```
      integer  mynum, hostnum, bytes, msgtype, ...
      double precision result, data(100), ...
      character*16 host, ...

c     Enroll this program in PVM
      call fenroll( "nodeprogram\0", mynum )

c ------- Begin user program --------
                    .
                    .
c     Receive data from host
      msgtype  = 1
     call frcv( msgtype )
     call fgetndfloat( data, 100 )
     call frcvinfo( bytes, msgtype, host, hostnum )
                    .
                    .
     result = user_routine( data )

c    Send result to host
     call finitsend()
     call fputndfloat( result, 1 )
     msgtype  = 2
     call fsnd( host, hostnum, msgtype )
                    .
                    .
c --------- End user program --------

c    Program finished. Leave PVM before exiting
     call fleave()
     stop
     end
```

Figure 5: Fortran node program template using PVM routines.

```
      integer i, nproc, msgtype, mynum, inst(4), ...
      double precision result(4), data(100), ...

c     Enroll this program in PVM
      call fenroll( "hostprogram\0", mynum )

c     Initiate nproc instances of node program
      nproc = 4
      arch  = "\0"
      do 10 i=1,nproc
         call finitiate( "nodeprogram\0", arch, inst(i) )
10    continue

c ------- Begin user program --------
                    .
                    .
c     broadcast data to all node programs
      call finitsend()
      call fputndfloat( data, 100 )
      msgtype  = 1
      call fsnd( "nodeprogram\0", -1, msgtype )
                    .
                    .
c     wait for results from nodes
      msgtype  = 2
      do 20 i=1,nproc
         call frcv( msgtype )
         call fgetndfloat( result(i), 1 )
20    continue
                    .
                    .
c --------- End user program --------

c     program finished leave PVM before exiting
      call fleave()
      stop
      end
```

Figure 4: Fortran host program template using PVM routines.

The space requirement for PVM depends on the number of architectures for which it is built. A set up for five different architectures requires about 1 Mbyte of disk space.

6. References

[1] SUN Network Programming Manual Part Two: Protocol Specification, (1988).

fbarrier(barrier_name, n)
fenroll(component_name, instance_number)
fgetbytes(array, num)
fgetncplx(array, num)
fgetndcplx(array, num)
fgetndfloat(array, num)
fgetnfloat(array, num)
fgetnint(iarray, num)
fgetstring(string)
fgetstringl(string, length)
finitiate(component_name, architecture, instance_number)
finitiatem(component_name, machine, instance_number)
finitsend( )
fleave( )
fputbytes(array, num)
fputncplx(array, num)
fputndcplx(array, num)
fputndfloat(array, num)
fputnfloat(array, num)
fputnint(iarray, num)
fputstring(string)
fputstringl(string, length)
frcv(msg_id)
frcvinfo(length, msg_id, component_name, instance_number)
frcvmulti(n, types)
fready(event_name)
fsnd(component_name, instance_number, msg_id)
fstatus(component_name, instance_number, istatus)
fterminate(process_name, instance_number)
fwaituntil(event_name)

Table 3: Routines in Fortran-to-PVM interface.

in the interface routines. A second problem, common to Fortan-to-C interfaces, was correct passing of arguments. Fortran passes arguments by reference and C passes arguments by value. Because of problems on some supported machines with passing values to Fortran functions, only subroutines are used in the interface. This causes the user interface to PVM to be slightly different when programming in Fortran rather than C. A third problem encountered was string termination. Several PVM routines pass strings, such as program names and signals. C terminates strings with NULLs, but this is not a requirement in Fortran so some Fortran compilers do not terminate strings. Instead, they keep track of the length of strings in an internal table. Sending a C routine a pointer to the beginning of a nonterminated string leads to nondeterministic behavior at best and a memory fault at worst. The solution to this problem requires that Fortran programmers manually append all the string arguments in their codes with NULLs For example,

```
call finitiate( 'program\0', instancenum ).
```

Not all Fortrans recognize '\0' as NULL. One example is Cray's cf77. For such machines there is a "NULFIX" switch in the interface *makefile* that causes the Fortran interface routines to recognize '\0' as the end of a string.

Table 3 lists the names and argument lists of the supported Fortran calls in PVM version 2.3.

Figures 4 and 5 depict the same program as the previous section. But now the examples are written in Fortran.

5. Obtaining PVM

PVM is available from `netlib`. For information about this package send the following message to `netlib@ornl.gov`.

```
send index from pvm
```

A short description of PVM and a list of available files in the package will be returned.

The source files, which consume less than 350Kbytes, are available in the *shar* file `pvm_shar`. To receive this file send the message:

```
send pvm_shar from pvm
```

```
main()
{
  int  mynum, hostnum, bytes, msgtype, ...
  double result, data[100], ...
  char host[16], ...

  /* Enroll this program in PVM */
  mynum = enroll( "nodeprogram" ) ;

  /* ------- Begin user program -------- */
                    .
                    .
  /* Receive data from host */
  msgtype  = 1 ;
  rcv( msgtype ) ;
  getndfloat( data, 100 ) ;
  rcvinfo( &bytes, &msgtype, host, &hostnum )
                    .
                    .
  result = user_routine( data ) ;

  /* Send result to host */
    initsend() ;
    putndfloat( &result, 1 ) ;
    msgtype  = 2 ;
    snd( host, hostnum, msgtype ) ;
  }
                    .
                    .
  /* --------- End user program -------- */

  /* Program finished. Leave PVM before exiting */
  leave() ;
}
```

Figure 3: Simple node program template using PVM routines.

```
main()
{
  int i, nproc, msgtype, mynum, inst[4], ...
  double  result[4], data[100], ...

  /* Enroll this program in PVM */
  mynum = enroll( "hostprogram" ) ;

  /* Initiate nproc instances of node program */
  nproc = 4 ;
  for( i=0 ; i<nproc ; i++ )
      inst[i] = initiate( "nodeprogram", NUL ) ;

  /* ------- Begin user program -------- */
                  .
                  .
  /* broadcast data to all node programs */
  initsend() ;
  putndfloat( data, 100 ) ;
  msgtype  = 1 ;
  snd( "nodeprogram", -1, msgtype ) ;
                  .
                  .
  /* wait for results from nodes */
  msgtype  = 2 ;
  for( i=0 ; i<nproc ; i++){
     rcv( msgtype ) ;
     getndfloat( &result[i], 1 );
  }
                  .
                  .
  /* --------- End user program -------- */

  /* program finished leave PVM before exiting */
  leave() ;
}
```

Figure 2: Simple host program template using PVM routines.

Sample programs are supplied with the software distribution. Figures 2 and 3 give a template of simple host and node programs that use PVM routines. The host calls `enroll()` to allow it to use the PVM system and enable interprocessor communication. It then calls `initiate()` to execute node program(s) on other machines in PVM. Each node program must also call `enroll()` to enable interprocessor communication. Subsequently, `snd()` and `rcv()` are used to pass messages between processes. The node program contains an example of broadcasting messages in PVM. This is accomplished by sending a message to an enrolled name with the instance number set to -1. The message is sent to all instances of enrolled processes with this name.

When finished, all PVM programs should call `leave()` to allow the *pvmds* to disconnect any sockets to the processes, and to allow the *pvmds* to keep track of which processes are running.

In PVM the `snd()` buffer remains valid until the next call to `initsend()`. Thus, the same message can be sent multiple times and can even be appended to with `put*()` commands between sends. On the receiving end, the processor is idle (or blocked) from the time it issues the `rcv()` command until a message satisfying the request arrives and is copied into the specified user buffer. Note that a program will not terminate if a `rcv()` command is never satisfied by an arriving message of the correct type. Moreover, only the type field distinguishes different messages in PVM. A common mistake made by new users is not using enough distinct types in their `snd()` and `rcv()` calls to uniquely identify different messages in their program. This often leads to nondeterministic behavior of the user's algorithm.

4. Fortran Routines

The Fortran routines are built on top of the C routines described above so their functionality is identical, but their names and arguments differ. Their naming convention is to prepend an "f" to the C routine name. The Fortran-to-C interface routines handle the distinctions between C and Fortran.

Several problems arose during the development of this interface. The first problem was the different calling conventions of C from Fortran by different compilers. For example, some compilers prepend C routine names with underscores; others do not. This problem was resolved by having *ifdef's* for each of the different calling conventions

`int` **barrier**`(char *barrier_name, int num)`	- blocks caller until num calls with same barrier name made. Returns < 0 if error.
`int` **enroll**`(char *component_name)`	- enrolls process in PVM and returns instance number ($>=0$) if successful or < 0 if error.
`int` **get[type]**`([type] *x, int num)`	- extracts num values of datatype [type] from received message and assigns it to x, eg. getnfloat(x, 5). Returns -1 if buffer empty. [type] must be nint, nfloat, ndfloat, ncplx, ndcplx, string, or bytes.
`int` **initiate**`(char *object_file, char *arch)`	- initiates a new process and returns instance number ($>= 0$) if successful or < 0 if error. If architecture is NULL, then PVM chooses an architecture.
`int` **initiatem**`(char *object_file, char *machine)`	- initiate a process on the specified machine and returns instance number ($>=0$) if successful or < 0 if error. If machine = ".", then initiating machine is used.
`void` **initsend**`()`	- initializes send buffer
`void` **leave**`()`	- process exiting PVM.
`int` **probe**`(int msgtype)`	- probe for message arrival of specified type or 'any' if msgtype=-1. Returns message type or -1 (not arrived).
`int` **probemulti**`(int num, int *msgtypes)`	- same as probe, but permits specifying an array of num message types.
`int` **put[type]**`([type] *ptr, int num)`	- inserts num values beginning at ptr into send buffer. Returns -1 if out of memory. [type] must be nint, nfloat, ndfloat, ncplx, ndcplx, string, or bytes.
`int` **rcv**`(int msgtype)`	- receives a message of specified type or 'any' if msgtype=-1 (Blocking). Returns actual message type.
`int` **rcvinfo**`(int *bytes, int *msgtype, char *component, int *instance)`	- returns the length, type, and sender of last rcv or probe. Returns -1 if rcv or probe not called.
`int` **rcvmulti**`(int num, int *msgtypes)`	- same as rcv, but permits specifying an array of num message types.
`void` **ready**`(char *event_name)`	- sends signal with specified (abstract) name.
`int` **snd**`(char *component, int instance, int msgtype)`	- sends message in send buffer to the specified instance of component. If instance $= -1$, then broadcast to all instances. Returns < 0 if error.
`int` **status**`(char *component, int instance)`	- returns 1 if specified component is active, 0 otherwise.
`int` **terminate**`(char *component, int instance)`	- terminates a specified component. Returns < 0 if error.
`void` **waituntil**`(char *event_name)`	- suspends caller until specified signal name occurs.
`void` **whoami**`(char *component, int *instance)`	- returns component name and instance number of caller.

Table 2: PVM C user routines

When a PVM application asks for an executable program to be started up on some machine, *pvmd* looks in `~/pvm/`*ARCH* on that machine for this executable program. Therefore, each user of PVM is responsible for having a directory `~/pvm/`*ARCH* on each machine he wishes to configure into PVM.

To configure a subset of machines into PVM, the user must first create a file containing the host names, one per line. The host running the *master pvmd* must be the first one in the file. (Blank lines and lines in the host file beginning with "#" are ignored.) Several options can be specified on each line after the hostname. If `pw` follows the hostname, then PVM will ask you for a password for this machine. Your default login name can be changed by specifying `lo=` *login name*. Using these options allows users to configure machines for which they have different login names and or passwords. The `pw` option is also needed for machines that do not allow the user to set up a *.rhosts* file. There is a third option available to allow the user to specify a nonstandard location for the *pvmd* executable on this machine. This option is `dx=` *location of pvmd*.

The user starts the pvmds with the command "`/tmp/pvm/pvmd filename`". The master pvmd will automatically start up a *pvmd* on each of the other hosts. Shortly after these have all been entered, all the pvmds will be running. If *pvmd* is invoked with a "-i" flag, i.e., "`/tmp/pvm/pvmd -i filename`", then the master *pvmd* will run a command editor that allows the user to query PVM about the status of processes and the present configuration of PVM. Typing "`help`" displays all the available commands.

The user can now run PVM programs. Currently, standard out and standard error streams of the slave *pvmds* (and any client programs they initiate) are passed back to the master *pvmd* and then to its standard out. Hostnames are added to the beginning of each output so that output from remote machines can be distinguished from local output.

To stop PVM the user should kill the master *pvmd* with the `quit` command in interactive mode or with control-C otherwise. Doing this will kill all the other pvmds and all processes enrolled in this PVM.

3. C Routines

Table 2 describes the communication and PVM interface routines available to C programs.

configure overlapping PVMs and execute several PVM applications simultaneously.

The second part of the package is a library of PVM interface routines. Application programs must be linked with this library to use PVM. Descriptions of the available routines are given in the next two sections.

If you are responsible for installing PVM on your network, then the top level of the PVM distribution should be in your home directory, and named `pvm`.

Check in the `~/pvm/src` directory for a subdirectory named for the architecture of your machine (ARCH). Table 1 contains a list of names currently in use in PVM version 2.3. If the correct one does not exist, you will have to make a new directory.

ARCH	Machine
PMAX	Dec/Mips arch (Ultrix)
SUN3	Sun 3
SUN4	Sun 4
RIOS	IBM/RS6000
SYMM	Sequent Symmetry
CRAY	Cray (UNICOS)
I860	Intel RX Hypercube
IPSC	Intel iPSC Hypercube
CM2	Thinking Machines CM2
AFX8	Alliant FX/8
TITN	Stardent Titan

Table 1: Current names used in PVM.

The name choice is arbitrary, but should be concise and accurate. PVM will be using the name to identify the machine type it is running on.

If you need to make an architecture directory, copy `Makefile.generic` to the new directory and customize it for your architecture. When customizing the *Makefile*, the only thing you definitely have to set is the ARCH definition. You may additionally need to define certain switches from the list given, or compiler/loader flags.

Make the PVM daemon (*pvmd*) and user library (*libpvm.a*) by typing "`make`" in the architecture directory of the machine you are logged onto. Once *pvmd* is made for each of the architectures on your network, pvmd needs to be installed as `/tmp/pvm/pvmd` on every host machine. Typically, this is done by setting up "soft" links to a centralized executable in `~/pvm/src/`*ARCH*`/pvmd`, but if this is not possible, then *pvmd* must be copied to `/tmp/pvm/pvmd` on each machine.

Application programs view PVM as a general and flexible parallel computing resource that supports a message-passing model of computation. This resource may be accessed at three different levels: the *transparent* mode in which component instances are automatically located at the most appropriate sites, the *architecture-dependent* mode in which the user may indicate specific architectures on which particular components are to execute, and the *low-level* mode in which a particular machine may be specified. Such layering permits flexibility while retaining the ability to exploit particular strengths of individual machines on the network. The PVM user interface is strongly typed; support for operating in a heterogeneous environment is provided in the form of special constructs that selectively perform machine-dependent data conversions where necessary. All communication done inside PVM uses the external data representation standard, XDR [1]. Inter-instance communication constructs include those for the exchange of data structures as well as high-level primitives such as broadcast, barrier synchronization, mutual exclusion, and rendezvous.

Application programs under PVM may possess arbitrary control and dependency structures. In other words, at any point in the execution of a concurrent application, the processes in existence may have arbitrary relationships between each other and, further, any process may communicate and/or synchronize with any other. This is the most unstructured form of crowd computation, but in practice a significant number of concurrent applications are more structured. Two typical structures are the tree and the "regular crowd" structure. We use the latter term to denote crowd computations in which each process is identical; frequently such applications also exhibit regular communication and synchronization patterns. Any specific control and dependency structure may be implemented under the PVM system by appropriate use of PVM constructs and host language control-flow statements.

2. Installation

The PVM package is composed of two parts. The first part is a daemon, called *pvmd*, that resides on all the computers on the network. *Pvmd* is designed so any user with a valid login can install this daemon on a machine. When a user wants to run a PVM application, he initiates some subset of the installed *pvmds*. This collection of running *pvmds* then defines the present PVM configuration for that user. Multiple users can

1. Introduction

This users' guide to PVM (Parallel Virtual Machine) version 2.3 contains examples and information needed for the straightforward use of PVM's basic features. Full documentation of all PVM options and error conditions will appear in *The PVM Reference Manual* (Documentation on all error conditions is presently included in the software distribution under `pvm/doc`).

PVM is a software package that enables concurrent computing on loosely coupled networks of processing elements. PVM may be implemented on a hardware base consisting of different machine architectures, including single CPU systems, vector machines, and multiprocessors. These computing elements may be interconnected by one or more networks, which may themselves be different (e.g. Ethernet, the Internet, and fiber optic networks). These computing elements are accessed by applications via a library of standard interface routines. These routines allow the initiation and termination of processes across the network as well as communication and synchronization between processes.

Application programs are composed of *components* that are subtasks at a moderately large level of granularity. During execution, multiple *instances* of each component may be initiated. Figure 1 depicts a simplified architectural overview of the PVM system.

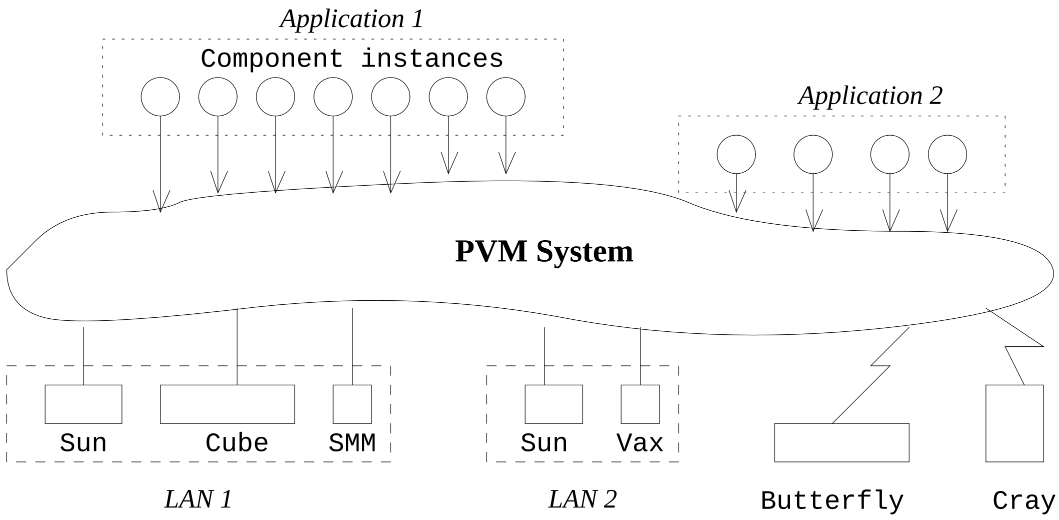

Figure 1: PVM Architecture Model

A USERS' GUIDE TO PVM
PARALLEL VIRTUAL MACHINE

Adam Beguelin
Jack Dongarra
Al Geist
Robert Manchek
Vaidy Sunderam

Abstract

This report is the PVM version 2.3 users' guide. It contains an overview of PVM and how it is installed and used. Example programs in C and Fortran are included.

PVM stands for Parallel Virtual Machine. It is a software package that allows the utilization of a heterogeneous network of parallel and serial computers as a single computational resource. PVM consists of two parts: a daemon process that any user can install on a machine, and a user library that contains routines for initiating processes on other machines, for communicating between processes, and syncronizing processes.

Contents

ORNL/TM-11826

Engineering Physics and Mathematics Division

Mathematical Sciences Section

A USERS' GUIDE TO PVM PARALLEL VIRTUAL MACHINE

Adam Beguelin *
Jack Dongarra *
Al Geist *
Robert Manchek *
Vaidy Sunderam **

* Oak Ridge National Laboratory
Mathematical Sciences Section
P.O. Box 2008, Bldg. 6012
Oak Ridge, TN 37831-6367

** Department of Math & Computer Science
Emory University
Atlanta, GA 30322

Date Published: July, 1991

Research was supported by the Applied Mathematical Sciences Research Program of the Office of Energy Research, U.S. Department of Energy.

Prepared by the
Oak Ridge National Laboratory
Oak Ridge, Tennessee 37831
operated by
Martin Marietta Energy Systems, Inc.
for the
U.S. DEPARTMENT OF ENERGY
under Contract No. DE-AC-05-84OR21400