Converting PVM Applications to MPI

PVM and MPI are two popular message-passing libraries that are in use across a variety of platform. MPI assimilates the most attractive features of a number of existing message-passing systems, including PVMs (see “Choosing Between MPI and PVM” on page 142).

Silicon Graphics has adopted MPI as the message-passing model for the POWER CHALLENGEarray and other Array products, and provides a low-latency, high-bandwidth implementation of MPI for these systems. Programmers are encouraged to write new message-passing applications using MPI, and to port existing applications to MPI when that is feasible.

Many existing message-passing applications use the PVM library, owing to its widespread use for the last five years. In order to support this application base, Silicon Graphics, Inc. also supports PVM for Array systems. However, the design of the MPI interface is such that the performance of the MPI implementation on these systems will always be better than the performance of PVM. To obtain best performance, porting parallel programs from PVM to MPI is recommended.

This appendix covers the following main topics:

- “Differences Between PVM and MPI” on page 282 gives an overview of the differences that are likely to cause difficulty in porting.
- “Comparing Library Routines” on page 285 lists the PVM routines and their MPI counterparts, when a counterpart exists.
- “Converting a PVM Program to an MPI Program” on page 289 covers the tasks involved in porting.
- “Example Programs” on page 294 shows example conversions.

Differences Between PVM and MPI

This section discusses the main differences between PVM and MPI from the user's perspective, focusing mainly on PVM functions that are not available in MPI.

Although to a large extent the library calls of MPI and PVM provide similar functionality, some PVM calls do not have a counterpart in MPI, and vice versa. Additionally, the semantics of some of the equivalent calls are inherently different for the two libraries (owing, for example, to the concept of dynamic groups in PVM). Hence, the process of converting a PVM program into an MPI program can be straightforward or complicated, depending on the particular PVM calls in the program and how they are used. For many PVM programs, conversion is straightforward.

In addition to a message-passing library, PVM also provides the concept of a *parallel virtual machine session*. A user starts this session before invoking any PVM programs; in other words, PVM provides the means to establish a parallel environment from which a user invokes a parallel program.

Additionally, PVM includes a *console*, which is useful for monitoring and controlling the states of the machines in the *virtual machine* and the state of execution of a PVM job. Most PVM console commands have corresponding library calls.

The MPI standard does not provide mechanisms for specifying the initial allocation of processes to an MPI computation and their binding to physical processors. Mechanisms to do so at load time or run time are left to individual vendor implementations. However, this difference between the two paradigms is not, by itself, significant for most programs, and should not affect the port from PVM to MPI.

The chief differences between the current versions of PVM and MPI libraries are as follows:

- PVM supports dynamic spawning of tasks, whereas MPI does not.
- PVM supports dynamic process groups; that is, groups whose membership can change dynamically at any time during a computation. MPI does not support dynamic process groups.

  MPI does not provide a mechanism to build a group from scratch, but only from other groups that have been defined previously. Closely related to groups in MPI are communicators, which specify the communication context for a communication operation and an ordered process group that shares this communication context. The chief difference between PVM groups and MPI communicators is that any PVM task can join/leave a group independently, whereas in MPI all communicator operations are collective.
- A PVM task can add or delete a host from the virtual machine, thereby dynamically changing the number of machines a program runs on. This is not available in MPI.
- A PVM program (or any of its tasks) can request various kinds of information from the PVM library about the collection of hosts on which it is running, the tasks that make up the program, and a task's parent. The MPI library does not provide such calls.
- Some of the collective communication calls in PVM (for instance, **pvm_reduce()**) are nonblocking. The MPI collective communication routines are not required to return as soon as their participation in the collective communication is complete.
- PVM provides two methods of signaling other PVM tasks: sending a UNIX signal to another task, and notifying a task about an event (from a set of predefined events) by sending it a message with a user-specified tag that the application can check. A PVM call is also provided through which a task can kill another PVM task. These functions are not available in MPI.
- A task can leave/unenroll from a PVM session as many times as it wants, whereas an MPI task must initialize/finalize exactly once.
- A PVM task need not explicitly enroll: the first PVM call enrolls the calling task into a PVM session. An MPI task must call **MPI_Init()** before calling any other MPI routine and it must call this routine only once.
- A PVM task can be registered by another task as responsible for adding new PVM hosts, or as a PVM resource manager, or as responsible for starting new PVM tasks. These features are not available in MPI.

- A PVM task can multicast data to a set of tasks. As opposed to a broadcast, this multicast does not require the participating tasks to be members of a group. MPI does not have a routine to do multicasts.
- PVM tasks can be started in debug mode (that is, under the control of a debugger of the user's choice). This capability is not specified in the MPI standard, although it can be provided on top of MPI in some cases.
- In PVM, a user can use the **pvm_catchout()** routine to specify collection of task outputs in various ways. The MPI standard does not specify any means to do this.
- PVM includes a receive routine with a timeout capability, which allows the user to block on a receive for a user-specified amount of time. MPI does not have a corresponding call.
- PVM includes a routine that allows users to define their own receive contexts to be used by subsequent PVM receive routines. Communicators in MPI provide this type of functionality to a limited extent.

On the other hand, MPI provides several features that are not available in PVM, including a variety of communication modes, communicators, derived datatypes, additional group management facilities, and virtual process topologies, as well as a larger set of collective communication calls. However, the set of MPI functions that are not available in PVM is not discussed here, since they are not directly relevant to porting from PVM to MPI.

Comparing Library Routines

Some PVM routines have close counterparts in MPI and others do not.

Comparable Routines

Table B-1 lists all the PVM routines (showing both C and Fortran names) and the corresponding MPI routines. As can be seen, most PVM routines have direct MPI counterparts. Of the remaining routines, many can simply be removed owing to changes in initial environment setup between PVM and MPI. These are marked by an asterisk (*) in the MPI column, and also include utility routines and routines that can be easily implemented at the application level (for example, **pvm_mcast()** and **pvm_trecv()**).

Routines that have a conceptual counterpart in MPI but which are not directly translatable to a single MPI call are listed with a phrase such as "communicators." Finally, nonportable routines are noted in the MPI column. Most of these nonportable PVM routines do not have a Fortran counterpart, which is also noted in the PVM column.

Note that this table does not exhaustively cover all aspects of PVM routines. For instance, it does not mention the various options of PVM calls. Also, there are some PVM routines that do have MPI counterparts, but are needed in only special cases, such as **pvm_initsend()** and **MPI_Send_init()**. Some routines, such as **pvm_bufinfo()** and **MPI_Get_count()**, have more than one corresponding call in MPI; only one is listed in the table.

Table B-1 Corresponding PVM and MPI Routines

PVM Routine (C/Fortran)	MPI Routine (C/Fortran)
pvm_addhosts/pvmfaddhost	*
pvm_barrier/pvmfbarrier	*MPI_Barrier/MPI_BARRIER*
pvm_bcast/pvmfbcast	*MPI_Bcast/MPI_BCAST*
pvm_bufinfo/pvmfbufinfo	*MPI_Get_count/MPI_GET_COUNT*
pvm_catchout/pvmfcatchout	*
pvm_config/pvmfconfig	*
pvm_delhosts/pvmfdelhost	*

Table B-1 (continued) Corresponding PVM and MPI Routines

PVM Routine (C/Fortran)	MPI Routine (C/Fortran)
pvm_exit/pvmfexit	*MPI_Finalize/MPI_FINALIZE*
pvm_freebuf/pvmffreebuf	*MPI_Buffer_detach/MPI_BUFFER_DETACH*
pvm_gather/pvmfgather	*MPI_Gather/MPI_GATHER*
pvm_getinst/pvmfgetinst	*MPI_Group_rank/MPI_GROUP_RANK*
pvm_getopt/pvmfgetopt	*
pvm_getrbuf/pvmfgetrbuf	Communicators
pvm_getsbuf/pvmfgetsbuf	Communicators
pvm_gettid/pvmfgettid	*
pvm_gsize/pvmfgsize	*MPI_Group_size/MPI_GROUP_SIZE*
pvm_halt/pvmfhalt	*
pvm_hostsync/pvmfhostsync	*MPI_Wtime/MPI_WTIME*
pvm_initsend/pvmfinitsend	*MPI_Send_init/MPI_SEND_INIT*
pvm_joingrup/pvmfjoingroup	*MPI_Comm_group/MPI_COMM_GROUP*
pvm_kill/pvmfkill	PVM routine is nonportable
pvm_lvgroup/pvmflvgroup	*MPI_Group_free/MPI_GROUP_FREE*
pvm_mcast/pvmfmcast	*
pvm_mkbuf/pvmfmkbuf	*MPI_Buffer_attach/MPI_BUFFER_ATTACH*
pvm_mstat/pvmfmstat	*
pvm_mytid/pvmfmytid	*MPI_Init/MPI_INIT* followed by *MPI_Comm_rank/MPI_COMM_RANK*
pvm_notify/pvmfnotify	PVM routine is nonportable
pvm_nrecv/pvmfnrecv	*MPI_Irecv/MPI_IRECV*
pvm_pk/pvmfpack*	*MPI_Pack/MPI_PACK*
pvm_parent/pvmfparent	*

Table B-1 (continued) Corresponding PVM and MPI Routines

PVM Routine (C/Fortran)	MPI Routine (C/Fortran)
pvm_perror/pvmfperror	*MPI_Error_string/MPI_ERROR_STRING*
pvm_precv/pvmfprecv	*MPI_Recv/MPI_RECV*
pvm_probe/pvmfprobe	*MPI_Iprobe/MPI_IPROBE*
pvm_psend/pvmfpsend	*MPI_Bsend/MPI_BSEND*
pvm_pstat/pvmfpstat	*
pvm_recv/pvmfrecv	*MPI_Recv/MPI_RECV*
pvm_recvf/no Fortran counterpart	PVM routine is nonportable
pvm_reduce/pvmfreduce	*MPI_Reduce/MPI_REDUCE*
pvm_reg_hoster/no Fortran counterpart	PVM routine is nonportable
pvm_reg_rm/no Fortran counterpart	PVM routine is nonportable
pvm_reg_tasker/no Fortran counterpart	PVM routine is nonportable
pvm_scatter/pvmfscatter	*MPI_Scatter/MPI_SCATTER*
pvm_send/pvmfsend	*MPI_Send/MPI_SEND*
pvm_sendsig/pvmfsendsig	PVM routine is nonportable
pvm_setopt/pvmfsetopt	*
pvm_setrbuf/pvmfsetrbuf	Communicators
pvm_setsbuf/pvmfsetsbuf	Communicators
pvm_spawn/pvmfspawn	*
pvm_tasks/pvmftasks	*
pvm_tidtohost/pvmftidtohost	*
pvm_trecv/pvmftrecv	*
pvm_upk/pvmfunpack*	*MPI_Unpack/MPI_UNPACK*

Nonportable PVM Functions

The PVM routines listed in this section cannot be translated directly into MPI routines. These same routines are shown in Table B-1 with the notation “PVM routine is nonportable,” to distinguish them from the PVM routines that, while they have no MPI counterpart, are easily removed in the MPI environment.

If the PVM program in question uses any of the following PVM functions, or uses dynamic groups, it cannot be directly ported to an MPI program:

- **pvm_kill()**
- **pvm_notify()**
- **pvm_recvf()**
- **pvm_reg_hoster()**
- **pvm_reg_rm()**
- **pvm_reg_tasker()**
- **pvm_sendsig()**

You must change the PVM program to eliminate the use of these functions before it can be ported to an MPI program. In some cases, this may not be possible. Note that most of these functions are available in the PVM domain only as C routines, and are not commonly used.

Converting a PVM Program to an MPI Program

This section discusses the basic steps for converting a PVM program to an MPI program..

Initial Environment Setup

PVM supports three different models of programming, and the initial environment setup varies depending on the model in question. The initial environment setup consists of determining the total number of PVM tasks to be used in the PVM job (including those started by hand at a shell prompt and those started via a **pvm_spawn()**), and using that as the initial static number for MPI. If the program being ported relies on dynamic addition and deletion of hosts you must change the program to use a static number of hosts and tasks.

It is a common practice in PVM programs to start a task by hand, and then determine the machine configuration inside this task via the **pvm_config()** call, so as to dynamically spawn tasks on the machines in the current configuration. You must replace this practice with a static determination of the hosts and tasks that form an MPI parallel program.

The rest of this section discusses the three programming models supported by PVM and how to perform initial environment setup for each case.

Pure SPMD Program

In the pure SPMD program model, *n* instances of the same program are started as the *n* tasks of the parallel job, using the *spawn* command of the PVM console (or by hand at each of the *n* hosts simultaneously). No tasks are dynamically spawned in the tasks; that is, the tasks do not use **pvm_spawn()**. This scenario is essentially the same as the current MPI one where no tasks are dynamically spawned.

For this scenario, the initial parallel environment setup consists of specifying the hosts to run the *n* tasks on. You can accomplish this setup using the mechanism provided on top of the MPI library. For example, the setup can use a hostfile for *mpirun* or the *procgroup* file for the MPICH implementation.

General SPMD Model

In this model, *n* instances of the same program are executed as *n* tasks of the parallel job. However, one or more tasks are started by hand at the beginning, and these dynamically spawn the remaining tasks in turn.

Here, the change involves figuring out how many PVM tasks are spawned in total (including those started by hand and those dynamically spawned), and on what machines these tasks are run. These two pieces of information can be directly translated into information (number of MPI tasks and the hosts on which these are to be run) that the *hostfile/procgroup* file of the MPI setup requires.

You must remove all instances of the **pvm_spawn()** call from the program. Most of the options of this call can be dealt by a translation into the MPI initial setup. The option *PvmTaskDebug* has no counterpart in MPI, so the corresponding MPI task cannot be started in debug mode. The option *PvmTaskTrace* and its subsequent use with a tool such as XPVM can be translated to whatever profiling interface and tools are available in the given MPI implementation.

Similarly, you should also eliminate all calls to **pvm_addhosts()**, **pvm_delhosts()**, and **pvm_config()**. Finally, if the program has a **pvm_halt()** call, remove it also.

MPMD Model

In an MPMD programming model, one or more distinct tasks (having different executables) are started by hand, and these tasks dynamically spawn other (possibly distinct) tasks. The initial setup change required for this model is similar to the one required for the general SPMD model discussed in the previous section; that discussion applies here too. The main difference here is that the task executables are different programs, and this information is encapsulated in the *hostfile/procgroup* file in the MPI paradigm.

The initial MPI environment setup thus consists of figuring out the number of instances of each distinct executable that constitute the parallel job, and using the total as the static initial number for the MPI environment. Again, you must remove all the **pvm_spawn()**, **pvm_config()**, **pvm_addhosts()**, **pvm_delhosts()**, and **pvm_halt()** calls in each PVM executable.

Common Environment Setup Changes

For all the three models, you must remove from the program being ported all calls that query the library for virtual machine or tasks information, such as **pvm_mstat()**, **pvm_pstat()** and **pvm_tasks()**. Handle any semantic dependency to these calls in the program, other than initial environment setup, in the resulting MPI program.

Since tasks cannot enroll in and leave from an MPI run time environment more than once, you must change all PVM tasks to reflect this requirement. Typically, a PVM task enrolls via the **pvm_mytid()** call; in the absence of this call, the first PVM call enrolls the calling task. Additionally, a task can call **pvm_mytid()** several times in a program with or without interleaved **pvm_exit()** calls. If it is not interleaved with **pvm_exit()** calls, the calling task simply gets its task ID back from the PVM library on the second and subsequent **pvm_mytid()** calls. You can easily eliminate these subsequent **pvm_mytid()** calls from the program by saving the value of the task ID and passing it around.

Replace the first **pvm_mytid()** call in a PVM program with the **MPI_Init()** routine, which must precede all other MPI routines and must be called exactly once. Since an MPI implementation can add its own command-line arguments to be processed by **MPI_Init()**, you must place all the user's command-line processing (anything that accesses *argc* and *argv*) after **MPI_Init()**. This requirement is in contrast to PVM programs, since PVM does not add its own arguments to those of the tasks being started.

To find out the number of tasks in the parallel job and its own task ID, an MPI task must call the functions **MPI_Comm_size()** and **MPI_Comm_rank()**. Thus the initial portion of a typical MPI program looks like the following:

```
/* Initialize the MPI environment. */
MPI_Init(&argc, &argv);
/* Get task id and the total number of tasks. */
/* The rank is essentially the task id. */
MPI_Comm_rank(MPI_COMM_WORLD, &taskId);
MPI_Comm_size(MPI_COMM_WORLD, &numTasks);
```

Task Environment Cleanup

Replace the **pvm_exit()** call at the end of each PVM program with the **MPI_Finalize()** call, which cleans up all MPI states. This call should be the last MPI routine in a program. You must ensure that all pending communications involving a process complete before the process calls **MPI_Finalize()**.

Group Management Functions

As far as groups are concerned, the main difference between PVM and MPI is that PVM groups can be dynamic, whereas MPI groups are static. In PVM, a task can belong to multiple groups and can join and leave a group an arbitrary number of times, so that groups can change dynamically at any time during a computation. Additionally, arbitrary groups can be formed by tasks.

In contrast, in MPI a group cannot be built from scratch, but only from other groups that have been defined previously. MPI has two predefined groups: *MPI_GROUP_EMPTY* (a group with no members), and the group associated with the initial communicator *MPI_COMM_WORLD* (consisting of all processes), which forms the base group upon which all other groups are defined.

If the PVM program uses dynamic groups, modify it to use only static groups before it can be ported to an MPI program. Note that most applications do not need dynamic groups.

Once the PVM program to be ported deals only with static groups, replace all instances of **pvm_joingroup()** with **MPI_Comm_group()** or one of its variants. Replace all occurrences of **pvm_lvgroup()** with **MPI_Group_free()**.

Intertask Communication

All PVM intertask communication calls have counterparts in MPI, except for **pvm_mcast()** and **pvm_trecv()**. You can easily replace multicasting in the PVM library with multicasting at the application layer with a set of send calls or by defining a group and performing a broadcast in that group. Similarly, you can replace a timed receive in the PVM library by an equivalent function at the application layer.

Some PVM collective communication calls, namely, **pvm_gather()** and **pvm_reduce()**, are nonblocking. This characteristic should not lead to any changes in the application code unless the PVM application has explicit synchronization calls (for example, **pvm_barrier()**) after such nonblocking calls. In such a case, you can remove these synchronization calls from the translated MPI program.

To send contiguous data of a given type, MPI does not require packing and unpacking of data in send buffers, as PVM does. Additionally, for noncontiguous data, MPI provides derived datatypes that avoid explicit packing and unpacking. However, MPI also provides pack/unpack functions for sending noncontiguous data, for compatibility with previous versions of libraries.

Multiple message buffers and their functionality in PVM can be emulated by communicators in MPI.

Utility Functions

Most utility functions in PVM have corresponding setup options in the parallel setup facility that comes with a particular MPI implementation. Some of these utility functions may not be available; note, however, that these functions do not directly affect the basic characteristics of the application. Instead they are provided as a convenience to programmers. Such functions include **pvm_catchout()**, **pvm_getopt()**, **pvm_setopt()** and **pvm_tidtohost()**.

A PVM task has a parent task, whose task ID is returned by the **pvm_parent()** call. Since MPI tasks are not spawned by other MPI tasks, this concept of a parent task does not exist in MPI. Hence you must remove all instances of **pvm_parent()** and handle their logical consequences in the program. For instance, one of the most common reasons for finding out the parent's task ID is to send computation-result messages back to it; this functionality can be easily replicated in an MPI program (or even a PVM program) by a task declaring itself to be the logical parent to whom all the computation-result messages should be sent.

Example Programs

The two examples in this section illustrate some of the porting concepts presented in this chapter. The first one is a SPMD program where all the tasks are instances of the same executable; here the first task spawns the remaining ones in the PVM version. The second example is a general MPMD program based on the master-slave paradigm, with one master task and multiple slave tasks.

Both these examples are taken from the example set provided with the public domain PVM software. Please note that several different translations are possible for each example, and the ones given here may not be the most efficient ones.

Example 1: SPMD Program

Note: The group functions in the PVM version of the program are not necessary in the MPI counterpart, since the basic group corresponding to *MPI_COMM_WORLD* containing all the tasks already exists in MPI.

SPMD Program in PVM Version

```
/*
*    SPMD example using PVM 3
*    also illustrating group functions
*/
#define NPROC 4
#include <stdio.h>
#include <sys/types.h>
#include "pvm3.h"
void dowork(int me, int nproc);
main()
{
    int mytid;                    /* my task id */
    int tids[NPROC];              /* array of task id */
    int me;                       /* my process number */
    int i;
    /* enroll in pvm */
    mytid = pvm_mytid();
    /* Join a group and if I am the first instance */
    /* i.e. me=0 spawn more copies of myself       */
    me = pvm_joingroup("foo");
    printf("me = %d mytid = %d\n",me,mytid);
    if( me == 0 )
```

```
      pvm_spawn("spmd", (char**)0, 0, "", NPROC-1,&tids[1]);
    /* Wait for everyone to startup before proceeding. */
    pvm_barrier("foo", NPROC);
/*--------------------------------------------------------*/
     dowork(me, NPROC);
     /* Program finished. Leave group and exit pvm */
     pvm_lvgroup("foo");
     pvm_exit();
     exit(1);
}
/* Simple example passes a token around a ring */
void dowork(int me, int nproc)
{
     int token;
     int src, dest;
     int count  = 1;
     int stride = 1;
     int msgtag = 4;
     /* Determine neighbors in the ring */
     src = pvm_gettid("foo", me-1);
     dest= pvm_gettid("foo", me+1);
     if(me == 0)
        src = pvm_gettid("foo", NPROC-1);
     if(me == NPROC-1)
        dest = pvm_gettid("foo", 0);
     if(me == 0)
     {
        token = dest;
        pvm_initsend(PvmDataDefault);
        pvm_pkint(&token, count, stride);
        pvm_send(dest, msgtag);
        printf("token ring begun: value sent = %d\n", token);
        pvm_recv(src, msgtag);
        pvm_upkint(&token, count, stride);
        printf("token ring done: value recvd = %d\n", token);
     }
     else
     {
        pvm_recv(src, msgtag);
        pvm_upkint(&token, count, stride);
        pvm_initsend(PvmDataDefault);
        pvm_pkint(&token, count, stride);
        pvm_send(dest, msgtag);
    }
}
```

SPMD Program in MPI Version

```
/*
*    SPMD example using MPI,
*    illustrating porting from PVM to MPI.
*/
#include <stdio.h>
#include <sys/types.h>
#include <mpi.h>
void dowork(int me, int nproc);
main(int argc, char *argv[])
{
    int mytid;                  /* my task id */
    int ntasks;                 /* total number of tasks */
    int i;
    /* Initialize MPI */
    MPI_Init(&argc, &argv);
    /* Get our task id (our rank in the basic group) */
    MPI_Comm_rank(MPI_COMM_WORLD, &mytid);
    /* Get the number of MPI tasks */
    MPI_Comm_size(MPI_COMM_WORLD, &ntasks);
    if(mytid == 0)
       printf("mytid = %d, ntasks = %d\n", mytid, ntasks);
    /* Wait for everyone to startup before proceeding. */
    MPI_Barrier(MPI_COMM_WORLD);
/*---------------------------------------------------*/
     dowork(mytid, ntasks);
     MPI_Finalize();
     exit(0);
}
/* Simple example passes a token around a ring */
void dowork(int me, int nproc)
{
     int token;
     int src, dest;
     MPI_Status status;
     int count = 1;
     int msgtag = 4;
     /* Determine neighbors in the ring */
     src = me-1;
     dest= me+1;
     if(me == 0) src = nproc-1;
     if(me == nproc-1) dest = 0;
     if(me == 0)
     {
```

```
        token = dest;
        MPI_Send(&token, count, MPI_INT, dest, msgtag,
                                          MPI_COMM_WORLD);
        printf("token ring begun: value sent = %d\n", token);
        MPI_Recv(&token, count, MPI_INT, src, msgtag,
                                    MPI_COMM_WORLD, &status);
        printf("token ring done: value rcvd = %d\n", token);
     }
     else
     {
        MPI_Recv(&token, count, MPI_INT, src, msgtag,
                                    MPI_COMM_WORLD, &status);
        MPI_Send(&token, count, MPI_INT, dest, msgtag,
                                       MPI_COMM_WORLD);
     }
}
```

Example 2: MPMD Example

In this example, in the PVM version, the slaves are sent all the slave tids by the master and they use these to determine their logical ordering among each other. The MPI slaves determine their logical ordering by the information available to them about their individual task ID and the master's task ID. This is just one of the many schemes by which this can be implemented.

Also, instead of packing and unpacking used in the MPI version, MPI derived datatypes could have been used.

MPMD in PVM Version—Master Task

```
#include <stdio.h>
#include "pvm3.h"
#define SLAVENAME "slave1"
main()
{
    int mytid;               /* my task id */
    int tids[32];            /* slave task ids */
    int n, nproc, numt, i, who, msgtype, nhost, narch;
    float data[100], result[32];
    struct pvmhostinfo *hostp[32];
    /* enroll in pvm */
    mytid = pvm_mytid();
    /* Set number of slaves to start */
    /* Can not do stdin from spawned task */
    if(pvm_parent() == PvmNoParent){
       puts("How many slave programs (1-32)?");
       scanf("%d", &nproc);
    }
    else{
       pvm_config(&nhost, &narch, hostp);
       nproc = nhost;
       if(nproc > 32) nproc = 32 ;
    }
    /* start up slave tasks */
    numt=pvm_spawn(SLAVENAME, (char**)0, 0, "", nproc, tids);
    if( numt < nproc ){
       printf("Trouble spawning slaves. Aborting. ");
       printf("Error codes are:\n");
       for( i=numt ; i<nproc ; i++ ) {
          printf("TID %d %d\n",i,tids[i]);
       }
```

```
        for( i=0 ; i<numt ; i++ ){
           pvm_kill(tids[i]);
        }
        pvm_exit();
        exit();
    }
    /* Begin User Program */
    n = 100;
    /* initialize_data( data, n ); */
    for( i=0 ; i<n ; i++ ){
       data[i] = 1;
    }
    /* Broadcast initial data to slave tasks */
    pvm_initsend(PvmDataDefault);
    pvm_pkint(&nproc, 1, 1);
    pvm_pkint(tids, nproc, 1);
    pvm_pkint(&n, 1, 1);
    pvm_pkfloat(data, n, 1);
    pvm_mcast(tids, nproc, 0);
    /* Wait for results from slaves */
    msgtype = 5;
    for( i=0 ; i<nproc ; i++ ){
       pvm_recv(-1, msgtype);
       pvm_upkint(&who, 1, 1);
       pvm_upkfloat(&result[who], 1, 1);
       printf("I got %f from %d\n",result[who],who);
    }
    /* Program Finished. Exit PVM before stopping */
    pvm_exit();
}
```

MPMD in PVM Version—Slave Task

```
#include <stdio.h>
#include "pvm3.h"
float work(int me, int n, float *data, int *tids, int nproc );
main()
{
    int mytid;       /* my task id */
    int tids[32];    /* task ids   */
    int n, me, i, nproc, master, msgtype;
    float data[100], result;
    float work();
    /* enroll in pvm */
    mytid = pvm_mytid();
```

```
    /* Receive data from master */
    msgtype = 0;
    pvm_recv(-1, msgtype);
    pvm_upkint(&nproc, 1, 1);
    pvm_upkint(tids, nproc, 1);
    pvm_upkint(&n, 1, 1);
    pvm_upkfloat(data, n, 1);
    /* Determine which slave I am (0 -- nproc-1) */
    for(i=0; i<nproc ; i++)
       if(mytid == tids[i])
          { me = i; break; }
    /* Do calculations with data */
    result = work(me, n, data, tids, nproc);
    /* Send result to master */
    pvm_initsend(PvmDataDefault);
    pvm_pkint(&me, 1, 1);
    pvm_pkfloat(&result, 1, 1);
    msgtype = 5;
    master = pvm_parent();
    pvm_send(master, msgtype);
    /* Program finished. Exit PVM before stopping */
    pvm_exit();
}
float
work(int me, int n, float *data, int *tids, int nproc )
/*Simple ex: slaves exchange data with left neighbor*/
{
    int i, dest;
    float psum = 0.0;
    float sum = 0.0;
    for(i=0 ; i<n ; i++){
       sum += me * data[i];
    }
    /*illustrate node-to-node communication*/
    pvm_initsend(PvmDataDefault);
    pvm_pkfloat(&sum, 1, 1);
    dest = me+1;
    if(dest == nproc) dest = 0;
    pvm_send(tids[dest], 22);
    pvm_recv(-1, 22);
    pvm_upkfloat(&psum, 1, 1);
    return(sum+psum);
}
```

MPMD in MPI Version—Master Task

```
#include <stdio.h>
#include <mpi.h>
main(int argc, char *argv[])
{
    int mytid;                  /* my task id */
    int n, nproc, ntasks, i, who, msgtype;
    float data[100], result[32];
    char sbuff[1000], rbuff[1000];
    int position;
    MPI_Status status;
    /* Initialize MPI */
    MPI_Init(&argc, &argv);
    /* Get our task id (our rank in the basic group) */
    MPI_Comm_rank(MPI_COMM_WORLD, &mytid);
    /* Get the number of MPI tasks and slaves */
    MPI_Comm_size(MPI_COMM_WORLD, &ntasks);
    nproc = ntasks - 1;
    if(mytid == 0)
       printf("mytid = %d, ntasks = %d\n", mytid, ntasks);
    /* Begin User Program */
    n = 100;
    /* initialize_data( data, n ); */
    for( i=0 ; i<n ; i++ ){
       data[i] = 1;
    }
    /* Pack initial data to be sent to slave tasks */
    position = 0;
    MPI_Pack(&n, 1, MPI_INT, sbuff, 1000, &position,
             MPI_COMM_WORLD);
    MPI_Pack(data, 100, MPI_FLOAT, sbuff, 1000, &position,
             MPI_COMM_WORLD);
    /* Send initial data to slave tasks */
    msgtype = 0;
    for(i=0; i<ntasks; i++){
       if(i != mytid){
          MPI_Send(sbuff, position, MPI_PACKED, i, msgtype,
                   MPI_COMM_WORLD);
       }
    }
    /* Wait for results from slaves */
    msgtype = 5;
    for( i=0 ; i<nproc ; i++ ){
       MPI_Recv(rbuff, 1000, MPI_PACKED, MPI_ANY_SOURCE,
```

```
                  msgtype, MPI_COMM_WORLD, &status);
         position = 0;
         MPI_Unpack(rbuff, 1000, &position, &who, 1, MPI_INT,
                    MPI_COMM_WORLD);
         MPI_Unpack(rbuff, 1000, &position, &result[who], 1,
                    MPI_FLOAT, MPI_COMM_WORLD);
         printf("I got %f from %d\n",result[who],who);
      }
      /* Program Finished. Exit MPI before stopping */
      MPI_Finalize();
}
```

MPMD in MPI Version—Slave Task

Note the use of the buffered sends in the slave task (MPI version). Using standard sends instead would lead to deadlock in MPI implementations that do not use buffering for standard sends.

```
#include <stdio.h>
#include <mpi.h>
float work(int mytid, int me, int n, float *data, int ntasks, int master)
main(int argc, char *argv[])
{
    int mytid;       /* my task id */
    int me;          /* logical ordering among slaves. */
    int n, i, ntasks, master, msgtype;
    float data[100], result;
    float work();
    char rbuff[1000], sbuff[1000];
    int position;
    MPI_Status status;
    /* Initialize MPI */
    MPI_Init(&argc, &argv);
    /* Get our task id (our rank in the basic group) */
    MPI_Comm_rank(MPI_COMM_WORLD, &mytid);
    /* Get the number of MPI tasks */
    MPI_Comm_size(MPI_COMM_WORLD, &ntasks);
    /* Receive initial data from master. */
    msgtype = 0;
    MPI_Recv(rbuff, 1000, MPI_PACKED, MPI_ANY_SOURCE,
             msgtype, MPI_COMM_WORLD, &status);
    /* Find out master's task id. */
    master = status.MPI_SOURCE;
    /* Unpack data. */
```

```
    position = 0;
    MPI_Unpack(rbuff, 1000, &position, &n, 1, MPI_INT,
               MPI_COMM_WORLD);
    MPI_Unpack(rbuff, 1000, &position, data, n, MPI_FLOAT,
               MPI_COMM_WORLD);
    /* Determine which slave I am (value of me) */
    /* If mytid < master, me = mytid */
    /* Else me=mytid-1 */
    if(mytid > master)
       me = mytid-1;
    else
       me = mytid;
    /* Do calculations with data */
    result = work(mytid, me, n, data, ntasks, master);
    /* Pack result */
    position = 0;
    MPI_Pack(&me, 1, MPI_INT, sbuff, 1000, &position,
             MPI_COMM_WORLD);
    MPI_Pack(&result, 1, MPI_FLOAT, sbuff, 1000, &position,
             MPI_COMM_WORLD);
    /* Send result to master */
    msgtype = 5;
    MPI_Send(sbuff, position, MPI_PACKED, master, msgtype,
             MPI_COMM_WORLD);
    /* Program finished. Exit from MPI */
    MPI_Finalize();
}
float
work(int mytid, int me, int n, float *data, int ntasks, int master)
    /* Simple example: slaves exchange data with left
       neighbor (wrapping) */
{
    int i, dest;
    MPI_Status status;
    float psum = 0.0;
    float sum = 0.0;
    char outbuff[100];
    for(i=0 ; i<n ; i++){
       sum += me * data[i];
    }
    /* illustrate node-to-node communication */
    dest = mytid+1;
    if(dest == ntasks)
       dest=0;
    if(dest == master)
```

```
        dest++;
    MPI_Buffer_attach(outbuff, 100);
    MPI_Bsend(&sum, 1, MPI_FLOAT, dest, 22, MPI_COMM_WORLD);
    MPI_Recv(&psum, 1, MPI_FLOAT, MPI_ANY_SOURCE, 22,
             MPI_COMM_WORLD, &status);
    return(sum+psum);
}
```