To the Graduate Council:

I am submitting herewith a thesis written by Nirmala Kodali entitled "Implementation of Channel Interface for PVM". I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Micah Beck, Major Professor

We have read this thesis
and recommend its acceptance:

Accepted for the council:

Associate Vice Chancellor and
Dean of the Graduate School

IMPLEMENTATION OF A CHANNEL INTERFACE FOR PARALLEL VIRTUAL MACHINE

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Nirmala Kodali

August 1998

ACKNOWLEDGEMENTS

The author wishes to express her deep gratitude to her major professor, Dr. M. Beck for his guidance and encouragement throughout the course of this research. The author is also indebted to her committee members, Dr. Jack Dongarra and Dr. Robert Ward, for their cooperation and help.

The author wishes to thank the staff of the Department of Computer Science and her fellow students for their help.

The financial support provided by the Computer Science department is gratefully acknowledged.

ABSTRACT

This thesis presents the implementation of a *channel interface* for PVM. PVM is a software system built around the concept of virtual machine, which is a dynamic collection of computational resources managed as a single parallel computer. As the collection of available resources can include a wide range of architecture types such as PCs, MPPs, workstations, etc., portability was considered much more important than performance in the design of PVM. One interface that can be used to simplify porting PVM to new systems is the *channel interface*. This thesis describes the *channel interface* and the porting of PVM to the channel interface. The channel interface we used is the channel interface developed for the MPICH implementation of MPI. A channel represents the front end for receiving or sending data toward an underlying transport. A set of function definitions that describe the channel is called the channel interface. The user-callable standard PVM message passing functions like pvm_send() and pvm_recv() can be expressed in terms of these functions. The channel interface can be implemented in several different ways. This thesis describes the implementation of the channel interface in terms of direct TCP/IP socket calls in a workstation environment and in terms of native message passing of the massively parallel processor (MPP) system in an MPP environment. In the channel layer, messages are sent in two parts: the control part, containing information on PVM message size, tag etc., and the data part, containing the actual data. Separate routines are used to send and receive the control and data parts. The performance of PVM with and without the channel interface has been presented. It is measured in terms of bandwidth and latency.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Parallel Virtual Machine (PVM)[1] is a software package built around the concept of a *virtual machine*. A *virtual machine* is a set of heterogeneous hosts connected by a network that appears to the users as one large parallel computer. Hence, PVM software allows a collection of computers connected to a network to be used as a single, large parallel distributed computer.

1.1 Heterogeneous Distributed Computing

The process by which a set of computers, connected by a network, is used collectively to solve a single large problem is called distributed computing. If the set of computers includes a wide range of architecture types, it is called heterogeneous

distributed computing. Previously, massively parallel processors (MPP)[2] have been used to solve problems that require large computational power. But over the past decade, researchers have expressed widespread interest in distributed computing. Several factors, including improvement in network speeds over the past few years and the realization that the combined computational resources of a collection of computers connected by high speed networks may exceed the power of a single high performance computer, have spurred this interest[3]. Some of the advantages of heterogeneous distributed computing are:

1. It is cost effective because little or no additional hardware needs to be procured.
2. Optimum performance can be obtained by assigning each task to the most appropriate architecture[1].
3. User-level or program-level fault tolerance can be implemented with little effort either in the application or in the underlying operating system[1].
4. By using heterogeneous distributed computing one can take advantage of the latest computational and network technologies.

Influenced by these advantages, more and more scientists are turning toward heterogeneous distributed computing to solve their computational problems.

In distributed computing, data must be exchanged between cooperating tasks. Research groups have developed several message passing environments to assist programmers using distributed computing. The PVM system is one of them. The PVM message passing model allows programmers to take advantage of distributed computing across a wide variety of computer types. A key concept in PVM is that it makes a collection of

computers appear to the user as one large *virtual* machine.

1.2 Motivation

The primary motivation for this thesis is to make PVM software easily portable to new systems without affecting its performance. As already discussed, PVM is built around the concept of a *virtual machine,* which is a dynamic collection of computational resources managed as a single parallel computer. Since the collection of computers available can include a wide range of architecture types such as PCs, workstations, shared multiprocessors, vector supercomputers, and even large MPP systems[2], portability was considered much more important than performance in PVM. The portability achieved by PVM means that a program written for one architecture can be copied to a second architecture, compiled and executed without modification, and the resulting PVM executables can communicate with each other. In other words, a PVM program can be ported heterogeneously to run co-operatively across a broad collection of different architectures at the same time.

As portability is an important feature for PVM, research work is being conducted to improve the portability of PVM to new systems. One interface that can be used to simplify porting PVM to new systems is the *channel interface*. This thesis describes the implementation of a *channel interface*[4] and porting of PVM to this channel interface. A *channel* represents the front end for receiving or sending data toward the underlying transport. A set of function definitions that describe the channel is called a *channel interface*. In

other words, the channel interface defines a collection of simple data transfer operations that are based on the simple ability to send data from one process to another. The design of our channel interface for PVM is based on the channel interface for the MPICH[5] implementation of MPI[6]. The functions defined in the channel interface establish socket connections and read data coming in through a socket or write data to a socket. The PVM code uses these simple operations to implement higher level operations such as pvm_send and pvm_recv.

1.3 Thesis Overview

The next chapter gives a brief description of the implementation of PVM software. The basic components and communication model of PVM, message format, and message routing in PVM are also addressed. Chapter 3 describes the channel interface and porting of PVM to the channel interface. The set of function definitions that form the channel interface is also described in this chapter.

Chapter 4 compares the performance of PVM with and without the channel interface. Test programs were run on SUN workstations and IBM SP2[10]. The objective of these test programs was to measure bandwidth and latency. The performance of the test programs on the above two platforms is determined and an analysis of the findings is presented.

The last chapter summarizes the work done in this thesis and outlines future work on this subject.

CHAPTER II

PVM:PARALLEL VIRTUAL MACHINE

In order to get a better understanding of the design trade-offs involved in the implementation of a channel interface for PVM, it is necessary to first understand the basic concepts behind PVM. This chapter gives a brief description of the implementation of the PVM software and the reasons behind the basic design decisions.

2.1 Introduction

As discussed in the first chapter, PVM is a software system, developed at the University of Tennessee, Knoxville and the Oak Ridge National Laboratory, that permits a heterogeneous collection of UNIX computers linked together by a network, to be used as a single, large parallel computer. The PVM system is composed of two parts:

1. The PVM daemon[1], called *pvmd*, resides on all computers making up the vir-

tual machine. Any user with a valid login can install this daemon on a machine.

2. The library of PVM interface routines called *libpvm*, contains user-callable routines for message passing, spawning processes, coordinating tasks and modifying the virtual machine.

The partial anatomy of PVM is shown in Figure 2.1[8]. The PVM computing model is based on the notion that an application consists of several tasks that can be executed concurrently. Functional parallelism is achieved when each task performs a different function, and data parallelism is achieved when multiple tasks perform the same operations on different data.

2.2 PVM Daemon

One pvmd runs on each host of the virtual machine. The first pvmd is designated the master and the rest are called slaves. During normal operation all are considered equal, but only the master pvmd can start new slaves as well as delete slaves from the virtual machine.

A pvmd owned by one user does not interact with those owned by others in order to reduce security risks, and each pvmd maintains a list of all tasks under its management. It serves as a message router and controller, and provides a point of contact, authentication, process control, and fault detection. An idle pvmd occasionally checks if its peers are still running.

At start-up, a pvmd configures itself as a master or slave depending on its

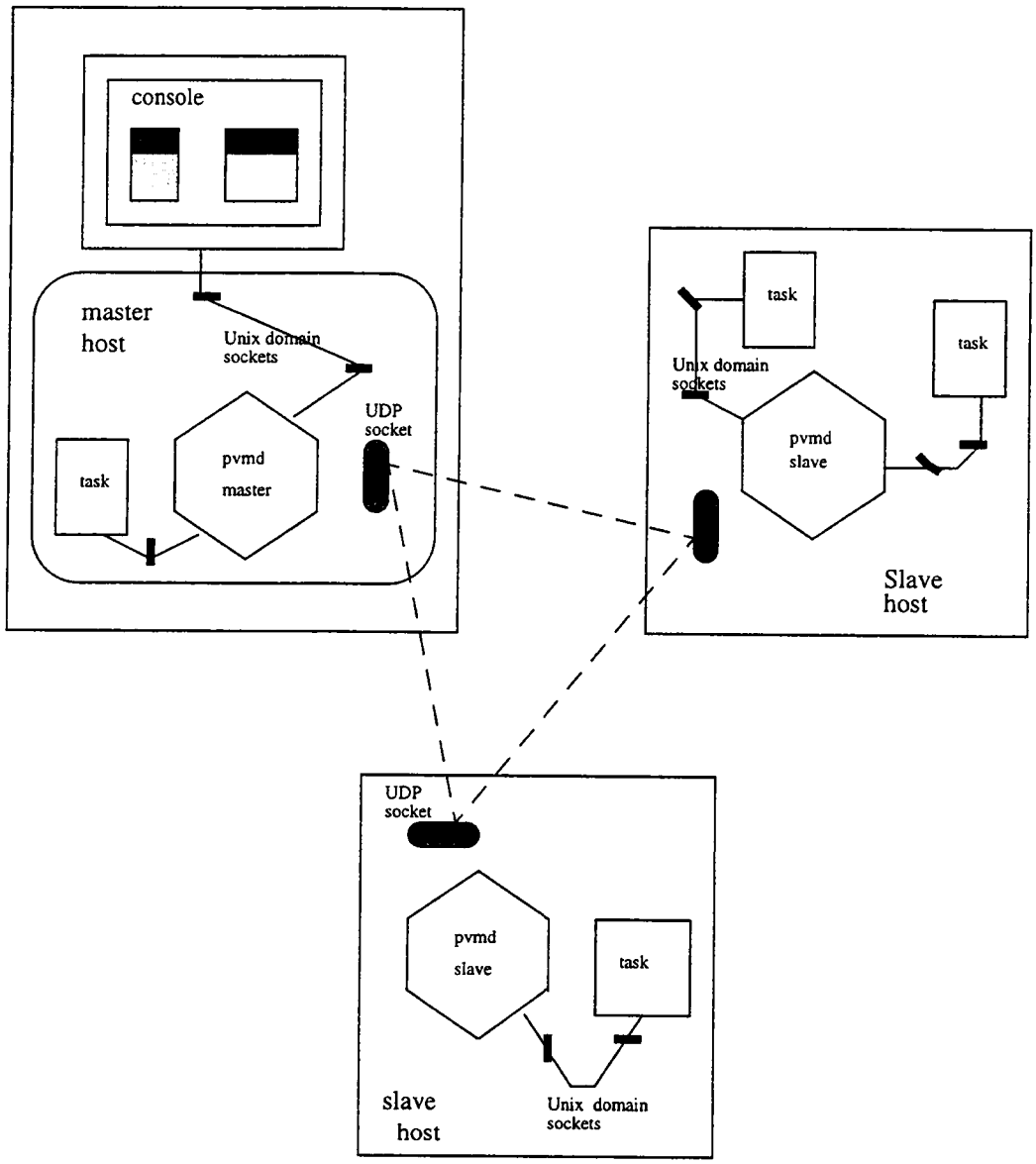


Figure 2.1. Partial anatomy of PVM.

command line arguments. It creates and binds sockets to tasks and other pvmds. After configuration, pvmd calls select(), which probes all sources of its input. Packets are received and routed to send queues. Messages to pvmd are reassembled and passed to the entry points.

A pvmd shuts down when it is deleted from the virtual machine, or when it is killed or when it looses contact with the master pvmd. When a pvmd shuts down it takes two final actions. First, it kills any tasks running under it with the signal SIGTERM. Second, it sends a final shutdown message to every other pvmd in its host table. If a slave pvmd is shut down, the virtual machine still continues to exist. On the other hand, if the master pvmd is shut down, all the slaves also have to shut down and the virtual machine does not exist any more.

2.3 Libpvm

The libpvm is a library that allows a task to interface with the pvmd and other tasks. It is written in C and supports C, C++, and Fortran applications. It contains functions for packing and unpacking messages and functions to perform PVM syscalls by using the message functions to send service requests to pvmd. Any executable using PVM should link to this library.

The upper layer of the libpvm library, including most of the programming interface functions, is machine independent. The bottom layer is kept separate and can be modified or replaced with a new machine specific file when ported to a new system.

2.4 The Communication Model

The PVM communication model provides asynchronous blocking sends and blocking and nonblocking receives. Therefore, the user does not have to worry either about any deadlocks for nonmatching pairs of send-receive or about rewriting into a buffer after it has been sent.

The communication model also assumes that any task can send a message to any other PVM task. There is no limit to the number of messages and no limit on their size. The communication does not restrict itself to a particular machine's limitations and always assumes that sufficient memory is available. The message buffers are allocated dynamically. Therefore, the maximum message size that can be sent or received is limited only by the amount of available memory on a given host.

2.5 Messages in PVM

In PVM, sending a message requires three steps. First, a send buffer has to be initialized. After initializing the send buffer, the message is packed from the user data space into the PVM buffer in one of the encoding forms available. Then, the complete message is sent to another process with pvm_send(). A message header is appended to each message before sending. The message header used in PVM is shown in Figure 2.2.

The encoding field is used to pass the encoding format of the message. The tag field contains an integer that is given by the user and it labels the content of the message.

The context field is also an integer, but it is given by the PVM software. The sender ID along with the tag and the context of the message help ensure that the message is sent to the correct receiver[9]. The sequence number field stores the sequence number of the message, the wait id field is used to pass the wait ids associated with a message and the checksum field is reserved for future use.

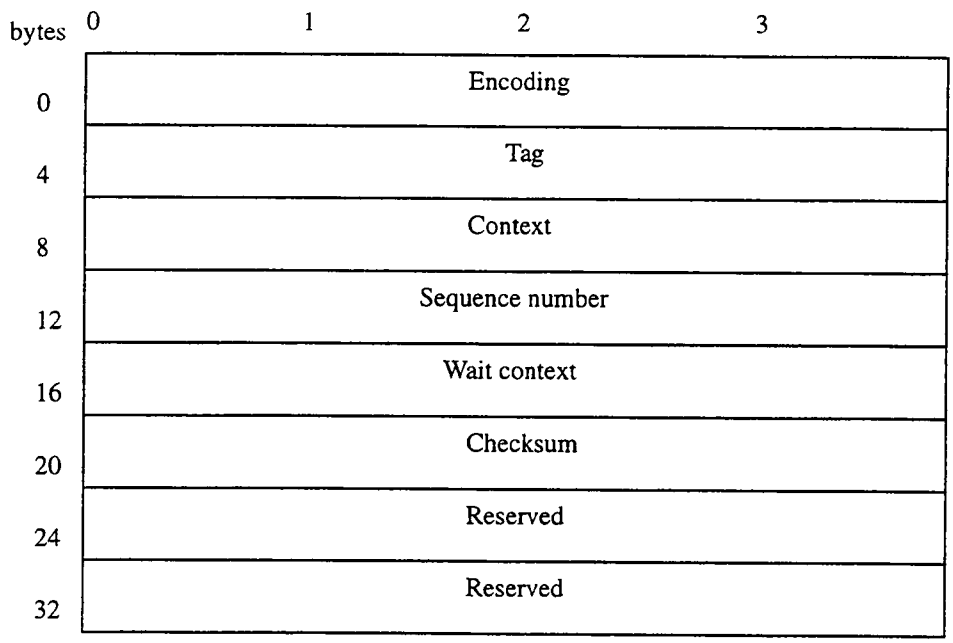


Figure 2.2. The message header.

Receiving a message in PVM involves two steps. First, the incoming message must be accepted by pvm_recv(), the blocking receive, or by pvm_nrecv(), the non-blocking receive. Once the message has arrived, it must be unpacked into the user data space with a combination of pvm_upk*() functions.

When receiving a message, the decoders are determined by the encoding field of the message header. The messages are buffered at the receiving end until they are

received. Once the message is received by the receiver, the message buffer is freed.

2.6 System Messages

System messages are special purpose messages that are sent to give specific information to tasks. They are sent like regular messages but have tags in the reserved space. System messages are passed to the function pvmmctl() causing the task to take some asynchronous action. For example, if task A needs to establish a direct connection with task B, task A sends a system message with the tag TC_CONREQ to task B requesting a connection. The list of tags is shown in Figure 2.3.

2.7 Communication in PVM

PVM communication is based on TCP, UDP, and UNIX domain sockets[10]. There are three types of connections within PVM: (1) pvmd-pvmd (2) task-task and (3) pvmd-task.

Pvmds communicate with each other through UDP sockets. As UDP is an unreliable delivery service, an acknowledgment and retry mechanism is used. In addition, as UDP limits packet length, messages are sent in one or more fragments, each with its own fragment header (see Figure 2.4) and with the message header at the beginning of the first fragment.

The communication between pvmd-task and task-task is through UNIX domain

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

Figure 2.3. System message tags list.

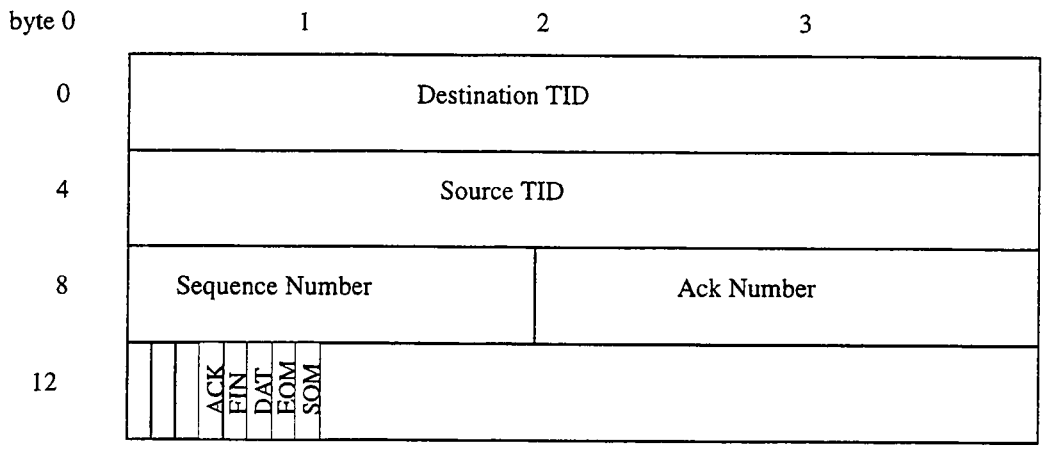


Figure 2.4. Pvmd-Pvmd packet header.

sockets. UDP is not used here because it is unreliable. Unreliable delivery requires retry at both ends, and since tasks cannot be interrupted while computing to perform I/O, UDP is not used[11]. The following fields exist in the pvmd-pvmd packet header.

Source TID	True source TID of the packet
Destination TID	True destination TID of the packet
SOM	Set for the first fragment of the message
EOM	Set for the last fragment of the message
DAT	If set, data is contained in the packet and the sequence number is valid
ACK	If set, the acknowledge number field is valid
FIN	If set, it indicates that the pvmd is closing down connection

The pvmd-task packet header looks similar to the pvmd-pvmd packet header except for two changes. Sequence numbers and acknowledge numbers are not used since this is a TCP connection and packet length is sent in the header as TCP provides no mechanism to distinguish back-to-back packets from one another.

2.8 Message Routing

2.8.1 Pvmd

Messages in the pvmd are sent by calling the PVM function sendmessage(), which routes the message by destination address. If the message is for other pvmds or tasks, sendmessage() attaches packet descriptions to the message and puts it on a send queue. If the message is to itself, sendmessage() calls netentry(), avoiding the packet layer entirely.

If a message is multicast, all the packets formed from the message would also be multicast, and the multicast packet descriptor is replicated placing one copy in each send queue.

When pvmd receives packets from the network, these packets are reassembled into messages in message reassembly buffers. Complete messages are then passed to entry points.

Pvmds usually do not communicate with tasks on other hosts. Every pvmd has message buffers for each foreign pvmd and each task it manages. The pvmd local to the sending task serves as a message repeater. The message is reassembled by the local pvmd as if it were the receiver, then forwarded all at once to the destination pvmd, which reassembles the message again.

2.8.2 Libpvm

The four functions that handle all traffic into and out of libpvm are mroute(), mxfer(), mxinput(), and pvmmctl(). Higher level functions, such as pvm_send() and pvm_recv(), call the function mroute() to copy messages into and out of tasks. The function mroute() establishes the necessary routes and calls mxfer(). The function mxfer() polls all routes for messages to be sent or received using select() and calls mxinput() to copy fragments into the tasks and reassemble messages. If a system message is received, mxinput() calls pvmmctl().

In PVM, messages are forwarded through pvmd by default. The default message route in PVM for a message passing between user processes task 1 and task 2 is shown in

Figure 2.5. Libpvm also provides the option of sending the messages directly from one task to another task through the TCP link instead of forwarding the message through pvmd. Direct routing is implemented with the help of notify and system messages. If PvmRouteDirect option is set using the function pvm_setopt(), mroute() attempts to create a direct TCP link if one does not exist. The route may be granted or refused by the destination task. If the route is refused, mroute() calls mxfer() to forward the message through pvmd. If the two tasks exchanging data are on the same host and the PvmRouteDirect option is set, UNIX domain sockets are used instead of TCP sockets. Figure 2.6 shows the direct route in PVM for a message passing between user processes task1 and task 2.

2.9 Massively Parallel Processors(MPP)

PVM was originally developed to join machines connected by a network into a single logical machine. A typical MPP[2] system has only one node, called the service node, that is actually connected to the network. The service node is the front-end of an MPP system and is used for user logins. The PVM daemon runs only on the service node and acts as a gateway to the outside world[12]. The programs running on the service node link to the regular PVM library, which relies on Unix sockets to transmit messages.

In addition to the service node, a typical MPP system has several compute nodes that are mainly used for computation. The compute nodes are preferred over service nodes for computation because communication between the processes running on service nodes and the node processes must go through the PVM daemon and a TCP link, whereas the

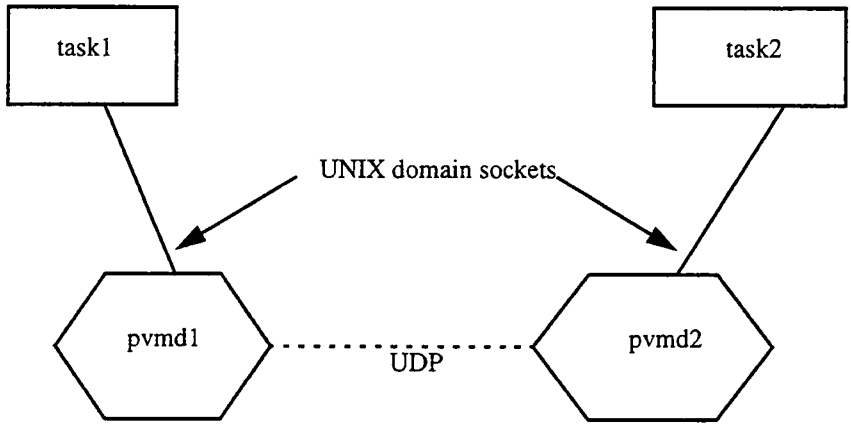


Figure 2.5. Default message route in PVM.

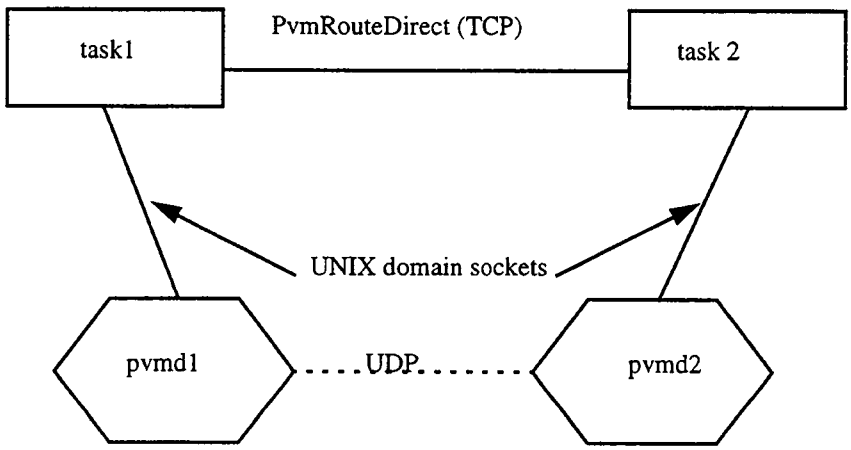


Figure 2.6. Direct message route in PVM.

communication between two compute nodes is through the fast interconnects between them.

The PVM message passing functions on an MPP system are implemented in terms of the native message passing functions of the MPP system. For example, pvm_send is implemented with the help of isend/csend on an iPSC860 system, CMMD_send_async/ _noblock on a CM-5 system, etc.

The way PVM routes messages is system dependent. On a Paragon, messages between two nodes of a multiprocessor go directly between them, while messages destined for a machine out on the network go to the single pvmd on the front end node for further routing.

On CM5 and IBM SP2, each set of tasks created by pvm_spawn() form a unit. If the tasks exchanging data are on the same unit, messages are passed directly. For each unit, PVM spawns an additional task to relay messages to and from pvmd, as tasks in the unit cannot communicate with pvmd directly. Communication between tasks belonging to different units must go through pvmd.

CHAPTER III

THE CHANNEL INTERFACE

In PVM, portability is considered much more important than performance because communication across the Internet is slow and research is focussed on problems with scaling, fault tolerance, and heterogeneity of the virtual machine. In order to simplify porting PVM to new systems, it should be made more layered. One interface that can be used is the channel interface. This interface is a set of simple data transfer function definitions, in terms of which user-callable standard PVM functions can be expressed. This chapter describes the implementation of a channel interface for PVM.

3.1 Description of Channel Interface

A *channel* represents the front end for receiving or sending data toward the underlying transport. A set of function definitions that describe the channel is called a *channel*

interface. The objective of these functions is to establish connections and read data coming in through the connection or write data to the connection.

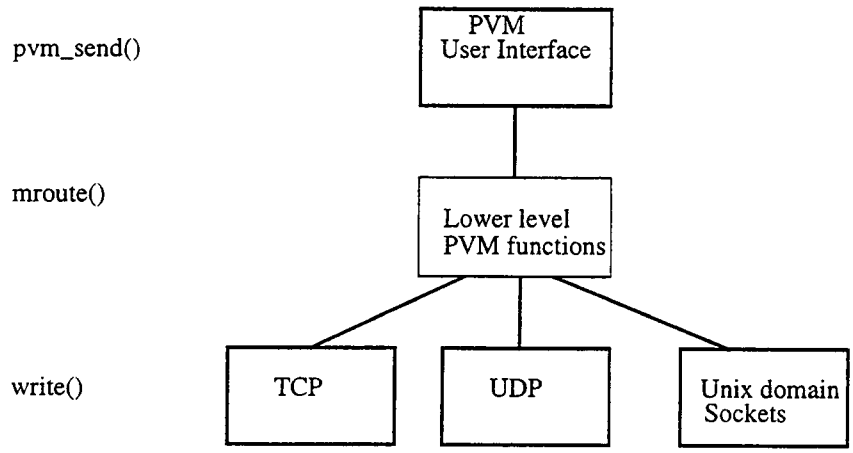
In PVM, at the lowest level, what is really needed is just a way to transfer data, possibly in small amounts, from one process address space to another. To achieve this no more functionality is required than what is provided by UNIX, in the *select, read,* and *write* operations. The channel interface uses these simple operations to provide the data transfer functions that can be used by the standard PVM user interface calls like pvm_send(), pvm_recv(), etc. All PVM message passing is converted into these lower level messages.

The structure of PVM without and with channel interface is shown in Figure 3.1(a) and Figure 3.1(b), respectively. Sample functions at each layer are shown on the left. The channel interface has been placed below the PVM user interface layer and above the functions that establish connections. The only functions below the channel interface are the functions that establish connections and hence the channel interface cannot be moved down any further.

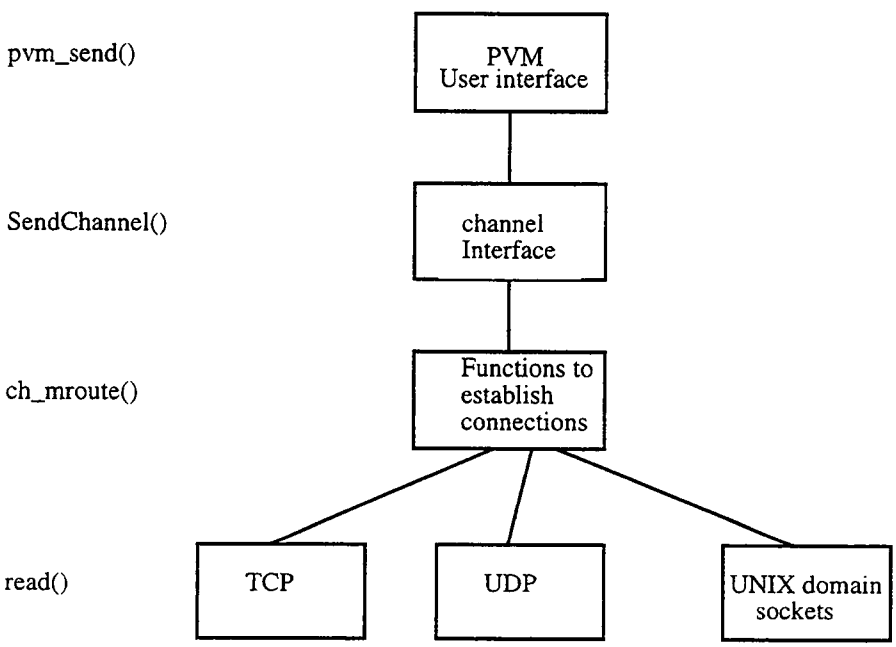
3.2 Design of Channel Interface

3.2.1 The Bare Minimum

The channel interface for PVM has been designed based on the channel interface for the MPICH[5] implementation of MPI[6]. The channel layer contains a simple set of routines that send and receive messages from one process to another. In this layer, mes-



(a). PVM without channel interface.



(b). PVM with channel interface.

Figure 3.1. PVM with and without channel interface

sages are sent in two parts: control part, containing information on the PVM message tag, size, etc., and a data part, containing the actual data. Separate routines are used to send and receive the control and data parts along with a routine to check if any control messages are available. A set of function definitions that define the send and receive routines of the channel layer is called channel interface. This specification for the channel interface can be done with a small number of definitions. They are:

ControlMsgAvail: Indicates whether a control message is available

RecvanyControl: Reads the next control message

SendControl: Sends a control message

RecvfromChannel: Receives data from a particular channel

SendChannel: Sends data on a particular channel

In UNIX terms, ControlMsgAvail is similar to *select* with an fd mask on all the file descriptors that can be read for control messages, and RecvAnyControl is like a *select* followed by a *read*, while the others are similar to *read* and *write* on appropriate file descriptors.

3.2.2 Data Exchange Mechanisms

This section describes the different ways in which data can be transferred from one process to another in the channel layer and the mechanism used in PVM.

1. The data is sent to the destination immediately, but the receiver must allocate some space to store the data locally.

2. Data is sent to the destination only when requested. When a receive is posted that matches the message, the destination sends a request for the data. In addition, it provides a way for the sender to send the data.

3. Data is read directly by the receiver. This choice requires a method to directly transfer data from one process memory to another.

PVM uses the third mechanism in shared memory architectures and the first mechanism in all other architectures.

3.2.3 Buffering in Channels

The channel interface has been designed assuming no buffering. The functions that call this interface have to perform the necessary buffer management. Hence the pvm_send() routine has to create a send buffer before passing on the message to the channel layer to perform the actual data transfer.

3.3 Implementation of Channel Interface

A channel interface can be implemented in several different ways. It can be implemented (i) in terms of p4[13], which is a library of macros and subroutines for programming a variety of parallel machines; (ii) in terms of Nexus[14], which is a portable library providing the multithreading, communication, and resource management facilities required to implement libraries and applications in heterogeneous parallel and distributed computing environments; (iii) in terms of the native message passing system on an MPP

machine; and (iv) in terms of direct TCP/IP[3] connection calls.

In this thesis, the channel interface for PVM has been implemented in terms of TCP/IP socket calls in a workstation environment. On an MPP machine these socket calls can be replaced by the native message passing calls of the MPP machine. To implement this channel interface, changes have been made only to the libpvm part of the PVM software, which is a library of PVM user interface routines. The pvmd part has not been touched.

3.3.1 Sending a Message

In PVM, a message is sent in three steps. A send buffer is initialized first, then the message is packed from user data space into the PVM buffer, and finally the complete message is sent to another process with pvm_send(). But, with the introduction of a channel interface, pvm_send() passes the message to be sent to the channel layer instead of sending the message itself. For every single PVM message, the channel layer sends two messages to the destination task. They are a control message and a data message.

A control message is sent by calling the function SendControl(). In addition to sending a control message, SendControl() also establishes a route from source task to destination task for the control and data messages to travel. It accomplishes this task with the help of a lower level function called ch_mroute() which opens a socket by calling the function sockcon(). The function sockcon() opens a socket with the help of the C function socket() and returns an identifier for the socket that may be used for future invocations of

commands like *read*, *write,* etc. Once the socket is opened, the function conreq() sends a system message to the destination task with the context SYSCTX_TC and tag TC_CONREQ requesting a connection to the socket. If the destination task accepts the connection, ch_mroute() creates a structure "ttpcb" that has all the information regarding the direct TCP connection established.

A data message is sent by calling the function SendChannel(). Once a route is established by SendControl(), SendChannel() sends the message through that route by calling the function ch_mxfer(). ch_mxfer() first checks if there is a direct route to the destination task. If one exists, the message to be sent is written to the TCP socket by calling the C function write(). If a direct route does not exist the message is sent to the pvmd, which forwards the message to the actual destination.

3.3.2 Receiving a Message

As discussed in the last chapter, receiving a message in PVM involves two steps. First, the message is accepted by pvm_recv(), then the received message is unpacked into the user data space. But, with the implementation of a channel interface, pvm_recv() passes the information regarding the message it is expecting to receive to the channel layer instead of receiving the message itself. The channel layer first checks if there are any control messages available by calling the function ControlMsgAvail(). Then, the available control messages are received by calling the function RecvanyControl(). The actual data messages are received by RecvfromChannel(). The channel functions RecvanyControl()

and RecvfromChannel() are implemented in terms of lower level functions like ch_mxfer() and ch_mroute(). The function path in the libpvm after introducing the channel interface is shown in Figure 3.2.

3.3.3 Message Ordering

Control messages between any pair of processors must arrive in the order in which they were sent. When a message is sent using SendControl() and SendChannel(), it is guaranteed that the sending process performs the SendChannel() after the SendControl() for that message, and before performing SendControl() for any other message. However, there is no required ordering of messages sent by different processors.

3.3.4 Message Routing

In PVM, messages were forwarded through pvmd by default. To send a message from task 1 to task 2, the message is first sent from task 1 to task 2, then from pvmd1 to pvmd2, and finally from pvmd2 to task 2. But, with the implementation of a channel interface, PvmRouteDirect is made as the default routing option. With this option, messages are sent directly from one task to another through sockets. If the two tasks are located on the same host, UNIX domain sockets are used and if they are located on different hosts, TCP sockets are used. The default message route in PVM after the implementation of channel interface, for a message passing between user processes task 1 and task 2 is shown in Figure 3.3. Figure 3.4 shows the message route in PVM for a message passing

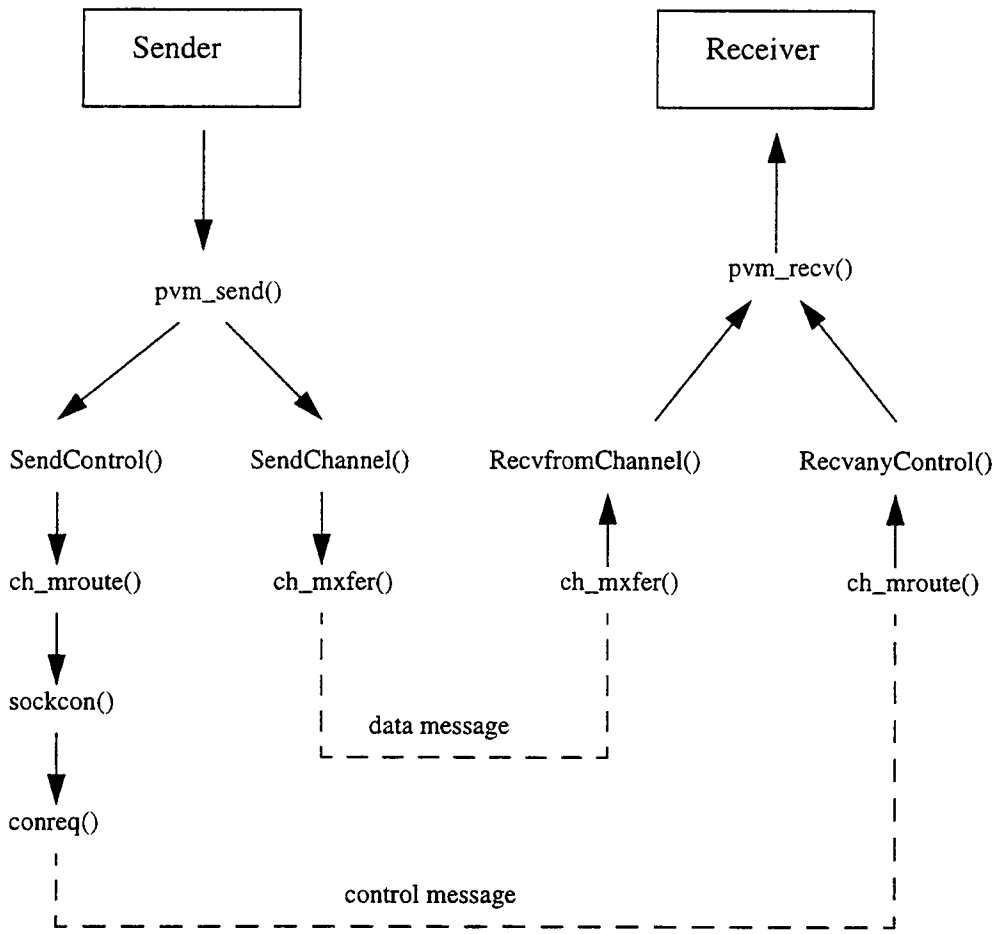


Figure 3.2. The function path in libpvm.

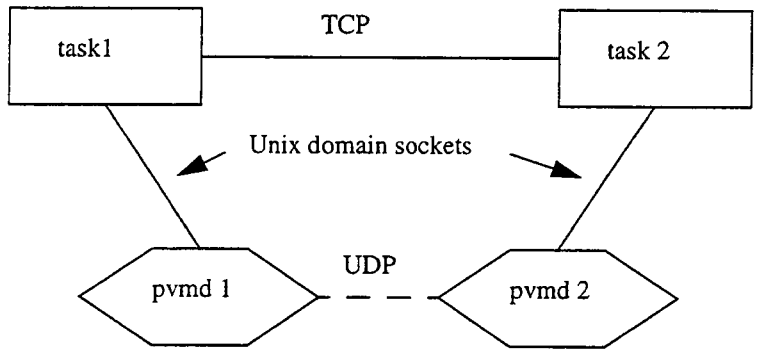


Figure 3.3. The default route in PVM with channel interface.

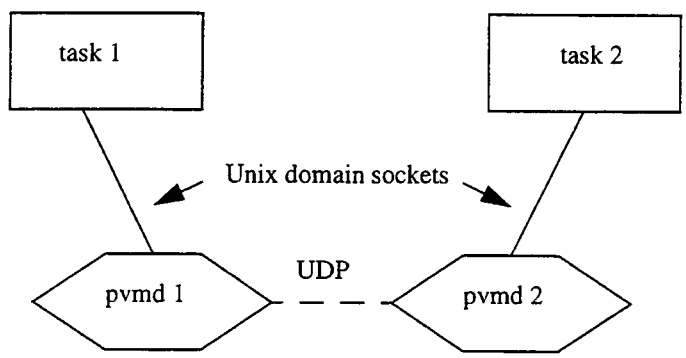


Figure 3.4. Forwarding messages through pvmd
(PvmDontRoute option).

between user processes task 1 and task 2 with PvmRouteDirect option set. If one of the two tasks has requested PvmDontRoute or no resources are available, the message is forwarded through the pvmd.

3.4 Massively Parallel Processors

The implementation of a channel interface for PVM on an MPP machine is very similar to its implementation in a workstation environment. While sending a message, the function pvm_send() calls the functions SendControl() and SendChannel() to send the control and data messages, respectively. SendControl() and SendChannel() are implemented in terms of the native send calls of the MPP system. Similarly, RecvfromChannel() and RecvanyControl() receive the control and data parts of a message. They are implemented in terms of native receive calls of the MPP system. However, the message routing of PVM on an MPP system has not been changed with the implementation of channel interface. For example, on an IBM SP2, PVM is built on top of IBM's implementation of MPI and on CM-5 it is built on top of CMMD which is the native message passing library of CM-5. Hence, the channel calls SendChannel() and RecvfromChannel() are implemented in terms of MPI_Isend() and MPI_Irecv(), respectively on SP2 and in terms of CMMD_send_async and CMMD_receive_async, respectively, on a CM-5 system.

CHAPTER IV

PERFORMANCE

The design goal of our channel interface is to simplify porting PVM to new systems without affecting its performance. This chapter discusses the performance of PVM with and without channel interface. The test programs were run on SUN workstations and IBM SP2 machine. The objective of these test programs was to determine the bandwidth and latency. The latency is measured as half the round-trip time for zero length messages, and bandwidth is measured as twice the datasize divided by the round-trip time.

4.1 Measurement Methodology

To measure latency and bandwidth, a simple echo test has been used between two SUN workstations and between two nodes of an IBM SP2. A receiving node simply echoes back whatever it receives and the sending node measures round-trip time. Times are

collected for several iterations over various message sizes. The pseudo-code for echo test is shown in Figure 4.1.

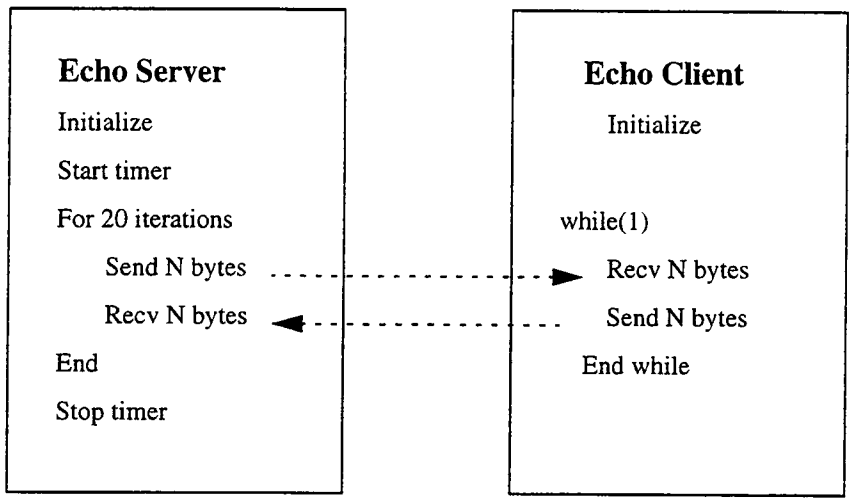


Figure 4.1. Pseudo-code for echo test.

4.2 Performance on Workstations

The bandwidth and transfer times between two SUN workstations are measured for short and long messages using:

- *pvm_send()/pvm_recv()* with channel interface
- *pvm_send()/pvm_recv()* without channel interface.
- *MPI_Send()/MPI_Recv()* (MPICH implementation of MPI).

4.2.1 Short Messages

The performance of PVM with channel interface is compared with the perfor-

mances of the original PVM run with the PvmRouteDirect option and MPICH implementation of MPI. The channel interface for MPICH is implemented in terms of p4 macros in the SUN workstation environment. Tables 4.1 and 4.2 show the bandwidth and message passing transfer times, respectively, obtained between two workstations for short messages. Graphs for these data are shown in Figures 4.2 and 4.3, respectively.

The original PVM software is run with the PvmRouteDirect option in order to make the message routing in both cases the same. The first point to notice in Figure 4.2 is that MPI has the better bandwidth and lower latency of all three. The reason for this is that unlike in PVM, sending a message in MPI does not require initializing a buffer and packing the message into the buffer. Similarly, receiving a message in MPI does not involve unpacking the received message. Hence, these extra steps in PVM lead to its poor performance. when compared to MPI. In addition, it is obvious from Figures 4.2 and 4.3 that PVM without channel interface has better bandwidth, lower latency (see Table 4.3), and lower transfer times than PVM with channel interface. The reason for this inefficient performance of PVM with channel interface is that the default buffer size for a PVM packet per send is 4096 bytes. The smaller the size of the message, the lesser the fragmentation and fewer the number of packets to be sent. With a channel interface, pvm_send() sends two messages, control message and data message, to the destination task for every one user message. As small messages can be sent in fewer number of packets, the extra cost of the control message becomes a penalty.

Table 4.1. Bandwidth between two workstations for short messages.

Message size (bytes)	Bandwidth		
	PVM with channel interface (MB/s)	PVM without channel interface (MB/s)	MPICH (MB/s)
0	0.0000	0.0000	0.0000
8	0.0072	0.0076	0.0085
800	0.4105	0.6164	0.6757
1600	0.7944	1.0296	1.1436
2400	1.1283	1.2409	1.4109
3200	1.1561	1.5793	1.7961

Table 4.2. Transfer times between two workstations for short messages.

Message size (bytes)	Transfer time		
	PVM with channel interface (μs)	PVM without channel interface (μs)	MPICH (μs)
0	940	925	899
8	1120	1052	962
800	1778	1298	1184
1600	2014	1554	1399
2400	2227	1934	1701
3200	2768	2320	2005

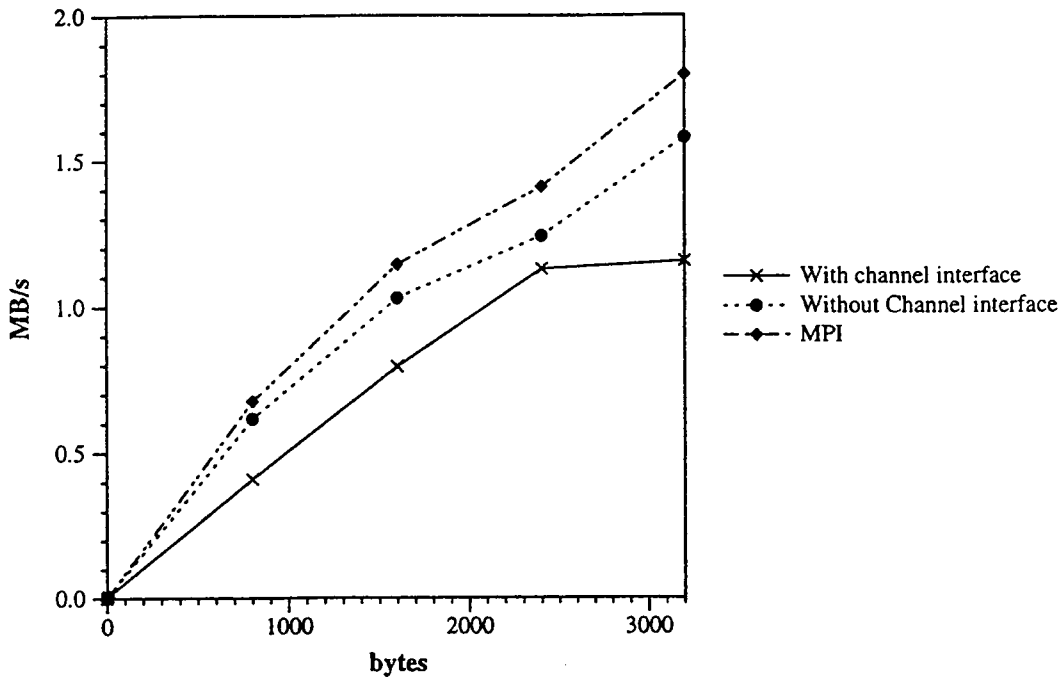


Figure 4.2. Bandwidth between two workstations for short messages.

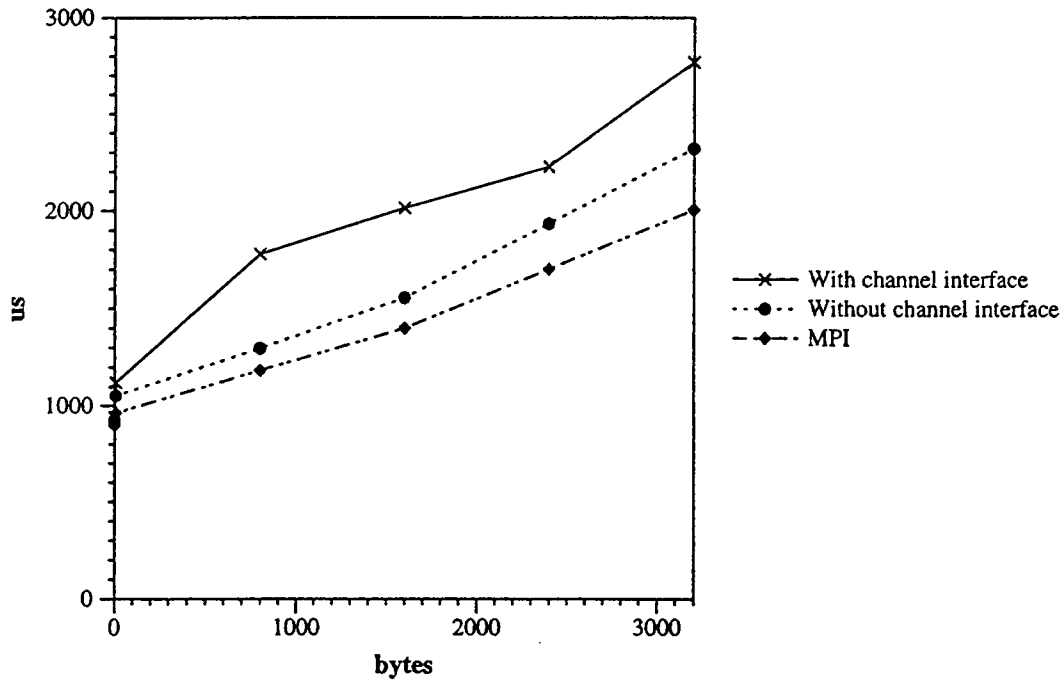


Figure 4.3. Transfer times between two workstations for short messages.

Table 4.3. Latency between two workstations.

System	Latency (μs)
PVM with channel interface	940
PVM without channel interface	925
MPICH	899

4.2.2 Long Messages

Tables 4.4 and 4.5 show the bandwidth and transfer time obtained between two workstations for long messages. Figures 4.4 and 4.5 show the graphs corresponding to these data, respectively.

As in the case of short messages, original PVM is run with the PvmRouteDirect option. The first point to notice in Figure 4.4 is that, as expected, MPI is the most efficient of all three for the same reason as in the case of short messages. The second point to notice is that, for long messages, the performance of PVM with channel interface is fairly close to the performance of PVM without channel interface. As previously discussed, the buffer size for a PVM packet per send is 4096 bytes. The number of packets sent from source to destination depends on the size of the message. As the message size increases, the number of fragments per message increases, and the cost of sending a control message becomes

Table 4.4. Bandwidth between two workstations for long messages.

Message size (bytes)	Bandwidth		
	PVM with channel interface (MB/s)	PVM without channel interface (MB/s)	MPICH (MB/s)
40000	0.7393	0.7527	0.9554
48000	0.7363	0.7500	0.9514
56000	0.8069	0.8091	0.9349
64000	0.8487	0.8533	0.9555
72000	0.8782	0.8887	1.0360
80000	0.9497	0.9526	1.0567

Table 4.5. Transfer times between two workstations for long messages.

Message size (bytes)	Transfer time		
	PVM with channel interface (μs)	PVM without channel interface (μs)	MPICH (μs)
40000	54105	52447	41869
48000	65189	63998	52093
56000	69805	69213	59897
64000	75403	74999	67013
72000	81982	81012	69498
80000	84247	83979	75689

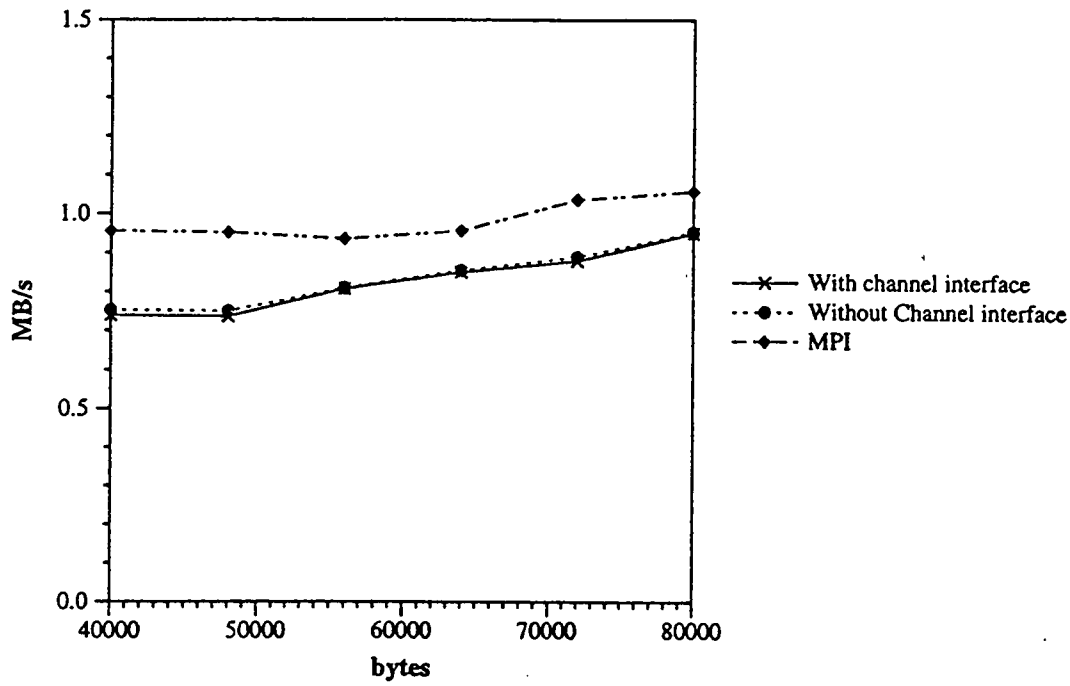


Figure 4.4. Bandwidth between two workstations for long messages.

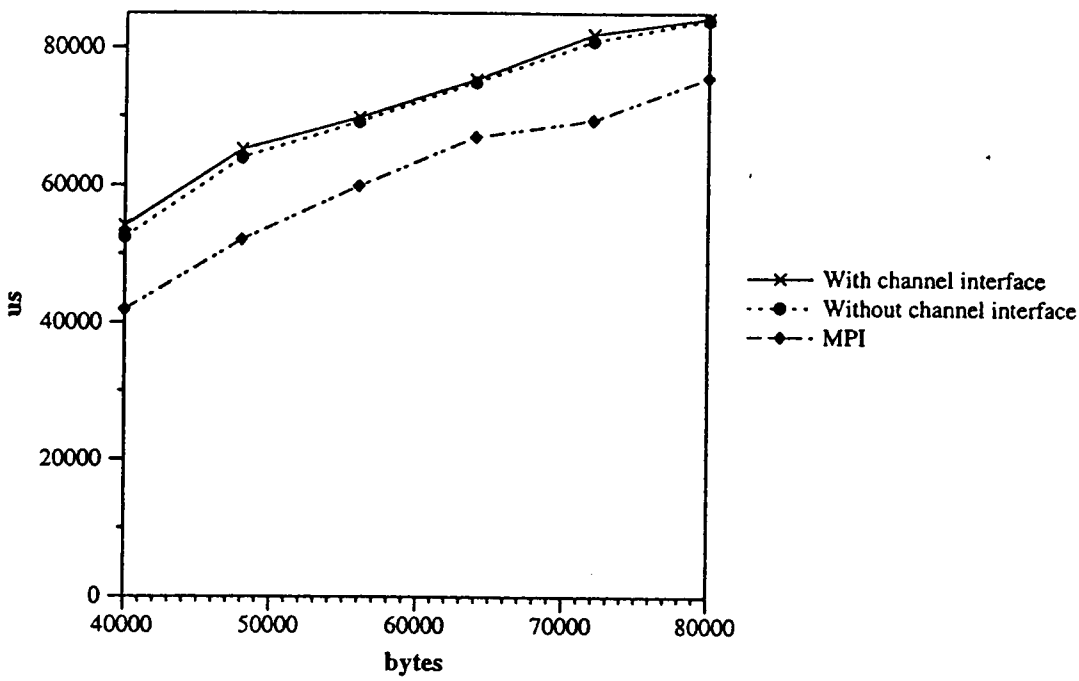


Figure 4.5. Transfer times between two workstations for long messages.

negligible.

4.3 Performance on IBM SP2

The bandwidth and transfer times between two nodes of an IBM SP2 are measured for short and long messages using:

- *pvm_send()/pvm_recv()* with channel interface.
- *pvm_send()/pvm_recv()* without channel interface
- *pvm_psend()/pvm_precv()* with channel interface.
- *pvm_psend()/pvm_precv()* without channel interface.
- *MPI_Send()/MPI_Recv()* (IBM's implementation of MPI).

4.3.1 Short Messages

Tables 4.6 and 4.7 show the bandwidth and transfer time, respectively, obtained between two nodes of an IBM SP2 for short messages. The graphs corresponding to these data are shown in Figures 4.6 and 4.7, respectively.

The first point to notice in Figures 4.6 and 4.7 is that IBM's implementation of MPI is the most efficient. The reason for this is that on SP2, MPI is implemented at the same level as MPL (the native message passing library of SP2) but not on top of it, whereas PVM is built on top of MPI. However, the performance of pvm_psend()/ pvm_precv() is fairly close to the performance of MPI and has better bandwidth and lower latency (see Table 4.8) than pvm_send()/pvm_recv(). This is because, unlike pvm_send()/

Table 4.6. Bandwidth on IBM SP2 for short messages.

Message size (bytes)	Bandwidth				
	PVM with channel interface		PVM without channel interface		MPI
	pvm_send/pvm_recv (MB/s)	pvm_psend/pvm_precv (MB/s)	pvm_send/pvm_recv (MB/s)	pvm_psend/pvm_precv (MB/s)	(MB/s)
0	0.0000	0.0000	0.0000	0.0000	0.0000
8	0.0050	0.0070	0.0060	0.0090	0.0110
800	0.4640	0.5578	0.5609	0.7692	0.9049
1600	0.8385	1.0107	1.0145	1.2818	1.7957
2400	1.0887	1.3330	1.2525	1.7686	2.1324
3200	1.2772	1.5984	1.4217	2.0492	2.6197

Table 4.7. Transfer times on IBM SP2 for short messages.

Message size (bytes)	Transfer time				
	PVM with channel interface		PVM without channel interface		MPI
	pvm_send/pvm_recv (μs)	pvm_psend/pvm_recv (μs)	pvm_send/pvm_recv (μs)	pvm_psend/pvm_precv (μs)	(μs)
0	1405	1105	1285	835	560
8	1609	1204	1344	884	720
800	1723	1434	1440	1050	884
1600	1908	1583	1585	1254	891
2400	2204	1799	1922	1361	1125
3200	2505	2002	2256	1565	1221

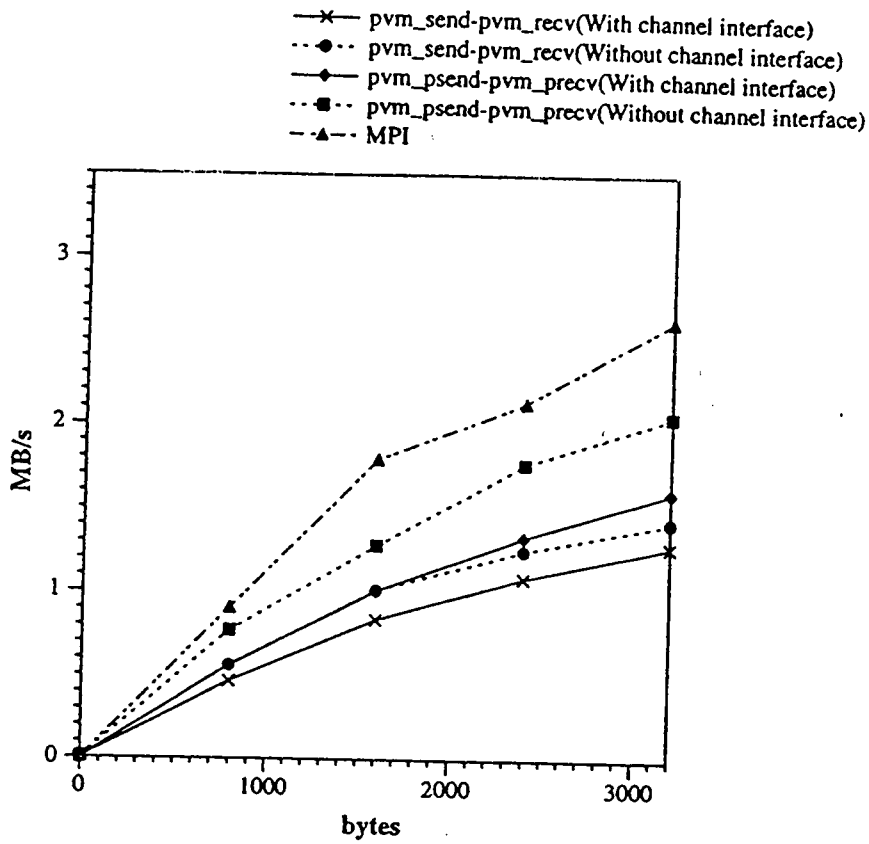


Figure 4.6. Bandwidth on IBM SP2 for short messages.

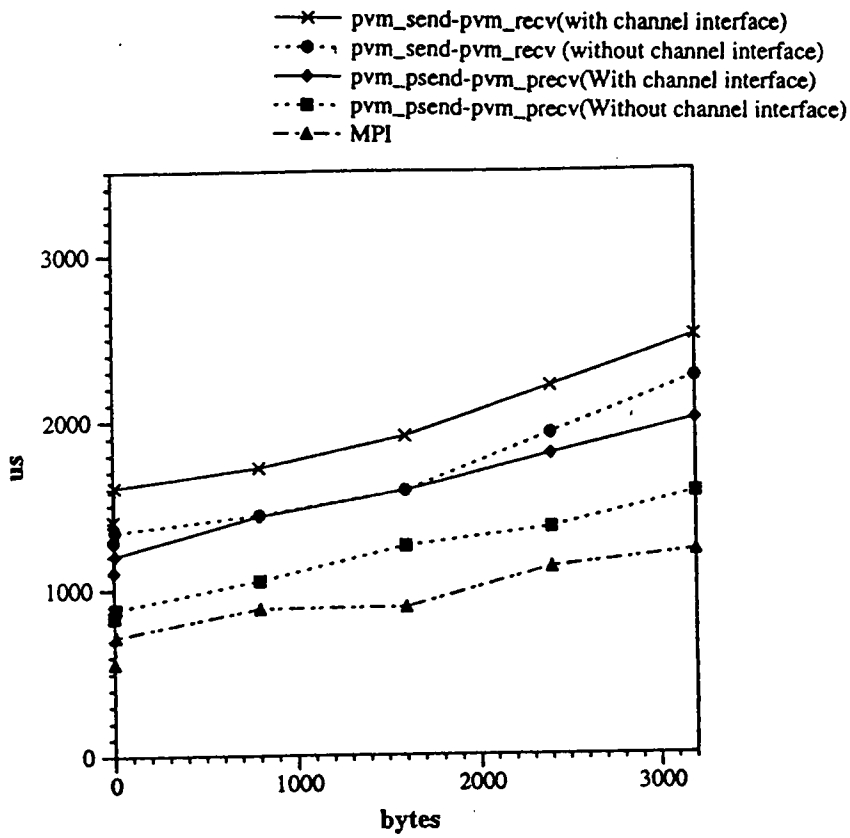


Figure 4.7. Transfer times on IBM SP2 for short messages.

pvm_recv(), it does not involve packing and unpacking of messages, and it is implemented directly on top of the native message passing system.

pvm_send()/pvm_recv() and pvm_psend()/pvm_precv() with channel interface have considerably higher transfer times and lower bandwidth than pvm_send/pvm_recv and pvm_psend()/pvm_precv() without channel interface, respectively. The reason for this poor performance of PVM with channel interface is the extra cost of sending a control message.

Table 4.8. Latency on IBM SP2

System	Latency (μs)
pvm_send/pvm_recv with channel interface	1405
pvm_send/pvm_recv without channel interface	1285
pvm_psend/pvm_recv with channel interface	1105
pvm_psend/pvm_precv without channel interface	835
MPI	560

4.3.2 Long Messages

Tables 4.9 and 4.10 show the bandwidth and transfer time, respectively, between two nodes of an IBM SP2 for long messages. The graphs corresponding to these tables are shown in Figures 4.8 and 4.9, respectively.

Table 4.9. Bandwidth on IBM SP2 for long messages.

Message size (bytes)	Bandwidth				
	PVM with channel interface		PVM without channel interface		MPI
	pvm_send/pvm_recv (MB/s)	pvm_psend/pvm_precv (MB/s)	pvm_send/pvm_recv (MB/s)	pvm_psend/pvm_precv (MB/s)	(MB/s)
40000	1.8392	2.2925	1.8895	2.4716	5.3001
48000	1.8538	2.6798	1.8705	2.7902	5.3416
56000	1.8642	3.0653	1.8833	3.1364	5.4302
64000	1.8813	3.3956	1.8805	3.5030	5.5106
72000	1.8814	3.6616	1.8993	3.7937	5.6203
80000	1.9043	3.6655	1.9181	3.2987	5.6912

Table 4.10. Transfer times on IBM SP2 for long messages.

Message size (bytes)	Transfer time				
	PVM with channel interface		PVM without channel interface		MPI
	pvm_send/pvm_recv (μs)	pvm_psend/pvm_precv (μs)	pvm_send/pvm_recv (μs)	pvm_psend/pvm_precv (μs)	(μs)
40000	21749	17448	21170	16184	7547
48000	25893	17912	25662	17203	9076
56000	30879	18269	29735	17855	10313
64000	35019	18848	34033	18270	11614
72000	38269	19663	37908	18969	12811
80000	42010	21825	41708	20107	14057

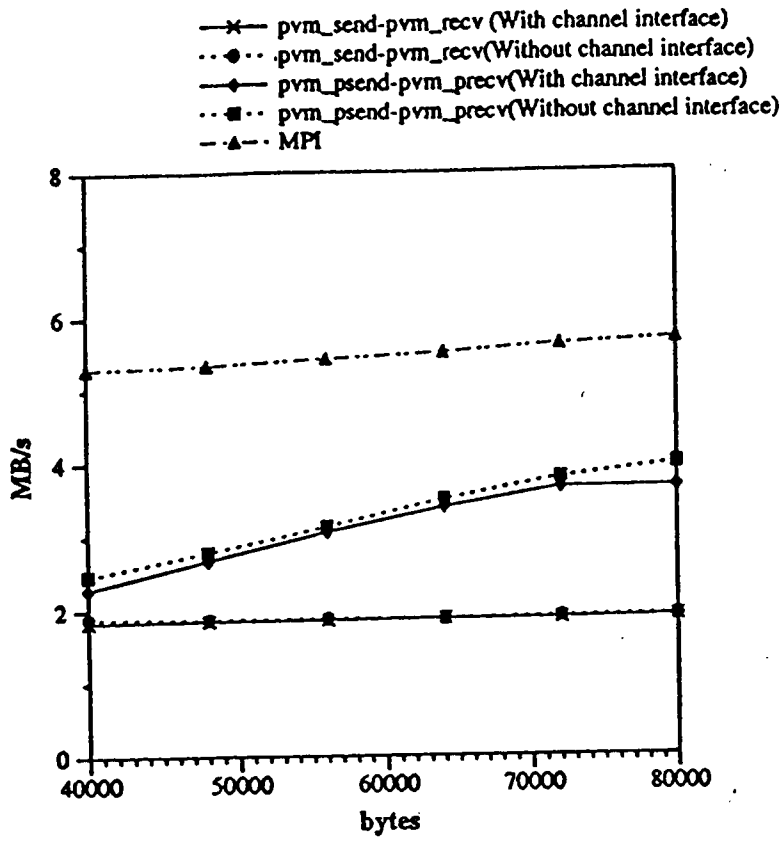


Figure 4.8. Bandwidth on IBM SP2 for long messages.

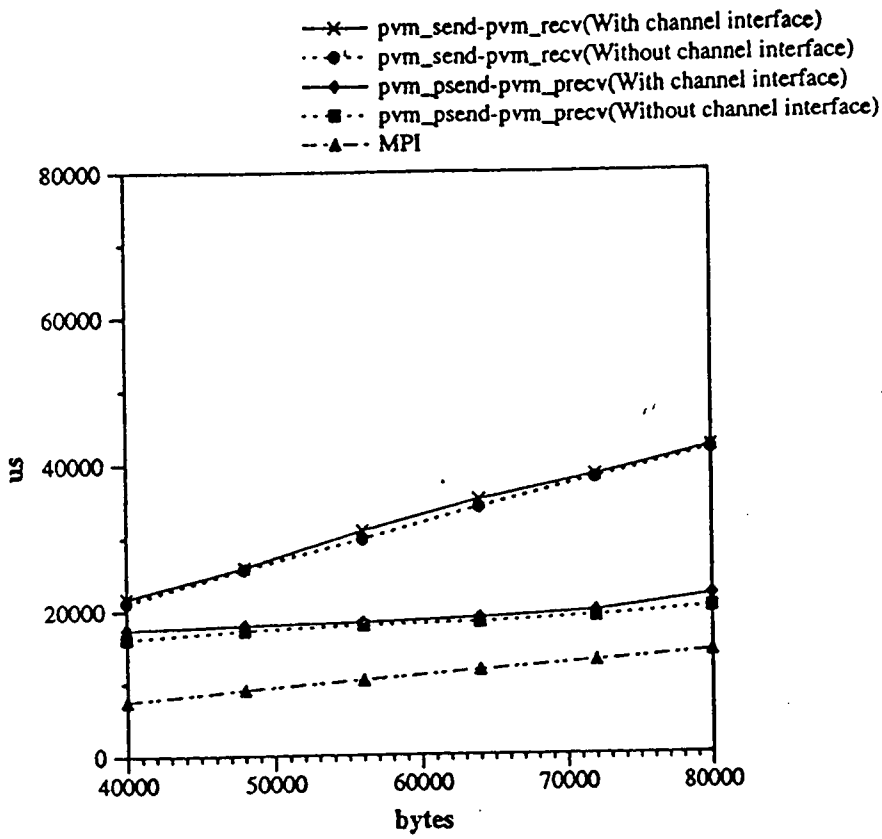


Figure 4.9. Transfer times on IBM SP2 for long messages.

MPI_Send()/MPI_Recv() and pvm_psend()/pvm_precv() are more efficient than the others for the same reason as in the case of short messages. However, PVM with channel interface performs better for long messages than for short messages. For long messages, the performance of PVM with channel interface is fairly close to the performance of the PVM without channel interface. This is because the number of fragments per message is more for long messages, and hence, the extra cost of sending a control message becomes negligible.

CHAPTER V

SUMMARY AND FUTURE WORK

This thesis focused on the implementation of a channel interface for PVM. The design and implementation of a channel interface and its impact on the performance of PVM were studied. The channel interface was implemented in terms of direct TCP/IP socket calls in a workstation environment and in terms of IBM's implementation of MPI on an IBM SP2. The performance of PVM has been measured on both the platforms.

The channel layer contains a simple set of routines that send and receive messages from one process to another. A set of function definitions that define the send and receive routines of the channel layer is called channel interface. In this layer, messages are sent in two parts: control part and data part. All PVM message passing is converted into these lower level messages.

In addition, with the implementation of the channel interface, all the messages are sent through direct TCP channels instead of forwarding them through pvmds. A sender task establishes a TCP connection to the destination task (if one does not already exist) and then sends the message through the connection. A TCP connection once established can be used for all future communication between those two tasks.

This implementation of our channel interface has few limitations. It affects the performance of PVM while sending short messages. First, a task has to completely fragment the message before sending it to its destination. For small messages, the fragmentation is less. Hence, the control message becomes a penalty in the case of short messages. Also, sending all the messages through TCP channels limits the number of tasks that can run on a single host as each TCP connection consumes a file descriptor.

5.1 Future Work

In addition to direct TCP/IP links and the native message passing system of an MPP machine, the channel interface for PVM can be implemented in several other ways. One of them involves implementing a channel interface in terms of p4 macros. This implementation of a channel interface has not been discussed in this thesis. The advantage of this implementation is that we can provide portability to a shared memory model and the distributed memory model at one stroke.

The channel interface affects the performance of PVM while sending small messages. However, this limitation can be overcome to a certain extent by sending the data

along with the control message, if the data length is short enough.

In this thesis, a channel interface has not been implemented for shared memory abstraction. This can be done in terms of shared memory malloc and locks.

BIBLIOGRAPHY

1. A. Geist, A. Beguelin, J. Dongarra, W. Tiang, R. Manchek, V. Sundaram, *PVM: Parallel Virtual Machine - A Users Guide and Tutorial for Network Parallel Computing,* MIT Press (1994).

2. J. Dongarra, Henri Casanova, Weicheng Jiang, *Performance of PVM on MPP systems*, Technical Report, Department of Computer Science, University of Tennessee (1995).

3. A. Beguelin, J. Dongarra, A. Geist, R. Manchek, V. Sundaram, *Recent Enhancements to PVM,* Technical Report, ORNL (1994).

4. W. Gropp, E. Lusk, *MPICH Working Note: Creating a New MPICH Device Using the Channel Interface*, Technical Report ANL/MCS-TM-000, Argonne National Laboratory.

5. William Gropp, Ewing Lusk, Anthony Skjellum, *A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard*, Technical Report, Argonne National Laboratory.

6. Message Passing Interface Forum *MPI: A Message Passing Interface Standard.* Technical Report CS-94-230, Department of Computer Science, University of Tennessee, Knoxville (1994).

7. Stephanie Wolf, *A Beginners Guide to IBM SP2*, Joint Institute of Computational Science, University of Tennessee (1997)

8 T. Dunigan, N. Venugopal, *The Design, Implementation, and Evaluation of Cryptographic Distributed Applications: Secure PVM*, Technical Report, University of Tennessee (1996).

9. P. M. Papadopolous, J. Manchek, A. Geist, J. Dongarra, *Adding Context and Static Groups into PVM* (1995).

10. D.Comer, Internetworking with TCP/IP, Second Edition, Prentice Hall (1991).

11. P. J. Mucci, J. Dongarra, *Possibilities for Active Messaging in PVM*, Technical Report, University of Tennessee, Knoxville (1994).

12. J. Dongarra, T. Dunigan, *Message Passing performance of Various Computers*, Techincal Report, Applied Mathematical Sciences subprogram, U.S Depart ment. of Energy (1997).

13. Ralph M. Butler, Swing L. Lusk, *Monitors, Messages, and Clusters:The p4 Parallel Programming System,* Technical Report, University of North Florida.

14. Ian Foster, Carl Keelson, Steven Tuecke, *The Nexus Task-parallel Runtime System,* Technical Report, Argonne National Laboratory.

15. G. A. Geist, J. A. Kohl, P. M. Papadopolous, S. L. Scott, *Beyond PVM 3.4: What We Have Learned, What's Next and Why*, Technical Report, Oak Ridge National Laboratory.

VITA

Nirmala Kodali was born in Andhra Pradesh, India on June 7, 1972. She graduated from St. Theresa's Girls High School, Eluru in 1989 and received the Bachelor of Science degree in Electrical Engineering from Andhra University in May 1993. In August 1996, she entered the Computer Science program at the University of Tennessee and was awarded the Master of Science in August 1998.