12-1996

An Efficient Collective Communication Library for PVM

Chirapol Mathawaphan

AN EFFICIENT COLLECTIVE COMMUNICATION LIBRARY FOR PVM

by

Chirapol Mathawaphan

A Thesis
Submitted to the
Faculty of The Graduate College
in partial fulfillment of the
requirements for the
Degree of Master of Science
Department of Computer Science

Western Michigan University
Kalamazoo, Michigan
December 1996

ACKNOWLEDGMENTS

I wish to express much appreciation to my advisor, Professor Ajay Gupta, for his valuable ideas and time spent on this thesis. I have gain much from his outstanding guidance and continual encouragement. Also I would like to express my gratitude to the thesis committee members, Professor Ben Pinkowski and Professor Elise de Doncker, for their help, comments and suggestions.

Finally, but not lastly, I would like to thank my mother, sister, brothers and all my friends for their love, encouragement and support.

Chirapol Mathawaphan

AN EFFICIENT COLLECTIVE COMMUNICATION LIBRARY FOR PVM

Chirapol Mathawaphan, M.S.

Western Michigan University, 1996

PVM enables the use of network of workstations for parallel and distributed computation. PVM provides the message-passing primitives which include those for point-to-point communication and collective communications. The current approaches for the collective communication operations in PVM use algorithms that do not exhibit good performance.

In this thesis, we develop new approaches to improve the performance of collective communication operations in PVM by using shared memory with IP broadcasting and IP multicasting mechanism. We have implemented these approaches and have run extensive tests. This reports shows the comparison of the time used by current approaches and our approaches.

TABLE OF CONTENTS

Table of Contents—Continued

Table of Contents—Continued

LIST OF TABLES

LIST OF FIGURES

List of Figures—Continued

CHAPTER I

INTRODUCTION

Statement of the Problem

In the last decade, a new trend of parallel and distributed computing has been developed namely Heterogeneous Network Computing. It involves a parallel program running on a cluster of workstations of different architecture bases, connected together by a high-speed network. The advent of high-performance workstations and gigabit networks, such as 100 Mb Ethernet, FDDI, ATM and HIPPI networks have made these workstations grow in power over the past few years and represent a significant computing resource [11].

Parallel programming environments have been developed to run on top of clusters of workstations and offer the user a convenient way for expressing parallel computation and communication. In message-passing paradigm, the communication part consists of regular point-to-point communication as well as collective communication, such as broadcast, scatter, gather and reduce [6]. It has been recognized that many parallel numerical algorithms can be effectively implemented by formulating the required communication as collective communications [16]. The performance of many algorithms depends heavily on the performance of Collective Communication Library (CCL) [1, 3, 6, 16].

Parallel Virtual Machine (PVM) provides the user a parallel programming environment. PVM enables users to use their existing computer hardware to solve much larger problems at minimal cost. Hundreds of sites around the world are using PVM to solve important scientific, industrial, and medical problems. PVM is also used as an educational tool to teach parallel programming [25]. PVM supports the message-passing model of computation. It provides user interface primitives for point-to-point communication and collective communication.

Collective Communication Library currently implemented in PVM is not very efficient, as we shall see later. The current approaches simply use linear algorithms, for example, in gather, scatter, and reduce operations and use the library function, pvm_mcast() which uses 1:N fanout mechanism, to perform broadcast operation [11, 13].

A new model is proposed here and we show that the performance of Collective Communication Library in PVM can be improved by using the advantage of LAN and PVM software point of view.

In this thesis, we develop new approaches using shared memory with IP broadcasting or IP multicasting mechanism, and report the comparision timings of the current algorithms and the new algorithms. Our experimental results show the improvement of the performance of Collective Communication Library using new approaches.

Organization of the Study

Chapter II is a review of the background and related literature, and it is intended to build some problem specific background, required to understand how PVM and collective communication operations work. This study is based on PVM version 3.3.

Chapter III is a review of current approaches of collective communication library in PVM.

Chapter IV is devoted to our new approaches that were developed during the course of study. We present an overview, describe our algorithms in detail, and compare them to the current approaches.

Chapter V is a presentation of our experimental results. The algorithms were implemented and run a large number of times for several problem sizes, in order to collect accurate performance statistics.

The new algorithms are implemented in PVM version 3.3. The source codes developed for the experiments in this study are included in Appendices. All implementations were done on Sun SparcStation 5 workstations and measurements of timing simply use the C library gettimeofday() function.

A summary and conclusion of the study and suggestions for further study are included in Chapter VI.

CHAPTER II

BACKGROUND INFORMATION

Heterogeneous Network Computing

A heterogeneous network computing system is typically a collection of independent machines of different types interconnected by a high-speed network [24]. The environment of local area network of the same type of workstations is also called a "cluster" [23].

Concurrent computing environments based on networks of computers can be an effective, viable, and economically attractive complement to single hardware multiprocessors [12]. When the networks become more high-speed, the combination of computational power of each workstation can be applied to solve a variety of computationally intensive applications. It may provide supercomputer-level computational power. Further, under the right circumstances, the network-based approach can be nearly as effective as multiprocessors. This configuration may be economically and technically difficult to achieve with supercomputer hardware [11]. As an example, Table 1 shows the comparison of the execution times of PVM using a network of RS/6000 workstations over a 1.2 MB/sec Ethernet and the iPSC/860 hypercube [12].

Table 1

Comparing Execution Time (secs) for a Molecular Dynamics Application [12]

Molecular Dynamics Simulation			
PVM procs	Problem size 5x5x5	8x8x8	12x12x12
1	23	146	1030
2	15	91	622
4	12	62	340
8	6	34	184
iPSC/860 procs			
1	42	202	992
2	22	102	500
4	11	52	252
8	6	27	129

Overall, the performance of a molecular dynamics application shows the viability of using network of computers to achieve supercomputer performance. Even higher performance is expected as faster networks become available [12].

Parallel Virtual Machine

Overview

Parallel Virtual Machine (PVM) was developed by the Heterogeneous Network Computing research project team, a collaborative effort by researchers at Oak Ridge National Laboratory, the University of Tennessee, Emory University, and Carnegie Mellon University specifically to make heterogeneous parallel computing

easier. This software is distributed freely in the interest of advancement of science and is being used in computational applications around the world [4, 23].

PVM enables a collection of heterogeneous computer systems to be viewed as a single parallel virtual machine. PVM may be implemented on a hardware base consisting a different machine architectures, including single CPU systems, vector machines, and multiprocessors. These computers may be interconnected by one or more networks. Application programs access these computers via a standard interface that supports common concurrent processing paradigms in the form of well-defined primitives that are available for procedural languages, such as C, C++, and Fortran [23]. This programming interface is straight forward and allows simple program structures to be implemented in an intuitive manner. The application program is a collection of cooperating tasks. PVM provides routines that allow initiation and termination of these tasks as well as communication and synchronization among tasks. Figure 1 shows architectural overview of the PVM system.

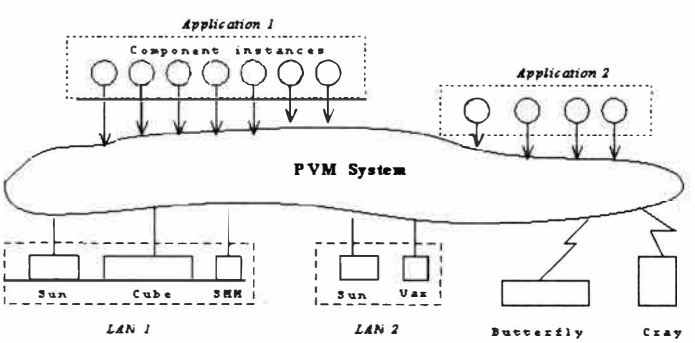


Figure 1. PVM Architectural Model [12].

Application programs view the PVM system as a single general and flexible computing resource. PVM transparently handles all message routing, data conversion, task scheduling and group management across a network of incompatible computer architectures. Communication constructs include those for sending and receiving data (in the form of message) as well as high-level primitives such as broadcast, barrier synchronization, gather, scatter, and global reduction. These high-level operations are also known as collective communication operations.

PVM System

PVM system consists of two parts. The first part is the daemon called *pvmd* which takes care of communication between machines such as send and receive data or command. The second part is a library function called libpvm which provides functions for developer to interface with *pvmd*, pack/unpack data, and send/receive data between tasks. Libpvm is now available for C/C++ and Fortran [11].

On each host of a virtual machine, there is one *pvmd* running. *Pvmds* owned by one user do not interact with those owned by others to reduce security risks. This feature provides different users to have their own configuration and host mapping can be overlapped over the network [11].

PVM also provides group management for tasks such as join group, leave group, and get group members list. The *pvmd* does not perform the group functions. By having a group server running as one of the tasks in the system, every task which

wants to maintain a group has to contact group server. In PVM, the group functions are designed to be very general and transparent to the user. Any PVM task can join or leave any user named group at any time without having to inform any other task in the affected group [11].

How PVM works

Task Identifier

PVM system uses a unique task identifier (TID) to address *pvmds*, tasks, and group of tasks within a virtual machine. TID is designed to fit into the 32-bit integer data type which is available on a wide range of machines. TID has four fields which are S, G, H, and L. Figure 2 shows the contents of TID.

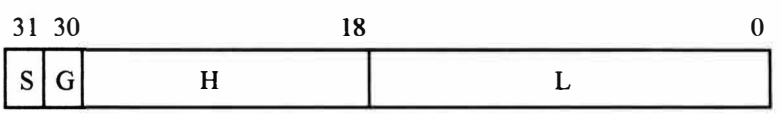


Figure 2. 32-bit Generic Task Identifier in PVM.

For *pvmd's* address, S bit is set, with host number set in the H field and L field is cleared. Therefore, the maximum number of hosts in a virtual machine is limited to $2^H - 1$, which is $2^{12} - 1 = 4095$.

For the address of each task, H field is set to host number where the task is running and L field is assigned by *pvmd*. Therefore, up to $2^{18} - 1$ tasks can exists

concurrently on each host. *Pvmd* maintains a mapping between L values and real process id's in the native operating system.

The G bit is set to form multicast addresses (GIDs). This addresses refer to groups of tasks. Multicasting is described later in this chapter. Table 2 shows the use of TID spaces.

Table 2

The Use of TID Spaces

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

Communication Model

PVM communication is based on TCP, UDP, and Unix-domain sockets, because these protocols are generally available. There are three connections to consider: Between pvmds, between pvmd and task, and between tasks.

Pvmd-pvmd. PVM daemons (*pvmds*) communicate with one another through UDP sockets which are unreliable, therefore an acknowledgment and retry mechanism is used to avoid losing, duplicating, or reordering packets. UDP also limits packet length, thus PVM fragments long messages. The main reason why UDP sockets are

used is because TCP sockets consume file descriptors. In a virtual machine of N hosts, each *pvmd* must have TCP connections to other N - 1 hosts. This occupies a number of file descriptors.

Pvmd-Task and Task-Task. A task communicates with *pvmd* and other tasks through TCP sockets which are reliable. In PVM version 3.3, the Unix Domain sockets are used for Pvmd-Task communication. UDP is unacceptable here because it needs acknowledgment and retry mechanism at both ends but tasks cannot be interrupted while computing to perform I/O.

PVM daemons and tasks can compose and send messages of arbitrary lengths containing typed data. The sender of messages does not wait for an acknowledgment from the receivers, but continues as soon as the message has been transferred to the network and the message buffer can be safely deleted or reused. Messages are buffered at the receiving end until received by a task. Blocking and non-blocking receive primitives are provided. There is no acknowledgment used between sender and receiver. Messages are reliably delivered and buffered by the system.

Message Routing

Default Message Routing. The message is routed through *pvmds*. For example, if task T_A wants to send a message to task T_B, the message will be sent to local *pvmd* on the machine on which T_A is running. There are two cases: (1) T_A and

T_B are running on the same machine, (2) T_A and T_B are running on different machines. When T_A and T_B are on the same machine, the local *pvmd* will send message to T_B as shown in Figure 3 (a). When T_A and T_B are on different machines, the local *pvmd* of T_A routes the message to remote *pvmd* using UDP in the machine where T_B is running, then the remote *pvmd* transfers the message to T_B when requested by a pvm_recv() function call by T_B as shown in Figure 3 (b).

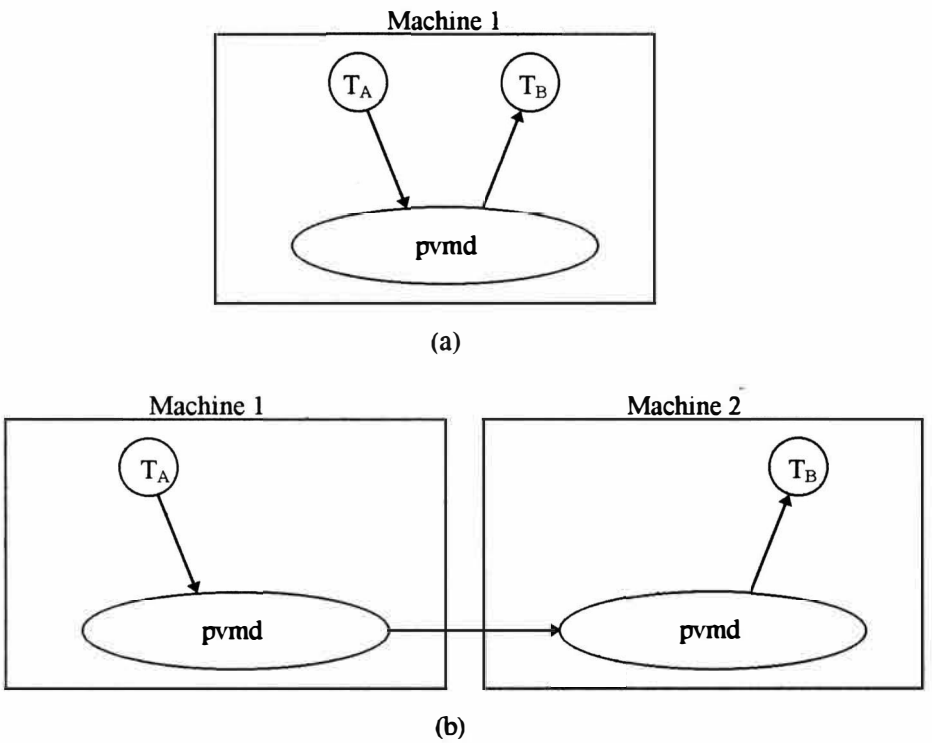


Figure 3. (a) T_A and T_B Are on the Same Machine, (b) T_A and T_B Are on Different Machines.

Direct Message Routing. PVM provides task-to-task direct route communication, so that a task can communicate with other tasks directly through TCP

sockets. This direct route mechanism reduces the overhead of forwarding a message through the *pvmds*, but it consumes a file descriptor for each direct route to task.

Multicasting. The libpvm function pvm_mcast() sends a message to multiple destinations in one call. The current implementation only routes multicast message through the *pvmds* using 1:N fanout mechanism where message is one-by-one routed from local *pvmd* to other *pvmds* and one-by-one routed from each *pvmd* to its local tasks.

Libpvm Communication Routines

pvm_send() and pvm_recv()

Sending a message in PVM version 3.3 requires three function calls. First, a send buffer is initialized by pvm_initsend() function call. This step includes clearing the previous send buffer. Second, the message is packed into this buffer using any number and combination of pvm_pk*() function calls. At this step the message is encoded and the buffer is also fragmented if the message is too long. Third, the message is sent to destination process by pvm_send() call which uses the Default Message Routing as a default routing.

There are some advantages of having this three step method. First, the user can pack a message with several data with different types. The advantage is that only one pvm_send() function call is required instead of every call for each data type. This

causes the startup latency for communication over a network to be reduced because packing is faster than sending data over a network. Another advantage is that the encoding and fragmenting of the message is done only once if this message is to be sent to several different destinations. PVM takes advantage of packing only once when a user broadcasts a message.

The pvm_recv() function call is a blocking receive method provided by PVM to receive a message. This blocking receive waits until the message is copied to the receive buffer. Wildcards can be specified in the receive for the source. This makes the pvm_recv() call to receive the message from any source.

pvm_mcast()

The multicast operation has two steps. First, during the initiation step, a task sends a TM_MCA message to its *pvmd*, containing a list of destination TIDs. The *pvmds* create a multicast descriptor (struct mca) and group identifier (GID). Then it notifies each host, one by one, which has destination tasks running by sending a message DM_MCA to the *pvmd* on that host, containing a list of TIDs running on that host and GID, and return the GID back to the task. After receiving DM_MCA message, each *pvmd* on each host then creates a multicast descriptor for that GID. Second step is the message sending step, in which the task sends a message using GID as a destination address. The message is routed to the local *pvmd*. The local *pvmd* knows that the address is multicast address. It sends the message to each *pvmd* listed

in a multicast descriptor (struct mca) of that GID one by one. *Pvmds* then send the message to each task belonging to them. This local message multicasting is done sequentially, that is, a message is sent to a task before sending to another local task.

Collective Communication Library

A Collective Communication Library (CCL) for parallel computers includes frequently used operations such as broadcast, reduce, scatter, and gather. This library provides users with a convenient programming interface, efficient communication operations, and the advantage of portability.

Process Group in CCL

A process group is an ordered set of processes that has a unique name in the system. Each process group is identified by a unique process Group Identifier (GID) in an application program. We next define the operations typically contained in a CCL.

Broadcast

Broadcast a message from one process to all processes in the group. The sending process will be referred as the *root* and the other processes in the group will be referred as the *client* in the rest of the thesis. Figure 4 depicts this operation pictorially.

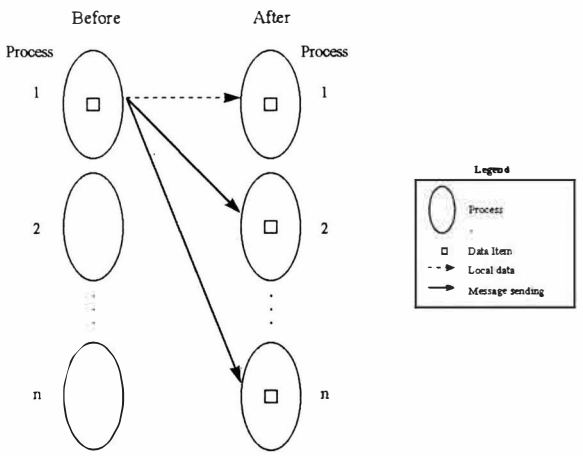


Figure 4. Broadcast Operation.

Scatter

Distribute distinct message from a single source to each process in a group. From Figure 5, process 1 is the root of scatter operation.

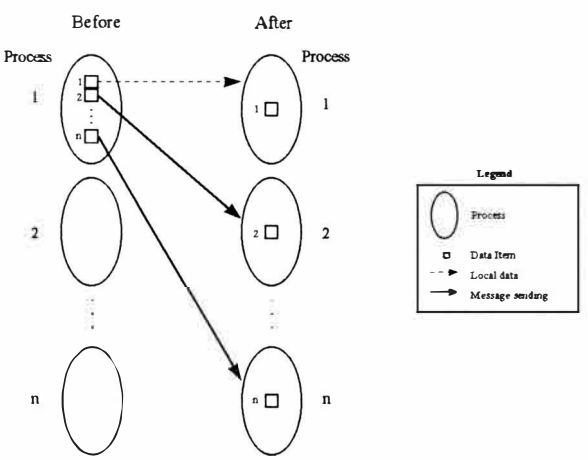


Figure 5. Scatter Operation.

Gather

Gather distinct messages from each process in a group to a single destination process. It is opposite of the scatter operation. From Figure 6, process 1 is the root of gather operation.

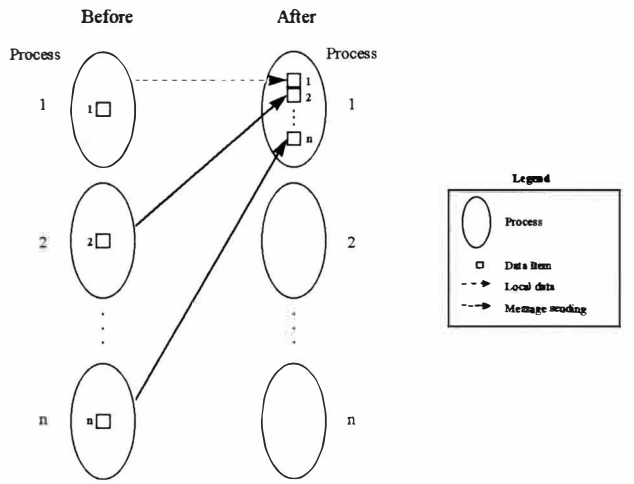


Figure 6. Gather Operation.

Reduce

Apply an associative reduction operation on local data of all the processes in a group and place the reduction result in a specified destination process. From Figure 7, process 1 is the specified destination process which is referred to as the root of reduce operation.

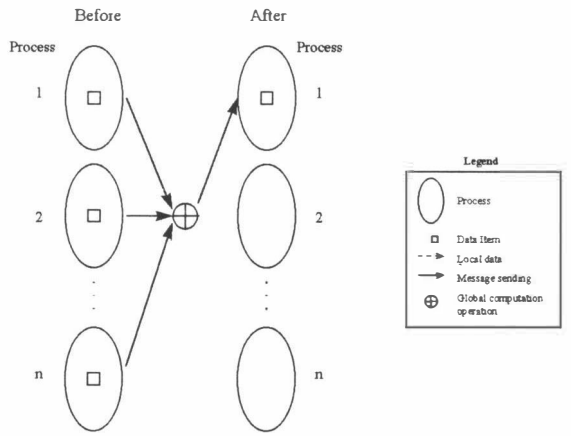


Figure 7. Reduce Operation.

CHAPTER III

PREVIOUS APPROACHES

Iterative Pvmd

Pvmd is an iterative server. It means that pvmd will repeatedly check if there is any packet in the output queue, then it will send the packet out to destination, or if there is any input from UDP or Unix domain sockets, then it will receive from the socket. The algorithm of the main working part of pvmd is as follows.

Assume:

netoutput() is the function to send all packets in which destination is a remote pvmd.

netinput() is the function to read a packet from another pvmd and process the message.

locloutput(*i*) is the function to send a packet to a local task *i*.

loclinput(*i*) is the function to read a packet from a local task *i* and process the message.

In(*s*) indicate that there is an input from socket *s*.

Out(*s*) indicate that there is an output to socket *s*.

netsock is the UDP socket for pvmd-pvmd communication.

loclsock^i is the Unix domain socket for pvmd-task communication of task *i*.

The algorithm of main working part of pvmd:

```
for (;;) {          / *infinite loop */
        netoutput();            / *send all packets in outgoing queue */
        if ( In( netsock ) )
                netinput();
        if ( In( loclsock^i ) )
                loclinput(i);
        for (i = 1; i ≤ number of local tasks; i++) {
                if ( Out(loclsock^i ) )
                        locloutput(i);
        }
}
```

Timing Model

The timing model for sending a d-byte message from process p to process q is $T = t_s + dt_c$ where t_s is the overhead (startup time) associated with each send or receive operation and t_c is the communication time for sending each additional byte.

The following definitions are used in Chapter III and Chapter IV.

Define:

t_{s1} is startup time for pvmd-task communication using Unix domain socket.

t_{s2} is startup time for pvmd-pvmd communication using UDP socket.

t_{c1} is communication time for sending additional byte using Unix domain, TCP or UDP socket in the same machine.

t_{c2} is communication time for sending additional byte using TCP or UDP socket over the network.

p is the number of pvmds (hosts).

s is size of TID (4 bytes in PVM 3.3).

m is the number of TID in TID-list, group size.

n_i is the number of TID on host i, $\sum n_i = m$.

d is size of data.

Timing Model for pvm_send() and pvm_recv()

Terminology

Synchronous Send: A synchronous send returns only when the receiver has posted a receive.

Asynchronous Send: An asynchronous send does not depend on the receiver calling a matching receive.

Blocking Send: A blocking send returns as soon as the send buffer is free for reuse, that is, as soon as the last byte of data has been sent or placed in an internal buffer.

Blocking Receive: A blocking receive returns as soon as the data is ready in the receive buffer.

Non-blocking Receive: A non-blocking receive returns as soon as possible, that is, either with a flag that the data has not arrived yet or with the data in the receive buffer.

According to our terminology, the PVM communication model provides only asynchronous blocking sends. Therefore, the PVM user does not have to worry either about any deadlocks for non-matching pairs of pvm_send()/pvm_recv() or about rewriting into a buffer after it has been sent. PVM provides blocking receives and non-blocking receives [7], but we will consider only blocking receives because the current implementation of collective operations uses only blocking receives.

We still use the dynamic groups which are provided by group server, because the current approaches use it, but note that this will results in more costs than static groups facility. Discussion of static groups facility is out of the scope of this thesis, further a group of PVM developing team is working on it. Our goal in this thesis was to improve CCL in dynamic group environment.

The time for a *d*-byte message for a matching pvm_send() and pvm_recv() using the default routing mechanism, according to the previous chapter, can be modeled as shown in Figure 8.

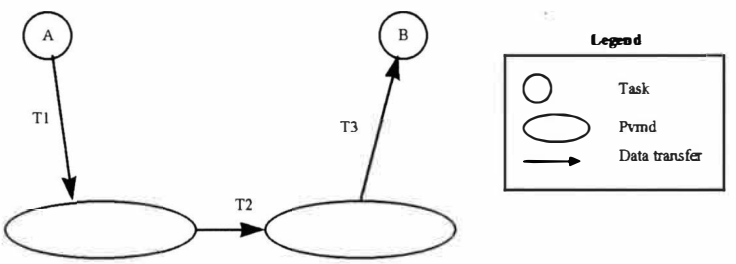


Figure 8. Times Used to Perform pvm_send()/pvm_recv() Routines.

Let T_1 be the time used for routing message from sender to local pvmd.

T_2 be the time used for routing message from local pvmd to remote pvmd.

T_3 be the time used for routing message from local pvmd to receiver.

Hence, T_i's could be expressed as

$$T_1 = t_{s1} + dt_{c1}$$

$$T_2 = t_{s2} + dt_{c2}$$

$$T_3 = t_{s1} + dt_{c1}$$

Note that T_2 can be 0 if A and B are on the same machine and this model does not include pack/unpack timings.

Pvmd being an iterative server, T1 and T2 cannot be overlapped, so does T2 and T3. T1 and T3 can be overlapped only if task A and B are on different machines.

These timing models of T_1, T_2 and T_3 will be used later to describe the times used to perform scatter, gather and reduce operations.

Timing Model for pvm_mcast()

According to how pvm_mcast() works from the previous chapter, the timing model of pvm_mcast() operation can be modeled as shown in Figure 9.

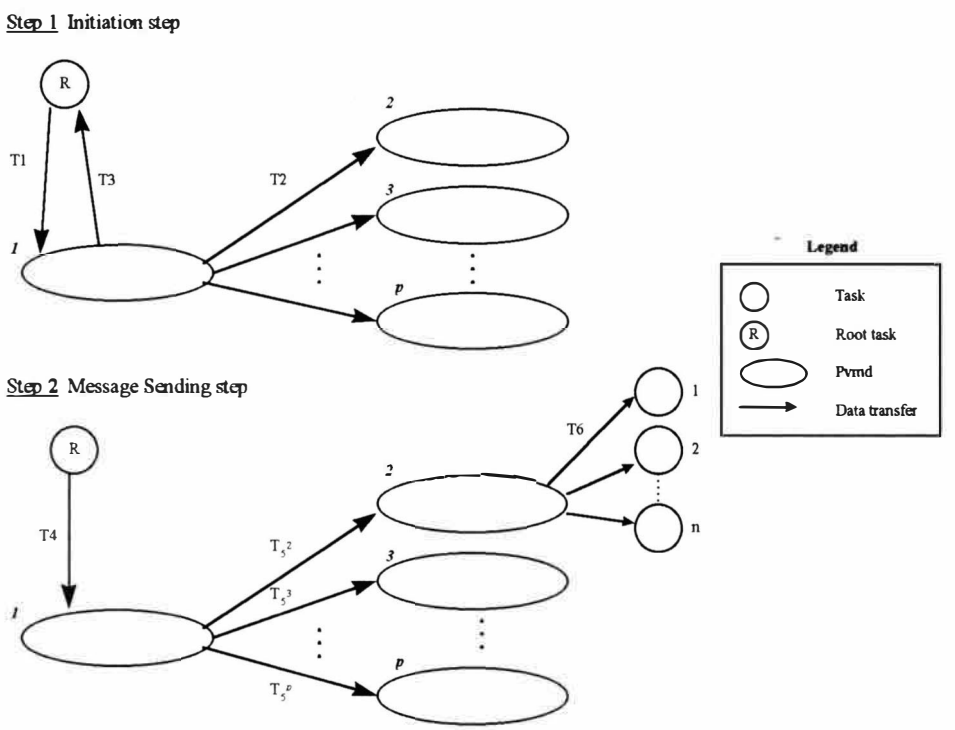


Figure 9. Timing for Performing pvm_mcast().

Let T_1 be the time used for sending TIDs list from sender to local pvmd.

T_2 be the time used for notifying other pvmds in pvmds list.

T_3 be the time used for sending GID back to sender.

T_4 be the time used for sending the message to local pvmd.

T_5^i be the time used for sending the message to the $pvmd^i$.

T_6^i be the time used for sending the message from $pvmd^i$ to local tasks.

$T_{5,6}^i$ be the time used at $pvmd^i$ assuming messages have been sent to $pvmd^2, \ldots, pvmd^{i-1}$.

$T_{5,6}$ be the total time used by pvm_mcast() for p hosts, once message has been read by $pvmd^1$ in step 2.

Hence T_i's could be expressed as

$$T_1 = t_{s1} + smt_{c1}$$

$$T_2 = \sum_{i=2}^{p} (t_{s2} + sn_i t_{c2}) = (p\text{-}1)t_{s2} + s\sum_{i=2}^{p} n_i t_{c2}$$

$$T_3 = t_{s1} + st_{c1}$$

$$T_4 = t_{s1} + dt_{c1}$$

$$T_6^i = n_i(t_{s1} + dt_{c1})$$

$$T_{5,6}^i = (i\text{-}1)(t_{s2} + dt_{c2}) + T_6^i$$

$T_{5,6} = \max_{i=2}^{p} \left((i\text{-}1)(t_{s2} + dt_{c2}) + T_6^i \right)$ (since messages have to be first sent to $pvmd^2$, ..., $pvmd^{i-1}$, all of which potentially could be sending messages to their local tasks in parallel.)

From Figure 9, the time used to perform pvm_mcast() can be modeled as:

$$T_{pvm\_mcast} = T_1 + T_2 + T_3 + T_4 + T_{5,6}$$

For load balanced situations, we have $n_i = m/p$ for $\forall\ i$. Therefore,

$$T_{5,6} = (p\text{-}1)(t_{s2} + dt_{c2}) + (m/p)(t_{s1} + dt_{c1})$$

and

$$T_{pvm\_mcast} = T_1 + T_2 + T_3 + T_4 + (p\text{-}1)(t_{s2} + dt_{c2}) + (m/p)(t_{s1} + dt_{c1})$$

This timing model for pvm_mcast() will be used later to describe the time used to perform broadcast operation because the current pvm_bcast() simply call pvm_mcast() to broadcast the data.

Applying Timing Model to Current Approaches of CCL

In this section, we define some more variables which are used to model the timing of current approaches of each collective operations in PVM.

Define:

$S = \{z \mid z \text{ is TID}\}$.

$|G|$ where $G \subseteq S$ is the number of members of G.

g_i is the i^{th} member of G, $g_i \in G$ where $G \subseteq S$.

T_g is the time used for obtaining TIDs list of the specified group from the group server.

t_m is the time used for copying data byte by byte.

Broadcast

The current approach in the broadcast operation uses pvm_mcast() function to perform the operation. The root of the operation first obtains TIDs list of the specified group from the group server then sends data to the other tasks (*clients*) in the group using pvm_mcast() function and copying the *data* to *result* buffer for itself. The clients simply call pvm_recv() function to wait and receive the data.

The current pvm_bcast() algorithm is as follows:

Assume:

G, G' $\subseteq$ S.

root is the TID of the root task.

data is the buffer for data.

result is the buffer for result.

Copy(*src*, *dst*) is the function to copy from *src* to *dst*.

GetTidList(*gname*) is the function to get the list of TIDs in group *gname* from group server.

Root task of the broadcast operation:

```
G  = GetTidList(gname);        /* obtain from group server */
G' = G - {root};
Copy(data, result);                    /* for itself */
pvm_mcast(G', data);
```

Client tasks of broadcast operation:

```
pvm_recv(root, result);        /* receive message from root and
                                  put it into result */
```

From the pvm_mcast() timing, assuming that $n_i = m/p$ for $\forall\ i$, the time used to perform the broadcast operation is

$$\begin{aligned} T_{bcast} &= T_g + T_{pvm\_mcast} + dt_m \quad ; \text{for } m = |G'| \\ &= T_g + T_1 + T_2 + T_3 + T_4 + \\ &\quad (p\text{-}1)(t_{s2} + dt_{c2}) + (m/p)(t_{s1} + dt_{c1}) + dt_m \quad \ldots\ldots\ldots\ldots\ldots\ldots (3.1) \end{aligned}$$

where T_i's are T_i's of pvm_mcast() timing and dt_m is the time used to copy *data* to local *result* buffer.

Scatter

The current approach of scatter operation uses a linear algorithm to distribute data[i] to process i. The root of the operation first obtains TIDs list of the specified

group from the group server then sends data[i] to the client task i by using pvm_send() function. The data are received by pvm_recv() function called by tasks other than root (clients tasks) in the group. Data for root is directly copied to result buffer.

The algorithm of current pvm_scatter() operation is as follows:

Assume:

$G \subseteq S$.

root is the TID of root task.

data is the buffer for data.

$data^i$ is the data for task i.

result is the buffer for result.

Copy(*src*, *dst*) is the function to copy from *src* to *dst*.

GetTidList(*gname*) is the function to get the list of TIDs in group *gname* from group server.

Root task of the scatter operation:

```
G = GetTidList(gname);
for (i = 1; i <= |G|; i++) {
        if (g_i == root)
                Copy(data^i, result);
        else
```

pvm_send(g_i, *data*^{i});

}

Client tasks of scatter operation:

pvm_recv(*root*, *result*); /* receive message from root and put it into result */

Figure 10 shows the time used by this scatter algorithm.

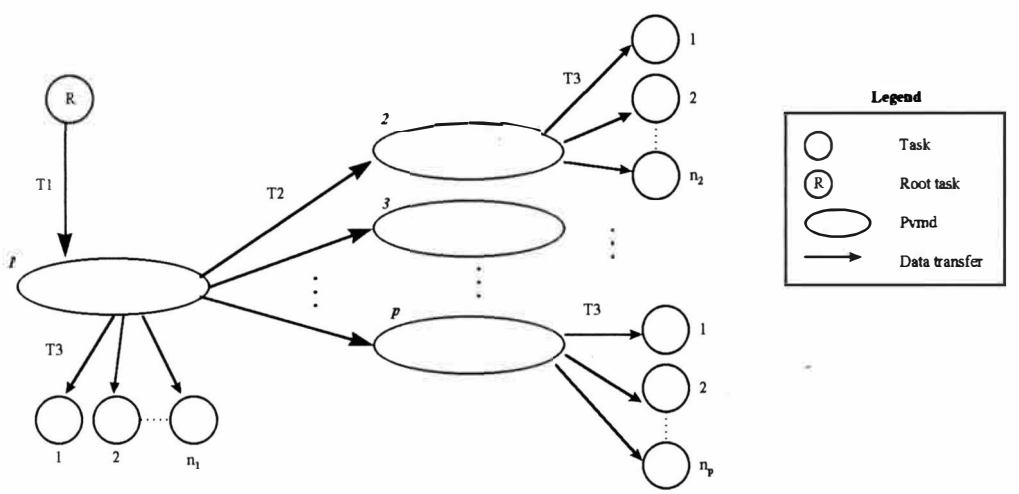


Figure 10. Time Used by Current Scatter Algorithm.

Consider T_3's, all T_3's on the same machine cannot be overlapped. but T_3's on different machine can be overlapped. T_1 and T_3 can be overlapped except T_1 and T_3 that are used for sending data from root task to the clients task on the same machine. The sending paths are not ordered. This means that the root task will send a message to each client tasks in any order. From current implementation of pvm_scatter(), the

message routes are ordered by *instance number* of the client tasks which can be placed in arbitrary order when the tasks join the group.

According to the timing model for pvm_send() and pvm_recv(), assume that T_1 is equal to T_3, all T_1's are equal and all T_3's are equal, because they are the time used to send data within machine using Unix domain sockets. Assume that $n_i = m/p$ for $\forall\ i$, the time used by this pvm_scatter() can be modeled as:

$$T_{scatter} = T_g + (m\text{-}1)T_1 + (m - m/p)T_2 + m/pT_3 + dt_m \quad \text{.....(3.2)}$$

Where m is the number of tasks in the group.

p is the number of hosts (pvmd).

T_i's are T_i's from pvm_send()/pvm_recv() timing model.

This model uses m-1 because the data for root task is copied to the result buffer instead of sent and received by a pair of pvm_send() and pvm_recv() and the time that the root task used to copy data is indicated by the term dt_m. T_1's occur (m-1) times to send (m-1) messages. T_2's occur m-m/p times to send m-m/p messages to (p-1) remote pvmds because T_1's and T_2's cannot be overlapped. T_3's can be overlapped with T_1's except T_3's and T_1's perform by the root pvmd which occur m/p-1 times. On other hosts, T_3's are overlapped with T_1's. Therefore, T_3's are reduced from (m-m/p)T_3 to one T_3. Therefore, the extended time performed by T_3's are m/p times.

Gather

The current approach of gather operation uses a linear algorithm to gather *data*[*i*] from process *i*. The root of the operation first obtains TIDs list of the specified group from the group server and then calls pvm_recv() *m* times to wait and receive data from client task 1 to client task *m*. The client tasks simply call pvm_send() to send their data to root. Data of root is locally copied to the result buffer. The current pvm_gather() algorithm is as follows:

Assume:

$G \subseteq S$.

data is the buffer for data.

result is the buffer for result.

result^i is the result buffer for result from client task *i*.

Copy(*src*, *dst*) is the function to copy from *src* to *dst*.

GetTidList(*gname*) is the function to get the list of TIDs in group *gname* from group server.

Root task of the gather operation:

```
G = GetTidList(gname);
for (i = 1; i <= |G|; i++) {
    if (g_i == root)
```

Copy(*data*, *result*[*i*]);

else

pvm_recv(g_i, *result*[*i*]);

}

Client tasks of gather operation:

pvm_send(*root*, *data*);

Figure 11 shows the time used by pvm_gather().

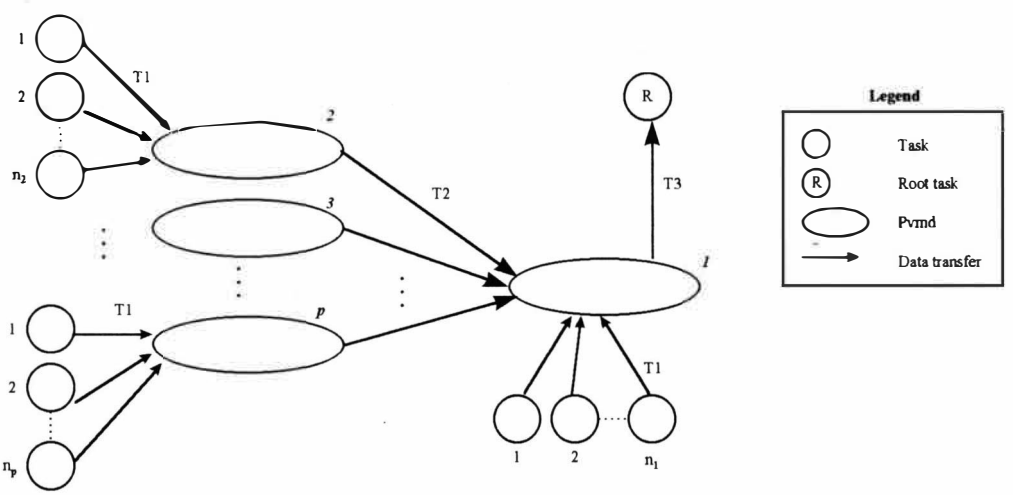


Figure 11. Time Used by Current Gather Algorithm.

Consider T_1's, all T_1's on the same machine cannot be overlapped but T_1's on different machines can be overlapped. T_1 and T_3 can be overlapped except T_1 and T_3 that are used for sending data from client tasks to the root task on the same machine. All T_3's cannot be overlapped. The sending paths are not ordered. This means that

the root task will receive a message from each client tasks in any order. From current implementation of pvm_gather(), the message routes are ordered by *instance number* of the client tasks which can be placed in arbitrary order when the tasks join the group.

According to the timing model for pvm_send() and pvm_recv(), assume that T_1 is equal to T_3 because they are the time used to send data within machine using Unix domain sockets. Assume that $n_i = m/p$ for $\forall\ i$, the time used by this pvm_gather() can be modeled as:

$$T_{gather} = T_g + (m/p-1)T_1 + (m-m/p)T_2 + (m-1)T_3 + dt_m \quad \text{.....(3.3)}$$

Where m is the number of tasks in the group.

T_i's are T_i's of pvm_send()/pvm_recv() timing.

This model use $(m-1)$ T_3 because the data for root task is copied to the result buffer instead of sent and received by a pair of pvm_send() and pvm_recv() and the time that the root task used to copy data is indicated by the term dt_m. T_1's can be overlapped with T_3's except T_1's and T_3's performed by the root pvmd which occur $m/p-1$ times, therefore, T_1's occur $m/p-1$ times. T_2's occur $m-m/p$ times to receive $m-m/p$ messages from remote pvmds.

Reduce

The current approach of reduce operation uses a linear algorithm. First, all tasks in the group obtain the TIDs list of the local tasks in the group on their hosts

along with which task is the coordinator task on that host. Then, each task on host *i* sends its data to coordinator on host *i* using pvm_send(). Coordinator on host *i* receives data using pvm_recv() and then performs the global computation operation on those data and sends the result to root task using pvm_send(). The root task then receives results from each coordinator on host *i* and performs the global computation operation on those result and puts the final result to the result buffer. Figure 12 shows the time used by pvm_reduce().

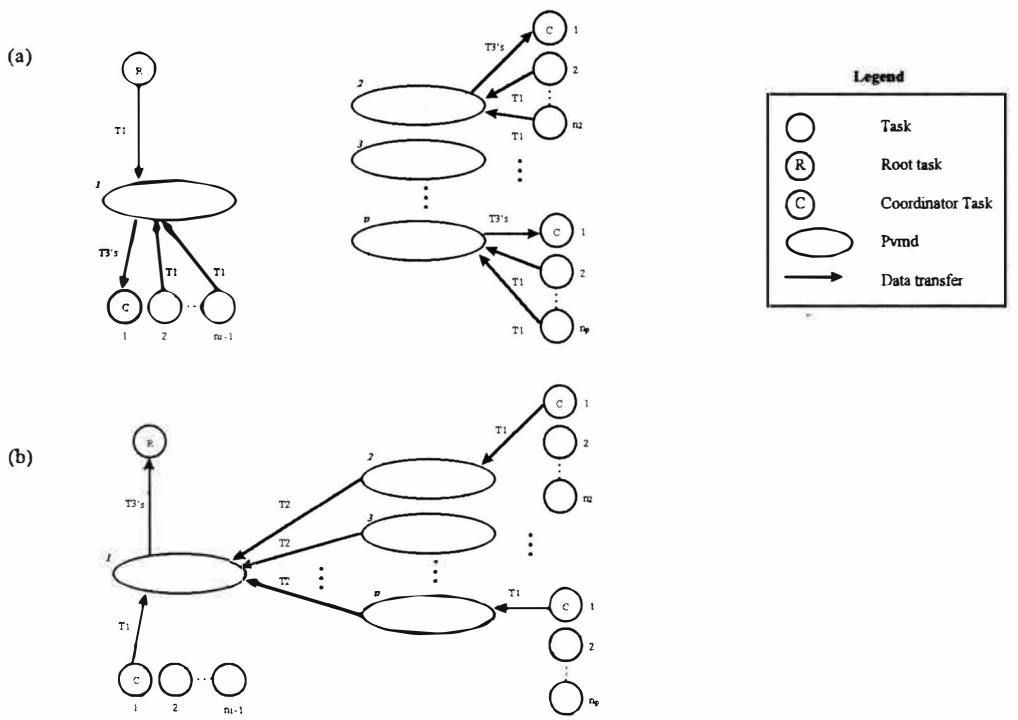


Figure 12. Time Used by pvm_reduce(): (a) All Local Tasks Send Their Data to Coordinator to Perform the Local Reduction, (b) All Coordinator Send Their Result to Root to Perform the Global Reduction.

The current pvm_reduce() algorithm is as follows:

Assume:

$G \subseteq S$.

root is the TID of root task.

coordinator is the TID of local coordinator.

coordinator ^{i} is the TID of coordinator of host *i*.

data is the buffer for data.

data' is the buffer for temporary data.

result is the buffer for result.

result^i is the result buffer for result from client task *i*.

Copy(*src*, *dst*) is the function to copy from *src* to *dst*.

IsCoordinator() indicate that this task is coordinator.

GetLocalTidList(*gname*) is the function to get the list of local TIDs in group *gname* from group server.

Root task of the reduce operation:

```
G = GetLocalTidList(gname);
/* Compute reduce operation on data residing on this host */
if ( IsCoordinator() ) {
        for (i = 0; i < (|G|-1); i++) {
                pvm_recv(g_i, data');
```

```
                    data = GlobalOperation(data, data');
                    /* root & coordinator, so keep data for global op. of
                       coordinator's data */
                }
            }
            else {  /* not a coordinator but root */
                pvm_send(coordinator, data);
                pvm_recv(coordinator, data);  /* receive the final result */
            }
            /* Compute reduce op. on data from different coordinators, one per
               host */
            for (i = 0; i < nhosts; i++) {
                pvm_recv(coordinator i, data');
                data = GlobalOperation(data, data');
}
```

Client tasks of reduce operation:

```
        G = GetLocalTidList(gname);
        if ( IsCoordinator() ) {
            for (i = 0; i < (|G|-1); i++) {
                pvm_recv(g_i, data');
```

```
            data = GlobalOperation(data, data');
        }
        pvm_send(root, data);
    }
    else
        pvm_send(coordinator, data);
```

From the pvm_reduce() algorithm, we ignore the time used to perform global computation operation. The time used to perform pvm_reduce() (T_{reduce}) can be expressed as:

$$T_{reduce} = mT_{g+}(m/p - 1)T_1 + (m/p - 1)T_3 + T_1 + (p - 1)T_2 + (p - 1)T_3 \quad .....(3.4)$$

Where m is the number of tasks in the group.

T_i's are T_i's of pvm_send()/pvm_recv() timing.

Tg occurs m times because every task obtain the TIDs list from a group server. This model use m/p -1 because the data of coordinator task are not sent to other tasks. The term $(m/p - 1)T_1 + (m/p - 1)T_3$ is the time that the each local task uses to send data to local coordinator, and $T_1 + (p - 1)T_2 + (p - 1)T_3$ is the time that root task used to receive remote result from other coordinator tasks.

CHAPTER IV

OUR SOLUTION

Different Approaches

We propose new approaches to make collective communication operations in PVM more efficient. The timing model of the current pvm_bcast(), pvm_scatter(), pvm_gather() and pvm_reduce() from equation 3.1, 3.2, 3.3 and 3.4 in previous chapter show that the performance depends on the times used by the pvmd-task and pvmd-pvmd communication. We propose new solutions which try to reduce the overhead and the communication over the local area network.

The improvement in performance of a CCL in PVM can be achieved by using the properties of LAN. From a software point of view, PVM considers the network of workstations as a number of pvmds. We divide the virtual machine into each host running a pvmd at the lowest layer. Instead of handling by tasks, the new approaches of collective communication operations are handled by pvmd. Each pvmd takes care of data of local tasks.

For the Pvmd-Task communication, the data exchange between tasks and local pvmd in the same host can be efficiently achieved by using Interprocess Communication (IPC). The current implementation uses Unix domain socket to achieve this. We select shared memory since it exhibits a better performance [18] and

CCL involves a group of data which can be viewed as an array of data. For the pvmd-pvmd communication, because CCL involves the communication to a group of pvmds we develop the new approach – the one-to-all communication using IP broadcasting or IP multicasting mechanism. This approach uses the advantage of LAN such as Ethernet.

We refer to our approach for Collective Communication Library in PVM as Shared Memory with IP Broadcasting or IP Multicasting mechanism.

Using Shared Memory for Pvmd-Task Communication

The use of shared memory for data exchange between pvmd and tasks on a host i can be viewed as Figure 13 where n_i is the number of PVM tasks running on host i. Semaphores are used for synchronization.

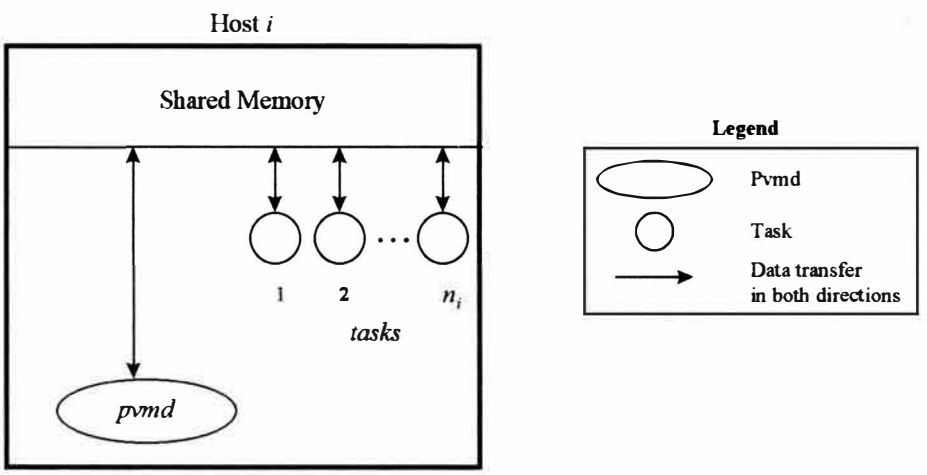


Figure 13. Data Exchange Using Shared Memory.

The shared memory is created at the time pvmd is starting up. More than one process can read the shared memory simultaneously. This means that it does not have to wait until other processes finish their reading. The shared memory is divided into sections of size of the data size. Only one process can write the data to each shared memory section. All writings to each section of the shared memory have to wait for the mutual exclusion semaphore and release after finish. All sections of shared memory, on the other hand, can be viewed as an array of data buffer. This has an advantage because the collective operations such as scatter and gather view the data as an array of data. Therefore, the single copy call can copy all continuous data instead of iterative copy data to or from each section of memory.

Using IP-Broadcast or IP-Multicast for Pvmd-Pvmd Communication.

This approach adds the IP broadcasting or IP multicasting to the UDP socket. A packet-loss recovery mechanism is added to make it reliable. IP Broadcasting sends a packet to all machines on the local area network while IP multicasting sends a packet to a group of machines [17]. We implement this approach in both IP broadcasting and IP multicasting mechanisms to compare their performance. The pvmd-pvmd communication using IP broadcasting or IP multicasting mechanism can be viewed as Figure 14.

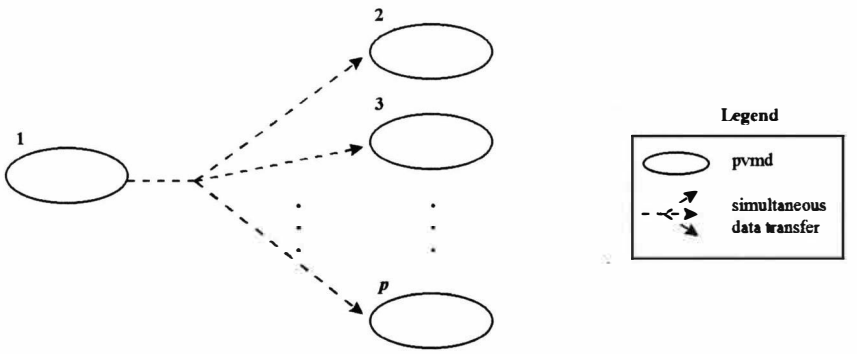


Figure 14. The Pvmd-Pvmd Communication Using IP Broadcasting or IP Multicasting Mechanism.

CCL Using IP Broadcasting and IP Multicasting Mechanism

We next describe how collective communication operations in PVM can be implemented using IP broadcasting or IP multicasting mechanism instead of using unicast UDP sockets. This mechanism reduces the time that the root pvmd sends data to all client pvmds almost down to the time used for sending message to one pvmd.

Define:

S = {z | z is TID}.

|G| where $G \subseteq S$ is the number of members of G.

g_i is the i^{th} member of G, $g_i \in G$.

T_g is the time used for obtaining TIDs list of the specified group from the group server.

t_m is the time used for copying data byte by byte.

p is the number of pvmds (hosts).

s is size of TID (4 bytes in PVM 3.3).

m is the number of TID in TID-list, group size.

n_i is the number of TID on host i, $\sum n_i = m$.

d is size of data.

In all algorithms of this chapter, for simplicity, the code for mutual exclusion to access shared memory is not indicated but exists in the implementation. The message type TM_* means the control message sent between task and pvmd, and the message type DM_* means the control message sent between pvmd and pvmd. The *pvmd-list* is the array of the TIDs of pvmds involved in the collective operation.

Broadcast

The root task of the broadcast operation first obtains TIDs list of the specified group from the group server and copies them and the data to the shared memory. Then, it sends the request for broadcast operation to local pvmd (root pvmd) along with the number of TIDs in the group. Root pvmd reads the TIDs list and data from shared memory, sorts the TIDs list and creates the pvmd-list. Then, root pvmd sends the pvmd-list and data to other pvmds using IP broadcasting or IP multicasting mechanism. After receiving, each pvmd writes the data to the shared memory and gives access to local tasks. Tasks then read the data from shared memory

simultaneously. Figure 15 shows the time used in broadcast operation using IP broadcasting or IP multicasting mechanism.

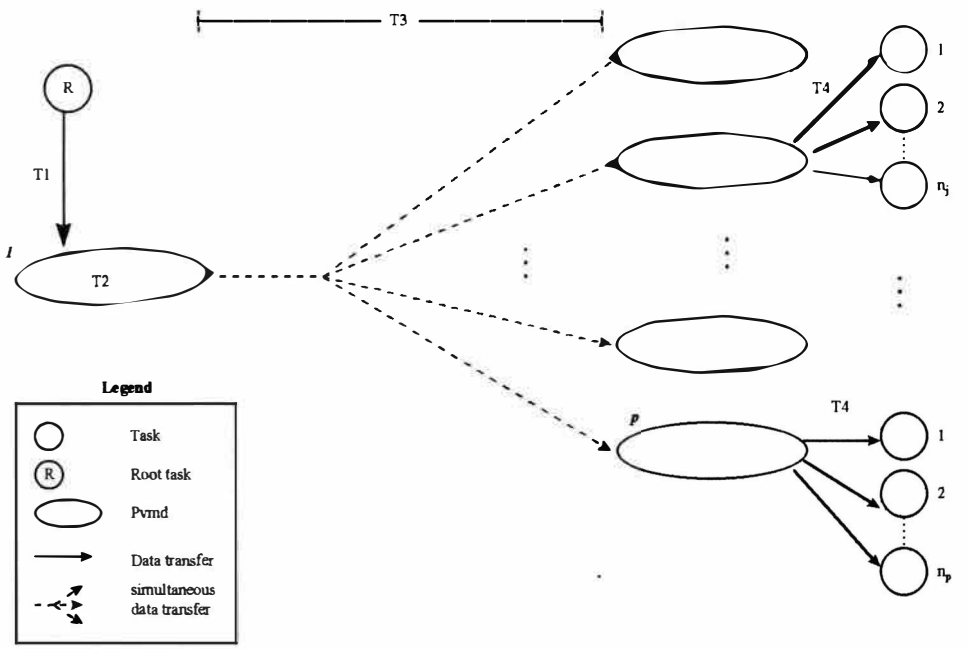


Figure 15. Time Used by Broadcast and Scatter Operation Using IP Broadcasting or IP Multicasting Mechanism.

In this algorithm of the broadcast operation, the message type TM_BCAST is the request message which is sent to root pvmd by root task. The message type DM_BCAST is the message that contains pvmd-list which is sent by root pvmd to other pvmds. The algorithm of the broadcast operation using shared memory and IP broadcasting or IP multicasting mechanism is as follows:

Assume:

G, G' $\subseteq$ S.

root is the TID of the root task.

data is the buffer for data.

result is the buffer for result.

Shmem is the shared memory.

Copy(*src*, *dst*) is the function to copy from *src* to *dst*.

GetTidList(*gname*) is the function to get list of TIDs in group *gname* from group server.

wait(*sem*) is the function to wait for semaphore *sem* $\geq$ 1 then subtract by 1.

signal(*sem*, *n*) is the function to increase the value of semaphore *sem* by *n*.

Bcast(*msg_type*, *msg*) is the function to broadcast message type *msg_type* along with message *msg* to all pvmds using IP-broadcasting or IP-multicasting mechanism.

Recv(*msg*) is the function to receive message *msg*.

read_sem is a semaphore initialized to 0.

At the root task of the broadcast operation:

G = GetTidList(*gname*);

G' = G - {*root*};

Copy(G', *Shmem*);

Copy(*data*, *result*);

Copy(*data*, *Shmem*);

Send(*local pvmd*, TM_BCAST, |G'|);

At the pvmds:

TM_BCAST:

Copy(*Shmem*, G);

Copy(*Shmem*, *data*);

Sort G and create pvmd-list;

Bcast(DM_BCAST, pvmd-list and *data*);

DM_BCAST:

Recv(pvmd-list and *data*);

Copy(*data*, *Shmem*);

signal(*read_sem*, the number of local tasks);

At the client tasks of the broadcast operation:

wait(*read_sem*);

Copy(*Shmem*, *result*);

From Figure 15, let

T_1 be the time used to send the number of TIDs in TID-list (request).

T_2 be the time used to copy TIDs list form shared memory.

T_3 be the time used to send pvmd-list and data to all pvmds using IP broadcasting or IP multicasting mechanism.

T_4 be the time used for all local tasks to copy data from shared memory.

We assume that $n_i = m/p$ for $\forall\ i$. Hence, T_i's can be expressed as

$$T_1 = t_{s1} + st_{c1}$$

$$T_2 = (sm + d)t_m$$

$$T_3 = t_{s2} + (sp + d)t_{c2}$$

$$T_4 = (m/p)dt_m$$

Therefore, the time used by broadcast operation using IP-broadcasting or IP-multicasting mechanism can be modeled as:

$$T_{bcast\text{-}1} = T_g + T_1 + T_2 + T_3 + T_4 \quad \text{.....(4.1)}$$

Scatter

The root task of the scatter operation first obtains TIDs list of the specified group from the group server and copies them and all data to the shared memory. Then, it sends the request for scatter operation to local pvmd (root pvmd) along with

the number of TIDs in the group. Root pvmd reads the list of TIDs and all data from the shared memory, sorts them and creates the pvmd-list. Then, root pvmd sends the TIDs list and their data to all pvmd using IP broadcasting or IP multicasting mechanism. Each pvmd writes the data of its local tasks to the shared memory and gives access to the tasks. Tasks read its own data from shared memory simultaneously.

In this algorithm of the scatter operation, the message type TM_SCATTER is the request message which is sent to root pvmd by root task. The message type DM_SCATTER is the message containing pvmd-list and is sent by root pvmd to other pvmds. The algorithm of the scatter operation using shared memory and IP broadcasting or IP multicasting mechanism is as follows:

Assume:

$G \subseteq S$.

data is the buffer of data for all tasks (*m* tasks).

$data^i$ is the data of task *i*.

result is the buffer for result.

Shmem is the shared memory.

Copy(*src*, *dst*) is the function to copy from *src* to *dst*.

GetTidList(*gname*) is the function to get list of TIDs in group *gname* from group server.

wait(*sem*) is the function to wait for semaphore *sem* ≥ 1 then subtract by 1.

signal(*sem*, *n*) is the function to increase the value of semaphore *sem* by *n*.

Bcast(*msg_type, msg*) is the function to broadcast message type *msg_type* along with message *msg* to all pvmds using IP-broadcasting or IP-multicasting mechanism.

Send(*dst*, *msg_type*, *msg*) is the function to send the message type *msg_type* along with message *msg* to destination *dst*.

Recv(msg) is the function to receive message msg.

read_sem is a semaphore initialized to 0.

At the root task of the scatter operation:

G = GetTidList(*gname*);

Copy(G, *Shmem*);

Copy(*data*, *Shmem*);

Send(*local pvmd*, TM_SCATTER, |G|);

wait (*read_sem*);

Copy(*Shmem*, *result*); /* copy $data^i$ to *result* where *i* is root */

At the pvmds:

TM_SCATTER:

Copy(*Shmem*, G);

Copy(*Shmem*, *data*);

Sort G and create pvmd-list;

Bcast(DM_SCATTER, pvmd-list and *data*);

DM_SCATTER:

Recv(pvmd-list and *data*);

Copy(*local data*, *Shmem*);

signal(*read_sem*, the number of local tasks);

At the client tasks of the scatter operation:

wait(*read_sem*);

Copy(*Shmem*, *result*); /* Copy *data*^{l} from shared memory to *result* */

By using IP Broadcast or IP Multicast mechanism. T3 will be the time used to send pvmd-list and the local TIDs and their data of each pvmd in pvmd-list by IP broadcast or IP multicast mechanism.

From Figure 15, let

T_1 be the time used to send the number of TID in TID-list (request).

T_2 be the time used to read TIDs list and all data form shared memory.

T_3 be the time used to send TIDs list and their data to other pvmds using IP broadcasting or IP multicasting mechanism.

T_4 be the time used for all local tasks to copy data from shared memory.

We assume that $n_i = m/p$ for $\forall\ i$, T_i's can be expressed as:

$$T_1 = t_{s1} + st_{c1}$$

$$T_2 = (sm + dm)t_m$$

$$T_3 = t_{s2} + (sp + dm)t_{c2}$$

$$T_4 = d(m/p)t_m$$

Therefore, the time used by scatter operation using IP-broadcasting or IP-multicasting mechanism can be modeled as:

$$T_{\text{scatter-1}} = T_g + T_1 + T_2 + T_3 + T_4 \quad \text{.....(4.2)}$$

Gather

The root task of the gather operation first obtains TIDs list of the specified group from group server and copies them to the shared memory. Then, it sends the request message for the gather operation (TM_GATHERA message) to local pvmd (root pvmd) along with the number of TIDs in the group. Root pvmd reads the list of TIDs from shared memory, sorts them and create the pvmd-list. Then, it sends the pvmd-list to other pvmds using IP broadcasting or IP multicasting mechanism (DM_GATHERA message). Each pvmd reads all local data of its tasks when it receives the TM_GATHERB message indicating that all local data is ready in the

shared memory and sends the data back to the root pvmd (DM_GATHERC message). When the data reach the root pvmd, they are copied to the shared memory and read by the root task.

The tasks can determine whether all data of local tasks are written in the shared memory by examining the shared memory header which is initialized to the number of local tasks by local pvmd. After a task writes its data, it decreases a count in the shared memory header. After the decrement, if the shared memory header is 0, the task will know that all data of local tasks are written in the shared memory.

The algorithm of the gather operation using shared memory and IP broadcasting or IP multicasting mechanism is as follows:

Assume:

$G \subseteq S$.

$data^i$ is the data of task i.

result is the buffer for result of gather operation.

Shmem is the shared memory.

Copy(*src*, *dst*) is the function to copy from *src* to *dst*.

GetTidList(*gname*) is the function to get list of TIDs in group *gname* from group server.

wait(*sem*) is the function to wait until the value of semaphore *sem* ≥ 1 then subtract by 1.

Bcast(*msg_type*, *msg*) is the function to broadcast message type *msg_type* along with message *msg* to all pvmds using IP-broadcasting or IP-multicasting mechanism.

signal(*sem*, *n*) is the function to increase the value of semaphore sem by *n*.

Send(*dst*, *msg_type*, *msg*) is the function to send the message type *msg_type* along with message *msg* to destination *dst*.

Recv(*msg*) is the function to receive message *msg*.

read_header_sem is a semaphore initialized to 0.

read_result_sem is a semaphore initialized to 0.

At the root task of the gather operation:

```
G = GetTidList(gname);
Copy(G, Shmem);
Send(local pvmd , TM_GATHERA, |G|);
wait (read_header_sem);
Copy(data, Shmem);    /* copy data of root to shared memory */
if  (all local data are written to shared memory)
        Send(local pvmd, TM_GATHERB, 0);
wait (read_result_sem);
```

Copy(*Shmem*, *result*); /*Copy *data*^{i} from task^i; data available in shared memory */

At the pvmds:

TM_GATHERA:

Copy(*Shmem* G);

Sort G and create pvmd-list;

Bcast(DM_GATHERA, pvmd-list);

TM_GATHERB:

Copy(*Shmem*, *data*); /*data is the data of all local tasks */.

Send(*root pvmd*, DM_GATHERC, *data*);

DM_GATHERA:

Recv(pvmd-list);

signal (*read_header_sem*, the number of local tasks);

DM_GATHERC:

Recv(*data*);

put *data*^{i} in appropriate place;

if (received all data) {

Copy(*all data*, *Shmem*);

signal(*read_result_sem*, 1);

}

At the client tasks of the gather operation:

```
wait (read_header_sem);
Copy(data, Shmem);
if (all local data are written to shared memory) {
        Send(local pvmd, TM_GATHERB, 0);
}
```

By using IP Broadcast or IP Multicast mechanism. T_3 will be the time used to send pvmd-list to other pvmds using IP broadcast or IP multicast mechanism.

From the algorithm, let

T_1 be the time used to send the number of TID in TID-list (request).

T_2 be the time used to read TIDs list form the shared memory.

T_3 be the time used to send the pvmd-list to other pvmds using IP broadcasting or IP multicasting mechanism.

T_4 be the time used by all local tasks to copy their data to the shared memory.

T_5 be the time used by all client pvmds to send data to root pvmd.

T_6 be the time used by root task to read data from shared memory.

We assume that $n_i = m/p$ for $\forall\ i$, T_i's can be expressed as:

$$T_1 = t_{s1} + st_{c1}$$

$$T_2 = smt_m$$

$T_3 = t_{s2} + spt_{c2}$

$T_4 = d(m/p)t_m$

$T_5 = (p\text{-}1)\ t_{s2} + d(m - (m/p))t_{c2}$

$T_6 = dmt_m$

Therefore, the time used by gather operation using IP broadcasting or IP multicasting mechanism can be modeled as:

$$T_{gather\text{-}1} = T_g + T_1 + T_2 + T_3 + T_4 + T_5 + T_6 \quad \text{.....(4.3)}$$

Reduce

All tasks of the reduce operation first obtains TIDs list of the specified group from group server and copies them to the shared memory. Then, the root task sends the request message for the reduce operation (TM_REDUCEA) to local pvmd (root pvmd) along with the number of TIDs in the group. Root pvmd reads the list of TIDs from the shared memory, sorts them and creates the pvmd-list. Then, it sends the pvmd-list to other pvmds using IP broadcasting or IP multicasting mechanism (DM_REDUCEA message). Each pvmd performs a reduction operation to all local data of its tasks when it receives the TM_REDUCEB message indicating that all local data are ready in the shared memory and sends the final result backward to the root pvmd (by DM_REDUCEC message). When the data reach the root pvmd, the root

pvmd performs the global computation operation on the received data and places the result in the shared memory to be read by the root task.

The tasks can determine whether all data of local tasks are written in the shared memory by examining the shared memory header which is initialized to the number of local tasks by local pvmd. After the task writes its data, it decreases a count in the shared memory header. After the decrement, if the shared memory header is 0, the task will know that all data of local tasks are written in the shared memory.

The algorithm of the reduce operation using shared memory and IP broadcasting or IP multicasting mechanism is as follows:

Assume:

$G \subseteq S$.

$data^i$ is the data of task i.

result is the buffer for result of reduce operation.

Shmem is the shared memory.

Copy(*src*, *dst*) is the function to copy from *src* to *dst*.

GetTidList(*gname*) is the function to get list of TIDs in group *gname* from group server.

wait(*sem*) is the function to wait until the value of semaphore *sem* ≥ 1 then subtract by 1.

Bcast(*msg_type*, *msg*) is the function to broadcast message type

msg_type along with message *msg* to all pvmds using IP-broadcasting or IP-multicasting mechanism.

signal(*sem*, *n*) is the function to increase the value of semaphore sem by *n*.

Send(*dst*, *msg_type*, *msg*) is the function to send the message type *msg_type* along with message *msg* to destination *dst*.

Recv(*msg*) is the function to receive message *msg*.

GlobalFunc(*data1*, *data2*) is the global computation function performs on *data1* and *data2* and place the result in *data1*.

read_header_sem is a semaphore initialized to 0.

read_result_sem is a semaphore initialized to 0.

At the root task of the reduce operation:

G = GetTidList(*gname*);

Copy(G, *Shmem*);

Send(*local pvmd* , TM_REDUCEA, |G|);

wait (read_*header_sem*);

Copy($data^i$, *Shmem*);

if (all local data are written to shared memory)

Send(*local pvmd*, TM_REDUCEB, 0);

wait (*read_result_sem*);

Copy(*Shmem*, *result*);

At the pvmds:

TM_REDUCEA:

Copy(*Shmem*, G);

Sort G and create pvmd-list;

Bcast(DM_REDUCEA, pvmd-list);

TM_REDUCEB:

Copy(*Shmem*, *result*); /* first data */

for (i = 1; i < the number of local tasks; i++) {

Copy(*Shmem*, *data[i]*); /* data of local task *i* */

GlobalFunc(*result*, *data[i]*); /* Compute reduction function */

}

Send(*root pvmd*, DM_REDUCEC, *result*);

DM_REDUCEA:

Recv(pvmd-list);

signal (*read_header_sem*, the number of local tasks);

DM_REDUCEC:

Recv(*data*);

GlobalFunc(*result*, *data*);

```
if (received all data) {
    Copy(all data, Shmem);
    signal(read_result_sem, 1);
}
```

At the client tasks of the reduce operation:

```
G = GetTidList(gname);
wait (read_header_sem);
Copy(data, Shmem);
if (all local data are written to shared memory) {
        Send(local pvmd, TM_REDUCEB, 0);
}
```

From the algorithm, let

T_1 be the time used to send the number of TID in TID-list (reduce).

T_2 be the time used to read TIDs list form shared memory.

T_3 be the time used to send the pvmd-list to other pvmds using IP broadcasting or IP multicasting mechanism.

T_4 be the time used for all local tasks to copy their data to the shared memory.

T_5 be the time used for all client pvmds to send the local result to the

root pvmd.

T_6 be the time used for root task to copy result from shared memory.

Assume that $n_i = m/p$ for $\forall\ i$, T_i's can be expressed as:

$$T_1 = t_{s1} + st_{c1}$$

$$T_2 = smt_m$$

$$T_3 = t_{s2} + spt_{c2}$$

$$T_4 = d(m/p)t_m$$

$$T_5 = (p\text{-}1)(t_{s2} + dt_{c2})$$

$$T_6 = dt_m$$

Therefore, the time used by reduce operation using IP-broadcasting or IP-multicasting mechanism can be modeled as:

$$T_{reduce\text{-}1} = mT_g + T_1 + T_2 + T_3 + T_4 + T_5 + T_6 \quad \text{.....(4.4)}$$

Here, we ignore the time for computing the reduce operation on a pair of data since it depends on the operation.

Comparison to Current Approaches

Now we will compare the timing model of our new algorithms to the existing collective communication operations algorithms.

Broadcast

The cost of the current algorithm of broadcast operation from equation 3.1 can be expanded as (assuming $n_i = m/p$; $\forall i$):

$$T_{bcast} = T_g + (t_{s1} + smt_{c1}) + ((p\text{-}1)t_{s2} + s(m\text{-}m/p)t_{c2})) + (t_{s1} + st_{c1}) + (t_{s1} + dt_{c1})$$
$$+ ((p\text{-}1)(t_{s2} + dt_{c2}) + (m/p)(t_{s1} + dt_{c1})) + dt_m$$
$$= T_g + \{ (m/p + 3)t_{s1} + (s(m + 1) + d(m/p + 1))t_{c1} + dt_m \} +$$
$$\{ 2(p\text{-}1)t_{s2} + (s(m\text{-}m/p) + (p\text{-}1)d)t_{c2} \}$$

The equation can be separated into two parts. The first part is the cost of communication within a host and the second part is the cost of communication between hosts.

The cost of the algorithm of broadcast operation using shared memory and IP broadcasting or IP multicasting mechanism, from equation 4.1, can be expanded as:

$$T_{bcast\text{-}1} = T_g + (t_{s1} + st_{c1}) + (sm + d)t_m + (t_{s2} + (sp+d)t_{c2}) + (m/p)dt_m$$
$$= T_g + \{ t_{s1} + st_{c1} + (sm + (m/p+1)d)t_m \} +$$
$$\{ t_{s2} + (sp+d)t_{c2} \}$$

By comparing T_{bcast} and $T_{bcast\text{-}1}$, it shows that the cost of communication within a host is reduced to the cost of memory access. The overhead of communication between hosts also reduce almost by a factor of $(p\text{-}1)$ for IP broadcasting or IP multicasting mechanism.

Scatter

The cost of the current algorithm of scatter operation from equation 3.2 can be expanded as:

$$T_{scatter} = T_g + (m\text{-}1)(t_{s1} + dt_{c1}) + (m\text{-}m/p)(t_{s2} + dt_{c2}) + (m/p)(t_{s1} + dt_{c1}) + dt_m$$

$$= T_g + \{ ((m\text{-}1)+m/p)t_{s1} + ((m\text{-}1)+m/p)dt_{c1} + dt_m \} + \{ (m\text{-}m/p)t_{s2} + (m\text{-}m/p)dt_{c2} \}$$

The equation can be separated into two parts. The first part is the cost of communication within a host and the second part is the cost of communication between hosts.

The cost of the algorithm of scatter operation using shared memory and IP broadcasting or IP multicasting mechanism, from equation 4.2, can be expanded as:

$$T_{scatter\text{-}1} = T_g + (t_{s1} + st_{c1}) + (sm + dm)t_m + (t_{s2} + (sp+dm)t_{c2}) + d(m/p)t_m$$

$$= T_g + \{ t_{s1} + st_{c1} + (sm + dm + d(m/p))t_m \} + \{ t_{s2} + (sp + dm)t_{c2} \}$$

By comparing $T_{scatter}$, and $T_{scatter\text{-}1}$, it shows that the cost of communication within a host is reduced to the cost of memory access. The overhead of

communication between hosts also reduce from (m - m/p) to 1 for IP broadcasting or IP-multicasting mechanism.

Gather

The cost of the current algorithm of gather operation from equation 3.3 can be expanded as:

$$T_{gather} = T_g + (m/p\text{-}1)(t_{s1} + dt_{c1}) + (m\text{-}m/p)(t_{s2} + dt_{c2}) + (m\text{-}1)(t_{s1} + dt_{c1}) + dt_m$$

$$= T_g + \{ (m+m/p\text{-}2)t_{s1} + (m+m/p\text{-}2)dt_{c1} + dt_m \} + \{ (m\text{-}m/p)t_{s2} + (m\text{-}m/p)dt_{c2} \}$$

The equation can be separated into two parts. The first part is the cost of communication within a host and the second part is the cost of communication between hosts.

The cost of the algorithm of gather operation using shared memory and IP broadcasting or IP multicasting mechanism, from equation 4.3, can be expanded as:

$$T_{gather\text{-}1} = T_g + (t_{s1} + st_{c1}) + smt_m + (t_{s2} + spt_{c2}) + d(m/p)t_m + ((p\text{-}1)ts2 + d(m - m/p)t_{c2}) + dmt_m$$

$$= T_g + \{ t_{s1} + st_{c1} + (sm + (m + m/p)d)t_m \} + \{ (p\text{-}1)t_{s2} + (sp + (m - m/p)d)t_{c2} \}$$

By comparing T_{gather}, and $T_{gather\text{-}1}$, it shows that the cost of communication within a host is reduced to the cost of memory access. The overhead of communication between hosts also reduce from $(m\text{-}m/p)$ to $(p\text{-}1)$ for IP broadcasting or IP multicasting mechanism.

Reduce

The cost of the current algorithm of reduce operation from equation 3.4 can be expanded as:

$$T_{reduce} = mT_g + (m/p\text{-}1)(t_{s1} + dt_{c1}) + (m/p\text{-}1)(t_{s1} + dt_{c1}) + (t_{s1} + dt_{c1}) + (p\text{-}1)(t_{s2} + dt_{c2}) + (p\text{-}1)(t_{s1} + dt_{c1}) + dt_m$$

$$= mT_g + \{ (2m/p+p-2)t_{s1} + (2m/p+p-2)dt_{c1} + dt_m \} + \{ (p\text{-}1)t_{s2} + (p\text{-}1)dt_{c2} \}$$

The equation can be separated into three parts. The first part is the cost of communication with the group server. The second part is the cost of communication within a host. The third part is the cost of communication between hosts.

The cost of the algorithm of reduce operation using shared memory and IP broadcasting or IP multicasting mechanism, from equation 4.4, can be expanded as:

$$T_{reduce\text{-}1} = T_g + (t_{s1} + st_{c1}) + smt_m + (t_{s2} + spt_{c2}) + d(m/p)t_m + (p\text{-}1)(ts2 + dt_{c2}) + dt_m$$

$$= T_g + \{ t_{s1} + st_{c1} + (sm + (m/p+1)d)t_m \} +$$

$$\{ pt_{s2} + (sp + (p-1)d)t_{c2} \}$$

By comparing T_{reduce}, and $T_{reduce\text{-}1}$, it shows that the cost of communication with the group server is reduced from m to 1. This will promote the better performance because having every task contacts a group server makes the bottleneck situation. The cost of communication within a host is reduced to the cost of memory access.

CHAPTER V

EXPERIMENTAL RESULTS

The new approaches presented in the previous section were implemented in C language and incorporated into PVM version 3.3.10. Sixteen Sun SparcStation 5 workstations were used for our experiments. The machines were not dedicated to the tests. Therefore, there were some additional loads from other users. The message sizes were up to 2048 bytes. The number of tasks running on each host is up to 2 tasks because more tasks running on the same uniprocessor machine are not preferable, in general, for parallel computation.

Each graph shows the results collected by running the program for different problem sizes. Each test was repeated 100 times in different times of the day to account for various load situations and the average was computed for better accuracy in the reported results. The results obtained are compared to the result of the current approaches of CCL in PVM. Note that the focus of our comparison is the relative performances improvements of the proposed approaches to the current ones, rather than the observation of absolute timings of the approaches.

Figure 16 shows the time in millisecond used by broadcast operation plotted against the number of machines in the configuration with 1 and 2 tasks per machine and data sizes of 4, 256, 512 and 2048 bytes.

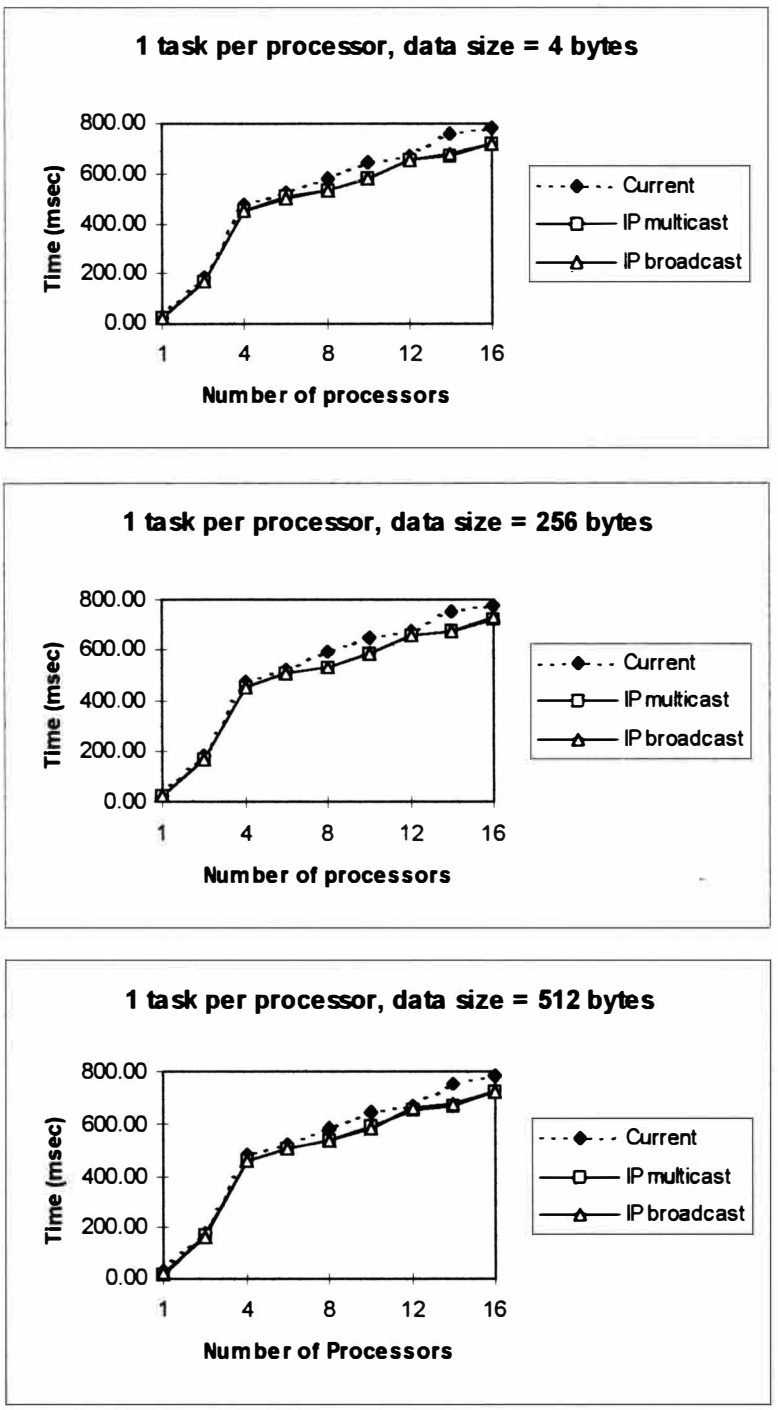
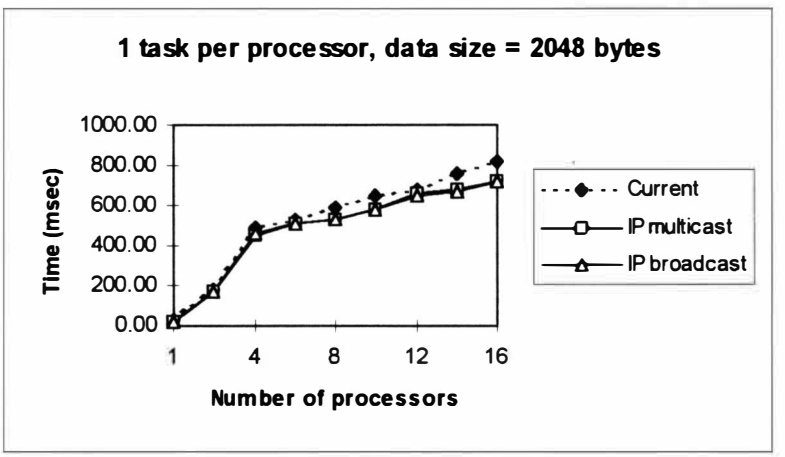
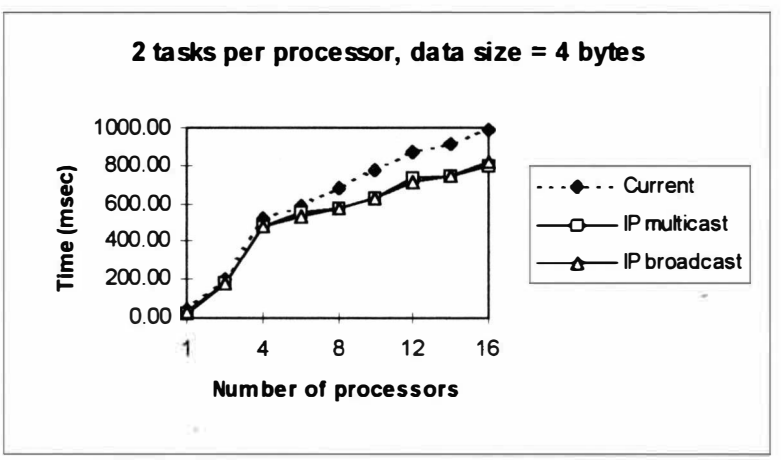


Figure 16. Time Used by Broadcast Operation of Different Data Sizes and Different Numbers of Tasks per Processor.

Figure 16—Continued





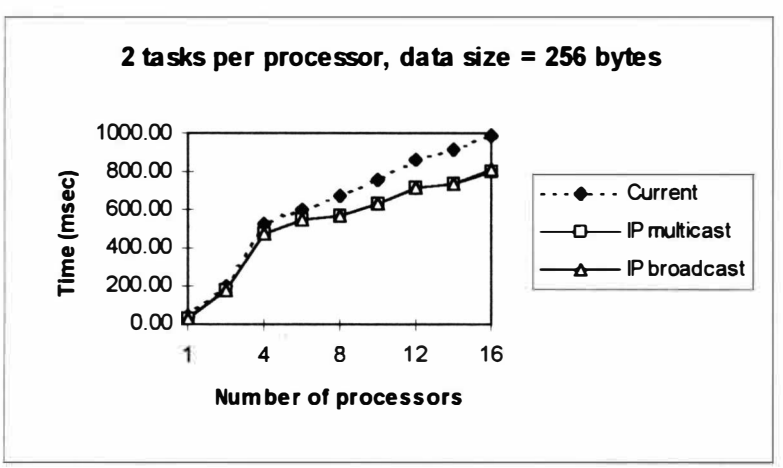
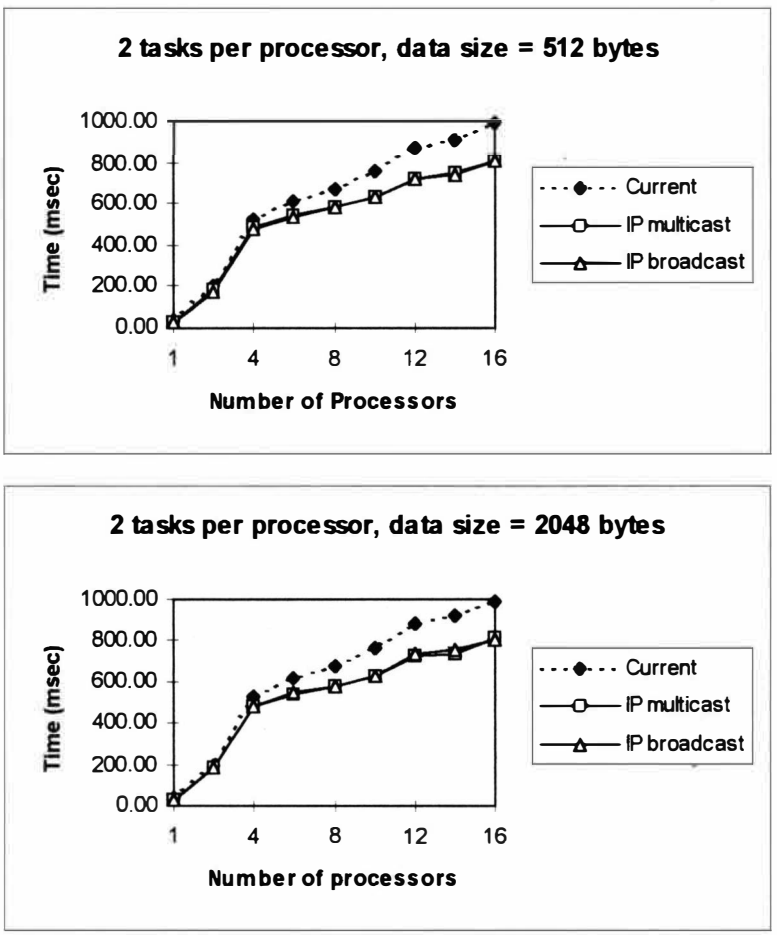


Figure 16—Continued



One might expect that the timing of our broadcast algorithms should remain constant as a function of number of hosts since IP broadcast mechanism is used. However, as can be seen from the plots in Figure 16, the timings of the new broadcast algorithms increase as a function of the number of machines. It is due to the fact that every task still needs some information from the group server, which creates a congestion a the group server. By having every task contact the group server, more

time is consumed when the number of tasks grow up. From the graphs, it is clear that the IP multicasting and IP broadcasting approach times are faster than the current approach.

Figure 17 shows the time in millisecond used by scatter operation plotted against the number of machines in the configuration with 1 and 2 tasks per machine and data sizes of 4, 32 and 64 bytes.

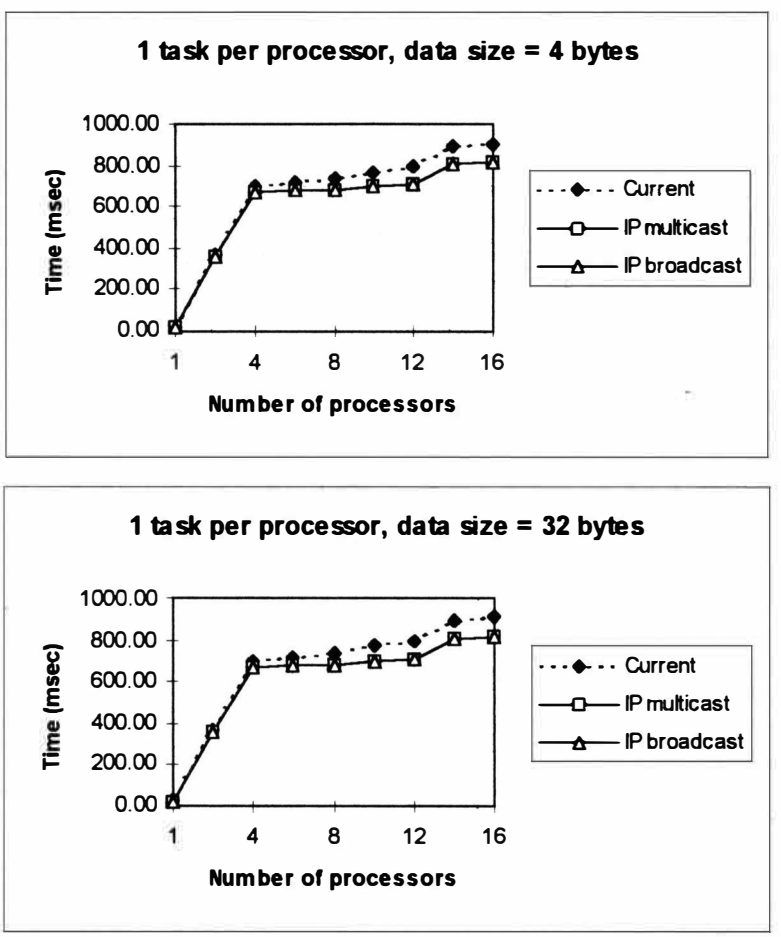
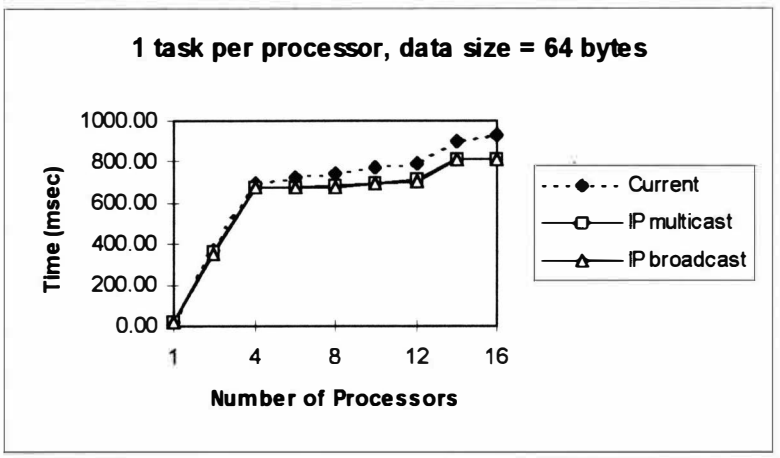
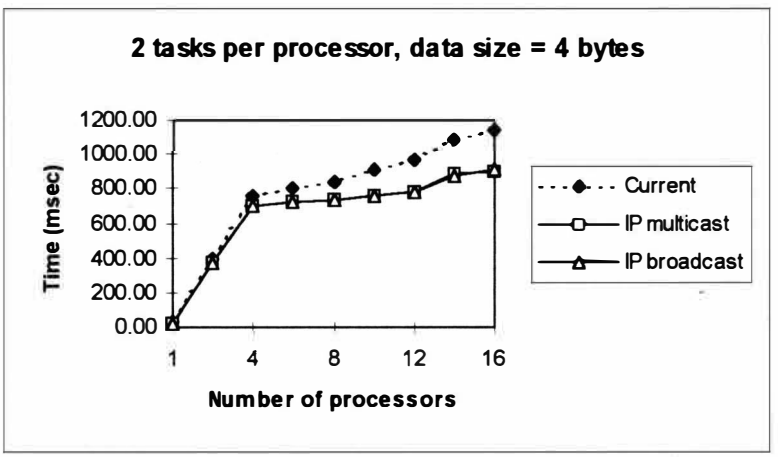


Figure 17. Time Used by Scatter Operation of Different Data Sizes and Different Numbers of Tasks per Processor.

Figure 17—Continued





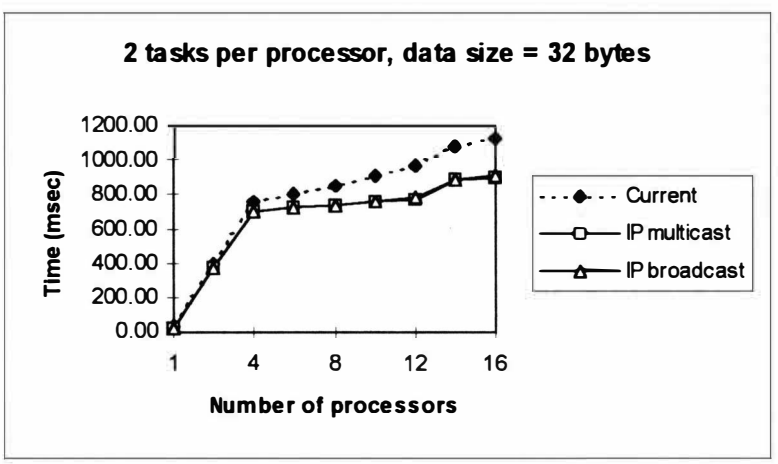
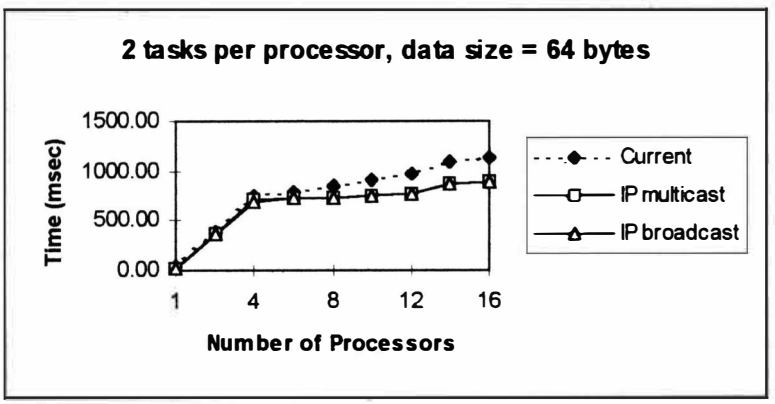


Figure 17—Continued



It is clear that the IP multicasting and IP broadcasting approach times are faster than the current approach.

Figure 18 shows the time in millisecond used by gather operation plotted against the number of machines in the configuration with 1 and 2 tasks per machine and data sizes of 4, 256, 512 and 1024 bytes.

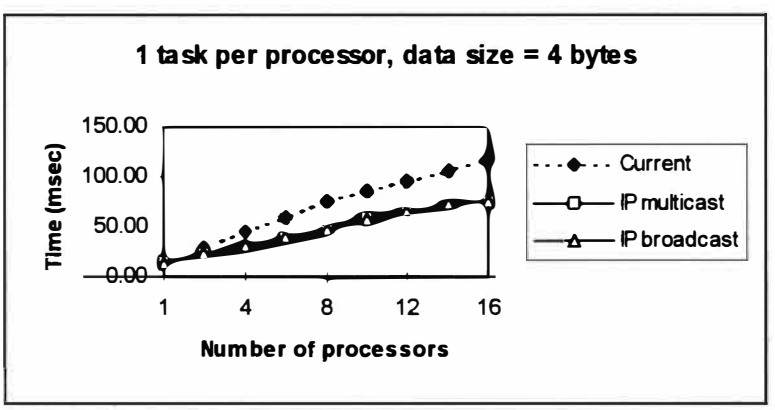
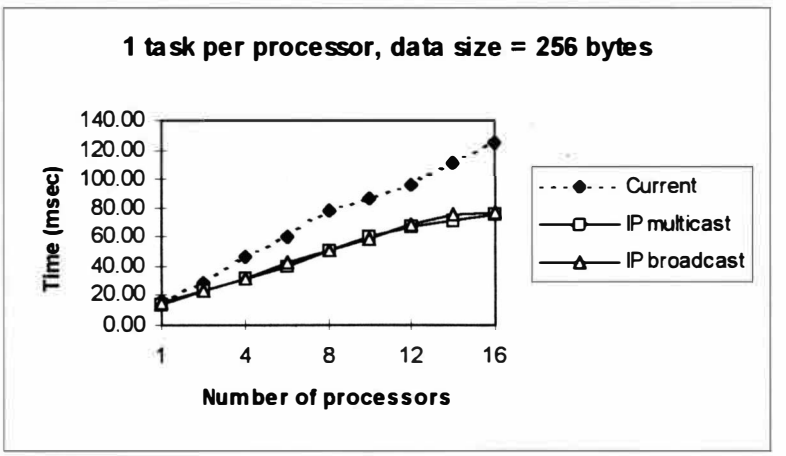
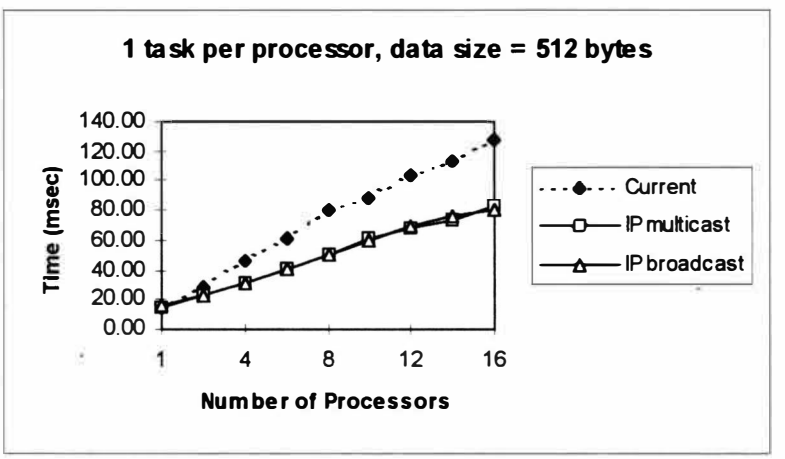


Figure 18. Time Used by Gather Operation of Different Data Sizes and Different Numbers of Tasks per Processor.

Figure 18—Continued





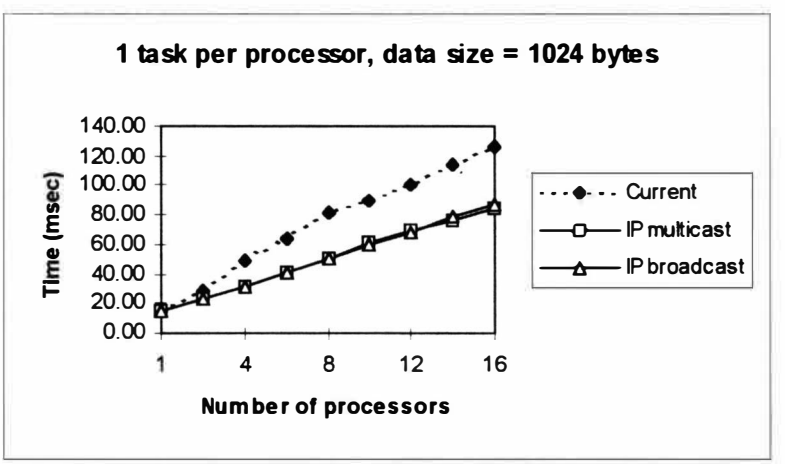
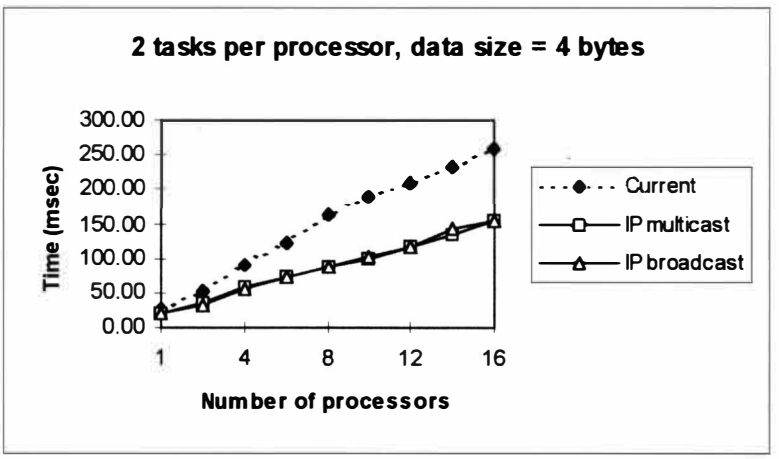
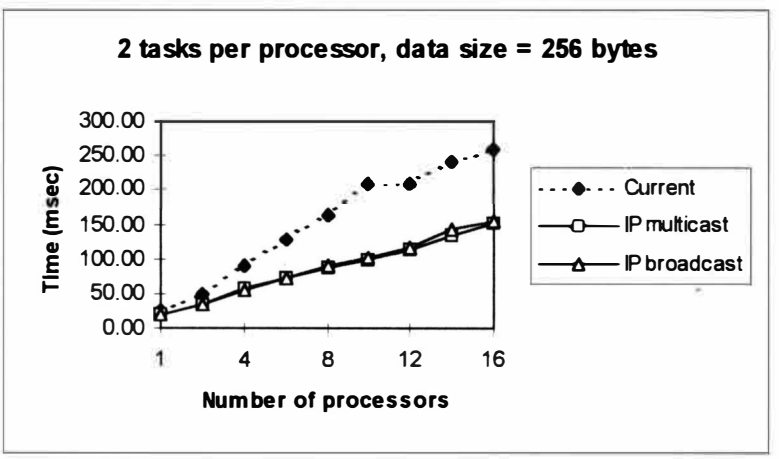


Figure 18—Continued





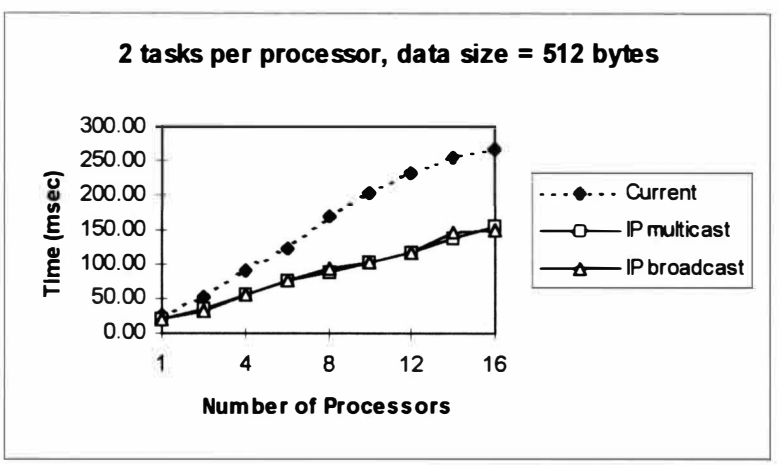
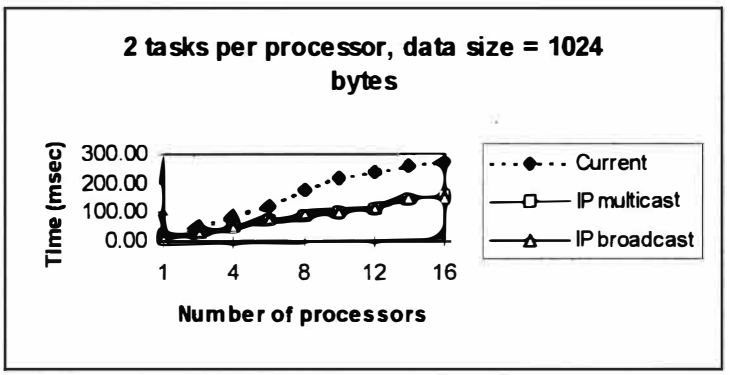


Figure 18—Continued



It is clear that the IP multicasting and IP broadcasting approach times are faster than the current approach.

Figure 19 shows the time in millisecond used by reduce operation of global summation plotted against the number of machines in the configuration with 1 and 2 tasks per machine and data size of 4-byte integer.

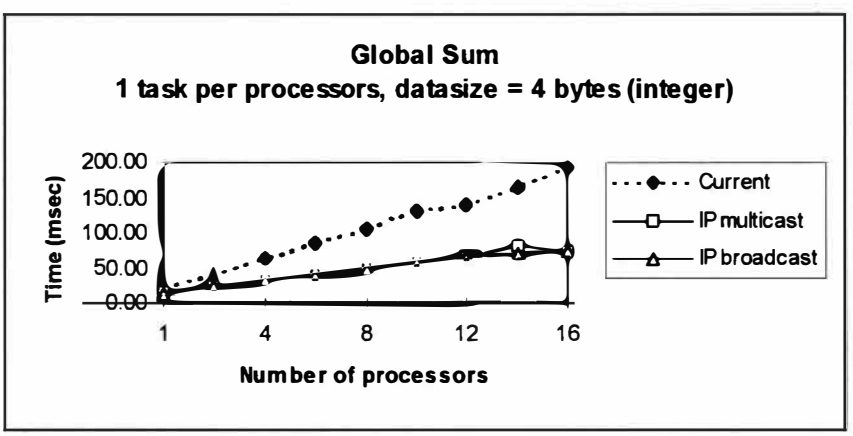
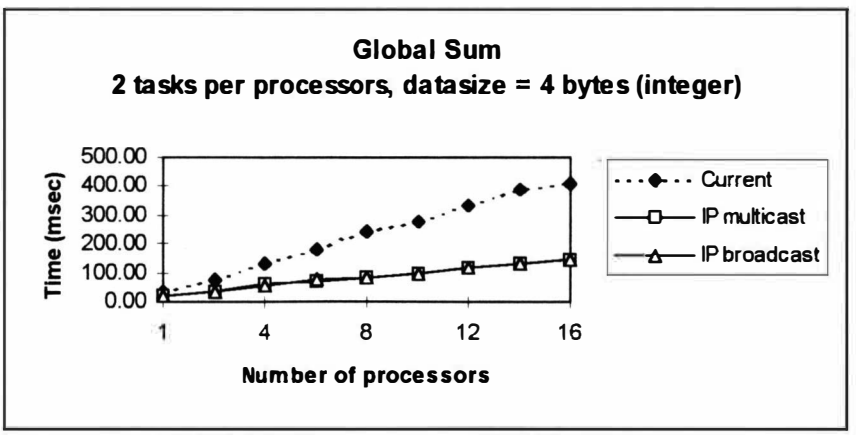


Figure 19. Time Used by Reduce Operation of Global Summation of 4-byte Integer Data and Different Numbers of Tasks per Processor.

Figure 19—Continued



It is clear that the IP multicasting and IP broadcasting approach times are faster than the current approach.

CHAPTER VI

CONCLUSIONS AND DISCUSSIONS

In this thesis, we developed the new approaches for Collective Communication Operations in PVM. These approaches intend to reduce the overhead of the communication over the network by using IP multicasting or IP broadcasting mechanism and reduce the communication time between tasks and local pvmd by using shared memory.

The experiments we conducted show that the performances of Collective Communication Library in PVM over local area network of 16 machines of the new approaches are about 13%, 35%, 15% and 60% faster than the current approaches for broadcast, gather, scatter and reduce operation respectively. In practice, actual performance improvements may differ from these numbers. Non the less, one can expect reasonable performance gains.

The proposed approaches still promote the problems in flexibility and portability. The implementation of broadcast or multicast over UDP sockets has a restriction. The port number of the UDP sockets of all pvmds in the system has to be the same. It may not be flexible when having a number of users running PVM systems because the port number of UDP sockets has to be unique for each user and has to be identical for all pvmds in one system. Next, using the shared memory for

communication within a machine has the portability problems because some operating systems do not provide shared memory facility.

For the future work, it would be nice to provide the ability to use these new approaches over a wide area network (WAN). The extension of the multicast mechanism can be applied, as well as the efficient packet-loss recovery algorithm over WAN. Because the broadcast and multicast over UDP sockets are not reliable, the performance of the operations will also depend on the packet-loss recovery algorithm.

Appendix A

Source Code of Broadcast-Receive Protocol for Pvmd

```
/*
 * New functions added to pvmd.c to perform IP multicast or IP
 * broadcast operation.
 *
 */


/*    bcastmessage()
*
*           broadcast a message. Make destination to broadcast
address or
*     multicast address.
*     - then put packet to bcast_opq queue by pkt_to_bcast().
*     - set pp->pk_dst to 0. (for broadcast address).
*     - the message is sent by bcastoutput().
*           - the message is received by netinput().
*/

int
bcastmessage(mp)
      struct mesg *mp;
{
      struct frag *fp;
      struct pkt *pp;
      int ff = FFSOM;
      int dst = mp->m_dst = 0;

      if (debugmask & PDMMESSAGE) {
            sprintf(pvmtxt, "bcastmessage() dst t%x code %s len
%d\n",
                        dst, pvmnametag(mp->m_cod, (int *)0), mp-
>m_len);
            pvmlogerror(pvmtxt);
      }

      /*
      *     add a frag to empty message to simplify handling
      */

      if ((fp = mp->m_frag->fr_link) == mp->m_frag) {
            fp = fr_new(MAXHDR);
            fp->fr_dat += MAXHDR;
            LISTPUTBEFORE(mp->m_frag, fp, fr_link, fr_rlink);
      }

      /*
      *     route message
      */

      /* to remote */

      /* packetize frags */

            do {
```

```
                pp = pk_new(0);
                if (ff & FFSOM) {
                        pp->pk_cod = mp->m_cod;
                        pp->pk_enc = mp->m_enc;
                        pp->pk_wid = mp->m_wid;
#ifdef      MCHECKSUM
                        pp->pk_crc = mesg_crc(mp);
#else
                        pp->pk_crc = 0;
#endif
                }
                pp->pk_buf = fp->fr_buf;
                pp->pk_dat = fp->fr_dat;
                pp->pk_max = fp->fr_max;
                pp->pk_len = fp->fr_len;
                da_ref(pp->pk_buf);
                if (fp->fr_link == mp->m_frag)
                        ff |= FFEOM;
                pp->pk_src = mp->m_src;
                pp->pk_dst = dst;
                pp->pk_flag = ff;
                ff = 0;
                if (mp->m_flag & MM_PRIO) {
                        if (debugmask & (PDMMESSAGE|PDMAPPL))
                                pvmlogerror("bcastmessage() PRIO
message to host? (scrapped)\n");

                } else {
                        pkt_to_bcast(pp);
                }
        } while ((fp = fp->fr_link) != mp->m_frag);

bail:
        mesg_unref(mp);
        return 0;
}

/*================================================================*/

/*      pkt_to_bcast()
*
*       Add data pkt to bcast queue (bcast_opq).
*       If data plus header length is greater than our host mtu,
*       refragment into >1 pkts.
*
*       We have to pay special attention to the FFSOM packet - make it
*       shorter so there's room to prepend the message header later.
*
*/

int
pkt_to_bcast(pp)
        struct pkt *pp;
{
```

```
        int maxl = ourudpmtu - DDFRAGHDR;
        int llim = pp->pk_flag & FFSOM ? maxl - TTMSGHDR : maxl;

        pp->pk_flag = (pp->pk_flag & (FFSOM|FFEOM)) | FFDAT;
        if (debugmask & PDMPACKET) {
                sprintf(pvmtxt, "pkt_to_bcast() pkt src t%x dst t%x f %s
len %d\n",
                                pp->pk_src, pp->pk_dst, pkt_flags(pp-
>pk_flag), pp->pk_len);
                pvmlogerror(pvmtxt);
        }

        if (pp->pk_len <= llim) {
                pp->pk_ack = hosts->ht_cnt;
                bcast_txseq = NEXTSEQNUM(bcast_txseq);
                pp->pk_seq = bcast_txseq;
                LISTPUTAFTER(bcast_opq, pp, pk_tlink, pk_trlink);
        }
        else {
                struct pkt *pp2;
                char *cp = pp->pk_dat;
                int togo;
                int n;
                int ff = pp->pk_flag & FFSOM;
                int fe = pp->pk_flag & FFEOM;

                for (togo = pp->pk_len; togo > 0; togo -= n) {
                        n = min(togo, llim);
                        if ((debugmask & PDMPACKET) && togo != pp->pk_len)
{
                                sprintf(pvmtxt, "pkt_to_bcast() refrag len
%d\n", n);
                                pvmlogerror(pvmtxt);
                        }
#ifdef          STATISTICS
                        stats.refrag++;
#endif
                        pp2 = pk_new(0);
                        pp2->pk_src = pp->pk_src;
                        pp2->pk_dst = pp->pk_dst;
                        if (n == togo)
                                ff |= fe;
                        pp2->pk_flag = ff | FFDAT;
                        ff = 0;
                        llim = maxl;
                        pp2->pk_cod = pp->pk_cod;
                        pp2->pk_enc = pp->pk_enc;
                        pp2->pk_wid = pp->pk_wid;
                        pp2->pk_crc = pp->pk_crc;
                        pp2->pk_buf = pp->pk_buf;
                        pp2->pk_max = pp->pk_max;
                        pp2->pk_dat = cp;
                        pp2->pk_len = n;
                        da_ref(pp->pk_buf);
```

```
                cp += n;

                pp->pk_ack = hosts->ht_cnt;
                bcast_txseq = NEXTSEQNUM(bcast_txseq);
                pp->pk_seq = bcast_txseq;

                LISTPUTAFTER(bcast_opq, pp2, pk_tlink, pk_trlink);
            }
            pk_free(pp);
        }
        return 0;
}

/*=============================================================*/


/*      bcastoutput()
*
*       Broadcast packets out the wire to remote pvmds.
*/

bcastoutput()
{
        struct timeval tnow, tx, tout;
        struct pkt *pp, *pp2;
        struct hostd *hp;
        char *cp;
        int len;
        int cc;
        char dummy[DDFRAGHDR];
        struct sockaddr_in  sin;

        gettimeofday(&tnow, (struct timezone*)0);

        tout.tv_sec = MCASTMAXTIMEOUT/1000000;
        tout.tv_usec = MCASTMAXTIMEOUT%10000000;

        for (pp=bcast_txq->pk_link; pp != bcast_txq; pp=pp->pk_link) {

            if (debugmask & PDMMESSAGE) {
                pvmlogerror("bcastoutput() Check bcast_txq
queue.\n");
            }

            if (pp->pk_ack == 0) {  /* recv all ACKs */

                if (debugmask & PDMMESSAGE) {
                    sprintf(pvmtxt, "bcastoutput() All ACKs
received for pkt %d.\n",
                                        pp->pk_seq);
                    pvmlogerror(pvmtxt);
                }

                LISTDELETE(pp, pk_link, pk_rlink);
```

```
                pk_free(pp);
                pp = bcast_txq;   /* reset pp for next loop */
            }
            else { /* Check for time out */
                TVXSUBY(&tx, &tnow, &pp->pk_at);
                if (TVXLTY(&tout, &tx)) {     /* Exceed time out,
then retry */

                    sprintf(pvmtxt, "bcastoutput() pkt %d exceed
timeout.\n",
                                        pp->pk_seq);
                    pvmlogerror(pvmtxt);

                    LISTDELETE(pp, pk_link, pk_rlink);
                    pp->pk_ack = hosts->ht_cnt;  /* reset pk_ack
*/
                    LISTPUTBEFORE(bcast_opq, pp, pk_tlink,
pk_trlink);
                    pp = bcast_txq; /* reset pp for next loop */
                }
            }
        }

        if (bcast_opq->pk_tlink == bcast_opq)
            return 0;          /* No pkt in queue */

        /*
        * send any pkts whose time has come
        */


        while ((pp = bcast_opq->pk_tlink) != bcast_opq
                /**&& TVXLTY(&pp->pk_rtv, &tnow)**/  ) {

            cp = pp->pk_dat;
            len = pp->pk_len;
            if (pp->pk_flag & FFSOM) {
                cp -= TTMSGHDR;
                len += TTMSGHDR;
                if (cp < pp->pk_buf) {
                    pvmlogerror("bcastoutput() no headroom for
message header\n");
                    return 0;
                }
                pvmput32(cp, pp->pk_cod);
                pvmput32(cp + 4, pp->pk_enc);
                pvmput32(cp + 8, pp->pk_wid);
                pvmput32(cp + 12, pp->pk_crc);
            }
            cp -= DDFRAGHDR;
            len += DDFRAGHDR;


        /*
```

```
      * save under packet header, because databuf may be shared.
      * we don't worry about message header, because it's only at the
      * head.
      */
            BCOPY(cp, dummy, sizeof(dummy));
            if (cp < pp->pk_buf) {
                  pvmlogerror("bcastoutput() no headroom for packet
header\n");
                  return 0;
            }

            if (debugmask & PDMPACKET) {
                  sprintf(pvmtxt,
                  "netoutput() pkt to bcast src t%x dst t%x f %s len
%d seq %d ack %d retry %d\n",
                              pp->pk_src, pp->pk_dst, pkt_flags(pp-
>pk_flag),
                              pp->pk_len, pp->pk_seq, pp->pk_ack, pp-
>pk_nrt);
                  pvmlogerror(pvmtxt);
            }
            pvmput32(cp, pp->pk_dst);
            pvmput32(cp + 4, pp->pk_src);
            pvmput16(cp + 8, pp->pk_seq);
            pvmput16(cp + 10, pp->pk_ack);
            pvmput32(cp + 12, 0);               /* to keep purify
happy */
            pvmput8(cp + 12, pp->pk_flag);

            /*
             * Select Multicast Address or Broadcast Address here
       *
       */
            BZERO((char *)&sin, sizeof(sin));
            sin.sin_family = AF_INET;
            sin.sin_port = EXPR_FIXED_PORT;
            sin.sin_addr.s_addr = inet_addr(EXPR_MCAST_ADDR);/*
multicast address */

                  if ((cc = sendto(netsock, cp, len, 0,
                              (struct sockaddr*)&sin, sizeof(sin)))
== -1
                  && errno != EINTR
                  && errno != ENOBUFS
#ifdef      IMA_LINUX
                  && errno != ENOMEM
#endif
                  ) {
                        pvmlogperror("bcastoutput() sendto");
#if defined(IMA_SUN4SOL2) || defined(IMA_X86SOL2) ||
defined(IMA_SUNMP) || defined(IMA_UXPM)
      /* life, don't talk to me about life... */
                        if (errno == ECHILD)
```

```
                              pvmlogerror("this message brought to
you by solaris\n");
                        else
#endif
                        pvmbailout(0);
                  }
#ifdef      STATISTICS
            if (cc == -1)
                  stats.sdneg++;
            else
                  stats.sdok++;
#endif
            TVCLEAR(&pp->pk_at);
            TVXADDY(&pp->pk_at, &pp->pk_at, &tnow);
            LISTDELETE(pp, pk_tlink, pk_trlink);
            LISTPUTBEFORE(bcast_txq, pp, pk_link, pk_rlink);

            return 0;
      }

      return 0;
}

/*=================================================================*/


/*    netinput()
*
*     Input from a remote pvmd.
*     Accept a packet, do protocol stuff then pass pkt to netinpkt().
*/

int
netinput()
{
      struct sockaddr_in osad;            /* sender's ip addr */
      struct sockaddr_in sin;
      int len;
      int oslen;                                /* sockaddr length */
      struct timeval tnow;
      struct pkt *pp, *pp2;
      struct hostd *hp;
      char *cp;
      int sqn;
      int aqn;
      int ff;
      int dst;
      int src;
      int hh;
      int already;
      struct timeval tdiff;               /* packet rtt */
      int rttusec;
      int cc;
```

```
	/*
	* alloc new pkt buffer and read packet
	*/

	pp = pk_new(ourudpmtu);
	if (TDFRAGHDR > DDFRAGHDR)
		pp->pk_dat += TDFRAGHDR - DDFRAGHDR;

	oslen = sizeof(osad);
	if ((pp->pk_len = recvfrom(netsock, pp->pk_dat,
			pp->pk_max - (pp->pk_dat - pp->pk_buf),
			0, (struct sockaddr*)&osad, &oslen)) == -1) {
		if (errno != EINTR)
			pvmlogperror("netinput() recvfrom(netsock)");
		goto scrap;
	}

#if 0
	/* drop random packets */
	if (!(random() & 3)) {
		pvmlogerror("netinput() oops, dropped one\n");
		goto scrap;
	}
#endif

#ifdef	STATISTICS
	stats.rfok++;
#endif

	cp = pp->pk_dat;
	pp->pk_len -= DDFRAGHDR;
	pp->pk_dat += DDFRAGHDR;
	dst = pp->pk_dst = pvmget32(cp);
	src = pp->pk_src = pvmget32(cp + 4);
	sqn = pp->pk_seq = pvmget16(cp + 8);
	aqn = pvmget16(cp + 10);
	ff = pp->pk_flag = pvmget8(cp + 12);
	if (ff & FFSOM) {
		if (pp->pk_len < TTMSGHDR) {
			sprintf(pvmtxt, "netinput() SOM pkt src t%x dst t%x
too short\n",
				src, dst);
			pvmlogerror(pvmtxt);
			goto scrap;
		}
		cp += DDFRAGHDR;
		pp->pk_cod = pvmget32(cp);
		pp->pk_enc = pvmget32(cp + 4);
		pp->pk_wid = pvmget32(cp + 8);
		pp->pk_crc = pvmget32(cp + 12);
		pp->pk_len -= TTMSGHDR;
		pp->pk_dat += TTMSGHDR;
	}
```

```
	/*
	* make sure it's from where it claims
	*/

	hh = (src & tidhmask) >> (ffs(tidhmask) - 1);
	if (hh < 0 || hh > hosts->ht_last || !(hp = hosts-
>ht_hosts[hh])
#ifndef IMA_LINUX
	/*
	* XXX removing these lines is a hack and reduces security
between
	* XXX pvmds somewhat, but it's the easiest fix for Linux right
now.
	*/
	|| (osad.sin_addr.s_addr != hp->hd_sad.sin_addr.s_addr)
	|| (osad.sin_port != hp->hd_sad.sin_port)
#endif
	) {
		sprintf(pvmtxt, "netinput() bogus pkt from %s\n",
				inadport_decimal(&osad));
		pvmlogerror(pvmtxt);
		goto scrap;
	}

	if (debugmask & PDMPACKET) {
		sprintf(pvmtxt,
		"netinput() pkt from %s src t%x dst t%x f %s len %d seq
%d ack %d\n",
				hp->hd_name, src, dst, pkt_flags(ff), pp-
>pk_len, sqn, aqn);
		pvmlogerror(pvmtxt);
	}

	if (!dst) { /* This is a broadcast packet */

		if (ff & FFACK) {

			if (debugmask & PDMMESSAGE) {
				sprintf(pvmtxt, "netinput() recv ACK pkt for
pkt %d.\n",
							pp->pk_ack);
				pvmlogerror(pvmtxt);
			}

			for (pp2 = bcast_txq->pk_link; pp2 != bcast_txq;
pp2 = pp2->pk_link)
				if (pp2->pk_seq == aqn) {
					if (pp2->pk_ack > 0)
						pp2->pk_ack--;
					else
						sprintf(pvmtxt, "netinput() Too
many Ack for pkt %d\n",
											pp-
>pk_seq);
```

```
                        break;
                    }
                goto scrap;
            }

            /*
            * send an ACK for the bcast pkt
            */
            pp2 = pk_new(DDFRAGHDR);/* XXX could reref a dummy
databuf here */
            pp2->pk_dat += DDFRAGHDR;
            pp2->pk_dst = 0;              /* set dst=0 for bcast pkt
ACK */
            pp2->pk_src = pvmmytid | TIDPVMD;
            pp2->pk_flag = FFACK;
            pp2->pk_nrt = 0;
            pp2->pk_seq = 0;
            pp2->pk_ack = sqn;                  /* put sequence
number in pk_ack */
            /* Send ack pkt here, if myself assume ack pkt can be
received */

            cp = pp2->pk_dat;
            len = pp2->pk_len;

            cp -= DDFRAGHDR;
            len += DDFRAGHDR;

            if (cp < pp2->pk_buf) {
                pvmlogerror("netinput() no headroom for ack pkt
header\n");
                goto afterack;
            }
            pvmput32(cp, pp2->pk_dst);
            pvmput32(cp + 4, pp2->pk_src);
            pvmput16(cp + 8, pp2->pk_seq);
            pvmput16(cp + 10, pp2->pk_ack);
            pvmput32(cp + 12, 0);               /* to keep purify
happy */
            pvmput8(cp + 12, pp2->pk_flag);

            BZERO((char *)&sin, sizeof(sin));
            sin.sin_family = AF_INET;
            sin.sin_port = EXPR_FIXED_PORT;
            sin.sin_addr.s_addr = osad.sin_addr.s_addr;

            if (debugmask & PDMMESSAGE) {
                sprintf(pvmtxt, "netintput() Send Ack pkt to
%s.\n",
                                    inet_ntoa(sin.sin_addr));
                pvmlogerror(pvmtxt);
            }

                if ((cc = sendto(netsock, cp, len, 0,
```

```
                              (struct sockaddr*)&sin, sizeof(sin)))
== -1
                  && errno != EINTR
                  && errno != ENOBUFS
#ifdef      IMA_LINUX
                  && errno != ENOMEM
#endif
                  ) {
                        pvmlogperror("netinput() sendto");
#if defined(IMA_SUN4SOL2) || defined(IMA_X86SOL2) ||
defined(IMA_SUNMP) || defined(IMA_UXPM)
                        /* life, don't talk to me about life... */
                        if (errno == ECHILD)
                              pvmlogerror("this message brought to
you by solaris\n");
                        else
#endif
                        pvmbailout(0);
                  }

afterack:
            if (SEQNUMCOMPARE(bcast_rxseq, sqn)) {  /* sqn >
bcast_rxseq ? */

                  if (debugmask & PDMMESSAGE) {
                        sprintf(pvmtxt, "netintput() Going to consume
pkt, bcast_rxseq=%d, sqn=%d.\n",
                                          bcast_rxseq, sqn);
                        pvmlogerror(pvmtxt);
                  }

                  bcast_rxseq = sqn;            /* Update bcast_rxseq
*/
                  bcast_txseq = bcast_rxseq;

                  /* Consume pkt - Make pk_dst of this packet to tid
of this pvmd. */
                  already = 0;
                  for (pp2=bcast_rxq->pk_link; pp2 != bcast_rxq;
pp2=pp2->pk_link)
                        if (pp2->pk_seq == sqn) {
                              already = 1;
                              break;
                        }
                  if (!already)
                        LISTPUTBEFORE(pp2, pp, pk_link, pk_rlink);

            }  /* end if -- (already received message) */
            else
                  goto scrap;

            /* consumes pkt from reorder q */

            while (pp = bcast_rxq->pk_link,
```

```
                        pp != bcast_rxq ) {

                    if (debugmask & PDMMESSAGE) {
                        pvmlogerror("netinput() Consuming pkts from
reorder q\n");
                    }

                    LISTDELETE(pp, pk_link, pk_rlink);
                    pp->pk_dst = pvmmytid | TIDPVMD;
                    netinpkt(hp, pp);
                }
                return 0;
            }   /* End if broadcast packet */

            if ((ff & (FFFIN|FFACK)) == (FFFIN|FFACK)) {
                if (hh == hosts->ht_master) {
            /*
            * FIN|ACK from master means we should bailout
            */
                    if (runstate == PVMDPRIME) {
                        if (debugmask & PDMSTARTUP)
                            pvmlogerror("work() PVMDPRIME
halting\n");
                        exit(0);
                    }
                    sprintf(pvmtxt, "netinput() FIN|ACK from master
(%s)\n",
                            hp->hd_name);
                    pvmlogerror(pvmtxt);
                    runstate = PVMDHALTING;

                } else {
            /*
            * FIN|ACK from slave means it croaked
            */
                    sprintf(pvmtxt, "netinput() FIN|ACK from %s\n",
                            hp->hd_name);
                    pvmlogerror(pvmtxt);
                    hd_dump(hp);
                    hostfailentry(hp);
                    clear_opq_of((int)(TIDPVMD | hp->hd_hostpart));
                    if (hp->hd_hostpart) {
                        ht_delete(hosts, hp);
                        if (newhosts)
                            ht_delete(newhosts, hp);
                    }
                }
                goto scrap;
            }

            /*
            * done with outstanding packet covered by this ack
            */
```

```
      if (ff & FFACK) {
            for (pp2 = hp->hd_opq->pk_link; pp2 != hp->hd_opq; pp2 =
pp2->pk_link)
                  if (pp2->pk_seq == aqn) {
                        if (pp2->pk_flag & FFDAT) {
                              if (pp2->pk_nrt == 1) {
                                    gettimeofday(&tnow, (struct
timezone*)0);

                                    TVXSUBY(&tdiff, &tnow, &pp2-
>pk_at);
                                    rttusec = tdiff.tv_sec * 1000000
+ tdiff.tv_usec;
                                    if (rttusec < 1)
                                          rttusec = 1000;   /* XXX
const */
                                    else
                                          if (rttusec >
DDMAXRTT*1000000)
                                                rttusec =
DDMAXRTT*1000000;
                                    rttusec += 3 * (hp->hd_rtt.tv_sec
* 1000000 + hp->hd_rtt.tv_usec);
                                    rttusec /= 4;
                                    hp->hd_rtt.tv_sec = rttusec /
1000000;
                                    hp->hd_rtt.tv_usec = rttusec %
1000000;
                              }
                        }
                        if (pp2->pk_flag & FFFIN) {
                              finack_to_host(hp);
                        }
                        hp->hd_nop--;
                        LISTDELETE(pp2, pk_link, pk_rlink);
                        pk_free(pp2);
                        break;
                  }
      }

      /*
      * move another pkt to output q
      */

/*
      if ((hp->hd_opq->pk_link == hp->hd_opq)
*/
      if (hp->hd_nop < nopax
      && (hp->hd_txq->pk_link != hp->hd_txq)) {
            if (debugmask & PDMPACKET) {
                  sprintf(pvmtxt, "netinput() pkt to opq\n");
                  pvmlogerror(pvmtxt);
            }
            pp2 = hp->hd_txq->pk_link;
```

```
            LISTDELETE(pp2, pk_link, pk_rlink);
            TVCLEAR(&pp2->pk_rtv);
            TVXADDY(&pp2->pk_rta, &hp->hd_rtt, &hp->hd_rtt);
            TVCLEAR(&pp2->pk_rto);
            TVCLEAR(&pp2->pk_at);
            pp2->pk_nrt = 0;
            pp2->pk_hostd = hp;
            pp2->pk_seq = hp->hd_txseq;
            hp->hd_txseq = NEXTSEQNUM(hp->hd_txseq);
            LISTPUTBEFORE(hp->hd_opq, pp2, pk_link, pk_rlink);
            hp->hd_nop++;
            LISTPUTAFTER(opq, pp2, pk_tlink, pk_trlink);
      }

      if (!(ff & (FFDAT|FFFIN)))
            goto scrap;

      /*
      * send an ack for the pkt
      */

      pp2 = pk_new(DDFRAGHDR);      /* XXX could reref a dummy
databuf here */
      pp2->pk_dat += DDFRAGHDR;
      pp2->pk_dst = hp->hd_hostpart | TIDPVMD;
      pp2->pk_src = pvmmytid;
      pp2->pk_flag = FFACK;
      TVCLEAR(&pp2->pk_rtv);
      TVCLEAR(&pp2->pk_rta);
      TVCLEAR(&pp2->pk_rto);
      TVCLEAR(&pp2->pk_at);
      pp2->pk_nrt = 0;
      pp2->pk_hostd = hp;
      pp2->pk_seq = 0;
      pp2->pk_ack = sqn;
      LISTPUTAFTER(opq, pp2, pk_tlink, pk_trlink);

      if (!(ff & FFDAT))
            goto scrap;

      /*
      * if we don't have it already, put it in reordering q
      */

      pp2 = 0;
      if (SEQNUMCOMPARE(sqn, hp->hd_rxseq))
            already = 1;
      else {
            already = 0;
            for (pp2 = hp->hd_rxq->pk_link; pp2 != hp->hd_rxq; pp2 =
pp2->pk_link)
                  if (pp2->pk_seq >= sqn) {
                        if (pp2->pk_seq == sqn)
                              already = 1;
```

```
                        break;
                }
        }
        if (already) {
                if (debugmask & PDMPACKET) {
                        sprintf(pvmtxt, "netinput() pkt resent from %s seq
%d\n",
                                        hp->hd_name, sqn);
                        pvmlogerror(pvmtxt);
                }
                goto scrap;
        }

        LISTPUTBEFORE(pp2, pp, pk_link, pk_rlink);

        /*
        * accept pkts from reordering q
        */

        while (pp = hp->hd_rxq->pk_link,
                        pp != hp->hd_rxq && pp->pk_seq == hp->hd_rxseq) {
                hp->hd_rxseq = NEXTSEQNUM(hp->hd_rxseq);
                LISTDELETE(pp, pk_link, pk_rlink);
                netinpkt(hp, pp);
        }
        return 0;

scrap:
        if (pp)
                pk_free(pp);
        return 0;
}

/*=================================================================*/


/* Create shared memory and semaphores */

static int
shmem_init()
{
        long shm_key;

        shm_key = SHM_BCAST;
        shm_hdr = sizeof(int);
        shm_size = SHM_SIZE;

        if ( (semid1 = semget(SEM_KEY1, 1, SEM_PERMS | IPC_CREAT)) < 0
) {
                sprintf(pvmtxt, "shmem_init(): semget(SEM_KEY1)
error\n");
                pvmlogerror(pvmtxt);
                return -1;
        }
```

```
      if ( (semid2 = semget(SEM_KEY2, 1, SEM_PERMS | IPC_CREAT)) < 0
) {
            sprintf(pvmtxt, "shmem_init(): semget(SEM_KEY2)
error\n");
            pvmlogerror(pvmtxt);
            return -1;
      }
      if ( (semid3 = semget(SEM_KEY3, 2, SEM_PERMS | IPC_CREAT)) < 0
) {
            sprintf(pvmtxt, "shmem_init(): semget(SEM_KEY3)
error\n");
            pvmlogerror(pvmtxt);
            return -1;
      }

      if ( (semid4 = semget(SEM_KEY4, 2, SEM_PERMS | IPC_CREAT)) < 0
) {
            sprintf(pvmtxt, "shmem_init(): semget(SEM_KEY4)
error\n");
            pvmlogerror(pvmtxt);
            return -1;
      }

   if ( (shmid = shmget(shm_key, shm_size, SHM_PERMS | IPC_CREAT)) <
   0 ) {
            return -1;
      }

      return 0;
}



/* Remove shared memory and semaphores */

shmem_cleanup()
{
      int cc;

      if (semid1 != -1)
            if (cc = semctl(semid1, 0, IPC_RMID, 0) < 0) {
                  sprintf(pvmtxt, "shmem_cleanup() can't remove
sem1.\n");
                  pvmlogerror(pvmtxt);
            }
      if (semid2 != -1)
            if (cc = semctl(semid2, 0, IPC_RMID, 0) < 0) {
                  sprintf(pvmtxt, "shmem_cleanup() can't remove
sem2.\n");
                  pvmlogerror(pvmtxt);
            }
      if (semid3 != -1)
```

```
            if (cc = semctl(semid3, 0, IPC_RMID, 0) < 0) {
                  sprintf(pvmtxt, "shmem_cleanup() can't remove
sem3.\n");
                  pvmlogerror(pvmtxt);
            }
      if (semid4 != -1)
            if (cc = semctl(semid4, 0, IPC_RMID, 0) < 0) {
                  sprintf(pvmtxt, "shmem_cleanup() can't remove
sem4.\n");
                  pvmlogerror(pvmtxt);
            }
      if (shmid != -1)
            if (cc = shmctl(shmid, IPC_RMID, (struct shmid_ds *) 0) <
0) {
                  sprintf(pvmtxt, "shmem_cleanup() shmctl() - can't
remove shmem.\n");
                  pvmlogerror(pvmtxt);
            }

}

/*=================================================================*/

/*    work()
*
*     The whole sausage
*/

work()
{
      static int lastpinged = 0;    /* host that got last keepalive
message */
      fd_set rfds, wfds;                  /* result of select */
/*
      fd_set efds;
*/
      int nrdy;                           /* number of fds ready
after select */
      struct timeval tbail;         /* time to bail if state =
STARTUP */
      struct timeval tping;         /* time to send next keepalive
packet */
      struct timeval tnow;
      struct timeval tout;
      struct mesg *mp;
      struct task *tp;
      struct hostd *hp;
         .
         .
         .

      for (; ; ) {

            /*
```

```
            *     clean up after any tasks that we got SIGCHLDs for
            */
            .
            .
            .


            bcastoutput();
            netoutput();

            .
            .
            .

            /*
            * send keepalive message to remote pvmd once in a while
            */
                        .
                        .
                        .


            rfds = wrk_rfds;
            wfds = wrk_wfds;

            if (debugmask & PDMSELECT) {
                  sprintf(pvmtxt, "work() wrk_nfds=%d\n", wrk_nfds);
                  pvmlogerror(pvmtxt);
                  print_fdset("work() rfds=", wrk_nfds, &rfds);
                  print_fdset("work() wfds=", wrk_nfds, &wfds);
            }

#if !defined(IMA_PGON) && !defined(IMA_I860)

            if ((nrdy = select(wrk_nfds,
#ifdef      FDSETISINT
                        (int *)&rfds, (int *)&wfds, (int *)0,
#else
                        &rfds, &wfds, (fd_set *)0,
#endif
                        &tout)) == -1) {
                  if (errno != EINTR) {
                        pvmlogperror("work() select");
                        sprintf(pvmtxt, " wrk_nfds=%d\n", wrk_nfds);
                        pvmlogerror(pvmtxt);
                        print_fdset(" rfds=", wrk_nfds, &wrk_rfds);
                        print_fdset(" wfds=", wrk_nfds, &wrk_wfds);
                        sprintf(pvmtxt, " netsock=%d, ppnetsock=%d,
loclsock=%d\n",
                                    netsock, ppnetsock, loclsock);
                        pvmlogerror(pvmtxt);
                        task_dump();
                        pvmbailout(0);
                  }
```

```
            }
#else /*IMA_PGON/IMA_I860*/

            .
            .
            .
#endif /*IMA_PGON/IMA_I860*/

                  .
                  .
                  .
      /*
      *     check network socket and local master socket for action
      */

            if (nrdy > 0) {
                  if (FD_ISSET(netsock, &rfds)) {
                        nrdy--;
                        netinput();
                  }
                  if (loclsock >= 0 && FD_ISSET(loclsock, &rfds)) {
                        nrdy--;
                        loclconn();
                  }
            }

      /*
      *     check tasks for action
      */

            .
            .
            .
}
```

Appendix B

Source Code of Pvmd-Task Protocol for Pvmd

```
static int
tid_elem_compare(i, j)
      tid_elem *i, *j;
{
      return i->tid - j->tid;
}

static int
gatids_compare(i, j)
      struct gatids *i, *j;
{
      return( i->tid - j->tid );
}

/*================================================================*/


/*    tm_bcasta()
*
*     Task wants to broadcast message to all tids in the list.
*
*/

int
tm_bcasta(tp, mp)
      struct task *tp;
      struct mesg *mp;
{
      struct cca *ccap;                         /* cca descriptor */
      char *shmptr;
      int  tidccamask = TIDCCA;
      int  ndst;                                      /* num of dst
tids */
      int *dsts;                                      /* dst tids */
      int  val;
      int  tid;
      int  il, i2, n;

      /* generate new cca local part */

      lastcca += TIDFIRSTCCA;

      if (lastcca > tidccamask)
            lastcca = TIDFIRSTCCA;

      /* Unpack request message from task */

      ccap = cca_new();
      ccap->cc_tid = myhostpart | TIDGID | lastcca;
      ccap->cc_type = CC_BCAST;

      upkint(mp, &(ccap->cc_datasize));
```

```
      ccap->cc_data = (char *) TALLOC(ccap->cc_datasize, char,
"cc_data");
      byteupk(mp, (char *)ccap->cc_data, ccap->cc_datasize, 1, 1);
      upkint(mp, &(ccap->cc_msgtag));
      upkint(mp, &(ccap->cc_roottid));
      upkint(mp, &ndst);
      dsts = TALLOC(ndst, int, "dsts");


/*
 * - Copy tid-list from shmem.
*/

   if ( (shmptr = (char *)shmat(shmid, (char *) 0, 0)) == (char *) -1
   ) {
                  sprintf(pvmtxt, "tm_bcasta(): shmat() error\n");
                  pvmlogerror(pvmtxt);
                  goto noguys;
      }

      /* Copy tid-list from shared memory */
      BCOPY((char *)(shmptr), (char *)dsts, ndst*sizeof(int));

      if (shmdt(shmptr) < 0) {
            sprintf(pvmtxt, "tm_bcasta(): shmdt() error\n");
            pvmlogerror(pvmtxt);
            goto noguys;
      }

      if (ndst < 1)           /* if no dsts then just return */
            goto noguys;

      /*
      * sort dst tids
      * XXX we could easily remove duplicates here
      * Make list of pvmds and its tids.
      */

      qsort((char*)dsts, ndst, sizeof(int), int_compare);

   ccap->cc_dsts = TALLOC(ndst, struct cc_elem, "tm_bcasta"); /*
   Cheat,         too much space */
      ccap->cc_ndst = 0;

/*
 * - Add tid of pvmd of local and remote tids to ccap-
>cc_dsts[].pvmdtid
 * - Create its tidlist (ccap->cc_dsts[].tids)
 * - send mesg DM_BCASTA along with pvmdtids and their tids to local
 *   pvmd first
 */


      for (i2 = 0; (i1 = i2) < ndst; ) {
```

```
            n = 0;
            tid = dsts[i1] & tidhmask;
            while (++i2 < ndst && tid == (dsts[i2] & tidhmask)) ;

            ccap->cc_dsts[ccap->cc_ndst].pvmdtid = tid;
            ccap->cc_dsts[ccap->cc_ndst].ntids = i2 - i1; /* number
of tids */
            ccap->cc_dsts[ccap->cc_ndst].tids = 0;
            ccap->cc_ndst++;
      }

   /* Create a broadcast message */

      mp = mesg_new(0);
      mp->m_dst = 0;                                    /* Broadcast to
all pvmds */
      mp->m_cod = DM_BCASTA;
      pkint(mp, ccap->cc_tid);                  /* cca address */
      pkint(mp, ccap->cc_type);
      pkint(mp, ccap->cc_datasize);
      bytepk(mp, (char *)ccap->cc_data, ccap->cc_datasize, 1, 1);
      pkint(mp, ccap->cc_msgtag);
      pkint(mp, ccap->cc_roottid);
      pkint(mp, ccap->cc_ndst);
      for (i1=0; i1<ccap->cc_ndst; i1++) {
            pkint(mp, ccap->cc_dsts[i1].pvmdtid);
            pkint(mp, ccap->cc_dsts[i1].ntids);
      }
      bcastmessage(mp);

noguys:
      PVM_FREE(dsts);

      /*
      * tag task descriptor with cca desc and send cca back to task
      */

      tp->t_cca = ccap;

      return 0;
}



/*=============================================================*/


int
tm_scattera(tp, mp)
      struct task *tp;
      struct mesg *mp;
{
      struct cca *ccap;                   /* cca descriptor */
      int  tidccamask = TIDCCA;
```

```
        char *shmptr, *tptr, *dptr;
        int  ndst;                              /* num of dst tids */
        tid_elem *dsts;                         /* dst tids */
        char *tdata=0;                          /* temp. data */
        int  intsize;
        int  val;
        int  tid;
        int  i1, i2, n;

        /* generate new cca local part */

        lastcca += TIDFIRSTCCA;

        if (lastcca > tidccamask)
              lastcca = TIDFIRSTCCA;

        /*
        * unpack request from message.
        */

        ccap = cca_new();
        ccap->cc_tid = myhostpart | TIDGID | lastcca;
        ccap->cc_type = CC_SCATTER;

        upkint(mp, &(ccap->cc_datasize));
        upkint(mp, &ndst);
        tdata = (char *) TALLOC(ccap->cc_datasize*ndst, char, "tdata");
        upkint(mp, &(ccap->cc_msgtag));
        upkint(mp, &(ccap->cc_roottid));
        dsts = TALLOC(ndst, tid_elem, "dsts");


/*
 *  Copy tid-list from shmem.
 */

    if ( (shmptr = (char *)shmat(shmid, (char *) 0, 0)) == (char *) -1
    ) {
                  sprintf(pvmtxt, "tm_scattera(): shmat() error\n");
                  pvmlogerror(pvmtxt);
                  goto noguys;
        }

        /* Copy tid-list from shared memory */
        intsize = sizeof(int);
        tptr = shmptr;
        dptr = (char *)tptr + ndst*intsize;

        BCOPY((char *)dptr, (char *)tdata, ccap->cc_datasize*ndst);

        for (i1 = 0; i1 < ndst; i1++) {
              BCOPY((char *)tptr, (char *)&dsts[i1].tid, intsize);
              tptr = (char *)tptr + intsize;
```

```
            /* dsts[i1].data = ((char *) dptr + i1*ccap-
>cc_datasize); */
            dsts[i1].data = ((char *) tdata + i1*ccap->cc_datasize);
      }

      if (ndst < 1)           /* if no dsts then just return */
            goto noguys;

      /*
      * sort dst tids
      * XXX we could easily remove duplicates here
      * Make list of pvmds and its tids.
      * Send DM_MCA messages containing pvmds and tids to local host
first.
      */

      qsort((char*)dsts, ndst, sizeof(tid_elem), tid_elem_compare);

      ccap->cc_dsts = TALLOC(ndst, struct cc_elem, "tm_scattera"); /*
                                    Cheat, too much space */
      ccap->cc_ndst = 0;

/*
 * - Add tid of pvmd of local and remote tids to ccap-
>cc_dsts[].pvmdtid
 * - Create its tidlist (ccap->cc_dsts[].tids)
 * - send mesg DM_BCASTA along with pvmdtids and their tids to local
 *   pvmd first
 */


      for (i2 = 0; (i1 = i2) < ndst; ) {
            n = 0;
            tid = dsts[i1].tid & tidhmask;
            while (++i2 < ndst && tid == (dsts[i2].tid & tidhmask)) ;

            ccap->cc_dsts[ccap->cc_ndst].pvmdtid = tid;
            ccap->cc_dsts[ccap->cc_ndst].ntids = i2 - i1; /* number
of tids */
            ccap->cc_dsts[ccap->cc_ndst].tids = TALLOC((i2 - i1),
tid_elem, "tm_scattera");

            while (i1 < i2) {
                  ccap->cc_dsts[ccap->cc_ndst].tids[n].tid =
dsts[i1].tid;
                  ccap->cc_dsts[ccap->cc_ndst].tids[n++].data =
dsts[i1++].data;
            }
            ccap->cc_ndst++;
      }

      /* Create scatter message */

      mp = mesg_new(0);
```

```
        mp->m_dst = (pvmmytid |= TIDPVMD);  /* send to local pvmd */
        mp->m_cod = DM_SCATTERA;
        pkint(mp, ccap->cc_tid);                    /* cca address */
        pkint(mp, ccap->cc_type);
        pkint(mp, ccap->cc_datasize);       /* NOTE: no ccap->cc_data
*/
        pkint(mp, ccap->cc_msgtag);
        pkint(mp, ccap->cc_roottid);
        pkint(mp, ccap->cc_ndst);
        for (i1=0; i1<ccap->cc_ndst; i1++) {
              pkint(mp, ccap->cc_dsts[i1].pvmdtid);
              pkint(mp, ccap->cc_dsts[i1].ntids);
              for (i2=0; i2<ccap->cc_dsts[i1].ntids; i2++) {
                    pkint(mp, ccap->cc_dsts[i1].tids[i2].tid);
                    bytepk(mp, (char *)ccap->cc_dsts[i1].tids[i2].data,
ccap->cc_datasize, 1, 1);
                    ccap->cc_dsts[i1].tids[i2].data = 0;
              }
        }


        if (shmdt(shmptr) < 0) {
              sprintf(pvmtxt, "tm_scattera(): shmdt() error\n");
              pvmlogerror(pvmtxt);
              goto noguys;
        }

        bcastmessage(mp);

noguys:
        PVM_FREE(dsts);
        PVM_FREE(tdata);

        /*
        * tag task descriptor with cca desc and send cca back to task
        */

        tp->t_cca = ccap;

        return 0;
}



/*============================================================================*/



int
tm_gathera(tp, mp)
        struct task *tp;
```

```
      struct mesg *mp;
{
      struct cca *ccap;                    /* cca descriptor */
      char *shmptr, *ptr;
      int  tidccamask = TIDCCA;
      int  ndst;                           /* num of dst tids */
      struct gatids *dsts;                 /* dst tids and inst. */
      int  val;
      int  tid;
      int  intsize;
      int  i1, i2, n;

      intsize = sizeof(int);

      /* generate new cca local part */

      lastcca += TIDFIRSTCCA;

      if (lastcca > tidccamask)
            lastcca = TIDFIRSTCCA;

      /*
      * unpack request from message.
      */

      ccap = cca_new();
      ccap->cc_tid = myhostpart | TIDGID | lastcca;
      ccap->cc_type = CC_GATHER;

      upkint(mp, &(ccap->cc_datasize));
      upkint(mp, &ndst);
      upkint(mp, &(ccap->cc_msgtag));
      upkint(mp, &(ccap->cc_roottid));
      dsts = TALLOC(ndst, struct gatids, "dsts");

/*
 * - read tids from shmem.
*/

      if ( (shmptr = (char *)shmat(shmid, (char *) 0, 0)) == (char *)
-1 ) {
                  sprintf(pvmtxt, "tm_gathera(): shmat() error\n");
                  pvmlogerror(pvmtxt);
                  goto noguys;
      }

      /* Copy tid-list from shared memory */
      ptr = shmptr;
      for (i1=0; i1<ndst; i1++) {
            BCOPY((char *)(ptr), (char *)&dsts[i1].tid, intsize);
            dsts[i1].inst = i1;
            ptr = (char *)ptr + intsize;
      }
```

```
      if (shmdt(shmptr) < 0) {
            sprintf(pvmtxt, "tm_gathera(): shmdt() error\n");
            pvmlogerror(pvmtxt);
            goto noguys;
      }

      if (ndst < 1)           /* if no dsts then just return */
            goto noguys;

      /*
      * sort dst tids
      * XXX we could easily remove duplicates here
      * Make list of pvmds and its tids.
      * Send DM_MCA messages containing pvmds and tids to local host
first.
      */

      qsort((char*)dsts, ndst, sizeof(struct gatids),
gatids_compare);

      ccap->cc_dsts = TALLOC(ndst, struct cc_elem, "tm_gathera"); /*
Cheat,                        too much space */
      ccap->cc_ndst = 0;

/*
 * - Add tid of pvmd of local and remote tids to ccap-
>cc_dsts[].pvmdtid
 * - Create its tidlist (ccap->cc_dsts[].tids)
 * - send mesg DM_BCASTA along with pvmdtids and their tids to local
 *   pvmd first
 */


      for (i2 = 0; (i1 = i2) < ndst; ) {
            n = 0;
            tid = dsts[i1].tid & tidhmask;
            while (++i2 < ndst && tid == (dsts[i2].tid & tidhmask)) ;

            ccap->cc_dsts[ccap->cc_ndst].pvmdtid = tid;
            ccap->cc_dsts[ccap->cc_ndst].ntids = i2 - i1; /* number
of tids */
            ccap->cc_dsts[ccap->cc_ndst].tids = TALLOC((i2 - i1),
tid_elem, "tm_gathera");

            /* Put insts. in tids for gather/reduce op. */
            /* will be used again in dm_gatherc() to put data in the
right place */
            while (i1 < i2) {
                  ccap->cc_dsts[ccap->cc_ndst].tids[n].tid =
dsts[i1++].inst;
                  ccap->cc_dsts[ccap->cc_ndst].tids[n++].data = 0;
            }

            ccap->cc_ndst++;
```

```
        }

        ga_ccap = ccap;         /* save for dm_gatherc(), will be freed
there */

        /* Create Gather message */

        mp = mesg_new(0);
        mp->m_dst = (pvmmytid |= TIDPVMD);  /* send to local pvmd */
        mp->m_cod = DM_GATHERA;
        pkint(mp, ccap->cc_tid);                    /* cca address */
        pkint(mp, ccap->cc_type);
        pkint(mp, ccap->cc_datasize);       /* NOTE: no ccap->cc_data
*/
        pkint(mp, ccap->cc_msgtag);
        pkint(mp, ccap->cc_roottid);
        pkint(mp, ccap->cc_ndst);
        for (i1=0; i1<ccap->cc_ndst; i1++) {
              pkint(mp, ccap->cc_dsts[i1].pvmdtid);
              pkint(mp, ccap->cc_dsts[i1].ntids);
        }
        bcastmessage(mp);

noguys:
        PVM_FREE(dsts);

}



/*      This message send from last task that writes data to shmem
        */
/* to tell local pvmd to read all data and continue the operation */
/*      Local pvmd will read data {tid, data} and sort them by tid
*/
/* then send only data to root pvmd in order.
              */
int
tm_gatherb(tp, mp)
        struct task *tp;
        struct mesg *mp;
{
        char *shmptr = 0;
        char *ptr;
        tid_elem *dd=0;
        int   cc_datasize, cc_type, cc_msgtag;
        int   intsize;
        int   inst, ntids;
        int   i, cc;
        int   semval;
        struct sembuf reset_head[1] = {
```

```
                        0, -1, IPC_NOWAIT }; /* decrease value of
sem#0 by 1 */


/*    upkint(mp, &cc_tid);    */
      upkint(mp, &cc_type);
      upkint(mp, &cc_datasize);
      upkint(mp, &cc_msgtag);

      if (debugmask & (PDMMESSAGE)) {
            sprintf(pvmtxt, "dm_gatherb(): start, cc_datasize =
%d.\n", cc_datasize);
            pvmlogerror(pvmtxt);
      }

   if ( (shmptr = (char *) shmat(shmid, (char *) 0, 0)) == (char *) -
   1 ) {
            sprintf(pvmtxt, "dm_gatherb(): shmat() error\n");
            pvmlogerror(pvmtxt);
            goto done;
      }

      /* Skip Header  # of tids */
      intsize = sizeof(int);
      ptr = shmptr;
      BCOPY((char *)ptr, (char *)&ntids, intsize);
      ptr = (char *)ptr + (intsize*2);
      dd = TALLOC(ntids, tid_elem, "tm_gatherb - dd");

      /* tid and data */
      for (i=0; i < ntids; i++) {
            BCOPY((char *)ptr, (char *)&dd[i].tid, intsize);
            ptr = (char *)ptr + intsize;
            dd[i].data = 0;
            dd[i].data = (char *) TALLOC(cc_datasize, char,
"dd[i].data");
            BCOPY((char *)ptr, (char *)dd[i].data, cc_datasize);
            ptr = (char *)ptr + cc_datasize;
      }

      /* sort by tid */
      qsort((char*)dd, ntids, sizeof(tid_elem), tid_elem_compare);

      /* this pvmd sends DM_GATHERC to root pvmd */
      mp = mesg_new(0);
      mp->m_dst = parent_tid;                    /* send data to root
pvmd */
      mp->m_cod = DM_GATHERC;
      pkint(mp, cc_type);
      pkint(mp, cc_datasize);
      pkint(mp, cc_msgtag);
      pkint(mp, ga_index);          /* pvmd index of this pvmd in
ga_ccap */
      pkint(mp, ntids);
```

```
        for (i=0; i < ntids; i++) {
                bytepk(mp, (char *)dd[i].data, cc_datasize, 1, 1);
        }

        sendmessage(mp);

        if (debugmask & (PDMMESSAGE)) {
                sprintf(pvmtxt, "dm_gatherb(): After send message.\n");
                pvmlogerror(pvmtxt);
        }

        if ((cc = shmdt(shmptr)) < 0) {
                sprintf(pvmtxt, "dm_gatherb(): shmdt() error\n");
                pvmlogerror(pvmtxt);
        }

done:

        /* reset sem#0 */
        if ((cc = semop(semid3, &reset_head[0], 1)) < 0) {
                sprintf(pvmtxt, "dm_gatherb(): semop(reset_head)
error\n");
                pvmlogerror(pvmtxt);
        }

        if (debugmask & (PDMMESSAGE)) {
                semval = semctl(semid3, 0, GETVAL, 0);
                sprintf(pvmtxt, "dm_gatherb(): sem#0 reset to %d.\n",
semval);
                pvmlogerror(pvmtxt);
        }

        if (dd) {
                for (i=0; i<ntids; i++)
                        if (dd[i].data)
                                PVM_FREE(dd[i].data);
                PVM_FREE(dd);
        }

        return 0;
}

/*============================================================*/


int
tm_reducea(tp, mp)
        struct task *tp;
        struct mesg *mp;
{
        struct cca *ccap;                       /* cca descriptor */
        char *shmptr, *ptr;
        int  tidccamask = TIDCCA;
```

```
      int  ndst;                                        /* num of dst
tids */
      int *dsts;                                        /* dst tids */
      int  val;
      int  tid;
      int  intsize;
      int  i1, i2, n;

      intsize = sizeof(int);

      /* generate new cca local part */

      lastcca += TIDFIRSTCCA;

      if (lastcca > tidccamask)
            lastcca = TIDFIRSTCCA;

      /*
      * unpack request from message.
      */

      ccap = cca_new();
      ccap->cc_tid = myhostpart | TIDGID | lastcca;
      ccap->cc_type = CC_REDUCE;

      upkint(mp, &(ccap->cc_datasize));
      upkint(mp, &ndst);
      upkint(mp, &(ccap->cc_msgtag));
      upkint(mp, &(ccap->cc_roottid));
      dsts = TALLOC(ndst, int, "dsts");

/*
 * - read tids from shmem.
*/

   if ( (shmptr = (char *)shmat(shmid, (char *) 0, 0)) == (char *) -1
   ) {
                  sprintf(pvmtxt, "tm_reducea(): shmat() error\n");
                  pvmlogerror(pvmtxt);
                  goto noguys;
      }

      /* Copy tid-list from shared memory */
      ptr = shmptr;
      BCOPY((char *)(ptr), (char *)dsts, intsize*ndst);

      if (shmdt(shmptr) < 0) {
            sprintf(pvmtxt, "tm_reducea(): shmdt() error\n");
            pvmlogerror(pvmtxt);
            goto noguys;
      }

      if (ndst < 1)           /* if no dsts then just reply to task
*/
```

```
            goto noguys;

      /*
      * sort dst tids
      * XXX we could easily remove duplicates here
      * Make list of pvmds and its tids.
      * Send DM_REDUCEA messages containing pvmds and tids to all
hosts.
      */

      qsort((char*)dsts, ndst, intsize, int_compare);

      ccap->cc_dsts = TALLOC(ndst, struct cc_elem, "tm_gathera"); /*
Cheat,                  too much space */
      ccap->cc_ndst = 0;

/*
 * - Add tid of pvmd of local and remote tids to ccap-
>cc_dsts[].pvmdtid
 * - Create its tidlist (ccap->cc_dsts[].tids)
 * - send mesg DM_BCASTA along with pvmdtids and their tids to local
 *   pvmd first
 */

      for (i2 = 0; (i1 = i2) < ndst; ) {
            n = 0;
            tid = dsts[i1] & tidhmask;
            while (++i2 < ndst && tid == (dsts[i2] & tidhmask)) ;

            ccap->cc_dsts[ccap->cc_ndst].pvmdtid = tid;
            ccap->cc_dsts[ccap->cc_ndst].ntids = i2 - i1; /* number
of tids */
            ccap->cc_dsts[ccap->cc_ndst].tids = 0;
            ccap->cc_ndst++;
      }

      /* Create reduce message */

      mp = mesg_new(0);
      mp->m_dst = (pvmmytid |= TIDPVMD);  /* send to local pvmd */
      mp->m_cod = DM_REDUCEA;
      pkint(mp, ccap->cc_tid);                  /* cca address */
      pkint(mp, ccap->cc_type);
      pkint(mp, ccap->cc_datasize);       /* NOTE: no ccap->cc_data
*/
      pkint(mp, ccap->cc_msgtag);
      pkint(mp, ccap->cc_roottid);
      pkint(mp, ccap->cc_ndst);
      for (i1=0; i1<ccap->cc_ndst; i1++) {
            pkint(mp, ccap->cc_dsts[i1].pvmdtid);
            pkint(mp, ccap->cc_dsts[i1].ntids);
      }
      bcastmessage(mp);
```

```
noguys:
      PVM_FREE(dsts);

}

/*    This message send from last task that write data to shmem
      */
/* to tell local pvmd to read all data and continue the operation */
/*    Local pvmd will read data and perform global computation op.
*/
/* then send final result data to root pvmd.
            */
int
tm_reduceb(tp, mp)
      struct task *tp;
      struct mesg *mp;
{
      char *shmptr = 0;
      char *ptr;
      char *data=0, *work=0;
      int   cc_datasize, cc_type, cc_msgtag;
      int   datatype, count;
      int   func;
      void (*reduce_func)();
      int   intsize;
      int   inst, ntids;
      int   i, cc;
      int   semval;
      struct sembuf reset_head[1] = {
                        0, -1, IPC_NOWAIT }; /* decrease value of
sem#0 by 1 */


      upkint(mp, &cc_type);
      upkint(mp, &cc_datasize);
      upkint(mp, &cc_msgtag);
      upkint(mp, &datatype);
      upkint(mp, &count);
      upkint(mp, &func);

      /* Add more function here */

      switch (func) {
            case REDUCESUM:   reduce_func = PvmSum;
                              break;
            default:          sprintf(pvmtxt, "tm_reduceb() [%d]
function unknown\n",
                                                      func);
                              pvmlogerror(pvmtxt);
                              cc = -1;
                              goto done;
      }
```

```
   if ( (shmptr = (char *) shmat(shmid, (char *) 0, 0)) == (char *) -
   1 ) {
            sprintf(pvmtxt, "dm_reduceb(): shmat() error\n");
            pvmlogerror(pvmtxt);
            goto done;
      }

      /* Skip Header  # of tids */
      intsize = sizeof(int);
      ptr = shmptr;
      BCOPY((char *)ptr, (char *)&ntids, intsize);
      ptr = (char *)ptr + (intsize*2);

      if (debugmask & (PDMMESSAGE)) {
            sprintf(pvmtxt, "dm_reduceb(): From shm header, ntids =
%d, shmptr = %x.\n", ntids, shmptr);
            pvmlogerror(pvmtxt);
      }

      /* perform global computation on data */

      data = (char *) TALLOC(cc_datasize, char, "data");
      work = (char *) TALLOC(cc_datasize, char, "work");

      BCOPY((char *)ptr, (char *)data, cc_datasize);
      ptr = (char *)ptr + cc_datasize;

      for (i=1; i < ntids; i++) {
            BCOPY((char *)ptr, (char *)work, cc_datasize);

            (*reduce_func)(&datatype, data, work, &count, &cc);

            ptr = (char *)ptr + cc_datasize;
      }

      if (debugmask & (PDMMESSAGE)) {
            sprintf(pvmtxt, "dm_reduceb(): Sending DM_REDUCEC, data =
%d.\n",
                              *(int *)data);
            pvmlogerror(pvmtxt);
      }

      /* this pvmd sends DM_REDUCEC along with final result to root
pvmd */
      mp = mesg_new(0);
      mp->m_dst = parent_tid;                  /* send data to root
pvmd */
      mp->m_cod = DM_REDUCEC;
      pkint(mp, cc_type);
      pkint(mp, cc_datasize);
      pkint(mp, cc_msgtag);
      pkint(mp, datatype);
      pkint(mp, count);
      pkint(mp, func);
```

```
        bytepk(mp, (char *)data, cc_datasize, 1, 1);

        sendmessage(mp);

        if ((cc = shmdt(shmptr)) < 0) {
                sprintf(pvmtxt, "dm_reduceb(): shmdt() error\n");
                pvmlogerror(pvmtxt);
        }

done:

        /* reset sem#0 */
        if ((cc = semop(semid4, &reset_head[0], 1)) < 0) {
                sprintf(pvmtxt, "dm_reduceb(): semop(reset_head)
error\n");
                pvmlogerror(pvmtxt);
        }

        if (debugmask & (PDMMESSAGE)) {
                semval = semctl(semid4, 0, GETVAL, 0);
                sprintf(pvmtxt, "dm_reduceb(): sem#0 reset to %d.\n",
semval);
                pvmlogerror(pvmtxt);
        }

        if (data)
                PVM_FREE(data);
        if (work)
                PVM_FREE(work);

        return 0;
}


/* PvmSum()*/

/*
  void PvmSum(int *datatype, void *x, void *y, *num, *info)

  Assigns the elements of x the sum of the corresponding elements of
x and y.
*/

void
PvmSum(datatype, x, y, num, info)
int *datatype;
void *x, *y;
int *num, *info;
{
    short  *xshort,  *yshort;
    int    *xint,    *yint;
    long   *xlong,   *ylong;
    float  *xfloat,  *yfloat;
    double *xdouble, *ydouble;
```

```
    int i, count;

    count = *num;

    switch(*datatype)
    {
        case (PVM_SHORT):
            xshort = (short *) x;
            yshort = (short *) y;
            for (i=0; i<count; i++) xshort[i] += yshort[i];
            break;
        case (PVM_INT):
            xint = (int *) x;
            yint = (int *) y;
            for (i=0; i<count; i++) xint[i] += yint[i];
            break;
        case (PVM_LONG):
            xlong = (long *) x;
            ylong = (long *) y;
            for (i=0; i<count; i++) xlong[i] += ylong[i];
            break;
        case (PVM_FLOAT):
            xfloat = (float *) x;
            yfloat = (float *) y;
            for (i=0; i<count; i++) xfloat[i] += yfloat[i];
            break;
        case (PVM_DOUBLE):
            xdouble = (double *) x;
            ydouble = (double *) y;
            for (i=0; i<count; i++) xdouble[i] += ydouble[i];
            break;
        case (PVM_CPLX):
            /* complex - complex*8 in fortran - treated as two floats
*/
            /* returns the sum of the two complex pairs */
            xfloat = (float *) x;
            yfloat = (float *) y;
            for (i=0; i<2*count; i++) xfloat[i]  += yfloat[i];
            break;
        case (PVM_DCPLX):
            /* double complex - complex*16 in fortran - treated as 2
doubles */
            /* returns the sum of the two complex pairs */
            xdouble = (double *) x;
            ydouble = (double *) y;
            for (i=0; i<2*count; i++) xdouble[i]   += ydouble[i];
            break;
        default:
            *info = PvmBadParam;
            return;
    }  /* end switch */

    *info = PvmOk;
```

```
    return;
}  /* end of PvmSum() */
```

Appendix C

Source Code of Pvmd-Pvmd Protocol for Pvmd

```
/*    dm_bcasta()
*
*   - This message is broadcast from tm_bcasta() of root pvmd.
*   - If this message does not contian our tid of pvmd, the drop it.
*   - If we are in pvmd-list, get our number of TIDs and send data to
*     them.
*
*/

int
dm_bcasta(hp, mp)
      struct hostd *hp;
      struct mesg *mp;
{
      struct cca *ccap;
      int i, j;
      int index = -1;
      int s1, s2;

      /* unpack struct cca from message */

      ccap = cca_new();
      upkint(mp, &ccap->cc_tid);
      upkint(mp, &ccap->cc_type);
      upkint(mp, &ccap->cc_datasize);
      ccap->cc_data = (char *) TALLOC(ccap->cc_datasize, char,
"dm_bcasta");
      byteupk(mp, (char *)ccap->cc_data, ccap->cc_datasize, 1, 1);
      upkint(mp, &ccap->cc_msgtag);
      upkint(mp, &ccap->cc_roottid);
      upkint(mp, &ccap->cc_ndst);
      ccap->cc_dsts = TALLOC(ccap->cc_ndst, struct cc_elem,
"bcastad");
      for (i = 0; i < ccap->cc_ndst; i++) {
            upkint(mp, &ccap->cc_dsts[i].pvmdtid);
            if ((ccap->cc_dsts[i].pvmdtid | TIDPVMD) == (pvmmytid |
TIDPVMD))
                  index = i;        /* if we are in, get our index */

            upkint(mp, &ccap->cc_dsts[i].ntids);
            ccap->cc_dsts[i].tids = 0;
      }

      /* Go on of we are in. If not, just drop it */
      if (index != -1)
            bcastd(ccap, index);     /* Send data to tasks */

      cca_free(ccap);

      return 0;
}


int bcastd(ccap, index)
```

```
      struct cca   *ccap;
      int  index;                   /* index of ccap->cc_dsts[] */
{
      int   i, cc;
      int   dst;
      char *shmptr = 0;
      struct sembuf r_empty[1] = {
                        0, 0, 0 };              /* wait for sem#0
become 0 */
      struct sembuf r_send[1] = {
                        0, 1, 0 };  /* increase value of sem#0 by 1
*/
      struct sembuf c_recv[1] = {
                        0, -1, 0 }; /* wait for sem#0 >= 1 and
decrement by 1 */

      /* create or attach to shared memory */
      if (ccap->cc_ndst) {
            if ( (shmptr = shmat(shmid, (char *) 0, 0)) == (char *) -
1 ) {
                        sprintf(pvmtxt, "bcastd(): shmat() error\n");
                        pvmlogerror(pvmtxt);
                        cc = -1;
                        goto done;
            }

            /* Copy data to shared memory */
            BCOPY((char *)ccap->cc_data, (char *)(shmptr + shm_hdr),
ccap->cc_datasize);

            if ((cc = shmdt(shmptr)) < 0) {
                  sprintf(pvmtxt, "bcastd(): shmdt() error\n");
                  pvmlogerror(pvmtxt);
                  goto done;
            }
      }

      r_send[0].sem_op = ccap->cc_dsts[index].ntids;
      if ((cc = semop(semid1, &r_send[0], 1)) < 0) {
            sprintf(pvmtxt, "bcastd(): shmdt() error\n");
            pvmlogerror(pvmtxt);
            goto done;
      }

      if (debugmask & (PDMMESSAGE)) {
            sprintf(pvmtxt, "bcastd() signal semaphore by sem_op =
%d.\n",
                                          ccap-
>cc_dsts[index].ntids);
            pvmlogerror(pvmtxt);
      }

      cc = 0;
```

```
done:

      if (cc < 0) {
            sprintf(pvmtxt, "bcastd() fail. return %d\n", cc);
            pvmlogerror(pvmtxt);

      }
      return cc;
}


/*=================================================================*/


int
dm_scattera(hp, mp)
      struct hostd *hp;
      struct mesg *mp;
{

      struct cca *ccap;
      int i, j;
      int index = -1;

      /* unpack struct cca from message */

      ccap = cca_new();
      upkint(mp, &ccap->cc_tid);
      upkint(mp, &ccap->cc_type);
      upkint(mp, &ccap->cc_datasize);
      upkint(mp, &ccap->cc_msgtag);
      upkint(mp, &ccap->cc_roottid);
      upkint(mp, &ccap->cc_ndst);
      ccap->cc_dsts = TALLOC(ccap->cc_ndst, struct cc_elem,
"dm_scattera");
      for (i = 0; i < ccap->cc_ndst; i++) {
            upkint(mp, &ccap->cc_dsts[i].pvmdtid);
            if ((ccap->cc_dsts[i].pvmdtid | TIDPVMD) == (pvmmytid |
TIDPVMD))
                  index = i;  /* use for MST */

            upkint(mp, &ccap->cc_dsts[i].ntids);
            ccap->cc_dsts[i].tids = (tid_elem *) TALLOC(ccap-
>cc_dsts[i].ntids, tid_elem, "dm_scattera");

            for (j=0; j<ccap->cc_dsts[i].ntids; j++) {
                  upkint(mp, &ccap->cc_dsts[i].tids[j].tid);
                  ccap->cc_dsts[i].tids[j].data = (char *)
TALLOC(ccap->cc_datasize, char, "dm_scattera");
                  byteupk(mp, (char *)ccap->cc_dsts[i].tids[j].data,
ccap->cc_datasize, 1, 1);
            }
      }
```

```
        /* Go on of we are in. If not, just drop it */
        if (index != -1)
                scatterd(ccap, index);   /* Send data to tasks */

        cca_free(ccap);

        return 0;
}

int scatterd(ccap, index)
        struct cca   *ccap;
        int  index;                     /* index of ccap->cc_dsts[] */
{
        int   i, cc;
        int   dst;
        int   size;
        char *shmptr = 0;
        char *ptr;
        struct sembuf r_empty[1] = {
                        0, 0, 0 };              /* wait for sem#0
become 0 */
        struct sembuf r_send[1] = {
                        0, 1, 0 };  /* increase value of sem#0 by 1
*/
        struct sembuf c_recv[1] = {
                        0, -1, 0 }; /* wait for sem#0 >= 1 and
decrement by 1 */


        /* create or attach to shared memory */
        if (ccap->cc_ndst) {
                if ( (shmptr = shmat(shmid, (char *) 0, 0)) == (char *) -
1 ) {
                        sprintf(pvmtxt, "scatterd(): shmat()
error\n");
                        pvmlogerror(pvmtxt);
                        cc = -1;
                        goto done;
                }

                /* Copy data to shared memory */

                ptr = shmptr;

                /* Header  = # of tids */
                size = sizeof(int);
                BCOPY((char *)&ccap->cc_dsts[index].ntids, (char *) ptr,
size);
                ptr += size;

                /* tid-list (sorted) */
                for (i=0; i < ccap->cc_dsts[index].ntids; i++) {
                      BCOPY((char *)&ccap->cc_dsts[index].tids[i].tid,
(char *) ptr, size);
```

```
                ptr += size;
            }

            /* data ordered by tid-list */
            size = ccap->cc_datasize;
            for (i=0; i < ccap->cc_dsts[index].ntids; i++) {
                BCOPY((char *)ccap->cc_dsts[index].tids[i].data,
(char *) ptr, size);
                ptr += size;
            }

            if ((cc = shmdt(shmptr)) < 0) {
                sprintf(pvmtxt, "scatterd(): shmdt() error\n");
                pvmlogerror(pvmtxt);
                goto done;
            }
      }

      r_send[0].sem_op = ccap->cc_dsts[index].ntids;
      if ((cc = semop(semid2, &r_send[0], 1)) < 0) {
            sprintf(pvmtxt, "scatterd(): shmdt() error\n");
            pvmlogerror(pvmtxt);
            goto done;
      }

      if (debugmask & (PDMMESSAGE)) {
            sprintf(pvmtxt, "scatterd() signal semaphore by sem_op =
%d.\n",
                                          ccap-
>cc_dsts[index].ntids);
            pvmlogerror(pvmtxt);
      }

      cc = 0;

done:

      if (cc < 0) {
            sprintf(pvmtxt, "scatterd() fail. returning %d\n", cc);
            pvmlogerror(pvmtxt);

      }

      return cc;
}


/*=============================================================*/


int
dm_gathera(hp, mp)
      struct hostd *hp;
      struct mesg *mp;
```

```
{
      struct cca *ccap;
      int i, j;
      int index = -1;
      int root;
      int tid_count = 0;
      struct btree *bt=0;

      /* keep to reply to */
      parent_tid = mp->m_src;
      child_count = 0;

      /* unpack struct cca from message */

      ccap = cca_new();
      upkint(mp, &ccap->cc_tid);
      upkint(mp, &ccap->cc_type);
      upkint(mp, &ccap->cc_datasize);
      upkint(mp, &ccap->cc_msgtag);
      upkint(mp, &ccap->cc_roottid);
      upkint(mp, &ccap->cc_ndst);
      ccap->cc_dsts = TALLOC(ccap->cc_ndst, struct cc_elem,
"dm_gathera");
      for (i = 0; i < ccap->cc_ndst; i++) {
            upkint(mp, &ccap->cc_dsts[i].pvmdtid);
            if ((ccap->cc_dsts[i].pvmdtid | TIDPVMD) == (pvmmytid |
TIDPVMD))
                  index = i;
            if ((ccap->cc_dsts[i].pvmdtid | TIDPVMD) ==
                        ((ccap->cc_roottid & TIDHOST) | TIDPVMD))
                  root = i;

            upkint(mp, &ccap->cc_dsts[i].ntids);
            ccap->cc_dsts[i].tids = 0;
            tid_count += ccap->cc_dsts[i].ntids;      /* for allocate
ga_data.data */
      }

      if (index == -1)  /* Not my msg, drop it */
            goto scrap;

      ga_index = index;       /* global var */

      if ( (pvmmytid | TIDPVMD) == ((ccap-
>cc_roottid&TIDHOST)|TIDPVMD) ) {
            is_root_pvmd = 1;
            child_count = ccap->cc_ndst;
      }

      if (is_root_pvmd) {     /* allocate ga_data.data */
            ga_data.ndata = 0;
            ga_data.data = TALLOC(tid_count*ccap->cc_datasize, char,
"ga_data.data");
      }
```

```
      gatherd(ccap, index);         /* Initialize shared memory for
tasks to write */

scrap:
      if (ccap)
            cca_free(ccap);               /* Check this out */
      if (bt)
            PVM_FREE(bt);

      return 0;
}

int
dm_gatherc(hp, mp)
      struct hostd *hp;
      struct mesg *mp;
{
      struct cca *ccap;
      char *ptr, *dptr;
      char *dd=0, *ddptr;
      int   intsize;
      int   inst;
      int   cc_type, cc_tid, cc_datasize, cc_msgtag;
      int   ntids;
      int   index;
      int   i, cc;
      int   pvmd_src;
      char *shmptr = 0;
      struct sembuf wait_write[1] = {
                        1, 0, 0 };              /* wait for sem#1
become 0 */
      struct sembuf sig_read[1] = {
                        1, 1, 0 };              /* increase value of
sem#1 by 1 */

      intsize = sizeof(int);

      pvmd_src = mp->m_src | TIDPVMD;

      upkint(mp, &cc_type);
      upkint(mp, &cc_datasize);
      upkint(mp, &cc_msgtag);
      upkint(mp, &index);
      upkint(mp, &ntids);
      dd = TALLOC(cc_datasize*ntids, char, "dm_gatherc - dd");
      ddptr = dd;
      for (i = 0; i < ntids; i++) {
            byteupk(mp, (char *)ddptr, cc_datasize, 1, 1);
            ddptr = (char *)ddptr + cc_datasize;
      }

      if (debugmask & PDMMESSAGE) {
```

```
            sprintf(pvmtxt, "dm_gatherc() index = %d ntids = %d\n",
                              index, ntids);
            pvmlogerror(pvmtxt);
            sprintf(pvmtxt, "dm_gatherc() ga_ccap-
>cc_dsts[index].ntids = %d\n",
                              ga_ccap->cc_dsts[index].ntids);
            pvmlogerror(pvmtxt);
      }

      /* Match data with inst in ga_ccap */
      if (ga_ccap->cc_dsts[index].ntids != ntids) {
            sprintf(pvmtxt, "dm_gatherc() number of data not match
%d, %d\n",
                              ga_ccap->cc_dsts[index].ntids, ntids);
            pvmlogerror(pvmtxt);
      } else {

            if (debugmask & PDMMESSAGE) {
                  for (i=0; i<ntids; i++) {
                        sprintf(pvmtxt, "dm_gatherc()   inst = %d\n",
                                          ga_ccap-
>cc_dsts[index].tids[i].tid);
                        pvmlogerror(pvmtxt);
                  }
            }

            /* Put data in place (indexed by inst of that data) */
            ddptr = dd;
            for (i=0; i<ntids; i++) {
                  ptr = (char *)ga_data.data + cc_datasize*ga_ccap-
>cc_dsts[index].tids[i].tid;
                  BCOPY((char *)ddptr, (char *)ptr, cc_datasize);
                  ddptr = (char *)ddptr + cc_datasize;
                  ga_data.ndata++;
            }
      }


      if (debugmask & PDMMESSAGE) {
            sprintf(pvmtxt, "dm_gatherc() from t%x cc_datasize = %d,
ntids = %d\n",
                              mp->m_src, cc_datasize, ntids);
            pvmlogerror(pvmtxt);
      }


      if (debugmask & PDMMESSAGE) {
            int i, inst;
            char *ptr2;
            char  buf[512];

            sprintf(pvmtxt, "dm_gatherc() list ga_data\n");
            pvmlogerror(pvmtxt);
            sprintf(pvmtxt, "ndata = %d\n", ga_data.ndata);
```

```
            pvmlogerror(pvmtxt);
            ptr = ga_data.data;
            for (i=0; i<ga_data.ndata; i++) {
                  BCOPY((char *)ptr, (char *)buf, cc_datasize);
                  ptr = (char *)ptr + cc_datasize;
                  sprintf(pvmtxt, "     {%d, %c}\n", inst, buf);
                  pvmlogerror(pvmtxt);
            }
      }

      --child_count;

      if (child_count <= 0) { /* get all data from children and
itself */
                  /* send data to roottid via shmem */

                  if ( (shmptr = shmat(shmid, (char *) 0, 0)) ==
(char *) -1 ) {
                        sprintf(pvmtxt, "dm_gatherc(): shmat()
error\n");
                        pvmlogerror(pvmtxt);
                        return -1;
                  }

                  /* going to write all data for root task */
                  if ((cc = semop(semid3, &wait_write[0], 1)) < 0) {
                        sprintf(pvmtxt, "dm_gatherc():
shmop(wait_write) error\n");
                        pvmlogerror(pvmtxt);
                        return -1;
                  }
                  ptr = shmptr;
                  dptr = ga_data.data;
                  BCOPY((char *)dptr, (char *)ptr,
cc_datasize*ga_data.ndata);

                  if ((cc = semop(semid3, &sig_read[0], 1)) < 0) {
                        sprintf(pvmtxt, "dm_gatherc():
semop(sig_read) error\n");
                        pvmlogerror(pvmtxt);
                        return -1;
                  }

                  if ((cc = shmdt(shmptr)) < 0) {
                        sprintf(pvmtxt, "dm_gatherc(): shmdt()
error\n");
                        pvmlogerror(pvmtxt);
                        return -1;
                  }

                  if (ga_data.data) {
                        PVM_FREE(ga_data.data);
                        ga_data.data = 0;
                  }
```

```
                if (ga_ccap) {
                        cca_free(ga_ccap);
                        ga_ccap=0;
                }
        }

done:
        if (dd)
                PVM_FREE(dd);
}


int gatherd(ccap, index)
        struct cca   *ccap;
        int  index;                     /* index of ccap->cc_dsts[] */
{
        int   i, cc;
        int   dst;
        int   size;
        struct mesg  *mp;
        char *shmptr = 0;
        char *ptr;
        int   semval = -1;
        struct sembuf wait_init[1] = {
                        0, 0, 0 };              /* wait for sem#0
become 0 */
        struct sembuf sig_init[1] = {
                        0, 1, 0 };              /* increase value of
sem#0 by 1 */

        /* create or attach to shared memory */
        if (ccap->cc_ndst) {
                if ( (shmptr = shmat(shmid, (char *) 0, 0)) == (char *) -
1 ) {
                        sprintf(pvmtxt, "gatherd(): shmat() error\n");
                        pvmlogerror(pvmtxt);
                        cc = -1;
                        goto done;
                }


                /* Wait for sem#1 is 0 */
                if ((cc = semop(semid3, &wait_init[0], 1)) < 0) {
                        sprintf(pvmtxt, "gatherd(): semop() error\n");
                        pvmlogerror(pvmtxt);
                        goto done;
                }

                /* Initialize header of shmem to 0 and ntids
                 *   +-------+
                 *   |   0   |          <-- shmptr
                 *   +-------+
                 *   | ntids |
                 *   +-------+
```

```
             *   |  ...  |          <-- data start here
             */

            ptr = shmptr;
            size = sizeof(int);
            cc = 0;
            BCOPY((char *)&cc, (char *)ptr, size);
            cc = ccap->cc_dsts[index].ntids;
            ptr = (char *)ptr + size;
            BCOPY((char *)&cc, (char *)ptr, size);

            /* signal sem#0 by 1 */
            if ((cc = semop(semid3, &sig_init[0], 1)) < 0) {
                  sprintf(pvmtxt, "gatherd(): semop() error\n");
                  pvmlogerror(pvmtxt);
                  goto done;
            }

            if ((cc = shmdt(shmptr)) < 0) {
                  sprintf(pvmtxt, "gatherd(): shmdt() error\n");
                  pvmlogerror(pvmtxt);
                  goto done;
            }
      }

      cc = 0;

done:

      if (cc < 0) {
            sprintf(pvmtxt, "gatherd() fail. returning %d\n", cc);
            pvmlogerror(pvmtxt);

      }

      return cc;

}

/*=================================================================*/


int
dm_reducea(hp, mp)
      struct hostd *hp;
      struct mesg *mp;
{
      struct cca *ccap;
      int i, j;
      int index = -1;
      int root;
      int tid_count = 0;

      /* keep to reply to */
```

```
        parent_tid = mp->m_src;
        child_count = 0;

        /* unpack struct cca from message */

        ccap = cca_new();
        upkint(mp, &ccap->cc_tid);
        upkint(mp, &ccap->cc_type);
        upkint(mp, &ccap->cc_datasize);
        upkint(mp, &ccap->cc_msgtag);
        upkint(mp, &ccap->cc_roottid);
        upkint(mp, &ccap->cc_ndst);
        ccap->cc_dsts = TALLOC(ccap->cc_ndst, struct cc_elem,
"dm_gathera");
        for (i = 0; i < ccap->cc_ndst; i++) {
                upkint(mp, &ccap->cc_dsts[i].pvmdtid);
                if ((ccap->cc_dsts[i].pvmdtid | TIDPVMD) == (pvmmytid |
TIDPVMD))
                        index = i;
                if ((ccap->cc_dsts[i].pvmdtid | TIDPVMD) ==
                                ((ccap->cc_roottid & TIDHOST) | TIDPVMD))
                        root = i;

                upkint(mp, &ccap->cc_dsts[i].ntids);
                ccap->cc_dsts[i].tids = 0;
                tid_count += ccap->cc_dsts[i].ntids;      /* for allocate
ga_data.data */
        }

        if (index == -1)  /* Not my msg, drop it */
                goto scrap;

        if ( (pvmmytid | TIDPVMD) == ((ccap-
>cc_roottid&TIDHOST)|TIDPVMD) ) {
                is_root_pvmd = 1;
                child_count = ccap->cc_ndst;

                if (debugmask & PDMMESSAGE) {
                        sprintf(pvmtxt, "dm_reducea(): This pvmd is root
pvmd, %x\n",
                                                pvmmytid|TIDPVMD);
                        pvmlogerror(pvmtxt);
                }
        }

        reduced(ccap, index);          /* Initialize shared memory for
tasks to write */

scrap:
        if (ccap)
                cca_free(ccap);                 /* Check this out */

        return 0;
}
```

```
int
dm_reducec(hp, mp)
      struct hostd *hp;
      struct mesg *mp;
{
      char *work=0;
      int   intsize;
      int   cc_type, cc_tid, cc_datasize, cc_msgtag;
      int   func, datatype, count;
      void (*reduce_func)();
      int   i, cc;
      char *shmptr = 0;
      struct sembuf wait_write[1] = {
                        1, 0, 0 };              /* wait for sem#1
become 0 */
      struct sembuf sig_read[1] = {
                        1, 1, 0 };              /* increase value of
sem#1 by 1 */

      intsize = sizeof(int);

      upkint(mp, &cc_type);
      upkint(mp, &cc_datasize);
      upkint(mp, &cc_msgtag);
      upkint(mp, &datatype);
      upkint(mp, &count);
      upkint(mp, &func);
      work = TALLOC(cc_datasize, char, "dm_reducec - work");
      byteupk(mp, (char *)work, cc_datasize, 1, 1);

      if (!re_data) {
            re_data = TALLOC(cc_datasize, char, "re_data");
            BCOPY((char *)work, (char *)re_data, cc_datasize);
      }
      else {
            switch (func) {
                  case REDUCESUM:   reduce_func = PvmSum;
                                          break;
                  default:          sprintf(pvmtxt, "dm_reducec()
[%d] function unknown\n",
                                                      func);
                                    pvmlogerror(pvmtxt);
                                    cc = -1;
                                    goto done;
            }
            (*reduce_func)(&datatype, re_data, work, &count, &cc);
      }


      --child_count;
```

```
      if (child_count <= 0) { /* get all data from children and
itself */
                  /* send data to roottid via shmem */

                  if ( (shmptr = shmat(shmid, (char *) 0, 0)) ==
(char *) -1 ) {
                        sprintf(pvmtxt, "dm_gatherc(): shmat()
error\n");
                        pvmlogerror(pvmtxt);
                        return -1;
                  }

                  /* going to write final result for root task */
                  if ((cc = semop(semid4, &wait_write[0], 1)) < 0) {
                        sprintf(pvmtxt, "dm_reducec():
shmop(wait_write) error\n");
                        pvmlogerror(pvmtxt);
                        return -1;
                  }
                  BCOPY((char *)re_data, (char *)shmptr,
cc_datasize);

                  if ((cc = semop(semid4, &sig_read[0], 1)) < 0) {
                        sprintf(pvmtxt, "dm_reducec():
semop(sig_read) error\n");
                        pvmlogerror(pvmtxt);
                        return -1;
                  }

                  if ((cc = shmdt(shmptr)) < 0) {
                        sprintf(pvmtxt, "dm_reducec(): shmdt()
error\n");
                        pvmlogerror(pvmtxt);
                        return -1;
                  }

                  if (re_data) {
                        PVM_FREE(re_data);
                        re_data = 0;
                  }
      }

done:
      if (work)
            PVM_FREE(work);
}


int reduced(ccap, index)
      struct cca   *ccap;
      int  index;                  /* index of ccap->cc_dsts[] */
{
      int   i, cc;
      int   dst;
```

```
        int   size;
        struct mesg  *mp;
        char *shmptr = 0;
        char *ptr;
        int   semval = -1;
        struct sembuf wait_init[1] = {
                        0, 0, 0 };              /* wait for sem#0
become 0 */
        struct sembuf sig_init[1] = {
                        0, 1, 0 };              /* increase value of
sem#0 by 1 */

        /* create or attach to shared memory */
        if (ccap->cc_ndst) {
            if ( (shmptr = shmat(shmid, (char *) 0, 0)) == (char *) -
1 ) {
                sprintf(pvmtxt, "gatherd(): shmat() error\n");
                pvmlogerror(pvmtxt);
                cc = -1;
                goto done;
            }


            /* Wait for sem#1 is 0 */
            if ((cc = semop(semid4, &wait_init[0], 1)) < 0) {
                sprintf(pvmtxt, "reduced(): semop() error\n");
                pvmlogerror(pvmtxt);
                goto done;
            }

            /* Initialize header of shmem to 0 and ntids
             *   +-------+
             *   |   0   |          <-- shmptr
             *   +-------+
             *   | ntids |
             *   +-------+
             *   |  ...  |          <-- data start here
             */

            ptr = shmptr;
            size = sizeof(int);
            cc = 0;
            BCOPY((char *)&cc, (char *)ptr, size);
            cc = ccap->cc_dsts[index].ntids;
            ptr = (char *)ptr + size;
            BCOPY((char *)&cc, (char *)ptr, size);

            /* signal sem#0 by 1 */
            if ((cc = semop(semid4, &sig_init[0], 1)) < 0) {
                sprintf(pvmtxt, "reduced(): semop() error\n");
                pvmlogerror(pvmtxt);
                goto done;
            }
```

```
            if ((cc = shmdt(shmptr)) < 0) {
                  sprintf(pvmtxt, "reduced(): shmdt() error\n");
                  pvmlogerror(pvmtxt);
                  goto done;
            }
      }

      cc = 0;

done:

      if (cc < 0) {
            sprintf(pvmtxt, "reduced() fail. returning %d\n", cc);
            pvmlogerror(pvmtxt);

      }

      return cc;

}
```

Appendix D

Source Code of Library Functions for Task

```
/*
 * pvm_bcast()
 *
 * Written by:  Chirapol Mathawaphan
 * Description: Use shared memory to transfer data between task and
 *              local pvmd.
*/

int
pvm_bcast(result, data, count, datatype, msgtag, gname, rootinst)
void *result, *data;
int  count, datatype, msgtag, rootinst;
char *gname;
{
      int   pvmtidlmask = TIDLOCAL;
      int   pvmtidhmask = TIDHOST;
      int   roottid, myginst;
      int   datasize, gsize, ntids, *tids = 0;
      int   i, cc;
      int   sbuf, rbuf;
      int   hdr_size, shm_size;
      int   shmid = -1;             /* shmem id */
      long  shm_key;
      char *shmptr = NULL;
      long  sem_key;
      int   semid = -1;             /* semaphore id */
      struct sembuf r_empty[1] = {
                        0, 0, 0 };              /* wait for sem#0
become 0 */
      struct sembuf r_send[1] = {
                        0, 1, 0 };  /* increase value of sem#0 by 1
*/
      struct sembuf c_recv[1] = {
                        0, -1, 0 }; /* wait for sem#0 >= 1 and
decrement by 1 */

      if ( (result == NULL) || (count <= 0) )   {     /* check some
parameters */
            cc = PvmBadParam;
            goto done;
      }

      /* root must be member of the group */
      if ( (roottid = pvm_gettid(gname, rootinst)) < 0 )
            goto done;

      /* get data size */
      if ( (cc = datasize = gs_get_datasize(datatype)) < 0 )
            goto done;

      /* calculate the value of message queue key */
      shm_key = (long) SHM_BCAST;
      sem_key = SEM_KEY1;
```

```
      /* calculate size of the shared memory */
      hdr_size = sizeof(int);       /* header size */
      shm_size = 0;

      /* I'm the root node for the bcast operation */
      if ( pvmmytid == roottid ) {

            if ( data == NULL ) {  /* check data parameter */
                  cc = PvmBadParam;
                  goto done;
            }

            /* get the list of tids */
            if ( (cc = gs_get_tidlist(gname, msgtag, &gsize, &tids,
1)) < 0 )
                  goto done;

            ntids = gsize;
            for (i=0; i<gsize; i++) {
                  if (i == rootinst) {
                        /* copy to  root itself */
                        BCOPY( (char *) data, (char *)result,
datasize*count );

                        /* mark inst as sent */
                        tids[i] = tids[--ntids];  /* remove root from
the list */
                        break;
                  }
            } /* end for-loop */

            /* copy tid-list to shmem for pvmd */
            if (shmid == -1) {
                  if ( (cc = shmid = shmget(shm_key, shm_size,
SHM_PERMS | IPC_CREAT)) < 0) {
                        perror("shmget()");
                        goto done;
                  }
                  /* attach shmem */
                  if ((shmptr=(char *)shmat(shmid, (char *)0, 0)) ==
(char *)-1)                   {
                        perror("shmat()");
                        goto done;
                  }
            }

            BCOPY((char *) tids, (char *)shmptr, ntids*sizeof(int));

            if (shmdt(shmptr) < 0) {
                  perror("shmdt()");
                  goto done;
            }

            /* send to all receivers */
```

```
            if ((cc = lpvm_bcast(data, count, datatype, msgtag,
roottid, tids, ntids)) > 0)
                  cc = 0;
      }
      else { /* other tasks receives data from shmem, prepared by
pvmd */
            if (shmid == -1) {            /* then get message queue
id */
                  if ( (cc = shmid = shmget(shm_key, shm_size,
SHM_PERMS | IPC_CREAT)) < 0) {
                        perror("shmget()");
                        goto done;
                  }
                  /* attach shmem */
                  if ((shmptr = (char *) shmat(shmid, (char *)0, 0))
== (char *)-1) {
                        perror("shmat()");
                        goto done;
                  }
            }


            if (semid == -1)  /* and get sem id */
                  if ( (cc = semid = semget(sem_key, 1, SEM_PERMS |
IPC_CREAT)) < 0 ) goto done;

            if (semop(semid, &c_recv[0], 1) != 0) {   /* wait for
access given by root */
                  perror("semop() - c_recv");
                  goto done;
            }
            BCOPY((char *) (shmptr+hdr_size), (char *) result,
datasize*count);

            if (shmdt(shmptr) < 0) {
                  perror("shmdt()");
                  goto done;
            }
      }

      cc = PvmOk;

done:
      if (tids)  PVM_FREE(tids);

      if (cc < 0) lpvmerr("pvm_bcast", cc);

      return(cc);
}


int
lpvm_bcast(data, count, datatype, msgtag, roottid, tids, ntids)
      void *data;
```

```
        int count, datatype, msgtag, roottid;
        int *tids;  /* dest tasks */
        int ntids;  /* number of tids */
{
        int cc;                 /* for return code */
        int ret;          /* return code of operation from pvmd */
        int i;
        int datasize;
        int sbf, rbf;
        static struct timeval ztv = { 0, 0 };

        if (!cc && ntids > 0) {
                datasize = count * get_datasize(datatype);

                /* send whole data to pvmd with TM_BCAST message */

                sbf = pvm_setsbuf(pvm_mkbuf(PvmDataFoo));
                rbf = pvm_setrbuf(0);

                /* pack data */
                pvm_pkint(&datasize, 1, 1);
                pvm_pkbyte((char *)data, datasize, 1);
                pvm_pkint(&msgtag, 1, 1);
                pvm_pkint(&roottid, 1, 1);
                pvm_pkint(&ntids, 1, 1);

                cc = mroute(pvmsbufmid, TIDPVMD, TM_BCASTA, &ztv);

                pvm_freebuf(pvm_setsbuf(sbf));

        }

        if (cc < 0)
                lpvmerr("lpvm_bcast", cc);
        return cc;
}


/*=============================================================*/


int
pvm_scatter(result, data, count, datatype, msgtag, gname, rootinst)
void *result, *data;
int  count, datatype, msgtag, rootinst;
char *gname;
{
    int   roottid, myginst, datasize, gsize, *tids = 0, i, cc;
        int   shmid = -1;               /* shmem id */
        long  shm_key;
        char *shmptr = NULL;
        int   hdr_size, shm_size;
        long  sem_key;
        int   semid = -1;               /* semaphore id */
```

```
      char *ptr;
      int   n, index;
      struct sembuf r_empty[1] = {
                        0, 0, 0 };              /* wait for sem#0
become 0 */
      struct sembuf r_send[1] = {
                        0, 1, 0 };  /* increase value of sem#0 by 1
*/
      struct sembuf c_recv[1] = {
                        0, -1, 0 }; /* wait for sem#0 >= 1 and
decrement by 1 */

    if ( (result == NULL) || (count <= 0) ) /* check some parameters
*/
    {
        cc = PvmBadParam;
        goto done;
    }

    /* root must be member of the group */
    if ( (roottid = pvm_gettid(gname, rootinst)) < 0 )
    {
        cc = roottid;
        goto done;
    }

    /* get the number of bytes per element of type datatype */
    if ( (cc = datasize = gs_get_datasize(datatype)) < 0  ) goto
done;

      /* calculate the value of message queue key */
      shm_key = (long) SHM_BCAST;
      sem_key = SEM_KEY2;

      /* calculate size of the shared memory */
      hdr_size = sizeof(int);       /* header size */
      shm_size = 0;

    /* everyone receives a result from the shared memory include the
root */

      if (shmid == -1) {            /* then get message queue id */
            if ( (cc = shmid = shmget(shm_key, shm_size, SHM_PERMS |
IPC_CREAT)) < 0) {
                  perror("shmget()");
                  goto done;
            }
            /* attach shmem */
            if ((shmptr = (char *) shmat(shmid, (char *)0, 0)) ==
(char *)-1) {
                  perror("shmat()");
                  goto done;
            }
      }
```

```
    /* I am the root node for the scatter operation */
    if (pvmmytid == roottid)
    {
        if ( data == NULL) /* check data parameter */
        {
            cc = PvmBadParam;
            goto done;
        }

        /* Get the list of tids.  These must be contiguous (no
holes). */
        if ( (cc = gs_get_tidlist(gname, msgtag, &gsize, &tids, 1)) <
0)
            goto done;

            ptr = shmptr;
            BCOPY((char *)tids, (char *)ptr, gsize*sizeof(int));
            ptr = (char *)ptr + gsize*sizeof(int);
            BCOPY((char *)data, (char *)ptr, datasize*count*gsize);

            if ( (cc = lpvm_scatter(data, count, datatype, msgtag,
roottid, tids, gsize)) >= 0 )
                  cc = 0;
            else
                  goto done;

    }

      if (semid == -1)  /* and get sem id */
            if ( (cc = semid = semget(sem_key, 1, SEM_PERMS |
IPC_CREAT)) < 0 ) goto done;

      if (semop(semid, &c_recv[0], 1) != 0) {   /* wait for access
given by root */
            perror("semop() - c_recv");
            goto done;
      }


      /* find tid in shmem */
      ptr = shmptr;
      n = *((int *)ptr);
      ptr += sizeof(int);

      if ( (index = find_tid(ptr, 0, n-1, pvmmytid)) < 0 ) {
            sprintf(pvmtxt, "pvm_scatter: can't find tid in
shmem.\n");
            pvmgslogerror(pvmtxt);
            goto done;
      }
```

```
      /* copy the data out */


      ptr=(char *)
shmptr+(sizeof(int)+n*sizeof(int)+index*datasize*count);
      BCOPY((char *) ptr, (char *) result, datasize*count);

      if (shmdt(shmptr) < 0) {
            perror("shmdt()");
            goto done;
      }

    cc = PvmOk;

done:
    if (tids) free(tids);

    if (cc < 0) lpvmerr("pvm_scatter",cc);

    return(cc);

}  /* end pvm_scatter() */


int
lpvm_scatter(data, count, datatype, msgtag, roottid, tids, ntids)
      void *data;
      int count, datatype, msgtag, roottid;
      int *tids;  /* dest tasks */
      int ntids;  /* number of tids */
{
      int cc;                 /* for return code */
      int ret;          /* return code of operation from pvmd */
      int i;
      int datasize;
      int sbf, rbf;
      static struct timeval ztv = { 0, 0 };

      if (!cc && ntids > 0) {
            datasize = count * get_datasize(datatype);

            /* send whole data to pvmd with TM_SCATTER message */

            sbf = pvm_setsbuf(pvm_mkbuf(PvmDataFoo));
            rbf = pvm_setrbuf(0);

            /* pack data */
            pvm_pkint(&datasize, 1, 1);
            pvm_pkint(&ntids, 1, 1);
            pvm_pkint(&msgtag, 1, 1);
            pvm_pkint(&roottid, 1, 1);

            cc = mroute(pvmsbufmid, TIDPVMD, TM_SCATTERA, &ztv);
```

```
            pvm_freebuf(pvm_setsbuf(sbf));

        }

        if (cc < 0)
            lpvmerr("lpvm_scatter", cc);
        return cc;
}

/*=============================================================*/


int
pvm_gather(result, data, count, datatype, msgtag, gname, rootinst)
void *result, *data;
int  count, datatype, msgtag, rootinst;
char *gname;
{
   int   roottid, myginst, datasize, gsize, *tids = 0, i, cc;
      int   shmid = -1;             /* shmem id */
      long  shm_key;
      char *shmptr = NULL;
      int   hdr_size, shm_size;
      long  sem_key;
      int   semid3 = -1;                  /* semaphore id */
      char *ptr;
      int   n1, n2;
      int   intsize, size;

      struct sembuf wait_head[1] = {
                        0, -1, 0 }; /* wait for sem#0 become 1 and
decrement by 1 */
      struct sembuf sig_head[1] = {
                        0, 1, 0 };  /* increase value of sem#0 by 1
*/
      struct sembuf wait_read[1] = {
                        1, -1, 0 }; /* wait for sem#1 become 1 and
decrement by 1 */
      int   semval = -1;

    if ( (data == NULL) || (count <= 0) ) /* check some parameters */
    {
        cc = PvmBadParam;
        goto done;
    }

    /* root must be member of the group */
      if ( (roottid = pvm_gettid(gname, rootinst)) < 0 )
      {
            cc = roottid;
            goto done;
      }

    /* get the number of bytes per element of type datatype */
```

```
    if ( (cc = datasize = gs_get_datasize(datatype)) < 0  ) goto
done;

      /* calculate the value of message queue key */
      shm_key = (long) SHM_BCAST;
      sem_key = SEM_KEY3;

      /* calculate size of the shared memory */
      intsize = sizeof(int);
      hdr_size = intsize*2;         /* header size */
      shm_size = 0;

      if ( (cc = shmid = shmget(shm_key, shm_size, 0)) < 0) {
            perror("shmget()");
            goto done;
      }
      /* attach shmem */
      if ((shmptr = (char *) shmat(shmid, (char *)0, 0)) == (char *)-
1) {
            perror("shmat()");
            goto done;
      }

    /* I am the root node for the gather operation */
    if (pvmmytid == roottid)
    {
        if ( result == NULL) /* check data parameter */
        {
            cc = PvmBadParam;
            goto done;
        }

        /* Get the list of tids.  These must be contiguous (no
holes). */
        if ( (cc = gs_get_tidlist(gname, msgtag, &gsize, &tids, 1)) <
0)
            goto done;

            /* root task copy tid-list to shm for pvmd */
            BCOPY((char *)tids, (char *)shmptr, gsize*sizeof(int));

            if ( (cc = lpvm_gather(count, datatype, msgtag, roottid,
tids, gsize)) >= 0 )
                  cc = 0;
            else
                  goto done;

    }

    /* everyone write data to the shared memory include the root for
pvmd */

      if ( (cc = semid3 = semget(SEM_KEY3, 2, 0)) < 0 )
            goto done;
```

```
      if (semop(semid3, &wait_head[0], 1) != 0) {     /* wait for
access to header of shmem */
            perror("semop() - wait_head");
            goto done;
      }

      /* update header in shmem */
      /* increase header[0] */
      ptr = shmptr;
      n1 = *((int *)ptr);
      n1++;
      BCOPY((char *)&n1, (char *)ptr, intsize);

      /* decrease header[1] */
      ptr = (char *)ptr + intsize;
      n2 = *((int *)ptr);
      n2--;
      BCOPY((char *)&n2, (char *)ptr, intsize);


      if (semop(semid3, &sig_head[0], 1) != 0) {      /* give access
to another task */
            perror("semop() - sig_head");
            goto done;
      }

      ptr = shmptr;
      size = intsize + datasize*count;
      ptr = (char *)ptr + hdr_size + size*(n1-1);

      BCOPY((char *) &pvmmytid, (char *) ptr, intsize);          /*
write tid */
      ptr = (char *)ptr + intsize;
      BCOPY((char *) data, (char *) ptr, datasize*count);        /*
write data */

      if (n2 == 0)      /* all data wrote in shmem */
            lpvm_gatherb(count, datatype, msgtag);

      /*
       * Root read all gathered data from shared memory.
       */
      if (pvmmytid == roottid) {
            /* if root, read all result from shmem */
            if ((cc = semop(semid3, &wait_read[0], 1)) < 0) {
                  perror("semop(wait_read)");
                  goto done;
            }

            BCOPY((char *)shmptr, (char *)result,
gsize*count*datasize);
      }
```

```
    cc = PvmOk;

done:

      if (shmdt(shmptr) < 0) {
            perror("shmdt()");
            goto done;
      }

    if (tids) free(tids);

    if (cc < 0) lpvmerr("pvm_gather",cc);

    return(cc);

}  /* end pvm_gather() */


int
lpvm_gather(count, datatype, msgtag, roottid, tids, ntids)
      int count, datatype, msgtag, roottid;
      int *tids;  /* dest tasks */
      int ntids;  /* number of tids */
{
      int cc;                  /* for return code */
      int ret;          /* return code of operation from pvmd */
      int i;
      int datasize;
      int sbf, rbf;
      static struct timeval ztv = { 0, 0 };

      if (!cc && ntids > 0) {
            datasize = count * get_datasize(datatype);

            /* send whole data to pvmd with TM_GATHER message */

            sbf = pvm_setsbuf(pvm_mkbuf(PvmDataFoo));
            rbf = pvm_setrbuf(0);

            /* pack data */
            pvm_pkint(&datasize, 1, 1);
            pvm_pkint(&ntids, 1, 1);
            pvm_pkint(&msgtag, 1, 1);
            pvm_pkint(&roottid, 1, 1);

            cc = mroute(pvmsbufmid, TIDPVMD, TM_GATHERA, &ztv);
            pvm_freebuf(pvm_setsbuf(sbf));

      }

      if (cc < 0)
            lpvmerr("lpvm_gather", cc);
      return 0;
}
```

```
int
lpvm_gatherb(count, datatype, msgtag)
int  count, datatype, msgtag;
{
      int  sbf;
      int  datasize;
      int  cc;
      int  cc_type = CC_GATHER;
      static struct timeval ztv = { 0, 0 };

      datasize = count * get_datasize(datatype);

      sbf = pvm_setsbuf(pvm_mkbuf(PvmDataFoo));

      /* pack data */
      pvm_pkint(&cc_type, 1, 1);
      pvm_pkint(&datasize, 1, 1);
      pvm_pkint(&msgtag, 1, 1);
      cc = mroute(pvmsbufmid, TIDPVMD, TM_GATHERB, &ztv);
      pvm_freebuf(pvm_setsbuf(sbf));
      return 0;
}


/*=================================================================*/


int pvm_reduce(func, data, count, datatype, msgtag, gname, rootinst)
    int  func;
    void *data;
    int count, datatype, msgtag, rootinst;
    char *gname;
{
   int   roottid, myginst, datasize, gsize, *tids = 0, i, cc;
      int   shmid = -1;             /* shmem id */
      long  shm_key;
      char *shmptr = NULL;
      int   hdr_size, shm_size;
      long  sem_key;
      int   semid4 = -1;                  /* semaphore id */
      char *ptr;
      int   n1, n2;
      int   intsize, size;

      struct sembuf wait_head[1] = {
                        0, -1, 0 }; /* wait for sem#0 become 1 and
decrement by 1 */
      struct sembuf sig_head[1] = {
                        0, 1, 0 };  /* increase value of sem#0 by 1
*/
      struct sembuf wait_read[1] = {
                        1, -1, 0 }; /* wait for sem#1 become 1 and
decrement by 1 */
```

```
      int   semval = -1;

    if ( (data == NULL) || (count <= 0) ) /* check some parameters */
    {
        cc = PvmBadParam;
        goto done;
    }

   /* root must be member of the group */
      if ( (roottid = pvm_gettid(gname, rootinst)) < 0 )
      {
            cc = roottid;
            goto done;
      }

    /* get the number of bytes per element of type datatype */
    if ( (cc = datasize = gs_get_datasize(datatype)) < 0  ) goto
done;

      /* calculate the value of message queue key */
      shm_key = (long) SHM_BCAST;
      sem_key = SEM_KEY4;

      /* calculate size of the shared memory */
      intsize = sizeof(int);
      hdr_size = intsize*2;         /* header size */
      shm_size = 0;


      if ( (cc = shmid = shmget(shm_key, shm_size, 0)) < 0) {
            perror("shmget()");
            goto done;
      }
      /* attach shmem */
      if ((shmptr = (char *) shmat(shmid, (char *)0, 0)) == (char *)-
1) {
            perror("shmat()");
            goto done;
      }

    /* I am the root node for the reduce operation    */
      /* Send request to local pvmd (root pvmd)       */
    if (pvmmytid == roottid)
    {
        /* Get the list of tids.  These must be contiguous (no
holes). */
        if ( (cc = gs_get_tidlist(gname, msgtag, &gsize, &tids, 1)) <
0)
            goto done;

            /* root task copy tid-list to shm for pvmd */
            BCOPY((char *)tids, (char *)shmptr, gsize*sizeof(int));
```

```
            if ( (cc = lpvm_reduce(count, datatype, msgtag, roottid,
gsize)) >= 0 )
                  cc = 0;
            else
                  goto done;

}

    /* everyone write data to the shared memory include the root for
pvmd */
#ifdef DEBUGCCL
      sprintf(pvmtxt, "pvm_reduce: ready to write, shmid = %d\n",
shmid);
      pvmgslogerror(pvmtxt);
#endif

      if ( (cc = semid4 = semget(SEM_KEY4, 2, 0)) < 0 )
            goto done;

      if (semop(semid4, &wait_head[0], 1) != 0) {     /* wait for
access to header of shmem */
            perror("semop() - wait_head");
            goto done;
      }

      /* update header in shmem */
      /* increase header[0] */
      ptr = shmptr;
      n1 = *((int *)ptr);
      n1++;
      BCOPY((char *)&n1, (char *)ptr, intsize);

      /* decrease header[1] */
      ptr = (char *)ptr + intsize;
      n2 = *((int *)ptr);
      n2--;
      BCOPY((char *)&n2, (char *)ptr, intsize);

      if (semop(semid4, &sig_head[0], 1) != 0) {      /* give access
to another task */
            perror("semop() - sig_head");
            goto done;
      }

      ptr = shmptr;
      size = datasize*count;
      ptr = (char *)ptr + hdr_size + size*(n1-1);

      BCOPY((char *) data, (char *) ptr, datasize*count);         /*
write data */

      if (n2 == 0)      /* all data wrote in shmem */
            lpvm_reduceb(func, count, datatype, msgtag);
```

```
        /*
         * Root read final result from shared memory.
         */
        if (pvmmytid == roottid) {
              /* if root, read final result from shmem */
              if ((cc = semop(semid4, &wait_read[0], 1)) < 0) {
                    perror("semop(wait_read)");
                    goto done;
              }

              BCOPY((char *)shmptr, (char *)data, count*datasize);
        }

    cc = PvmOk;


        if (shmdt(shmptr) < 0) {
              perror("shmdt()");
              goto done;
        }

done:
    if (cc < 0) lpvmerr("pvm_reduce",cc);

    return(cc);

} /* end pvm_reduce() */



int
lpvm_reduce(count, datatype, msgtag, roottid, ntids)
      int count, datatype, msgtag, roottid;
      int ntids;  /* number of tids */
{
      int cc;                  /* for return code */
      int ret;           /* return code of operation from pvmd */
      int i;
      int datasize;
      int sbf, rbf;
      static struct timeval ztv = { 0, 0 };

      if (!cc && ntids > 0) {
            datasize = count * get_datasize(datatype);

            /* send whole data to pvmd with TM_GATHER message */

            sbf = pvm_setsbuf(pvm_mkbuf(PvmDataFoo));
            rbf = pvm_setrbuf(0);

            /* pack data */
            pvm_pkint(&datasize, 1, 1);
            pvm_pkint(&ntids, 1, 1);
            pvm_pkint(&msgtag, 1, 1);
```

```
            pvm_pkint(&roottid, 1, 1);

            cc = mroute(pvmsbufmid, TIDPVMD, TM_REDUCEA, &ztv);
            pvm_freebuf(pvm_setsbuf(sbf));

      }

      if (cc < 0)
            lpvmerr("lpvm_reduce", cc);
      return 0;
}

int
lpvm_reduceb(func, count, datatype, msgtag)
int  func, count, datatype, msgtag;
{
      int  sbf;
      int  datasize;
      int  cc;
      int  cc_type = CC_GATHER;
      static struct timeval ztv = { 0, 0 };

      datasize = count * get_datasize(datatype);

      sbf = pvm_setsbuf(pvm_mkbuf(PvmDataFoo));

      /* pack data */
      pvm_pkint(&cc_type, 1, 1);
      pvm_pkint(&datasize, 1, 1);
      pvm_pkint(&msgtag, 1, 1);
      pvm_pkint(&datatype, 1, 1);
      pvm_pkint(&count, 1, 1);
      pvm_pkint(&func, 1, 1);
      cc = mroute(pvmsbufmid, TIDPVMD, TM_REDUCEB, &ztv);
      pvm_freebuf(pvm_setsbuf(sbf));
      return 0;
}
```

BIBLIOGRAPHY

[1] Bala, V., Bruck, J., Cypher, R., Elustondo, P., Ho, A. Ho, C. T., Kipnis, S., and Snir, M. "CCL: A Portable and Tunable Collective Communication Library for Scalable Parallel Computers." *International Parallel Processing Symposium*, pp. 835-844, April 1994.

[2] Barnett, M., Gupta, S., Payne, D., Shuler, L., van de Geijn, R., and Watts, J. "Building a High-Performance Collective Communication Library." *Proceedings of IEEE Scalable High Performance Computing*, pp. 835-834, 1994.

[3] Barnett, M., Gupta, S., Payne, D., Shuler, L., van de Geijn, R., and Watts, J. "Interprocessor Collective Communication Library (InterCom)." *Proceedings of Scalable High Performance Computing Conference*, pp. 357-364, May 23-24, 1994.

[4] Beguelin, A., Dongarra, J., Geist, A. Mancheck B., and Sunderam, V. "Recent Enhancements to PVM." *International Journal for Supercomputer Applications*, Vol. 9, No. 2, Summer 1995.

[5] Bernaschi, M. and Richelli, G. "Development and Results of PVMe on the IBM 9076 SP1." Journal of Parallel and Distributed Computing. Vol. 29, pp. 75-83, 1995.

[6] Bruck, J., Dolev, D., Ho, C.T., Orni, R., and Strong R, "PCODE: An Efficient and Reliable Collective Communication Protocol for Unreliable Broadcast Domains." *IBM Research Report RJ 9895*, September 1994.

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

[8] Crowl, L. A. "How to Measure, Present, and Compare Parallel Performance." *IEEE Parallel & Distributed Technology*, pp. 9-25, Spring 1994.

[9] Dongarra, J., Geist, A., Mancheck, R. and Runderam, V. "Supporting Heterogeneous Network Computing: PVM." *Chemical Design Automation News*, Vol. 8, No. 9/10, pp. 36-42, September/October 1993.

[10] Dongarra, J., Geist, A., Manchek, R. and Runderam, V. "Integrated PVM Framework Supports Heterogeneous Network Computing." *Computers in Physics*, Vol. 7, No. 2, pp. 166-75, April 1993.

[11] Geist, A., Beguelin, A., Dongarra, J. Jiang, W., Manchek, R. and Sunderam V. *PVM: Parallel Virtual Machine - A User's Guide and Tutorial for Network Parallel Computing*. MIT Press, 1994.

[12] Geist, G. A. and Sunderam, V. S. "Network Based Concurrent Computing on the PVM System." *Journal of Concurrency: Practice and Experience*, 4, 4, pp. 166-175, June 1992.

[13] Lowekamp B., Beguelin, A. "ECO Efficient Collective Operations for Communication on Heterogeneous Networks." School of Computer Science, Carnegie Mellon University, February 1996.

[14] McKinley P. K., Liu, J. "Multicast tree construction in bus-based networks." *Communications of the ACM*, 33(1), pp. 29-41, January 1990.

[15] McKinley, P. K., Tsai, Y., and Robinson, D. F. "Collective Communication in Wormhole-Routed Massively Parallel Computers." *Computer*, pp. 39-50, December 1995.

[16] Mitra, P., Payne D. G., Shuler, L., van de Geijn, R., and Watts, J. "Fast Collective Communication Library, Please." Technical Report TR-95-22, The University of Texas, June 1995.

[17] Stevens, R. *Unix Network Programming*, Prentice-Hall, 1989.

[18] Stevens, R. *Advanced Unix Network Programming*, Prentice-Hall, 1991.

[19] Stevens, R., Comer, P. *Internetworking with TCP/IP, Volume I*, Prentice-Hall, 1988.

[20] Stevens, R., Comer, P. *Internetworking with TCP/IP, Volume II*, Prentice-Hall, 1990.

[21] Stevens, R., Comer, P. *Internetworking with TCP/IP, Volume III*, Prentice-Hall, 1992.

[22] Sunderam, V. S. "PVM: A Framework for Parallel Distributed Computing." *Concurrency: Practice and Experience*, 2, 4, pp. 315-339, December 1990.

[23] Sunderam, V., Dongarra, J., Geist, A. and Mancheck, R. "The PVM Concurrent Computing System: Evolution, Experiences, and Trends." *Parallel Computing*, Vol. 20, No. 4, pp. 531-547, April 1994.

[24] Zomaya, A. Y. *Parallel & Distributed Computing Handbook*. McGraw-Hill, 1996.

[25] URL http://www.epm.ornl.gov/pvm/pvm_home.html