ORNL/TM-13228

OAK RIDGE NATIONAL LABORATORY

LOCKHEED MARTIN

Collective Communication Routines in PVM

J. M. Donato
G. A. Geist

MANAGED AND OPERATED BY
LOCKHEED MARTIN ENERGY RESEARCH CORPORATION
FOR THE UNITED STATES
DEPARTMENT OF ENERGY

ORNL-27 (3-96)

ORNL/TM-13228

Computer Science and Mathematics Division

Mathematical Sciences Section

COLLECTIVE COMMUNICATION ROUTINES IN PVM

J.M. Donato and G.A. Geist

Mathematical Sciences Section
Oak Ridge National Laboratory
P.O. Box 2008, Bldg. 6010
Oak Ridge, TN 37831-6414

Date Published: May 1996

Research was supported by the Mathematical, Information, and Computational Sciences Division of the Office of Energy Research, U.S. Department of Energy.

Prepared by the
Oak Ridge National Laboratory
Oak Ridge, Tennessee 37831
managed by
Lockheed Martin Energy Research Corporation
for the
U.S. Department of Energy
under Contract No. DE-AC05-96OR22464

Contents

List of Figures

List of Tables

COLLECTIVE COMMUNICATION ROUTINES IN PVM

J.M. Donato and G.A. Geist

Abstract

The collective communication routines of scatter, gather, and reduce are frequently implemented as part of the native library for parallel architectures. These operations have been implemented in PVM for use among a heterogeneous system of workstations and parallel computers forming a virtual parallel machine. In the case of the Intel Paragon machines, the PVM implementation of the reduce operation utilizes the corresponding native mode library routines whenever possible.

This paper describes the implementation of these collective communication routines in PVM including the utilization of the Intel Paragon native mode operations. Performance data is also given comparing the use of the native Intel Paragon collective routines and the PVM implementation on top of these routines on a dedicated Intel Paragon. For our timing results an average latency of 109 μs is incurred using PVM as compared to the native Intel global sum routine. This extra startup is independent of the size of the message being sent and the number of nodes in the group. It is demonstrated that the use of static groups is preferable in time efficiency over the use of dynamic groups.

1. Introduction/Background

PVM (Parallel Virtual Machine)[4] is a widely used system for programming parallelism across a network of heterogeneous machines. This network could contain a variety of machine architectures including massively parallel processors. Collective communication routines such as scatter, gather, and reduce are frequently implemented in some form as part of the native library for parallel machines. Here, collective communication means communication that is performed across a group of tasks. Each member of the group must participate by calling the collective communication operation. Such collective communication routines and extensions thereof are defined and extended upon under MPI [6].

In PVM versions 3.3.8 and higher scatter, gather, and reduce operations are implemented for use among a heterogeneous system of workstations and parallel computers forming a virtual parallel machine. In the case of the Intel Paragon machines, the PVM implementation of the reduce operation utilizes the corresponding native mode library routines whenever possible.

This paper describes the implementation of these collective communication routines in PVM including the utilization of the Intel Paragon native mode operations. Performance data is also given comparing the use of the native Intel Paragon collective routines and the PVM implementation on top of these routines on a dedicated Intel Paragon machine. The timings were performed using PVM release 3.3.10 on the Center for Computational Science (CCS)[1], XP/S 5, Intel Paragon machine.

For our timing results an average latency of 109 μs is incurred using PVM as compared to the native Intel global sum routine. This extra startup is independent of the size of the message being sent and the number of nodes in the group. It is demonstrated that the use of static groups is preferable in time efficiency over the use of dynamic groups.

Throughout this document the phrase "static group" actually refers to a "frozen dynamic group" where each member of a dynamic group has executed a pvm_freezegroup call. True static groups are to be implemented in PVM release 3.4.0.

This paper assumes basic knowledge of the PVM software system and Intel Paragon hardware and software. For background on the Intel Paragon and its native group operations, please see [8]. The PVM Users' Guide[4] provides the general background on installation, syntax and usage of the PVM software system. For more detail on the performance of PVM on Massively Parallel systems see reference [2] which describes the basics of the communication model of PVM along with performance results for send and receive operations on Intel Paragon, SP-2 and CM-5 machines.

[1]`http://www.ccs.ornl.gov/HomePage.html`

In the sections that follow, we describe the implementation of scatter, gather, and reduce operations in the general situation of a heterogeneous network of machines. A brief overview of the PVM syntax will be given for each command. Please refer to the appendices for more complete specification of the syntax of the commands and a discussion of the their usage along with example statements.

The next two sections will briefly describe the scatter and gather operations. The Intel Paragon `NX` routines do not include native scatter and gather operations, so no specific changes have been made for the Intel Paragon in the implementation of these two routines. After scatter and gather, the reduce operation is described. Since the `NX` library does provide a number of native reduction routines, these are utilized when possible. When and how these are used will be described.

This paper is written from a C language point of view in terms of the indexing of arrays. In C, multi-index arrays are arranged contiguously in memory in row ordering with a starting index value typically 0. For Fortran multi-index (multi-dimensional) arrays are arranged contiguously in memory in column ordering with starting index typically being 1. See Figure 1 for an example of how a two-dimensional $M \times N$ array (matrix) would be laid out in memory if ordered by columns versus being ordered by rows.

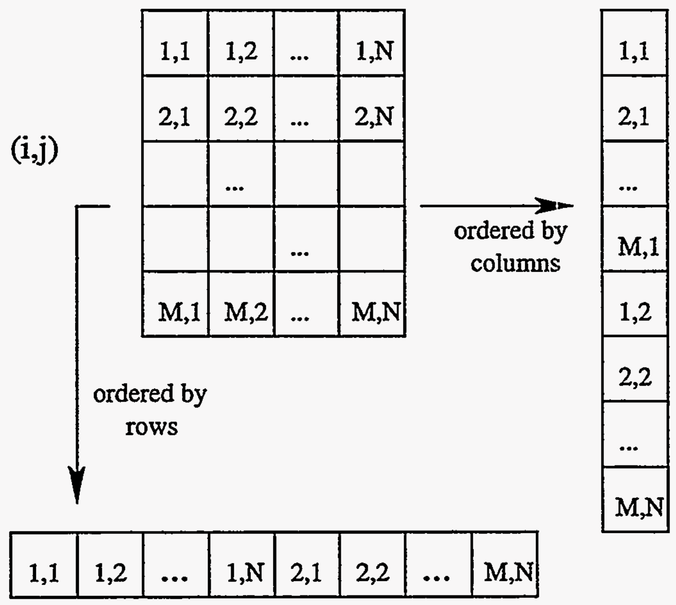

Figure 1: Row and Column Linear Orderings of a Matrix

2. Scatter

A scatter operation distributes data segments from one member of the group to the other members of the group. For example, a scatter operation can be used to disperse rows of a matrix from one task to all the members of the group in order to perform row operations in parallel.

The syntax of the scatter operation in PVM is as follows.

```
int info =
  pvm_scatter(void *result, int *data,  int count, int datatype,
              void msgtag, char *gname, int rootinst)
```

It performs a scatter of messages from the specified root member of the group to each of the other members of the group as shown in Figure 2.

Each member of the group *gname* receives a message *result* of type *datatype* and length *count* from the root member of the group. The root sends these messages from a single array *data* which is of length, at least, $M * count$. Here, M represents the number of members in the group, all of which must be participating in the scatter operation. The values sent to the i^{th} member of the group are taken from the *data* array starting at position $i * count$. The root member of the group is specified by its instance number, *rootginst*, in that group.

The message passing employed during the scatter operation in PVM is implemented using basic PVM commands, such as pvm_send and pvm_recv. The root member does not send to itself, rather it performs a memory to memory copy.

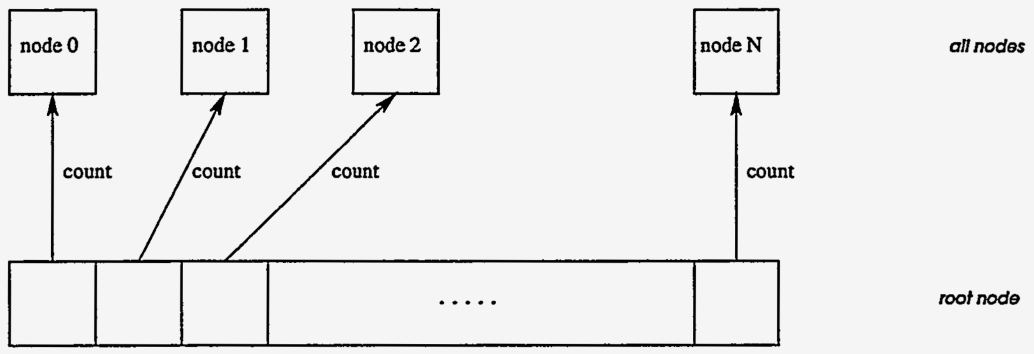

Figure 2: Scatter: The root distributes data sections to each group member

3. Gather

A gather, the inverse operation to a scatter, combines separate data segments from each group member into a single array on the root member of the operation.

The syntax of the gather operation in PVM is as follows.

```
int info =
  pvm_gather(void *result, void *data, int count, int datatype,
             int msgtag,  char *gname, int rootinst)
```

It performs a gather of messages from each member of the group to a specified member of the group. This is shown in Figure 3.

Each member of the group *gname* sends a message *data* of type *datatype* and length *count* to the root member of the group. The root receives these messages into a single array *result* which is of length, at least, $M * count$. Again, M represents the number of members in the group, all of which must be participating in the gather operation. On the root, the values received from the i^{th} member of the group are placed into the *result* array starting at position $i * count$. The root member of the group is specified by its instance number, *rootginst*, in that group.

The message communication that occurs as part of the gather operation, as with the scatter operation, is implemented using basic PVM commands, such as pvm_send and pvm_recv. Again, the root does not send or receive a message from itself, it performs a memory to memory copy.

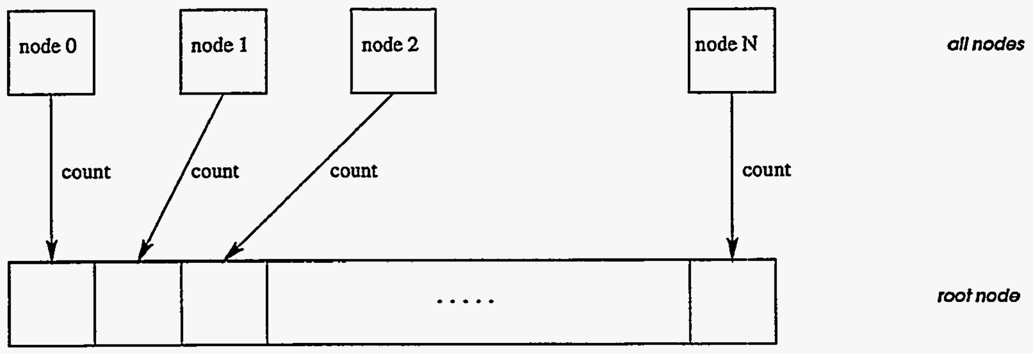

Figure 3: Gather: The root assembles data sections from each group member

4. Reduce

In a reduction operation, such as a global sum, an associative and commutative operation is performed on corresponding data segments by each member of the group. These global combine operations "reduce" the data segments from each member into one data segment on the root. For further information, see [5].

The PVM syntax for the reduce operation is as follows.

```
int info =
  pvm_reduce(void (*func)(), void *data, int count, int datatype,
             int msgtag, char *gname, int rootinst)
```

where

```
  void (*func)(int *datatype, void *data, void *work,
               int *num, int *info)
```

4.1. PVM Reduce Implementation

The current implementation uses a hierarchical fan-in algorithm to perform the reduce operation. Global min, max, sum, and product reduction operations are provided in PVM. This is done by specifying *func* as one of the PVM defined functions of PvmMin, PvmMax, PvmSum, or PvmProduct, respectively. A user written function may also be provided as the *func* argument. For predictable results, it is important that such a user-defined function be associative and commutative. See the appendix for the syntax and summary of these functions. The general heterogeneous implementation is described as follows.

The reduce operation, as with the scatter and gather operations, in PVM is implemented using basic PVM commands, such as pvm_send and pvm_recv.

For each host (a physical machine in the parallel virtual machine) a coordinator is designated for that host. During the reduce operation, each group member on a host communicates (via pvm_send) its data segment to the coordinator for that host. The coordinators on each host are then responsible for performing (combining or reducing) the specified function *func* on the data segments it has received and then communicating (via pvm_send) the result to the root member of the reduce operation. The root then performs the specified function *func* on the data received from the coordinators.

Figure 4 gives a pictorial view of the message flow from group members on a host to the coordinator on the same host and then to the root member of the reduce operation. Each host can be a multitasking multiprocessor.

4.2. Specifics of PVM Reduce on the Intel Paragon

This hierarchical fan-in technique is still used if an Intel Paragon is part of the virtual machine. However, the nodes on the Intel Paragon will utilize corresponding

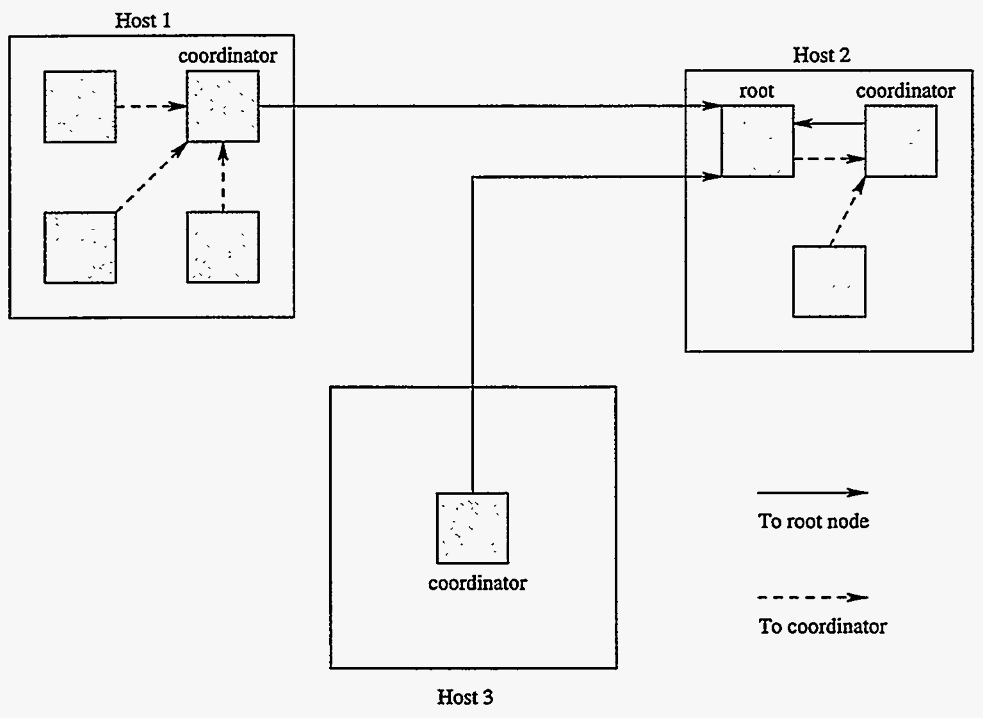

Figure 4: Message flow from group members to coordinators to the root

NX functions whenever possible. The Paragon is currently limited to executing only one PVM task per node. The PVM console and the PVM group server (pvmgs) run on the service nodes for the partition.

If all the nodes on the partition are participating in the reduce operation, then the NX function will be executed, if one exists. In this case, there is no need for the Intel Paragon nodes to explicitly send data to their coordinator node. This is because the NX collective routines return the final values to each of the nodes participating in the operation. Similarly, if all the nodes in the Paragon partition are part of a larger group, the NX native operations will be used for the Paragon part of the collective operation.

PVM determines which native mode NX routine to call by comparing the *func* function reference (e.g. pointer to the function) in the reduce call to the those functions for which an NX version exists. Currently, PVM recognizes that PvmSum, PvmMin, PvmMax, and PvmProduct which correspond to g*x*sum, g*x*min, g*x*max, g*x*prod, respectively.

For the NX native collective operations to be executed, the following two conditions must hold:

1. all the nodes in the paragon compute partition must be participating in the collective reduce operation, and

2. a corresponding NX collective operation must exist and be detected by PVM for the given *datatype* on the Intel Paragon.

If these two conditions do not hold, the collective operation still functions correctly, but will not use NX native operations. Instead, the nodes will send data to their coordinator as described in the general reduce case.

The three possible basic situations are show in Figure 5. Figure 5(a) shows via dotted lines (without arrows) that the native NX command is used and so PVM does not define the message flow. The results of the group operation will be communicated via NX to each of the nodes. Figure 5(b) shows the case where not all of the nodes of the partition are part of the group and hence the native NX function can not be used. Figure 5(c) shows the situation where, although all of the nodes in the partition are in the group, the specified *func* is a user-written function, and hence no appropriate native NX routine can be utilized.

Figure 6 gives a pictorial example of the message passing that would occur if a partition of Paragon nodes are part of the group operation. The figure shows two Intel Paragons, one with a 4 node partition allocated to PVM, and the other with a 6 node partition allocated. A third host of unspecified architecture is also pictured for variety.

Even in the case where an NX native operation is used, the overhead for the reduce operation could still be extreme if dynamic groups are being used. In the case of dynamic groups, a check must be made by each node to determine who is

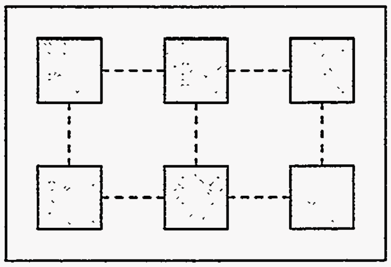
Figure 5(a) : Native NX routine is used for the 6 node partition

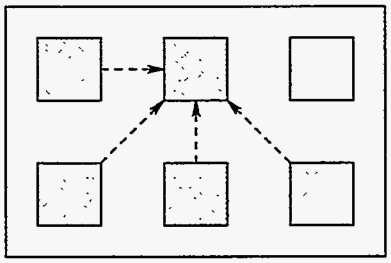
Figure 5(b) : Only 5 of the 6 nodes in the partition are in the group

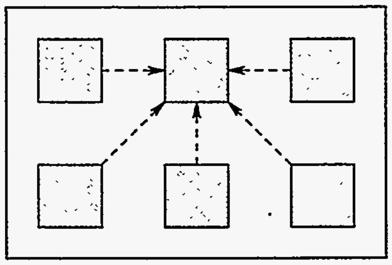
Figure 5(c) : There is no corresponding NX collective routine

Figure 5: Three cases on the Paragon

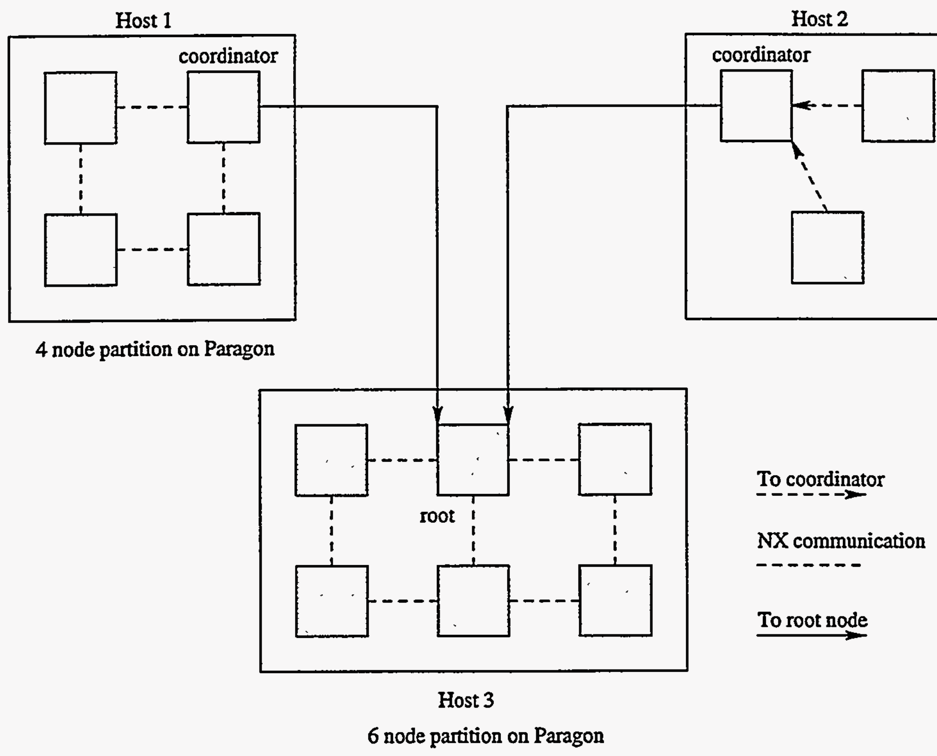

Figure 6: Message flow on a virtual machine consisting of 2 Paragons and another host architecture

part of the group operation. Hence, calls to the pvmgs are made by each member of the group.

If the user has called pvm_freezegroup, to designate that the group is static, this overhead is not incurred. The list of group members is cached to each member.

4.3. PVM and Intel Paragon notes

In this section we remind the user of notes and caveats on the use of PVM on the Intel Paragon. This section includes information from the PVM Readme.mp file that accompanied the PVM 3.3.10 release along with other useful notes. For further information and updates for new releases the reader should refer to the Readme.mp of the release of PVM being used.

- Tasks spawned onto the Intel Paragon run on the compute nodes by default. Host tasks run on the service nodes and should be started from a Unix prompt. The PVM console and group server (pvmgs) also run on the service nodes.
- By default PVM spawns tasks in your default partition. You can use the NX command-line options such as '-pn partition_name' to force it to run on a particular partition or '-sz number_of_nodes' to specify the number of nodes you want it to use. Setting the environmental variable NX_DFLT_SIZE would have the same effect. For example starting pvmd with the following command

  ```
  pvmd -pn pvm -sz 33
  ```

  would force it to run on the partition 'pvm' using only 33 nodes (there must be at least that many nodes in the partition).
- The current implementation only allows one task to be spawned on each node.
- There is a constant TIMEOUT in the file 'pvmmimd.h' that controls the frequency at which the PVM daemon probes for packets from node tasks. If you want it to respond more quickly you can reduce this value. Currently it is set to 10 millisecond.
- Be aware that mixing NX message passing calls in PVM may interfere with PVM message passing commands, such as pvm_send and pvm_recv, since the PVM system may have utilized NX message tags. This warning also applies to pvm_reduce, pvm_scatter, and pvm_gather since they are implemented using basic PVM commands.

- PVM programs compiled for versions earlier than 3.3.8 need to be recompiled. A small change in data passed to group members on startup will cause earlier programs to break.

5. Performance Measurements

5.1. Hardware Description

The timings were performed on the Center for Computational Science (CCS)[2], xps5, Intel Paragon machine.[3] At this time, the configuration of the Intel Paragon XP/S 5 consists of 70 General Purpose (GP) compute processors arranged in 10 by 7 mesh, 4 Multi-purpose (MP) compute processors in a 2 by 2 mesh, 3 service nodes, and 6 I/O nodes. Each GP compute node has 16MB of memory, while each MP compute node has 128MB of memory. Five of the I/O nodes are connected to 4.8 GB RAID disks, and the sixth to a 16 GB RAID disk. The system provides a total of 40 GB of system disk space. The system is connected to the ORNL network with an Ethernet connection and 2 HIPPI connections. Versions of release 1.3 of the Intel Paragon OS was running at the time of these performance tests.

5.2. Performance Timing Procedure

The program that produced the performance timing results is very straightforward. An integer global sum, via *gisum*() or via pvm_reduce using PvmSum, was performed for three different message lengths. The times given in the tables are an average over 100 such iterations of the *gisum*() or pvm_reduce command. Messages containing 1 integer, 1000 integers, and 10000 integers were used.

Elapsed time was used, rather than cpu time, since cpu time would not include the necessarily important wait for messages from other group members. However, initial startup overhead was not included in the timings. For example, partition allocation, pvmd startup, pvmgs startup, spawn (or pexec) of the executable, nor the first message passing cycles were included in the timings. All times are given in microseconds (μs).

All of these performance timing results were produced by execution runs performed on dedicated hosts. This was done to insure that there would be no interference from other processes being run on the hosts. This helps to produce repeatable performance results.

[2]`http://www.ccs.ornl.gov/HomePage.html`

[3]`http://www.ccs.ornl.gov/comp_resources/intel_par/5.hdwre.html`

Number of Integers	Number of Nodes 1	2	4	8	16
1	21251	46783	97420	208178	430380
1000	21147	50641	120648	247305	506010
10000	21344	117401	259629	866073	961402

Table 1: Times (μs) using Dynamic Groups on a SPARC Classic

Number of Integers	Number of Nodes 1	2	4	8	16
1	500	6947	19350	52654	90909
1000	477	12172	34005	94000	179776
10000	485	70043	178529	728407	832529

Table 2: Times (μs) using Static Groups on a SPARC Classic

5.3. Static versus Dynamic Groups

In this subsection we illustrate the importance of using static groups as opposed to dynamic groups whenever possible for group operations.

Tables 1 and 2 show the results of executing the test routine on a dedicated SPARC Classic. Table 1 gives the times for the test when dynamic groups are being used. Table 2 gives the times for the test when static groups are used. The timing differences are enormous, some as much as two orders of magnitude slower for dynamic groups as compared to the analogous static group timing. Figure 7 displays these results for comparison on a semi-log plot.

On the Intel Paragon, the difference in timings using dynamic and static groups is even more staggering. Tables 3 and 4 show the performance results using dynamic and static groups, respectively. The timings using dynamic groups are typically three orders of magnitude higher than those for static groups.

From these tables of results, both on the SPARC Classic and on the Intel Paragon, it is obvious that the efficient use of the collective communication routines in PVM relies upon using static groups directly (as will be available in PVM release 3.4.0) or by freezing a dynamic group via the pvm_freezegroup operation.

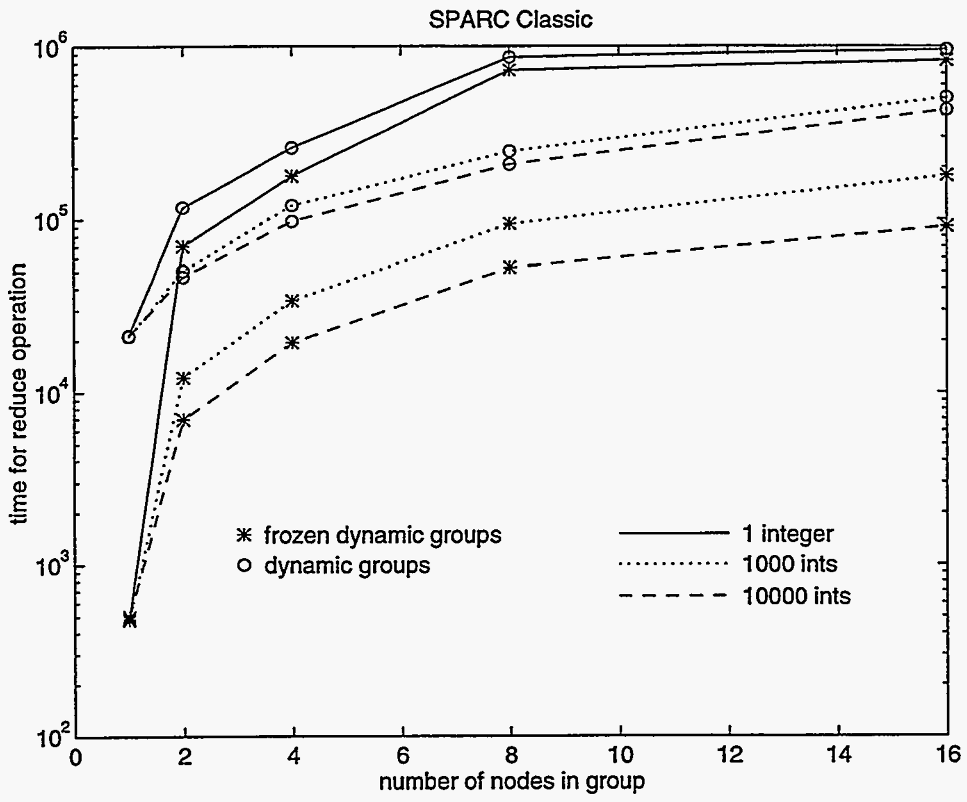

Figure 7: Times (μs) for Dynamic and Static Groups on a SPARC Classic

Number of Integers	Number of Nodes 1	2	4	8	16
1	458549	562425	858077	1483393	2988008
1000	462391	573367	809935	1608210	2768591
10000	475713	564680	807562	1669463	2848504

Table 3: Times (μs) using Dynamic Groups on a Paragon

Number of Integers	Number of Nodes 1	2	4	8	16
1	102	269	390	653	768
1000	115	755	1382	2122	2638
10000	133	3607	5592	7320	8808

Table 4: Times (μs) using Static Groups on a Paragon

5.4. Native NX versus PVM routines on Paragon

In this section we examine the performance of global sum via pvm_reduce with PvmSum as compared to a test implementing the same code using only native NX calls to *gisum*().

Table 5 lists the average time in microseconds (μs) to perform a *gisum*() for various length integer messages. This average is also calculated over 100 iterations of the *gisum*() call. Table 6 lists the averages for performing a pvm_reduce using PvmSum on the Intel Paragon.

Suppose we use a linear equation, $\alpha + \beta n$, to model the message communication based on latency (α), bandwidth (β) and size of data in bytes (n). For the native NX reduce operation, we would write

$$\text{msgtime}_{\texttt{NX}} = \alpha + \beta n.$$

Then, the data show that on average, the message communication for the PVM reduction operation would be

$$\text{msgtime}_{PVM} \approx \alpha + 109\mu s + \beta n.$$

For our timing results PVM added an average 109 μs latency term to the communication performance. This overhead appears independent of message length and the number of group members. Hence, communication bandwidth for the PVM reduction is the same as for the native NX commands. For most applications, this communication overhead is a small price to pay for easy portability of the code and for the ability to network different architectures into a single parallel machine.

5.5. Comparison to a Paragon Optimized PVM

As part of a diploma thesis[9], Bjarte Walaker implemented a version of PVM for the Paragon. The purpose of this work was to decrease the overhead that PVM incurs in performing group operations. At the time of Walaker's thesis, the native NX changes had not been implemented in PVM.

In this thesis, a number of hypotheses are made, however, most of the improvement in timings on the Paragon which were achieved by Walaker were due simply to utilizing the native NX commands.

For example, there is no need to have the root instance execute on the service node as Walaker describes in the thesis. All the node executables can easily be executed on the Paragon compute nodes even on the first release of the pvm_reduce function although the first release did not utilize the native NX calls. This is done using the "spawn" command from the PVM console.

Similarly, there is no need to force the PVM group server (pvmgs) to be executed on one of the compute nodes of the Paragon which Walaker does. This

Number of	Number of Nodes						
Integers	1	2	4	8	16	32	64
1	6	131	249	437	576	1118	1391
1000	11	573	1172	1918	2379	2838	3375
10000	12	3369	5360	7091	8524	8959	9767

Table 5: Times (μs) using Native NX *gisum* command

Number of	Number of Nodes						
Integers	1	2	4	8	16	32	64
1	102	269	390	653	768	1250	1537
1000	115	755	1382	2122	2638	3039	3610
10000	133	3607	5592	7320	8808	9124	9888

Table 6: Times (μs) using Static Groups on a Paragon

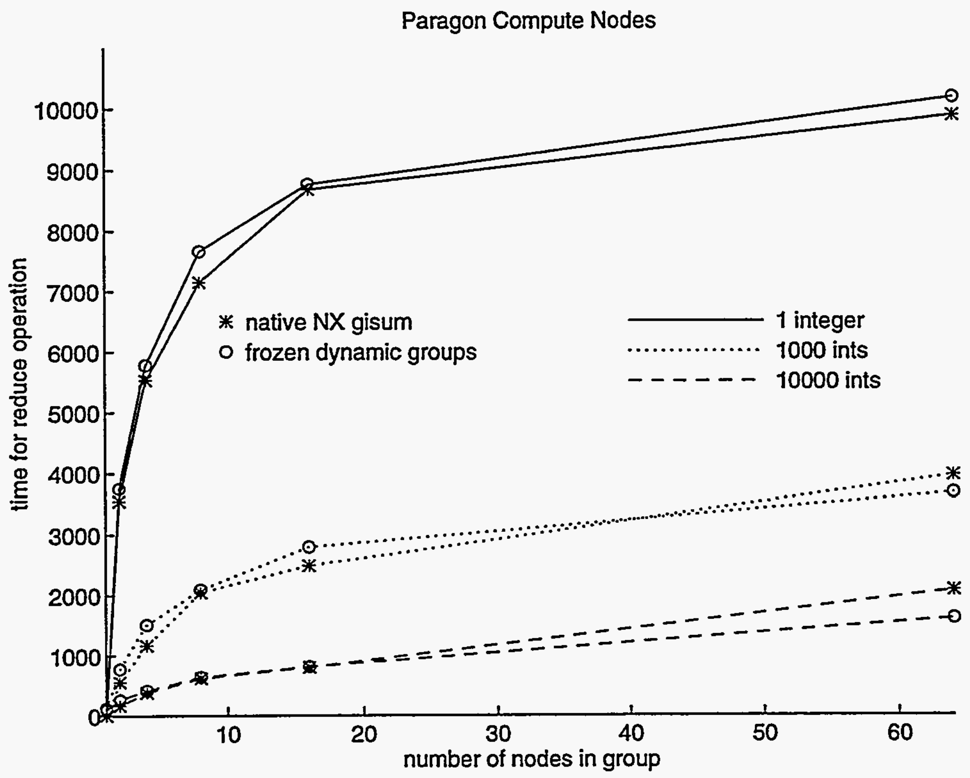

Figure 8: Times (μs) for Native NX versus Static Groups on a Paragon

version	Number of Nodes				
	2	4	8	16	64
Orig PVM	689075	768635	1208601	1680296	3798682
Walaker PVM	707	1428	2122	2847	4617

Table 7: Times (μs) from Walaker's Thesis

can be seen by comparing the timings in Table 6 to the results given in Walaker's thesis. The timing data from Walaker's thesis are presented in Table 7. For his tests, the message is always 1000 integers in length. Again, the times are given in microseconds (μs). It could not be determined from the thesis whether these timings were calculated from an average number of executions or not.

Refer to Figure 9 for a comparison of the data for a message of 1000 integers for the native NX *gisum*, pvm_sum, and Walaker's pvm_sum. The data from Walaker's thesis for Walaker's version of pvm_sum are the points noted with an #. The timings for PVM static groups and the implementation by Walaker are comparable. The differences may be due to random timing variations.

It is important to note that Walaker admits to making the restriction that his version of the NX based PVM can only be used on a single Intel Paragon. But from the tables and figure we can see that PVM can be implemented just as fast without this restriction.

The approach taken in the official PVM release gives the best combination of performance (using static groups) and in terms of keeping the crucial PVM feature of being able to network multiple hosts of different architectures into one Parallel Virtual Machine.

6. Conclusions

This paper has described the implementation of scatter, gather, and reduce collective communication routines in PVM as of release 3.3.10. Compared to native functions, we have seen that it is important to use static groups whenever performance is critical. Using direct static groups as will be implemented in PVM release 3.4.0 or making a simple change (such as, adding a call to pvm_freezegroup) in current PVM programs using dynamic groups can increase efficiency by two orders of magnitude when performing collective operations. Dynamic groups (not frozen) are still needed for fault tolerant applications.

Compared to native functions, we showed that there is an 109 μs average overhead incurred by using PVM. This overhead is independent of the number of nodes in the group and the message size. Hence, the message bandwidth for the PVM reduce operation is the same as the native NX routines upon which it is

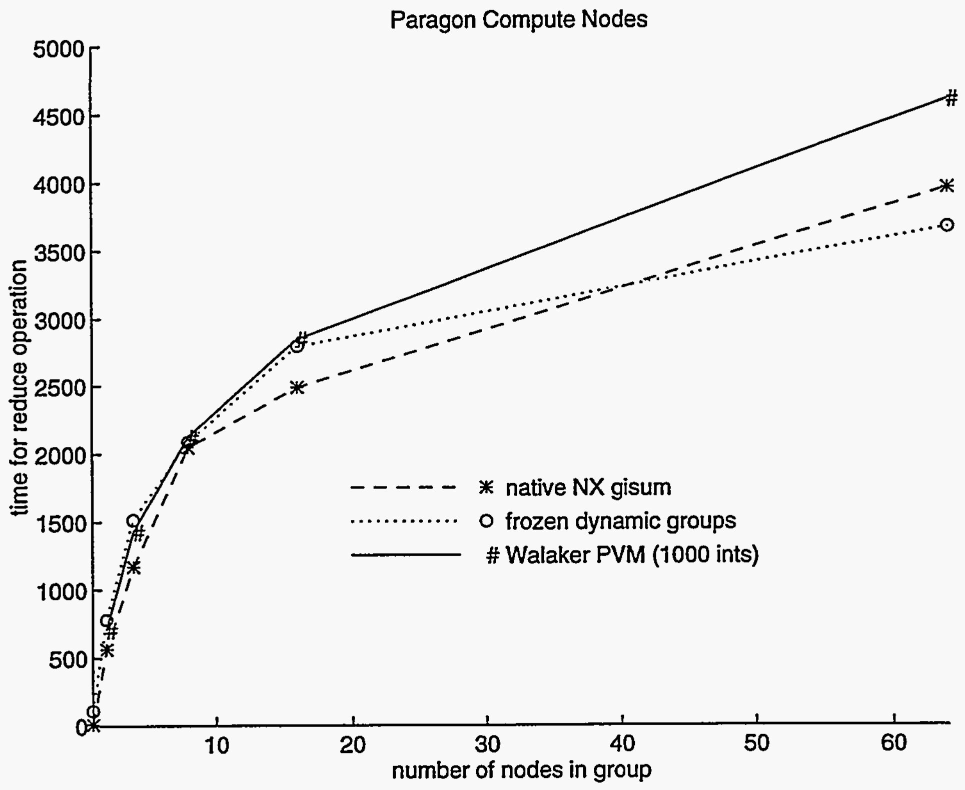

Figure 9: Comparison of NX *gisum*, pvm_sum, and Walaker's pvm_sum for 1000 integers

implemented. For most applications, the cost of this overhead in terms of time performance is well worth the generality and flexibility of being able to use PVM as it is intended - as a software system that allows a heterogeneous network of machines to be used as a Parallel Virtual Machine.

7. References

[1] A. Beguelin and P.M. Papadopoulos. Process Groups for Distributed Computing. September 1994. `http://www.epm.ornl.gov/~phil/procgroup.html`.

[2] H. Casanova, J.J. Dongarra, and W. Jiang. *The Performance of PVM on MPP Systems.* `http://www.netlib.org/utk/papers/pvmmpp/pvmmpp.html`.

[3] P.M. Papadopoulos, R.J. Manchek, G.A. Geist. *Context, Name Service, Static Groups for PVM.* Proceedings 1995 PVM User's Group Meeting, Pittsburgh, PA. May 1995. (Proceedings published on-line)

[4] G.A. Geist, A.L. Beguelin, J.J. Dongarra, W. Jiang, R.J. Manchek, and V.S. Sunderam. PVM: A Users' Guide and Tutorial for Networked Parallel Computing. MIT Press, 1994.

[5] P.K. McKinley, Y. Tsai, and D.F. Robinson. Collective Communication in Wormhole-Routed massively Parallel Computers. Computer, Vol. 28, No. 12, pp. 39–50, December 1995.

[6] MPI Forum. *MPI: A Message-Passing Interface Standard.* International Journal of Supercomputer Application, Vol. 8, No. 3/4, 165-416, 1994. See also `http://www.mcs.anl.gov/mpi/`.

[7] N. Nupairoj and L.M. Ni, *Benchmarking of Multicast Communication Services,* Technical Report MSU-CPS-ACS-103. Department of Computer Science, Michigan State University, 1995. Working Draft. `http://ftp.cps.msu.edu/pub/acs/msu-cps-acs-103.ps.Z`.

[8] *Paragon OSF/1 User's Guide,* Intel Supercomputer Systems Division, Beaverton, Oregon, April 1993.

[9] Bjarne Walaker, *PVM on PARAGON.* Diploma Thesis. Norwegian Institute of Technology. Faculty of Electrical Engineering and Computer Science. Fall 1994. Available via anonymous ftp. Host: `export.ssd.intel.com`. Directory: `pub/ISUG/PVM`.

A. PVM Manual Pages

The following are the man pages for the scatter, gather and reduce operations from the PVM 3.3.10 release.

A.1. Scatter

```
SCATTER(3PVM)     MISC. REFERENCE MANUAL PAGES      SCATTER(3PVM)

NAME
     pvm_scatter - Sends to each member of a group a  section  of
     an array from a specified member of the group.


SYNOPSIS
     C    int info = pvm_scatter( void *result, void *data,
                         int count, int datatype, int msgtag,
                         char *group, int rootginst)

     Fortran    call pvmfscatter(result, data, count, datatype,
                                msgtag, group, rootginst, info)


PARAMETERS
     result  Pointer to the  starting  address  of  an  array  of
             length  count  of datatype which will be overwritten
             by the message from the specified root member of the
             group.

     data    On the root  this  is  a  pointer  to  the  starting
             address  of  an array datatype of local values which
             are to be distributed to the members of  the  group.
             If  n  is  the  number of members in the group, then
             this array of datatype should be of length at  least
             n*count.   This  argument  is meaningful only on the
             root.

     count   Integer specifying the number of elements  of  data-
             type to be sent to each member of the group from the
             root.

     datatype
             Integer specifying the type of the  entries  in  the
             result  and  data  arrays.   (See  below for defined
             types.)
```

msgtag Integer message tag supplied by the user. msgtag should be >= 0. It allows the user's program to distinguish between different kinds of messages.

group Character string group name of an existing group.

rootginst
Integer instance number of group member who performs the scatter of its array to the members of the group.

info Integer status code returned by the routine. Values less than zero indicate an error.

DESCRIPTION

pvm_scatter() performs a scatter of data from the specified root member of the group to each of the members of the group, including itself. All group members must call pvm_scatter(), each receives a portion of the data array from the root in their local result array. It is as if the root node sends to the ith member of the group count elements from its array data starting at offset i*count from the beginning of the data array. And, it is as if, each member of the group performs a corresponding receive of count values of datatype into its result array. The root task is identified by its instance number in the group.

C and Fortran defined datatypes are:

C datatypes	FORTRAN datatypes
PVM_BYTE	BYTE1
PVM_SHORT	INTEGER2
PVM_INT	INTEGER4
PVM_FLOAT	REAL4
PVM_CPLX	COMPLEX8
PVM_DOUBLE	REAL8
PVM_DCPLX	COMPLEX16
PVM_LONG	

In using the scatter and gather routines, keep in mind that C stores multidimensional arrays in row order, typically starting with an initial index of 0; whereas, Fortran stores arrays in column order, typically starting with an offset of 1.

The current algorithm is very simple and robust. A future implementation may make more efficient use of the architecture to allow greater parallelism.

EXAMPLES

C:

```
   info =  pvm_scatter(&getmyrow, &matrix, 10, PVM_INT,
                       msgtag, "workers", rootginst);
```

Fortran:

```
   CALL PVMFSCATTER(GETMYCOLUMN, MATRIX, COUNT, INTEGER4,
  &                MTAG, 'workers', ROOT, INFO)
```

ERRORS

These error conditions can be returned by pvm_scatter

PvmNoInst
: Calling task is not in the group

PvmBadParam
: The datatype specified is not appropriate

PvmSysErr
: Pvm system error

SEE ALSO

pvm_bcast(3PVM), pvm_barrier(3PVM), pvm_psend(3PVM)

A.2. Gather

```
GATHER(3PVM)       MISC. REFERENCE MANUAL PAGES       GATHER(3PVM)

NAME
     pvm_gather - A specified member of the group  receives  mes-
     sages  from  each member of the group and gathers these mes-
     sages into a single array.


SYNOPSIS
     C    int info = pvm_gather( void *result,  void *data,
                      int count, int datatype, int msgtag,
                      char *group, int rootginst)

     Fortran    call pvmfgather(result, data, count, datatype,
                                 msgtag, group, rootginst, info)


PARAMETERS
     result  On the root  this  is  a  pointer  to  the  starting
             address  of  an array datatype of local values which
             are to be accumulated from the members of the group.
             If  n  if  the  number of members in the group, then
             this array of datatype should be of length at  least
             n*count.   This  argument  is meaningful only on the
             root.

     data    For each group member  this  is  a  pointer  to  the
             starting  address  of  an  array  of length count of
             datatype which will be sent to  the  specified  root
             member of the group.

     count   Integer specifying the number of elements  of  data-
             type  to  be sent by each member of the group to the
             root.

     datatype
             Integer specifying the type of the  entries  in  the
             result  and  data  arrays.  (See  below for defined
             types.)

     msgtag  Integer message tag supplied by  the  user.   msgtag
             should  be  >=  0.   It allows the user's program to
```

distinguish between different kinds of messages.

group Character string group name of an existing group.

rootginst
Integer instance number of group member who performs the gather of the messages from the members of the group.

info Integer status code returned by the routine. Values less than zero indicate an error.

DESCRIPTION

pvm_gather() performs a send of messages from each member of the group to the specified root member of the group. All group members must call pvm_gather(), each sends its array data of length count of datatype to the root which accumulates these messages into its result array. It is as if the root receives count elements of datatype from the ith member of the group and places these values in its result array starting with offset i*count from the beginning of the result array. The root task is identified by its instance number in the group.

C and Fortran defined datatypes are:

C datatypes	FORTRAN datatypes
PVM_BYTE	BYTE1
PVM_SHORT	INTEGER2
PVM_INT	INTEGER4
PVM_FLOAT	REAL4
PVM_CPLX	COMPLEX8
PVM_DOUBLE	REAL8
PVM_DCPLX	COMPLEX16
PVM_LONG	

In using the scatter and gather routines, keep in mind that C stores multidimensional arrays in row order, typically starting with an initial index of 0; whereas, Fortran stores arrays in column order, typically starting with an offset of 1.

Note: pvm_gather() does not block. If a task calls pvm_gather and then leaves the group before the root has called pvm_gather an error may occur.

The current algorithm is very simple and robust. A future implementation may make more efficient use of the architecture to allow greater parallelism.

EXAMPLES

C:

```
info =  pvm_gather(&getmatrix, &myrow, 10, PVM_INT,
                   msgtag, "workers", rootginst);
```

Fortran:

```
   CALL PVMFGATHER(GETMATRIX, MYCOLUMN, COUNT, INTEGER4,
  &                MTAG, 'workers', ROOT, INFO)
```

ERRORS

These error conditions can be returned by pvm_gather

PvmNoInst
: Calling task is not in the group

PvmBadParam
: The datatype specified is not appropriate

PvmSysErr
: Pvm system error

SEE ALSO

pvm_bcast(3PVM), pvm_barrier(3PVM), pvm_psend(3PVM)

A.3. Reduce

```
NAME
     pvm_reduce - Performs a reduction operation over members  of
     the specified group.


SYNOPSIS
     C    int info = pvm_reduce( void (*func)(),
                    void *data, int count, int datatype,
                    int msgtag, char *group, int rootginst)

     Fortran    call pvmfreduce(func, data, count, datatype,
                               msgtag, group, rootginst, info)


PARAMETERS
     func    Function which defines the  operation  performed  on
             the  global  data.  Predefined  are  PvmMax, PvmMin,
             PvmSum, and PvmProduct.  Users can define their  own
             function.

               SYNOPSIS for func
               C   void func(int *datatype, void *x, void *y,
                             int *num, int *info)
               Fortran    call func(datatype, x, y, num, info)

     data    Pointer to the starting address of an array of local
             values.   On return, the data array on the root will
             be overwritten with the result of the reduce  opera-
             tion  over  the  group.   For  the  other (non-root)
             members of the group the values of  the  data  array
             upon  return  from  the  reduce  operation  are  not
             defined; the values may be different than those ori-
             ginally passed to pvm_reduce.

     count   Integer specifying the number of elements  of  data-
             type  in  the data array.  The value of count should
             agree between all members of the group.
```

```
datatype
        Integer specifying the type of the  entries  in  the
        data array.  (See below for defined types.)

msgtag  Integer message tag supplied by  the  user.   msgtag
        should  be  >=  0.   It allows the user's program to
        distinguish between different kinds of messages.

group   Character string group name of an existing group.

rootginst
        Integer instance number of group member who gets the
        result.

info    Integer status code returned by the routine.  Values
        less than zero indicate an error.
```

DESCRIPTION

pvm_reduce() performs global operations such as max, min, sum, or a user provided operation on the data provided by the members of a group. All group members call pvm_reduce with the same size local data array which may contain one or more entries. The root task is identified by its instance number in the group.

The inner workings of the pvm_reduce call are implementation dependent; however, when the pvm_reduce call completes, the root's data array will be equal to the specified operation applied element-wise to the data arrays of all the group members.

A broadcast by the root can be used if the other members of the group need the resultant value(s).

PVM supplies the following predefined functions that can be specified in func.

```
          PvmMin
          PvmMax
          PvmSum
          PvmProduct
```

PvmMax and PvmMin are implemented for all the datatypes listed below. For complex values the minimum [maximum] is that complex pair with the minimum [maximum] modulus.

PvmSum and PvmProduct are implemented for all the datatypes listed below with the exception of PVM_BYTE and BYTE1.

C and Fortran defined datatypes are:

C datatypes	FORTRAN datatypes
PVM_BYTE	BYTE1
PVM_SHORT	INTEGER2
PVM_INT	INTEGER4
PVM_FLOAT	REAL4
PVM_CPLX	COMPLEX8
PVM_DOUBLE	REAL8
PVM_DCPLX	COMPLEX16
PVM_LONG	

A user defined function may be used in func. The argument func is a function with four arguments. It is the base function used for the reduction operation. Both x and y are arrays of type specified by datatype with num entries. The arguments datatype and info are as specified above. The arguments x and num correspond to data and count above. The argument y contains received values.

Caveat: pvm_reduce() does not block, a call to pvm_barrier may be necessary. For example, an error may occur if a task calls pvm_reduce and then leaves the group before the root has completed its call to pvm_reduce. Similarly, an error may occur if a task joins the group after the root has issued its call to pvm_reduce. Synchronization of the tasks (such as a call to pvm_barrier) was not included within the pvm_reduce implementation since this overhead is unnecessary in many user codes (which may already synchronize the tasks for other purposes).

The current algorithm is very simple and robust. A future implementation may make more efficient use of the architecture to allow greater parallelism.

ILLUSTRATION

The following example illustrates a call to pvm_reduce. Suppose you have three group members (instance numbers 0, 1, 2) with an array called Idata with 5 values as specified:

```
     instance       the 5 values in the integer array
```

```
        0                 1,   2,   3,   4,   5
        1                10,  20,  30,  40,  50
        2               100, 200, 300, 400, 500
```

And, suppose that a call to reduce (such as the ones following) are issued where the root is the group member with instance value of 1:

```
   C:
      root = 1;
      info = pvm_reduce(PvmSum, &Idata, 5, PVM_INT, msgtag,
                         "worker", root);
   Fortran:
      root = 1
      call pvmfreduce(PvmSum, Idata, 5, INTEGER4, msgtag,
                       "worker", root, info)
```

Then, upon completion of the reduce call, the following will result:

```
     instance       the 5 values in the integer array
        0                .... not defined.......
        1                111, 222, 333, 444, 555
        2                .... not defined ......
```

EXAMPLES

C:

```
   info =  pvm_reduce(PvmMax, &myvals, 10, PVM_FLOAT,
                      msgtag, "worker", rootginst);
```

Fortran:

```
   CALL PVMFREDUCE(PvmMax, MYVALS, COUNT, REAL4,
  &                MTAG, 'worker', ROOT, INFO)
```

ERRORS

These error conditions can be returned by pvm_reduce

PvmNoInst

Calling task is not in the group

PvmBadParam

The datatype specified is not appropriate for the specified reduction function.

```
     PvmSysErr
          Pvm system error

SEE ALSO
     pvm_bcast(3PVM), pvm_barrier(3PVM), pvm_psend(3PVM)
```

ORNL/TM-13228

INTERNAL DISTRIBUTION

1–2.	T. S. Darland	25–29.	S. A. Raby
3.	E. F. D'Azevedo	30.	C. H. Romine
4–8.	J. M. Donato	31.	T. H. Rowan
9.	J. J. Dongarra	32.	B. D. Semeraro
10–14.	G. A. Geist	33.	B. A. Shelton
15.	N. W. Grady	34–38.	R. F. Sincovec
16.	J. A. Kohl	39.	Central Research Library
17–21.	M. R. Leuze	40.	ORNL Patent Office
22.	E. G. Ng	41.	K-25 Appl Tech Library
23.	C. E. Oliver	42.	Y-12 Technical Library
24.	P. M. Papadopoulos	43.	Lab Records Dept - RC
		44–45.	Laboratory Records Dept

EXTERNAL DISTRIBUTION

46. Steven Ashby, Lawrence Livermore National Laboratory, P.O. Box 808, L-316, Livermore, CA 94551

47. Edward H. Barsis Computer Science and Mathematics Sandia National laboratories P.O. Box 5800 Albuquerque, NM 87185

48. Roger W. Brockett, Pierce Hall 29 Oxford Street Harvard University Cambridge, MA 02138

49. Tony Chan, Department of Mathematics, University of California, Los Angeles, 405 Hilgard Avenue, Los Angeles, CA 90024

50. Geoffrey C. Fox, Northeast Parallel Architectures Center, 111 College Place, Syracuse University, Syracuse, NY 13244-4100

51. J. Alan George, Vice President, Academic and Provost, Needles Hall, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1

52. Gene H. Golub, Department of Computer Science, Stanford University, Stanford, CA 94305

53. John Gustafson, Ames Laboratory, Iowa State University, Ames, IA 50011

54. Dr. Christian Halloy Asst. Director of JICS 104 South College University of Tennessee Knoxville, TN 37996-1301

55. Sven J. Hammarling The Numerical Algorithms Group Ltd. Wilkinson House Jordan Road Oxford OX2 8DR UNITED KINGDOM

56. Dr. Dan Hitchcock ER-31, Mathematical, Information, & Computational Sciences Div. Office of Computational & Technology Research Office of Energy Research, U.S. Department of Energy Washington, DC 20585

57. Malvyn H. Kalos, Cornell Theory Center, Engineering and Theory Center Bldg., Cornell University, Ithaca, NY 14853-3901

58. Hans Kaper, Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Bldg. 221, Argonne, IL 60439
59. Linda Kaufman, Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974
60. Kenneth Kennedy, Department of Computer Science, Rice University, P.O. Box 1892, Houston, TX 77001
61. James McGraw, Lawrence Livermore National Laboratory, L-306, P.O. Box 808, Livermore, CA 94550
62. Dr. David B. Nelson, Director Office of Scientific Computing ER-7 Applied Mathematical Sciences Office of Energy Research U.S. Dept. of Energy Washington, DC 20585
63. James M. Ortega, Department of Applied Mathematics, Thornton Hall, University of Virginia, Charlottesville, VA 22901
64. Robert J. Plemmons, Departments of Mathematics and Computer Science, Box 7311, Wake Forest University, Winston-Salem, NC 27109
65. Werner C. Rheinboldt, Department of Mathematics and Statistics, University of Pittsburgh, Pittsburgh, PA 15260
66. Ahmed H. Sameh, Center for Supercomputing R&D, 1384 W. Springfield Avenue, University of Illinois, Urbana, IL 61801
67. David S. Scott, Intel Scientific Computers, 15201 N.W. Greenbrier Parkway, Beaverton, OR 97006
68. Paul N. Swartztrauber, National Center for Atmospheric Research, P.O. Box 3000, Boulder, CO 80307
69. Andrew B. White, Computing Division, Los Alamos National Laboratory, P.O. Box 1663, MS-265, Los Alamos, NM 87545
70. Office of Assistant Manager for Energy Research and Development, Department of Energy, Oak Ridge Operations Office, P.O. Box 2001 Oak Ridge, TN 37831-8600

71–72. Office of Scientific & Technical Information, P.O. Box 62, Oak Ridge, TN 37831