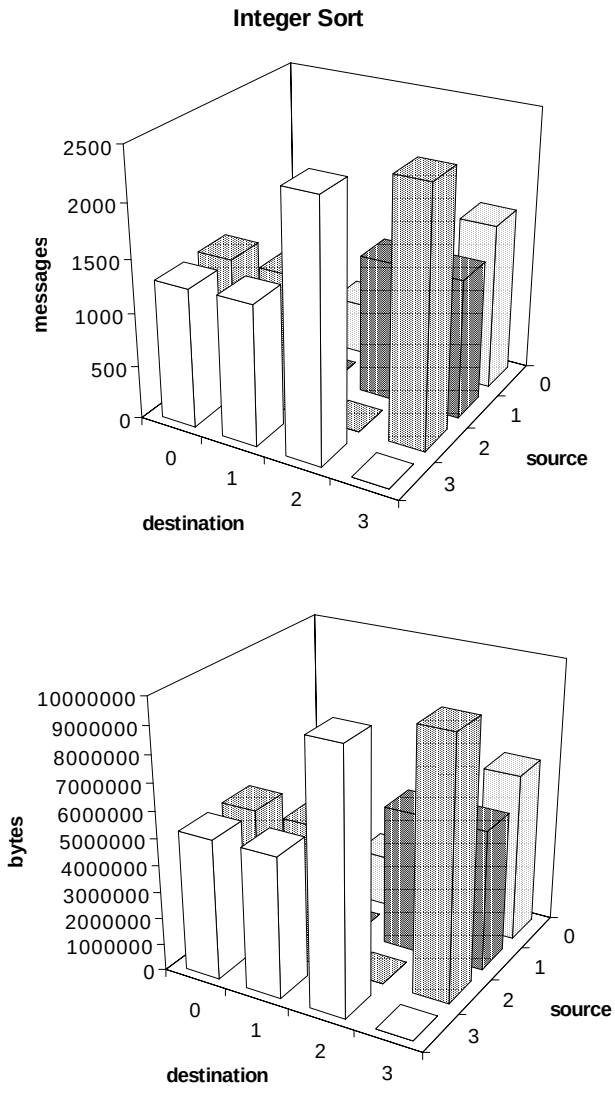


Figure 5.5: Aggregate Communications Profile for IS

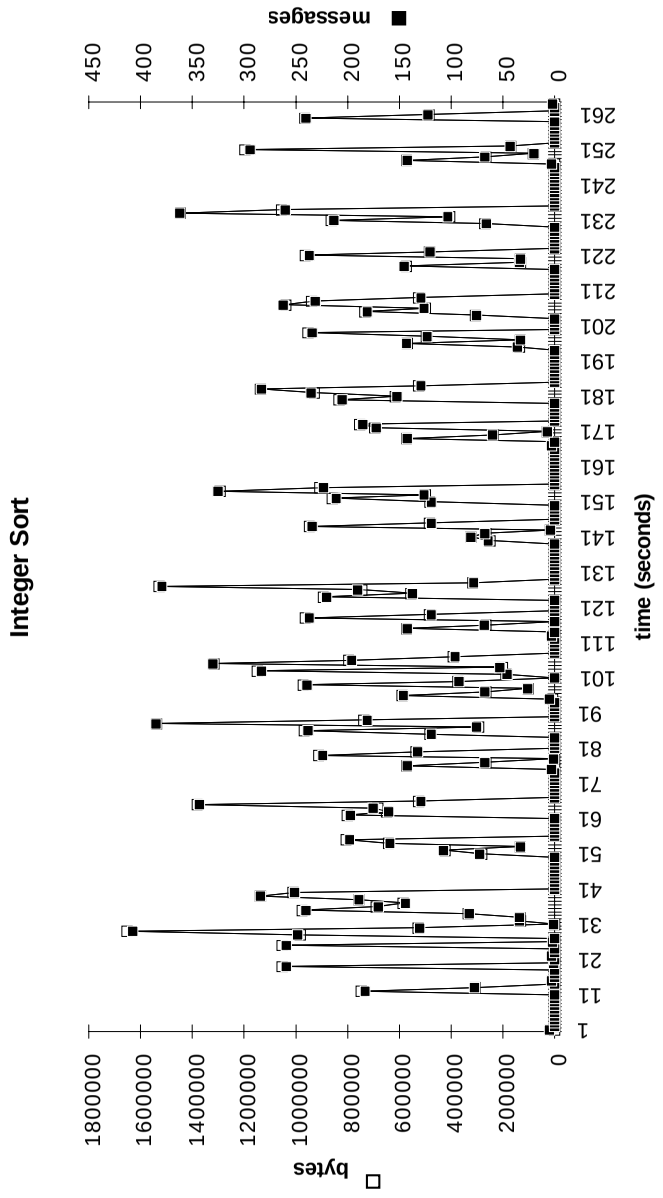


Figure 5.6: Dynamic Communication Profile for IS

5.4 Fast Fourier Transform

The FT benchmark solves a 3D partial differential equation using fast Fourier transform [9]. This kernel benchmark was implemented in Fortran as pvmft. We used a problem size of 64x64x64 in our tests.

The aggregate and dynamic communications profiles in Figures 5.7 and 5.8 show that each process participating in the FT application sends approximately the same amount of data to every other process in the system. The number of messages sent is consistent with the number of bytes sent as well.

5.5 Lower-Upper Diagonal

The Lower-Upper Diagonal (LU) benchmark uses a symmetric, successive overrelaxation numerical scheme to solve a regular-sparse, block lower and upper triangular system [9]. LU is a simulated computational fluid dynamics benchmark and was implemented in Fortran as pvmlu. For these tests, another machine was added to the four-node configuration to hold the master pvmlu task. This master task initializes the problem set and spawns and controls the other tasks as they perform the computation. Due to memory constraints, the problem size we selected was 12x12x12, and the program ran in 50 iterations. A complete solution to this benchmark requires 250 iterations.

Results of a test run shown in Figure 5.9 shows that each pvmlu process sent messages consistently to two other tasks, and the overall number of messages sent to each process is consistent with the number of bytes sent to that process. (The master process is station 0.) Figure 5.10 shows that the number of bytes per message varied considerably over time.

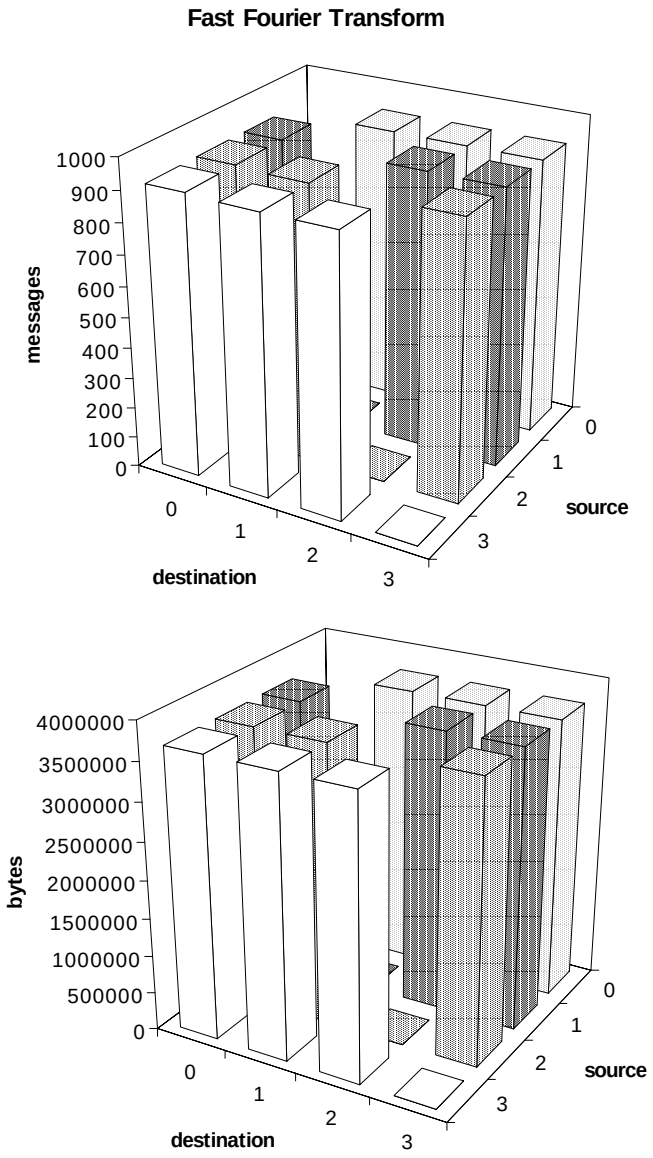


Figure 5.7: Aggregate Communications Profile for FT

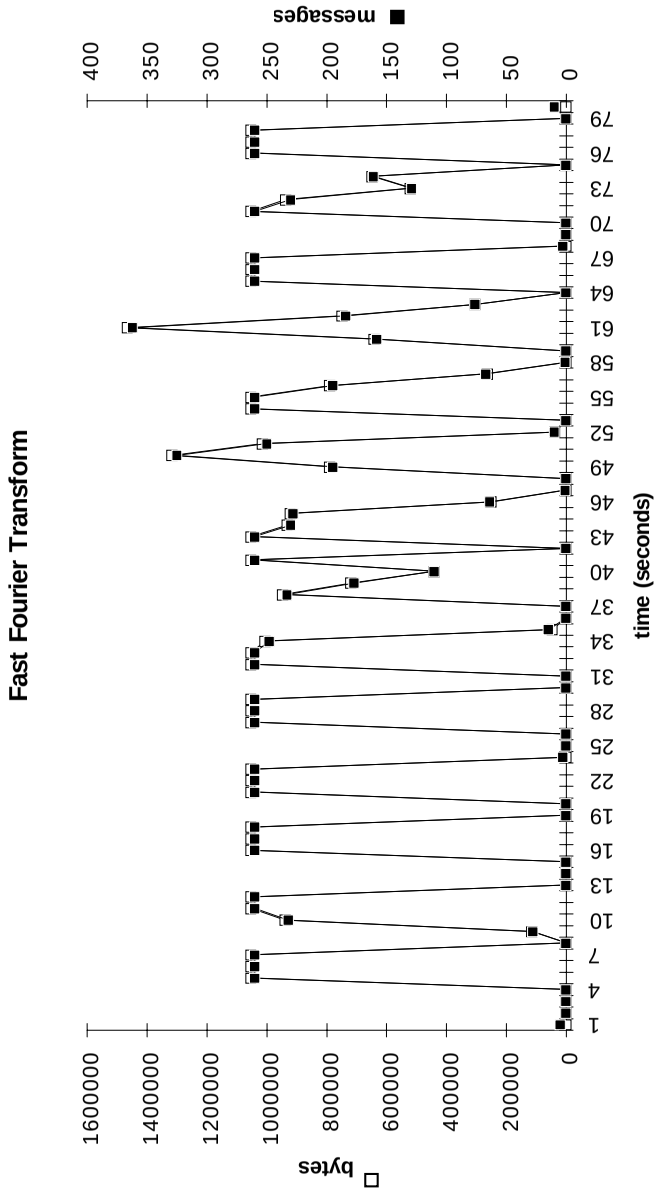


Figure 5.8: Dynamic Communication Profile for FT

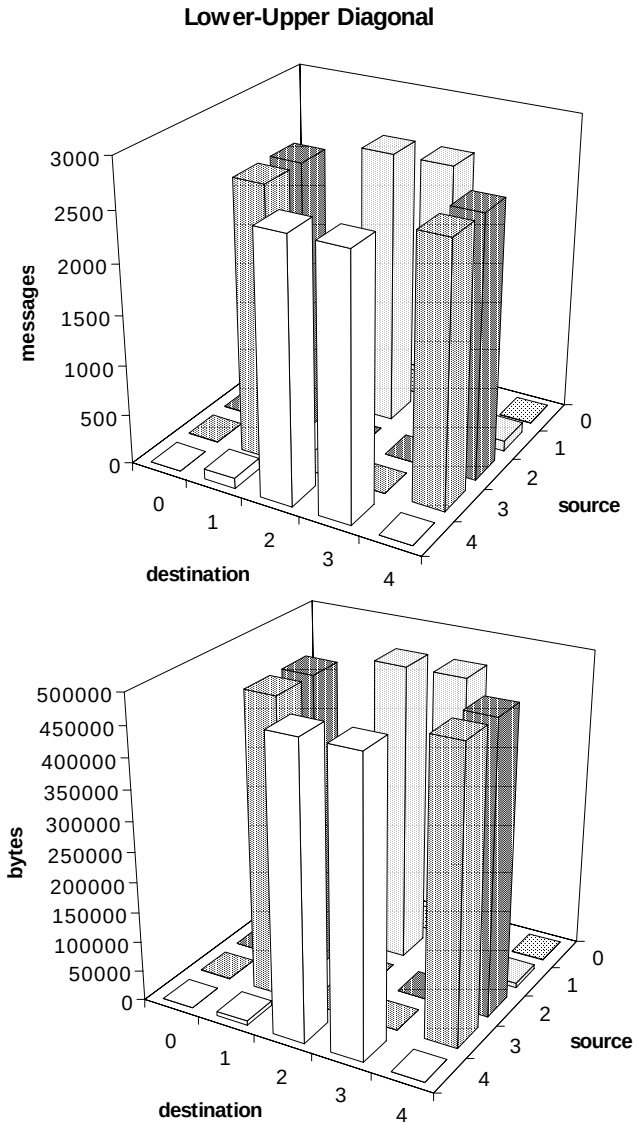


Figure 5.9: Aggregate Communications Profile for LU

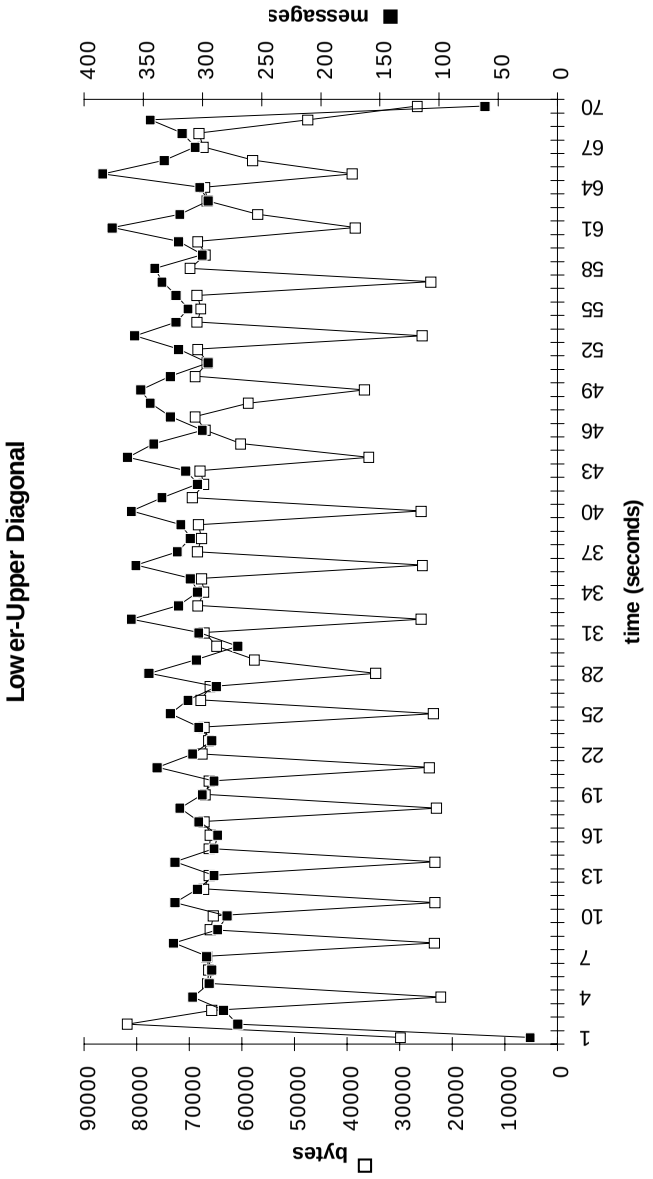


Figure 5.10: Dynamic Communication Profile for LU

5.6 Scalar Pentadiagonal

The Scalar Pentadiagonal (SP) benchmark solves multiple independent systems of nondiagonally dominant, scalar pentadiagonal equations [9]. SP was implemented in Fortran as pvmsp. We used a problem size of 12x12x12 in these tests, again due to memory constraints. An extra machine was added to each configuration to hold the master pvmsp process. The program ran in 100 iterations; a complete solution requires 400 iterations.

Figure 5.11 shows that each process sends a different amount of data to each other process, and the number of messages sent to a process matches the number of bytes sent to that process. Figure 5.12 indicates, however, that the number of bytes per message varied throughout the computation. (It is interesting to note that each process sent a considerable amount of data to itself. We assume that this is an error in the implementation of the benchmark. Though these messages to self never hit the network media, considerable time is spent in calls to read, write, and select to handle them.)

5.7 Block Tridiagonal

The Block Tridiagonal (BT) benchmark solves multiple independent systems of nondiagonally dominant, block tridiagonal equations with a 5x5 block size [9]. This implementation of BT, pvmbt, also requires a master task to initialize the problem set and spawn and control the other tasks as they performed the computation. We used a problem size of 12x12x12, executing in 60 iterations. A complete solution requires 200 iterations.

Figure 5.13 shows that each process sent different amounts of data to other processes, and the number of messages sent to a process matches the number of bytes sent to that process, much the same as the SP application. (In this benchmark implementation as well, each process sent a considerable number of messages to

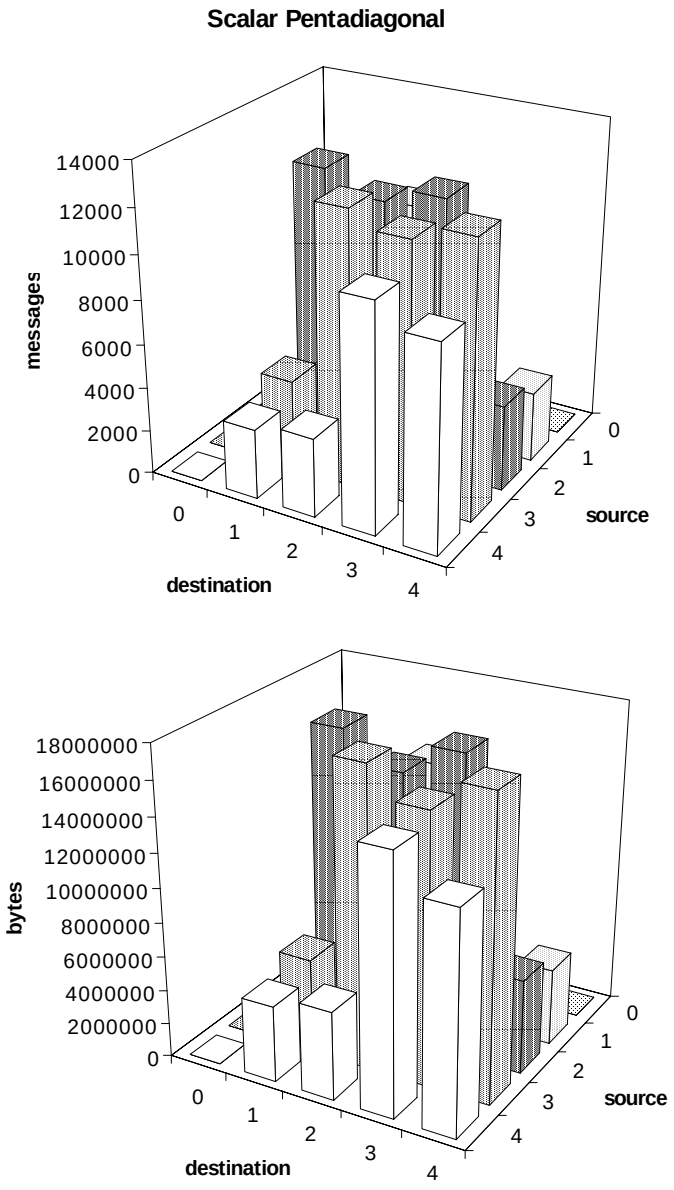


Figure 5.11: Aggregate Communications Profile for SP

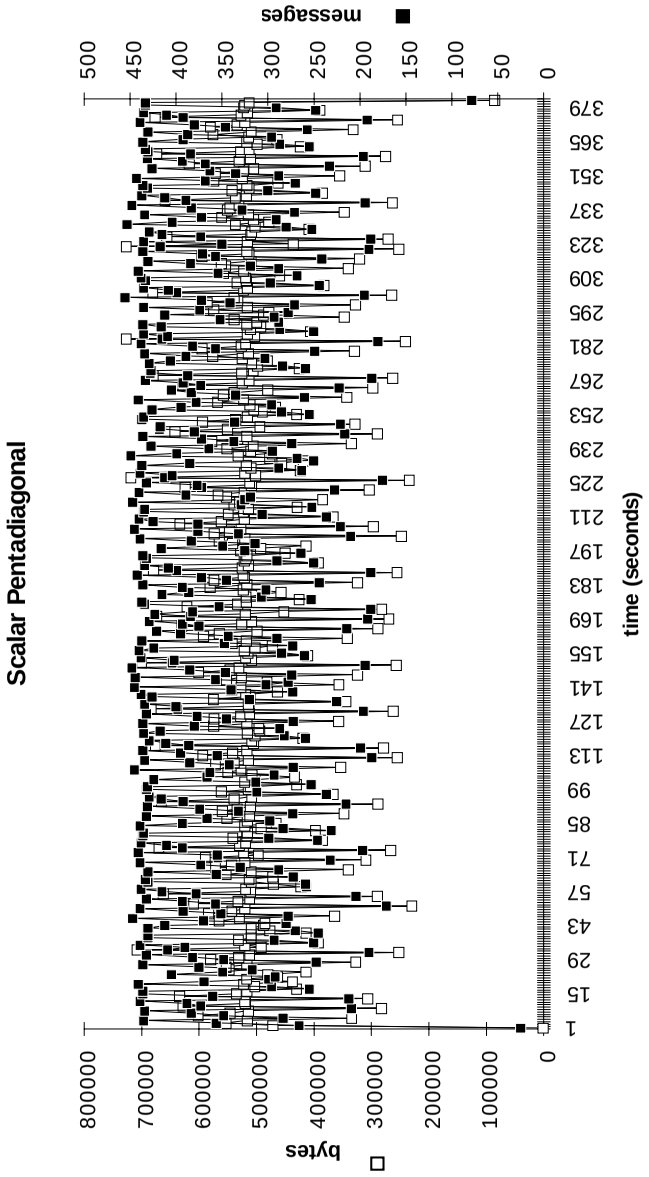


Figure 5.12: Dynamic Communication Profile for SP

itself.) Unlike SP, however, the graph of traffic over time indicates that the number of bytes per message remained relatively constant over the period of the computation.

5.8 Test Network Configuration

The test network used to run the NAS benchmark applications under PVM consisted of five computers on the University of Texas at San Antonio computer network. This network actually connects 85 computers in a series of Ethernet sub- LANs, connected by bridges and repeaters into one rather large and busy network. The computers used in these tests consisted of five Sun Sparcstation 5s configured as listed in Figure 5.15. These computers were all connected to the same sub-LAN.

We ran the benchmarks at night to reduce the amount and effect of ambient network traffic. Typically, we first start PVM on each machine in the four- or five-node configuration, depending of the type of benchmark run. A shell script is executed to run the benchmarks and convert the trace output to simulation input. These input files are used to drive the Ethernet simulator described in Chapter 4.

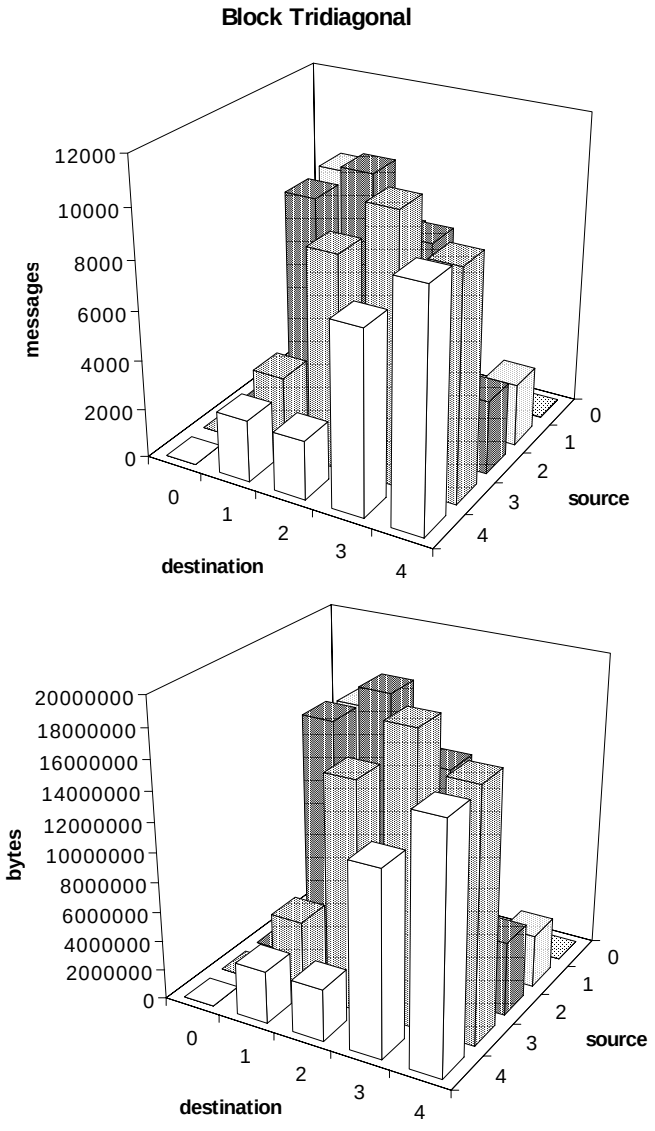


Figure 5.13: Aggregate Communications Profile for BT

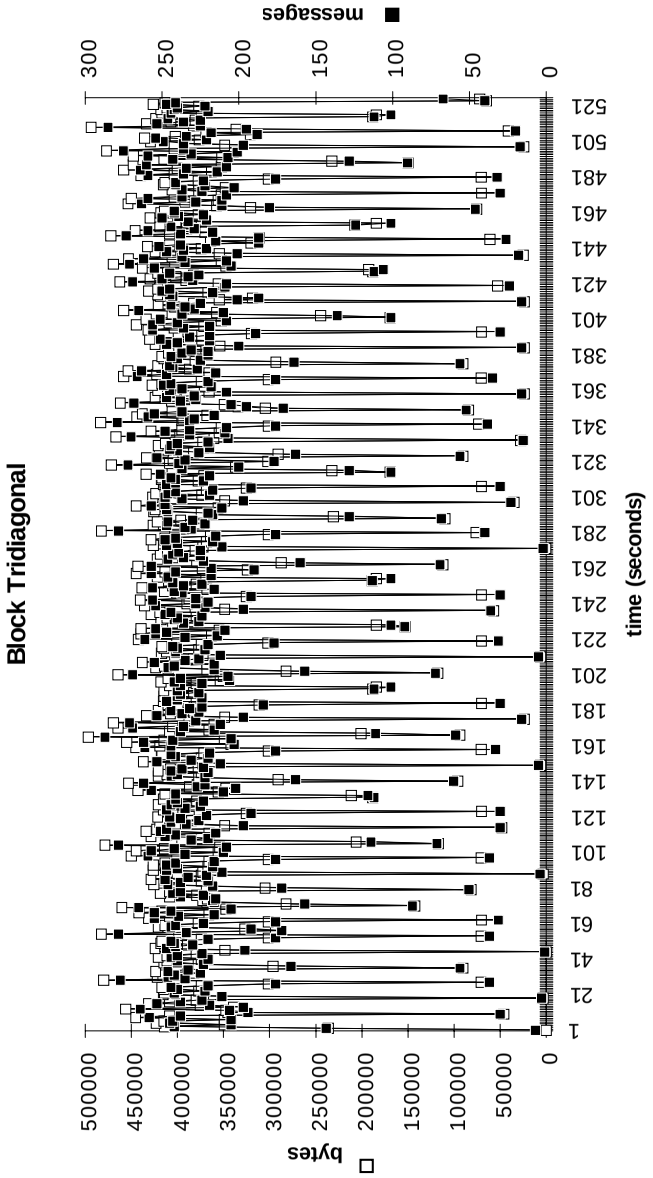


Figure 5.14: Dynamic Communication Profile for BT

Computer Type:	Sun Sparcstation 5
Operating System :	SunOS 5.4
Main Memory:	32 MB
Virtual Memory:	55 MB
No. of CPUs:	1
Kernel Architecture:	sun4m
Max No. of Processes per user:	485
Max STREAMS Message Size:	65536
Version of POSIX 1 Standard Supported:	199309
Version of X/Open Standard Supported:	3
Total Pages Physical Memory:	8192
Ethernet Interface Card:	AMD Lance Am7990, 10 Mb/sec
CPU:	microSPARC II
CPU Speed:	70 MHz
PVM Version Used:	3.3.7
PVM Architecture Class:	SUN4SOL2

Figure 5.15: Network Testbed Computer Configuration (from sysinfo [6])

Chapter 6

Performance Analysis

Several tests conducted in recent years [16, 17, 18, 19] have compared the performance of PVM to that of other parallel environments, such as P4 and Linda. In most cases, the comparisons begin with a simple ping/pong type of data transfer, much like the ring example described in section 2.4. Some of the papers extended the comparison to consider other, more complicated communications profiles, using applications like OVERFLOW-PVM and a subset of the NAS Parallel Benchmarks [17]. In all cases, however, message send and receive operations were timed externally to PVM; that is, either the PVM application was instrumented to report the start and end times of the data transfer, or the overall execution of the application was timed. We believe that our method of inserting the instrumentation within the PVM communications library enables a finer-grained analysis of the results. In particular, we are able to separate out the contributions that each element of PVM processing makes to the overall execution time as well as the time each packet spends on the actual network.

In this chapter, we present the results of our benchmark runs on a testbed workstation cluster and an analysis of the data. The performance analysis compares the actual execution times of the PVM benchmarks to the predicted execution times obtained by our simulator. For the simulated system configurations, we have used both the 10 Mb/sec Ethernet backbone (the one actually used in our execu-

tion runs) and the 100 Mb/sec fast-Ethernet backbone. We also recomputed the execution times after reducing the PVM message processing overhead by 90% to get an idea of the impact of this aspect of the intertask communications. Using these scenarios, we predict the performance improvement on the applications.

6.1 Test and Simulation Results

The simulations can measure and report a variety of factors. For these tests, we report the total simulation time, time blocked on receive, time required to send messages on the network, overhead added by PVM communications functions, and the number of bytes and messages sent. All stations start at simulation time 0, and the times at which they complete the processing of their trace input files are reported as the total simulation time. The time blocked on receive is measured by the station, and includes the time the station had to wait for the tcp_server facility to become available as well as the time the tcp_server facility had to wait for a message from the correct source. The network time is measured at the tcp_server facility, and reports the time required to send each packet through the simulated Ethernet. We take the overhead added by PVM for communications support functions directly from the traces, with the overlapping time segments removed from the enveloping function call as described in Section 3.3. We also subtract the actual process blocking time reported in the traces from the PVM overhead total.

Time values are reported per station, and as an average of the four stations that perform the actual computation (in several cases, station 0 does not participate in the actual computation but acts as a task coordinator). The simulation also reports the number of collisions on the simulated Ethernet, the amount of time spent in each PVM function call, and the time each process spent in select while attempting a receive. This last value represents the actual blocking time of the process generating the traces.

We validated the simulation model by comparing the overall time of the

simulation to the benchmark execution time, and by comparing the blocking time from our simulation with the total time the benchmark actually spent in select, as reported in the traces. We calculated the actual execution time by subtracting the time stamp on the last message trace from the time stamp on the first message trace. Both the overall times and blocking times are close in all cases. More accurately, the actual and simulated blocking times have differed consistently by the same amount, indicating that the differences have to do with some initializations that are not modeled in our simulations.

Figure 6.1 summarizes the overall execution times of the actual runs and in the simulations. Detailed information on message overheads and individual process statistics are given in Appendix A. Possible sources of time differences include our acknowledgment policy and the ambient traffic on the real network. Our simulation sends an acknowledgment after each complete message is received. TCP acknowledges at most every other message [15]. In order to reduce the effect of ambient network traffic, we ran the benchmarks and collected traces during the early hours of the morning. Despite this precaution, it was evident that we occasionally happened upon a period of heightened network and system activity, when automatic news feed updates and other such activities disrupted the benchmark execution. To minimize this effect, we ran each benchmark at least 12 times, and used the traces from the minimum execution time for each.

The instrumentation itself is intrusive, and adds approximately 438 μsec per message to the overall execution time. With message counts in the tens of thousands for some benchmarks, this additional processing adds up, and can tend to mask the degree of speedup we see from the network.

The communications profiles given in Chapter 5 show that the communications complexity of each benchmark is significant. Our 10 Mb/sec Ethernet simulation matches strongly to the real execution profile in each case. Based on these results, we also consider our 100 Mb/sec Ethernet configuration simulation valid, and draw conclusions about the effectiveness of that network type.

Benchmark	Actual Time (seconds)	Simulated Time (seconds) 10 Mb/sec	100 Mb/sec
MG	329.162	321.601	319.393
CG	47.184	37.396	31.594
IS	282.657	261.182	257.114
FT	81.505	79.066	47.318
LU	77.880	69.298	68.458
SP	392.106	381.611	364.642
BT	529.075	524.556	504.224

Figure 6.1: Benchmark Test and Simulation Results

6.2 Impact of Network Speed on Communications Time

Changing the network speed from 10 Mb/sec to 100 Mb/sec brought some improvement to the benchmark performance in each case. Figures 6.2 through 6.8 show the results of the simulations. In these figures, the time duration of each benchmark is shown for four possible system configurations. The first configuration represents the system configuration on which we traced each benchmark. This system is a four-workstation cluster connected by a 10 Mb/sec Ethernet LAN (Section 5.8). The second configuration shows the workstation cluster connected by a 100 Mb/sec Fast Ethernet LAN. In the third and fourth configurations, we reduced the PVM messaging overhead by 90% to predict the performance improvement. These results are presented for workstation cluster connected by a 10 Mb/sec Ethernet LAN as well as one connected by a 100 Mb/sec Fast Ethernet LAN. All time values are reported in seconds.

6.2.1 Multigrid (MG)

For MG, the change in network speed proved to have little effect on the overall performance of the benchmark (Figure 6.2). MG processes sent a total of 25,714,772 bytes in 329.16 seconds, averaging 6,428,693 bytes in 213 messages. As noted in Section

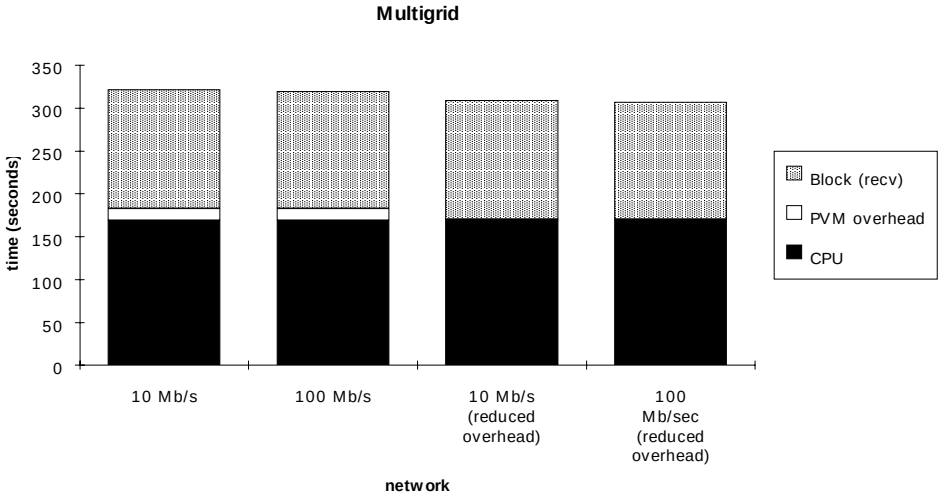


Figure 6.2: Comparison of Multigrid Simulation Results

2.5, when pvm_send is used with PvmDataDefault encoding, the process will divide the message into chunks of 4080 bytes each. Because of this, the number of messages sent averaged 1,721 per process. When the network speed is increased, the average send time dropped by more than a factor of 10. There is no similar decrease in blocking time, however, and the total effect is to reduce overall execution time by only 2.2 seconds, or 0.6%. MG processes send relatively few messages compared to the processes in the other benchmarks. It is possible that because MG sends fewer messages throughout an extended execution time, its processes enjoy a lower network time and collision rate than seen in other benchmarks. In that case, speeding up the network, which tends to reduce the collision rate, had little effect. Examination of the results shows that the average number of collisions per message for the 10 Mb/sec Ethernet was 3.01. As a comparison, FT processes suffered an average of 11.75 collisions per message on the 10 Mb/sec Ethernet LAN.

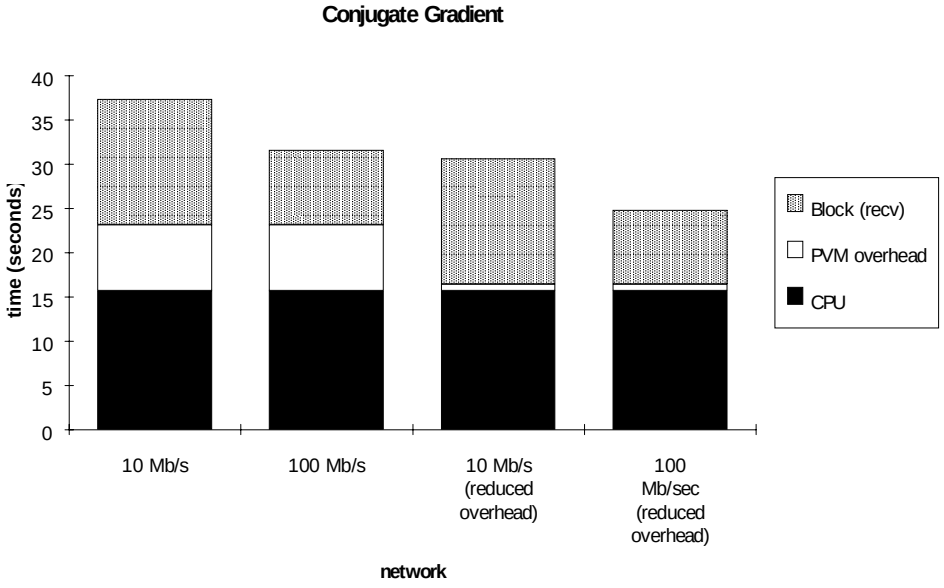


Figure 6.3: Comparison of Conjugate Gradient Simulation Results

6.2.2 Conjugate Gradient (CG)

The effect of changing the network speed for CG is more dramatic (Figure 6.3). In the test case, CG processes send a total of 13,414,084 bytes in 47.18 seconds, averaging 3,353,521 bytes in 2,386 messages per process. Changing the network reduces the send time by 95%. Blocking time is also reduced significantly, by 41%.

6.2.3 Integer Sort (IS)

The performance of IS shows little response to the change in network speed (Figure 6.4). IS processes send a total of 66,373,816 bytes in 282.66 seconds, averaging 16,593,454 bytes and 92 messages per process. The number of messages sent average 4,128 per process. In this case, changing the network reduces send time by 92%. However, blocking time remains within 3% of the original blocking time, so the overall effect of the network speedup is minimal.

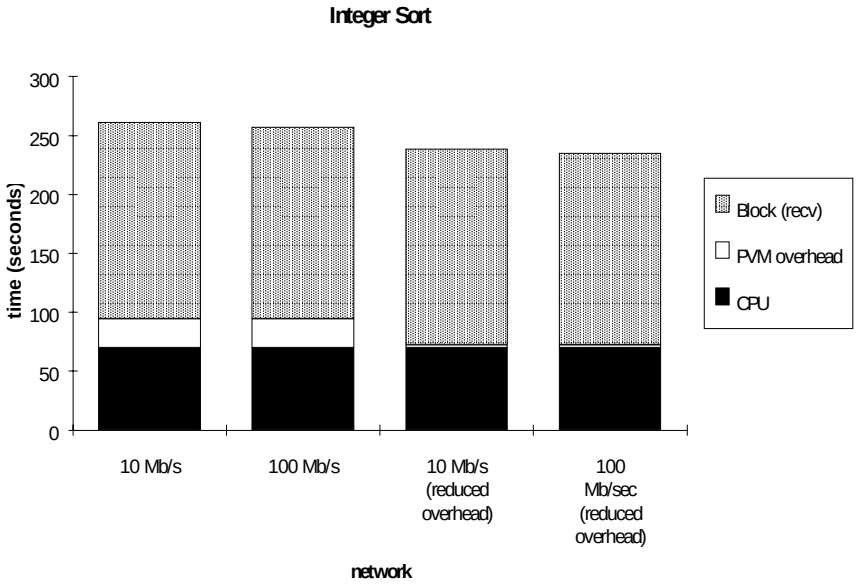


Figure 6.4: Comparison of Integer Sort Simulation Results

6.2.4 Fast Fourier Transform (FT)

The FT benchmark shows the most dramatic improvement from the change in network speed (Figure 6.5). FT processes send a total of 44,218,744 bytes in 81.51 seconds, averaging 11,054,686 bytes and 48 messages per process. The number of messages sent averages 2,735 per process. Using the faster network reduces the time on the network by 94%, and reduces blocking time by 93% as well.

6.2.5 Lower-Upper Diagonal (LU)

LU shows little effect when the network speed is changed (Figure 6.6). The four ”worker” LU processes send a total of 3,909,888 bytes in 77.88 seconds, averaging 977,472 bytes and 5,395 messages per process. Using the faster network reduces send time by 92%. Blocked time is reduced by only 6.7%. Of all the benchmarks, LU processes send the least number of bytes, and the least bytes per message (approx

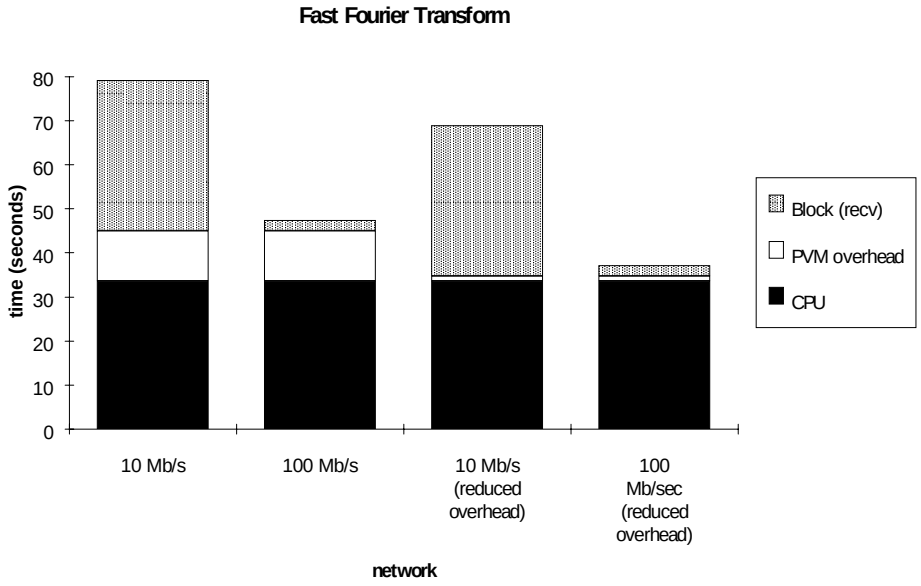


Figure 6.5: Comparison of Fast Fourier Transform Simulation Results

181 bytes per message in the test case).

6.2.6 Scalar Pentadiagonal (SP)

The SP benchmark shows a moderate change with the increase in network speed (Figure 6.7). The four ”worker” SP processes send a total of 187,350,080 bytes in 392.11 seconds, averaging 46,837,520 bytes and 33,639 messages per process. When the network speed is increased, the send time drops by 93%, and blocked time drops by 77%. SP stations each send a lot of messages to themselves. These messages cause no blocking and are never transmitted over the network, but affect overall execution time.

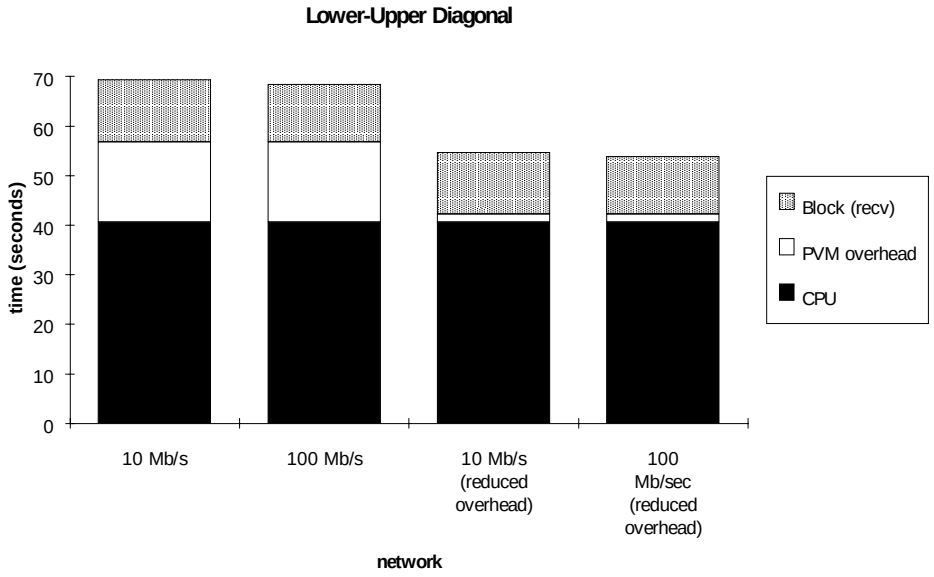


Figure 6.6: Comparison of Lower-Upper Diagonal Simulation Results

6.2.7 Block Tridiagonal (BT)

The BT benchmark also shows moderate effect with the network change (Figure 6.8). The four ”worker” BT processes send a total of 185,099,200 bytes in 529.07 seconds, averaging 46,274,800 bytes and 26,969 messages per process. When the network speed is increased, the send time drops by 93%, and blocked time drops by 73% of the original blocked time. Like SP, BT stations each send a lot of messages to themselves. These messages cause no blocking and are never transmitted over the network, but affect overall execution time.

6.3 Algorithmic Blocking Factors

The performance improvements with 100 Mb/sec Ethernet over 10 Mb/sec Ethernet varied considerably. Some benchmarks showed a great increase in performance while others showed small improvement. From the test results, it is apparent that

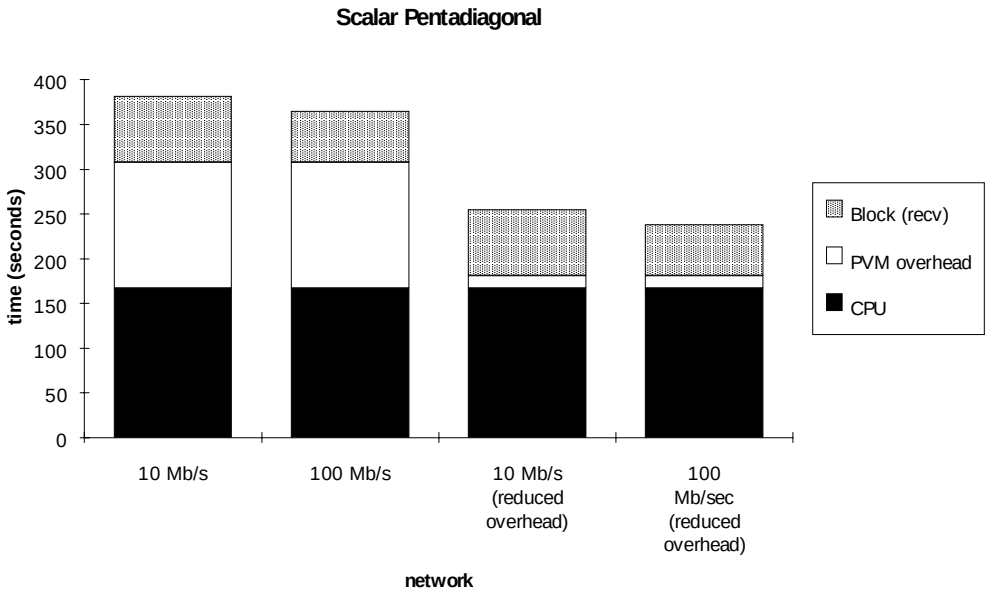


Figure 6.7: Comparison of Scalar Pentadiagonal Simulation Results

in each case, the amount of time that each benchmark has data on the network is reduced by 90% or more with the increase in network speed. This should have resulted in substantially reduced blocking time on the part of the station waiting for that data. But in most cases, the blocking time is reduced by only a small amount. These results highlight two areas that deserve more consideration: why the percent decrease in time on network is greater than the percent increase in network speed, and why this does not translate into a similar reduction in blocking time. To understand the results, we further analyzed the blocking time of PVM tasks in a benchmark execution.

By increasing the speed of the underlying network from 10 Mb/sec to 100 Mb/sec, we would expect that the time on the network would decrease by the same factor, yet the simulation results show that in all cases, the time on the network was reduced by more than 90%. The time on the network gives the time elapsed in the simulation's doether function, which simulates Ethernet, It does not include waiting

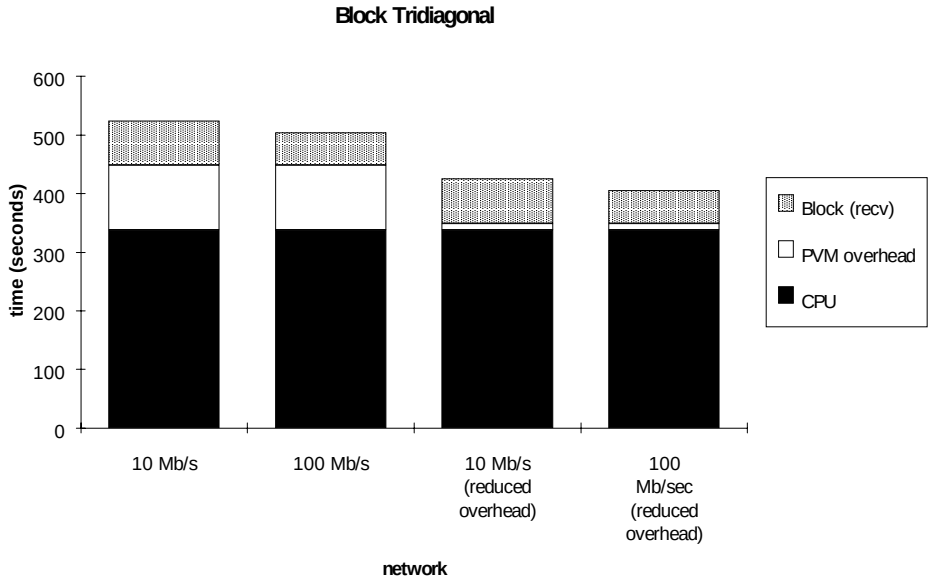


Figure 6.8: Comparison of Block Tridiagonal Simulation Results

for the tcp_server to become available or any other blocking factor. The most probable cause for the greater than 90% time difference is that having data on the network for less time resulted in fewer collisions per message. Therefore, the stations did not have to back off and retransmit as often and the total network time was reduced.

If the data spends less time on the network, one would assume that the station waiting for that data would not have to block as long. A naive prediction would be that the blocking time should be reduced by the amount of the reduction in time on the network. If so, the simulation results should show the blocking time for the 100 Mb/sec Ethernet configuration to be 57.3% of the blocking time for the 10 Mb/sec Ethernet configuration. Instead, the results show the 100 Mb/sec blocking time to be 72.3% of the blocking time for the 10 Mb/sec Ethernet configuration.

One possible cause for this difference is that a station may begin a receive operation on data that the other station has not sent yet. In that case, the receiving station's blocked time includes some computation time on the part of the sending

station as well as the time the data spent on the network. We refer to the time from the start of a receive operation to the start of the complementary send operation as algorithmic blocked time, and the time from the start of the send operation to the end of the receive as service blocking. Service blocking includes the time the sending station waits for its tcp_server to become free, the time the data spends on the network, and the time spent processing the message at the receiver's TCP station. These time elements are shown in Figure 6.9.

We computed the algorithmic and service blocking times for all the benchmarks (see Figure 6.10). These results show that only the service blocking time was reduced. Where algorithmic blocking was a large component of the total blocking time (e.g. MG and IS), the reduction in total blocking time was minimal.

6.4 Summary

These results show that for most cases, a PVM application would see an increase in performance by changing from a PVM configuration on 10 Mb/sec Ethernet to a PVM configuration on 100 Mb/sec Ethernet. The degree of change depends on the communications characteristics of the application.

Figures 6.2 through 6.8 also predict how well an application would perform if PVM overhead were reduced by 90%. PVM overhead constitutes a substantial portion of execution time for each benchmark. Reductions in the packing, buffer initialization, and send and receive processing in PVM would greatly improve the performance of most of these applications.

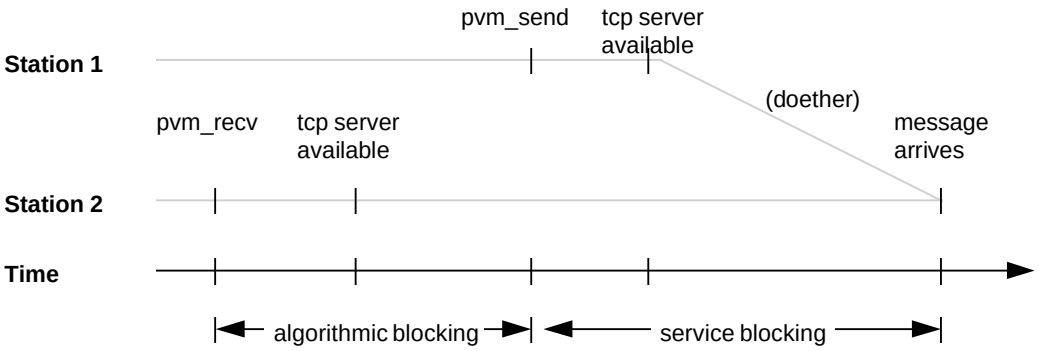


Figure 6.9: Components of Blocking Time

Benchmark	Total Time (seconds)	Total Blocking	Algorithmic Blocking	Service Blocking	%Service Blocking
MG	321.601	138.954	132.168	6.786	4.9%
CG	37.396	14.202	8.435	5.767	40.6%
IS	261.182	166.588	151.056	15.532	5.9%
FT	79.066	34.152	0.258	33.894	99.2%
LU	69.298	12.456	10.792	1.664	13.4%
SP	381.611	73.997	53.636	20.361	27.5%
BT	524.556	76.350	55.417	20.933	27.4%

Figure 6.10: Benchmark Blocking Times for 10 Mb/sec Ethernet Simulation

Chapter 7

Discussion and Conclusions

In this project, we demonstrated how PVM could be instrumented to report key characteristics of its communications processes. The resulting traces give insight into how a PVM program actually works, and can be used to drive simulations of the system. This approach is promising in that the traces generated may be used to simulate any LAN and predict the possible performance changes. We experimented with changing the type of network connecting a workstation cluster, but the same traces could drive simulations that use custom-designed Application Programmer Interfaces (APIs) [20, 21] or other protocols instead of TCP in task-to-task communications, or run on a massively parallel processor. Alternatively, one could reduce or even remove entirely the overhead associated with PVM processing and see the performance improvements.

The version of PVM we instrumented was 3.3.7, and we ran our tests and simulations on the SUN4SOL2 class architecture exclusively. This does not necessarily limit the instrumentation's effectiveness to that environment, however. PVM source code was designed for platform independence. There is a common set of source code files, and compile-time flags tell the system about the target architecture and environment. Our instrumentation should run just as effectively on any computer architecture that supports PVM. Because we use clock_gettime, a POSIX-4 function, in the instrumentation, however, the target architecture must support

the posix4 library.

In future work we would like to extend this implementation to a more current version of PVM, and test it on other architecture classes. There are other areas to address as well. Because our instrumented code prints trace lines directly, trace files generated are very large and the instrumentation somewhat intrusive. The instrumentation could be revised to output the traces in a compact, encoded form that can be processed later to create readable traces. Furthermore, it may be preferable to rewrite the instrumentation so that trace writes occur when a PVM task is supposed to be blocked for a receive operation. These approaches are complementary and minimize the extent of the intrusion caused by the instrumentation. Also, we do not account for the effect of context switching, and the instrumentation environment is not applicable to latency-dependent computations.

Future directions for research include addressing these improvements and adding simulations of other network types. In addition, we plan to investigate using these traces to predict the performance of workstation clusters connected by various hybrid or conceptual networks, and to estimate the performance improvement in workstation clusters as more and more machines are added to the PVM configuration.

Appendix A

Simulation Results

All time values are listed in seconds. Messages, bytes, and calls are literal numbers.

A.1 Benchmark: mg

10 Mbps Ethernet

Station	Exec. Time	Block Time			Net. Travel	Overhead		Msgs	Bytes
		Total	Serv.	Alg.		PVM	TCP		
0	324.4	218.3	9.9	208.4	9.2	8.1	0.6	1717	6426992
1	311.4	117.4	5.6	111.9	7.9	12.8	0.6	1729	6430540
2	326.2	3.1	2.5	0.6	7.6	25.0	0.6	1729	6430540
3	324.4	217.0	9.2	207.8	9.7	9.2	0.6	1709	6426700
Avg of 4	321.6	139.0	6.8	132.2	8.6	13.8	0.6	1721	6428693

5174 Collisions on ethernet

Block Time in the Actual Execution

Proc 0	Proc 1	Proc 2	Proc 3
223.4	134.8	5.6	221.4

100 Mbps Ethernet

Station	Exec. Time	Block Time			Net. Travel	Overhead		Msgs	Bytes
		Total	Serv.	Alg.		PVM	TCP		
0	322.3	216.2	0.8	215.4	0.6	8.1	0.6	1717	6426992
1	309.0	115.0	0.3	114.7	0.6	12.8	0.6	1729	6430540
2	324.0	0.9	0.0	0.9	0.6	25.0	0.6	1729	6430540
3	322.3	214.9	0.8	214.1	0.6	9.2	0.6	1709	6426700
Avg of 4	319.4	136.7	0.5	136.3	0.6	13.8	0.6	1721	6428693

514 Collisions on ethernet

PVM message overheads:

Processor 0	
PVM function	Time (calls)
initsend	0.085 (209)
send	4.813 (208)
mcast	0.009 (1)
pkstr	0.000 (4)
pkdouble	1.594 (208)
pkint	0.000 (17)
upkdouble	1.503 (227)
upkint	0.000 (39)
upkstr	0.000 (1)
nrecv	0.000 (1)
recv	223.455 (215)

Processor 1	
PVM function	Time (calls)
initsend	0.523 (221)
send	6.084 (221)
pkdouble	2.362 (225)
pkint	0.000 (9)
upkdouble	3.528 (216)
upkint	0.000 (16)
upkstr	0.000 (2)
nrecv	0.000 (1)
recv	135.091 (217)

Processor 2	
PVM function	Time (calls)
initsend	2.571 (221)
send	8.487 (221)
pkdouble	3.389 (225)
pkint	0.015 (9)
upkdouble	9.617 (216)
upkint	0.000 (16)
upkstr	0.000 (2)
nrecv	0.000 (1)
recv	6.482 (217)

Processor 3	
PVM function	Time (calls)
initsend	0.099 (201)
send	5.818 (201)
pkdouble	1.723 (205)
pkint	0.000 (9)
upkdouble	1.410 (204)
upkint	0.001 (16)
upkstr	0.000 (2)
nrecv	0.000 (1)
recv	221.483 (205)

A.2 Benchmark: cg

10 Mbps Ethernet

Station	Exec. Time	Block Time			Net. Travel	Overhead		Msgs	Bytes
		Total	Serv.	Alg.		PVM	TCP		
0	37.4	7.2	4.8	2.4	10.2	11.5	2.0	3991	4504264
1	37.3	18.6	7.8	10.8	7.2	5.3	1.0	1591	2235180
2	37.4	16.0	5.5	10.4	9.6	6.7	1.1	2371	4439460
3	37.4	15.1	4.9	10.2	6.2	6.6	1.0	1591	2235180
Avg of 4	37.4	14.2	5.8	8.4	8.3	7.5	1.3	2386	3353521

8030 Collisions on ethernet

Block Time in the Actual Execution

Proc 0	Proc 1	Proc 2	Proc 3
17.4	28.4	25.7	24.9

100 Mbps Ethernet

Station	Exec. Time	Block Time			Net. Travel	Overhead		Msgs	Bytes
		Total	Serv.	Alg.		PVM	TCP		
0	31.6	1.4	0.1	1.2	0.5	11.5	2.0	3991	4504264
1	31.5	12.8	0.6	12.2	0.2	5.3	1.0	1591	2235180
2	31.6	10.2	0.5	9.7	0.4	6.7	1.1	2371	4439460
3	31.6	9.3	0.5	8.8	0.3	6.6	1.0	1591	2235180
Avg of 4	31.6	8.4	0.4	8.0	0.4	7.5	1.3	2386	3353521

725 Collisions on ethernet

PVM message overheads:

Processor 0	
PVM function	Time (calls)
initsend	0.588 (3211)
send	8.271 (3210)
mcast	0.011 (1)
pkstr	0.000 (3)
pkdouble	1.006 (3210)
pkint	0.018 (405)
upkdouble	0.502 (2829)
upkint	0.000 (15)
upkstr	0.000 (1)
nrecv	0.000 (1)
recv	18.511 (2823)

Processor 1	
PVM function	Time (calls)
initsend	0.210 (1201)
send	2.993 (1201)
pkdouble	0.465 (1203)
pkint	0.000 (1)
upkdouble	0.973 (1590)
upkint	0.006 (404)
upkstr	0.000 (1)
nrecv	0.000 (1)
recv	29.085 (1591)

Processor 2	
PVM function	Time (calls)
initsend	0.278 (1591)
send	4.491 (1591)
pkdouble	0.941 (1593)
pkint	0.006 (391)
upkdouble	0.460 (1200)
upkint	0.000 (14)
upkstr	0.000 (1)
nrecv	0.000 (1)
recv	26.182 (1201)

Processor 3	
PVM function	Time (calls)
initsend	0.217 (1201)
send	4.237 (1201)
pkdouble	0.462 (1203)
pkint	0.000 (1)
upkdouble	0.914 (1590)
upkint	0.006 (404)
upkstr	0.000 (1)
nrecv	0.000 (1)
recv	25.649 (1591)

A.3 Benchmark: is

10 Mbps Ethernet

Station	Exec. Time	Block Time			Net. Travel	Overhead		Msgs	Bytes
		Total	Serv.	Alg.		PVM	TCP		
0	260.8	205.0	19.2	185.8	18.1	13.2	1.1	3629	14566444
1	260.3	229.3	22.7	206.6	16.4	11.5	0.9	2852	11419876
2	263.7	29.6	4.0	25.6	18.6	49.8	1.5	5022	20214820
3	259.9	202.5	16.3	186.2	19.6	24.9	1.5	5012	20172676
Avg of 4	261.2	166.6	15.5	151.1	18.2	24.8	1.3	4128	16593454

7189 Collisions on ethernet

Block Time in the Actual Execution

Proc 0	Proc 1	Proc 2	Proc 3
229.2	250.7	50.5	224.5

100 Mbps Ethernet

Station	Exec. Time	Block Time			Net. Travel	Overhead		Msgs	Bytes
		Total	Serv.	Alg.		PVM	TCP		
0	256.8	200.9	1.6	199.3	1.3	13.2	1.1	3629	14566444
1	256.2	225.2	1.8	223.3	1.1	11.5	0.9	2852	11419876
2	259.7	25.6	0.2	25.4	1.8	49.8	1.5	5022	20214820
3	255.8	198.4	1.7	196.7	1.8	24.9	1.5	5012	20172676
Avg of 4	257.1	162.5	1.3	161.2	1.5	24.8	1.3	4128	16593454

613 Collisions on ethernet

PVM message overheads:

Processor 0	
PVM function	Time (calls)
initsend	0.096 (105)
send	7.027 (93)
mcast	0.083 (12)
pkstr	0.000 (3)
pkint	3.544 (144)
upkdouble	0.000 (15)
upkint	2.340 (137)
upkstr	0.000 (1)
nrecv	0.000 (1)
recv	229.228 (96)

Processor 1	
PVM function	Time (calls)
initsend	0.088 (92)
send	5.815 (92)
pkdouble	0.000 (5)
pkint	2.794 (98)
upkint	2.664 (139)
upkstr	0.000 (1)
nrecv	0.003 (1)
recv	250.826 (123)

Processor 2	
PVM function	Time (calls)
initsend	2.012 (102)
send	16.752 (92)
mcast	0.521 (10)
pkdouble	0.656 (5)
pkint	5.898 (128)
upkint	19.485 (129)
upkstr	0.000 (1)
nrecv	0.003 (1)
recv	54.963 (103)

Processor 3	
PVM function	Time (calls)
initsend	0.152 (102)
send	13.442 (92)
mcast	0.105 (10)
pkdouble	0.000 (5)
pkint	5.659 (128)
upkint	5.291 (139)
upkstr	0.000 (1)
nrecv	0.003 (1)
recv	224.700 (113)

A.4 Benchmark: ft

10 Mbps Ethernet

Station	Exec.	Block Time			Net.	Overhead		Msgs	Bytes
	Time	Total	Serv.	Alg.	Travel	PVM	TCP		
0	79.1	33.6	33.3	0.3	37.0	11.9	0.8	2731	11054464
1	79.1	35.1	34.8	0.2	35.5	10.7	0.8	2737	11054760
2	79.0	32.9	32.7	0.2	37.1	12.1	0.8	2737	11054760
3	79.1	35.1	34.7	0.4	35.9	10.7	0.8	2737	11054760
Avg of 4	79.1	34.2	33.9	0.3	36.4	11.3	0.8	2735	11054686

32124 Collisions on ethernet

Block Time in the Actual Execution

Proc 0	Proc 1	Proc 2	Proc 3
33.3	34.0	32.1	34.1

100 Mbps Ethernet

Station	Exec.	Block Time			Net.	Overhead		Msgs	Bytes
	Time	Total	Serv.	Alg.	Travel	PVM	TCP		
0	47.3	1.8	0.4	1.4	2.1	11.9	0.8	2731	11054464
1	47.3	3.3	1.0	2.3	2.0	10.7	0.8	2737	11054760
2	47.3	1.1	0.3	0.8	2.2	12.1	0.8	2737	11054760
3	47.3	3.4	0.8	2.6	2.0	10.7	0.8	2737	11054760
Avg of 4	47.3	2.4	0.6	1.8	2.1	11.3	0.8	2735	11054686

9325 Collisions on ethernet

PVM message overheads:

Processor 0	
PVM function	Time (calls)
initsend	0.085 (43)
send	6.320 (42)
mcast	0.007 (1)
pkstr	0.000 (3)
pkdcplx	2.891 (42)
pkint	0.000 (15)
upkdcplx	2.490 (60)
upkdouble	0.000 (12)
upkint	0.000 (18)
upkstr	0.000 (1)
nrecv	0.000 (1)
recv	33.392 (63)

Processor 1	
PVM function	Time (calls)
initsend	0.072 (49)
send	5.471 (49)
pkdcplx	2.577 (48)
pkdouble	0.000 (4)
pkint	0.000 (2)
upkdcplx	2.423 (42)
upkint	0.001 (14)
upkstr	0.000 (1)
nrecv	0.003 (1)
recv	34.221 (43)

Processor 2	
PVM function	Time (calls)
initsend	0.071 (49)
send	6.536 (49)
pkdcplx	2.787 (48)
pkdouble	0.000 (4)
pkint	0.000 (2)
upkdcplx	2.438 (42)
upkint	0.001 (14)
upkstr	0.000 (1)
nrecv	0.004 (1)
recv	32.324 (43)

Processor 3	
PVM function	Time (calls)
initsend	0.074 (49)
send	5.455 (49)
pkdcplx	2.610 (48)
pkdouble	0.000 (4)
pkint	0.000 (2)
upkdcplx	2.443 (42)
upkint	0.001 (14)
upkstr	0.000 (1)
nrecv	0.002 (1)
recv	34.220 (43)

A.5 Benchmark: lu

10 Mbps Ethernet

Station	Exec.	Block Time			Net.	Overhead		Msgs	Bytes
	Time	Total	Serv.	Alg.	Travel	PVM	TCP		
0	69.5	68.8	0.0	68.8	0.0	0.1	0.0	8	784
1	69.5	14.4	1.4	13.0	2.3	17.2	3.3	5627	993940
2	69.1	14.1	1.8	12.2	2.3	14.4	3.2	5317	971908
3	69.1	14.4	2.1	12.4	2.3	14.2	3.2	5317	971876
4	69.4	6.9	1.4	5.5	2.4	19.3	3.2	5319	972164
Avg of 4	69.3	12.5	1.7	10.8	2.3	16.3	3.2	5395	977472

1343 Collisions on ethernet

Block Time in the Actual Execution

Proc 0	Proc 1	Proc 2	Proc 3	Proc 4
77.7	23.0	22.6	23.1	15.7

100 Mbps Ethernet

Station	Exec. Time	Block Time			Net. Travel	Overhead		Msgs	Bytes
		Total	Serv.	Alg.		PVM	TCP		
0	68.7	68.0	0.0	68.0	0.0	0.1	0.0	8	784
1	68.6	13.6	0.4	13.2	0.2	17.2	3.3	5627	993940
2	68.3	13.2	0.5	12.7	0.2	14.4	3.2	5317	971908
3	68.3	13.6	0.6	13.0	0.2	14.2	3.2	5317	971876
4	68.6	6.1	0.4	5.7	0.2	19.3	3.2	5319	972164
Avg of 4	68.5	11.6	0.5	11.2	0.2	16.3	3.2	5395	977472

273 Collisions on ethernet

PVM message overheads:

Processor 0	
PVM function	Time (calls)
initsend	0.002 (8)
send	0.010 (8)
pkstr	0.000 (3)
pkdouble	0.000 (4)
pkint	0.001 (16)
upkbyte	0.000 (8)
upkdouble	0.003 (124)
upkint	0.000 (23)
upkstr	0.000 (1)
nrecv	0.000 (1)
recv	77.789 (110)

Processor 1	
PVM function	Time (calls)
initsend	1.054 (5523)
send	12.399 (5523)
pkdouble	0.325 (5528)
pkint	0.000 (5)
upkbyte	0.000 (1)
upkdouble	0.340 (5423)
upkint	0.000 (11)
upkstr	0.000 (1)
nrecv	0.000 (1)
precv	0.000 (1)
recv	26.106 (5425)

Processor 2	
PVM function	Time (calls)
initsend	1.106 (5213)
send	10.677 (5213)
pkdouble	0.288 (5216)
pkint	0.000 (5)
upkbyte	0.000 (1)
upkdouble	0.283 (5211)
upkint	0.003 (11)
upkstr	0.000 (1)
nrecv	0.000 (1)
precv	0.000 (1)
recv	24.653 (5213)

Processor 3	
PVM function	Time (calls)
initsend	1.255 (5213)
send	10.360 (5213)
pkdouble	0.297 (5216)
pkint	0.000 (5)
upkbyte	0.000 (1)
upkdouble	0.296 (5211)
upkint	0.000 (11)
upkstr	0.000 (1)
nrecv	0.000 (1)
precv	0.000 (1)
recv	25.110 (5213)

Processor 4	
PVM function	Time (calls)
initsend	1.080 (5215)
send	13.851 (5215)
pkdouble	0.566 (5218)
pkint	0.000 (5)
upkbyte	0.000 (1)
upkdouble	0.418 (5209)
upkint	0.001 (11)
upkstr	0.000 (1)
nrecv	0.001 (1)
precv	0.000 (1)
recv	19.092 (5211)

A.6 Benchmark: sp

10 Mbps Ethernet

Station	Exec.	Block Time			Net.	Overhead		Msgs	Bytes
	Time	Total	Serv.	Alg.	Travel	PVM	TCP		
0	381.7	380.8	0.0	380.7	0.0	0.1	0.0	20	848
1	381.6	129.5	24.9	104.6	38.1	103.5	14.0	26921	37535888
2	381.6	30.3	17.7	12.7	58.1	171.1	20.0	40613	56159536
3	381.6	40.9	18.4	22.6	58.4	162.1	20.0	40613	56159536
4	381.6	95.2	20.5	74.7	39.9	125.9	13.7	26409	37495120
Avg of 4	381.6	74.0	20.4	53.6	48.6	140.7	16.9	33639	46837520

27133 Collisions on ethernet

Block Time in the Actual Execution

Proc 0	Proc 1	Proc 2	Proc 3	Proc 4
391.9	143.7	47.5	55.2	110.7

100 Mbps Ethernet

Station	Exec.	Block Time			Net.	Overhead		Msgs	Bytes
	Time	Total	Serv.	Alg.	Travel	PVM	TCP		
0	364.7	363.8	0.0	363.8	0.0	0.1	0.0	20	848
1	364.7	112.5	3.9	108.6	2.4	103.5	14.0	26921	37535888
2	364.6	13.4	0.5	12.9	3.9	171.1	20.0	40613	56159536
3	364.6	24.0	1.0	23.0	3.9	162.1	20.0	40613	56159536
4	364.6	78.2	2.9	75.4	2.5	125.9	13.7	26409	37495120
Avg of 4	364.6	57.0	2.1	55.0	3.2	140.7	16.9	33639	46837520

2177 Collisions on ethernet

PVM message overheads:

Processor 0	
PVM function	Time (calls)
initsend	0.002 (4)
send	0.030 (12)
psend	0.047 (8)
pkstr	0.000 (3)
pkbyte	0.001 (8)
pkdouble	0.000 (4)
pkint	0.000 (8)
upkdouble	0.005 (126)
upkint	0.001 (27)
upkstr	0.000 (1)
nrecv	0.000 (1)
recv	391.913 (114)

Processor 1	
PVM function	Time (calls)
initsend	6.046 (26719)
send	71.752 (26719)
pkdouble	10.336 (26722)
pkint	0.000 (4)
upkdouble	9.758 (29012)
upkint	0.000 (11)
upkstr	0.000 (1)
nrecv	0.000 (1)
recv	149.305 (29014)

Processor 2	
PVM function	Time (calls)
initsend	9.586 (40209)
send	133.711 (40209)
pkdouble	18.102 (40212)
pkint	0.000 (4)
upkdouble	13.911 (37808)
upkint	0.000 (11)
upkstr	0.000 (1)
nrecv	0.001 (1)
recv	43.373 (37810)

Processor 3	
PVM function	Time (calls)
initsend	9.296 (40209)
send	123.577 (40209)
pkdouble	16.808 (40212)
pkint	0.000 (4)
upkdouble	13.836 (37808)
upkint	0.000 (11)
upkstr	0.000 (1)
nrecv	0.000 (1)
recv	53.770 (37810)

Processor 4	
PVM function	Time (calls)
initsend	6.377 (26207)
send	95.834 (26207)
pkdouble	11.143 (26210)
pkint	0.000 (4)
upkdouble	10.810 (28606)
upkint	0.000 (11)
upkstr	0.000 (1)
nrecv	0.000 (1)
recv	112.465 (28608)

A.7 Benchmark: bt

10 Mbps Ethernet

Station	Exec. Time	Block Time			Net. Travel	Overhead		Msgs	Bytes
		Total	Serv.	Alg.		PVM	TCP		
0	524.6	523.5	0.0	523.4	0.0	991.3	0.0	20	848
1	524.6	121.5	24.5	97.0	29.4	84.7	11.2	22281	35993488
2	524.6	41.8	19.7	22.2	49.9	128.1	15.1	31813	56568496
3	524.5	47.8	19.7	28.1	50.3	126.1	15.1	31813	56568496
4	524.6	94.3	19.9	74.4	29.2	101.1	11.0	21969	35968720
Avg of 4	524.6	76.3	20.9	55.4	39.7	110.0	13.1	26969	46274800

24397 Collisions on ethernet

Block Time in the Actual Execution

Proc 0	Proc 1	Proc 2	Proc 3	Proc 4
529.9	129.1	51.3	57.2	102.6

100 Mbps Ethernet

Station	Exec. Time	Block Time			Net. Travel	Overhead		Msgs	Bytes
		Total	Serv.	Alg.		PVM	TCP		
0	504.3	503.1	0.0	503.1	0.0	991.3	0.0	20	848
1	504.2	101.1	3.9	97.3	2.0	84.7	11.2	22281	35993488
2	504.2	21.5	0.5	21.0	3.6	128.1	15.1	31813	56568496
3	504.2	27.5	0.9	26.6	3.6	126.1	15.1	31813	56568496
4	504.2	73.9	3.0	71.0	1.9	101.1	11.0	21969	35968720
Avg of 4	504.2	56.0	2.0	54.0	2.8	110.0	13.1	26969	46274800

2109 Collisions on ethernet

PVM message overheads:

Processor 0	
PVM function	Time (calls)
initsend	0.001 (4)
send	0.024 (12)
psend	0.044 (8)
pkstr	0.000 (3)
pkbyte	0.000 (8)
pkdouble	0.000 (4)
pkint	0.000 (8)
upkdouble	0.003 (86)
upkint	0.001 (27)
upkstr	0.000 (1)
nrecv	0.000 (1)
recv	520.080 (73)

Processor 1	
PVM function	Time (calls)
initsend	4.220 (22159)
send	55.739 (22158)
pkdouble	10.122 (22162)
pkint	0.000 (4)
upkdouble	11.627 (24492)
upkint	0.000 (11)
upkstr	0.000 (1)
nrecv	0.000 (1)
recv	132.035 (24494)

Processor 2	
PVM function	Time (calls)
initsend	7.234 (31569)
send	97.281 (31569)
pkdouble	15.907 (31571)
pkint	0.000 (4)
upkdouble	12.611 (29168)
upkint	0.000 (11)
upkstr	0.000 (1)
nrecv	0.000 (1)
recv	46.310 (29170)

Processor 3	
PVM function	Time (calls)
initsend	6.890 (31569)
send	94.457 (31569)
pkdouble	16.163 (31572)
pkint	0.000 (4)
upkdouble	12.665 (29168)
upkint	0.000 (11)
upkstr	0.000 (1)
nrecv	0.000 (1)
recv	53.096 (29170)

Processor 4	
PVM function	Time (calls)
initsend	6.280 (21847)
send	75.528 (21847)
pkdouble	9.053 (21850)
pkint	0.000 (4)
upkdouble	9.717 (24246)
upkint	0.000 (11)
upkstr	0.000 (1)
nrecv	0.000 (1)
recv	103.127 (24248)

Bibliography

[1] G. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. PVM: Parallel Virtual Machine: A User's Guide and Tutorial for Networked Parallel Computing. The MIT Press, 1994.

[2] V. S. Sunderam. PVM: A Framework for Parallel Distributed Computing, Concurrency: Practice and Experience, Vol. 2 No. 4. 1990. pp 315- 339.

[3] G. A. Geist and V. S. Sunderam. The Evolution of the PVM Concurrent Computing System. IEEE Computer, 1993. pp 549-557.

[4] A. S. Tanenbaum. Computer Networks. 3rd ed. Prentice Hall PTR, Upper Saddle River, New Jersey, 1996.

[5] W. R. Stevens. TCP/IP Illustrated, Vol I. Addison-Wesley Publishing Company, Reading, Massachusetts, 1994.

[6] Sun OS 5.3 Reference Manual, Rev A. Sun Microsystems, Inc. Mountain View, California, 1993.

[7] H. Casanova, J. Dongarra, and W. Jiang. The Performance of PVM on MPP Systems. Technical Report CS-95-301, University of Tennessee, 1995.

[8] J. A. Kohl and G. A. Geist. The PVM 3.4 Tracing Facility and XPVM. Proceedings of the 29th Hawaii International Conference on System Sciences. Maui, Hawaii, 1996.

[9] D. H. Bailey, E. Barszcz, L. Dagum, and H. D. Simon. NAS Parallel Benchmark Results. IEEE Parallel and Distributed Technology, Vol. 1, No. 1, 1993. pp 43-51.

[10] PVM-NAS Source Code Documentation and Readme files. Emory University, Tennessee. Obtained from http://www/mathcs.emory.edu/~vss/.

[11] AIMS: An Automated Instrumentation and Monitoring System. Obtained from http://www.nas.nasa.gov/NAS/Tools/Projects/AIMS/.

[12] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn Parallel Performance Measurement Tools. IEEE Computer, Vol 28, No. 11, 1995.

[13] Introduction to CSIM17 for C Programmers. Mesquite Software, Inc. Austin, Texas, 1994.

[14] M. H. MacDougall. Simulating Computer Systems, Techniques and Tools. The MIT Press, Cambridge, Massachusetts, 1992. pp 160-188.

[15] D. D. Clark, V. Jacobson, J. Romkey, and H. Salwen. An Analysis of TCP Processing Overhead. IEEE Communications Magazine, Vol. 27, No. 6, 1989. pp 23-36.

[16] K. Zielinski, M. Gajecki, and G. Czajkowski. Parallel Programming Systems For LAN Distributed Computing. Proceedings of the 14th International Conference on Distributed Computing Systems. IEEE Computer Society Press, Los Alamitos, California, 1994. pp 600-607.

[17] K. Castagnera, D. Cheng, R. Fatoohi, E. Hook, B. Kramer, C. Manning, J. Musch, C. Niggley, W. Saphir, D. Sheppard, M. Smith, I. Stockdale, S. Welch, R. Williams, and D. Yip. NAS Experiences with a Prototype Cluster of Workstations. Proceedings of Supercomputing '94. IEEE Computer Society Press, Los Alamitos, California, 1994. pp 410-419.

[18] T. Mattson. Programming Environments for Parallel Computing: A Comparison of CPS, Linda, P4, PVM, POSYBL, and TCGMSG. IEEE Proceedings of the Twenty-Seventh Annual Hawaii International Conference on System Sciences, Vol II. IEEE Computer Society Press, Los Alamitos, California, 1994, pp 586-594.)

[19] J. J. Dongarra and T. Dunigan. Message-Passing Performance of Various Computers. Technical Report, Oak Ridge National Laboratory. Obtained from http://www.netlib.org/utk/papers/commperf.ps.

[20] J. Bruck, D. Dolev, C-T. Ho, M-C. Rosu, and R. Strong. Efficient Message Passing Interface (MPI) for Parallel Computing on Clusters of Workstations. Technical Report, obtained from http://cs-tr.cs.cornell.edu/.

[21] P. L. Vaughan, A. Skjellum, D. S. Reese, F-C. Cheng. Migrating from PVM to MPI, part I: The Unify System, Proceedings, Frontiers '95: The Fifth Symposium on the Frontiers of Massively Parallel Computation. IEEE Computer Society Press, Los Alamitos, California. pp 488-495. (1994)

Vita

Sarah Elizabeth Zabel was born in Burbank, California, on July 9, 1965, the daughter of Patrick and Martha Zabel. After completing her work at Devine High School, Devine, Texas, in 1983, she entered the United States Air Force Academy in Colorado Springs, Colorado. She received the degree of Bachelor of Science in Computer Science from the United States Air Force Academy in May, 1987. Since that time, she has been a commisioned officer in the United States Air Force, and was stationed at Fort George G. Meade, Maryland and Misawa Air Base, Japan, before arriving at Kelly Air Force Base, San Antonio, Texas. In August, 1993, she entered the graduate program in computer science at the University of Texas at San Antonio.

Permanent Address: Route 1 Box 397D, Devine, Texas 78016

This thesis was typeset by Sarah Zabel.