USTL

PARALLEL PROGRAMMING WITH PVM

Philippe Marquet
marquet@lifl.fr
Laboratoire d'informatique fondamentale de Lille
Université des sciences et technologies de Lille
France

Maîtrise d'informatique

November 22, 1993
minor changes, November 24, 1994
revised, December 9, 1994
revised, January 10, 1995
revised, January 7, 1996
February 1998

Table of Contents

Thanks to

- This presentation of PVM is based on several presentations of PVM, mainly
  - *Parallel Programming with PVM*
    Giorgio Richelli IBM ECSEC, Rome Italy
  - *PVM 3 User's Guide and Reference Manual*
    Oak Ridge National Laboratory
    May 1994
  - *An Introduction to PVM*
    Robert Manchek
    Computer science department, university of Tennessee
    August 1993
- Other source of information on PVM
  - *PVM: Parallel Virtual Machine–A Users' Guide and Tutorial for Networked Parallel Computing*
    Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam
    The MIT Press, 1994
    available on URL:
    `ftp://netlib2.cs.utk.edu:pvm3/book/pvm-book.ps`
    `http://www.netlib.org/pvm3/book/pvm-book.html`
- PVM project home page:
  `http://www.epm.ornl.gov/pvm`

PVM—Parallel Virtual Machine

PVM is a software system that permits a heterogeneous collection of serial, parallel, and vector computers on a network to appear as one large computing resource

- It is a poor man's supercomputer allowing one to exploit the aggregate power of unused workstations during off-hours
- It is a metacomputer allowing multiple supercomputers distributed around the world to be linked together to achieve Grand Challenge performance
- It is an educational tool allowing people without access to parallel computers to teach/learn parallel programming courses
- It is a software tool allowing to efficiently use some of the distributed memory multi-processors (Cray T3D, IBM SP-n, Alpha farm...)

OVERVIEW

Table of Contents

Major advantages of PVM

- Easy to install – can be done by any user, can be shared between users
- Easy to configure – using your own host file
- Your configuration can overlap with other PVM user's without conflict
- Easy to write programs – standard message passing interface
- C and Fortran applications supported
- Multiple applications can run simultaneously on one PVM
- Package is small – requires only a few Mb disk space
- Public domain – available in source code
- Well known and accepted in the academic environment
- Not very widespread in the industry

Features

- Parallel machine control
- Process management
- Fault tolerance
- Signaling
- Supports all basic message passing programming constructs
- Multiprocessor integration (→ PVM uses native communication calls)

Heterogeneity

PVM supports heterogeneity at three levels:

Application Subtasks can exploit the architecture best suited to their solution

Machine Computers with different data formats and architecture (serial/parallel), different operating systems

Network Differents network, e.g. FDDI, Token Ring, Ethernet

Portability

- PVM source code is very portable across Unix machines, porting generally means setting options
- PVM currently runs on

80386/486 with Linux	Alliant FX/8
DEC Alpha/OSF-1	BBN TC2000
DEC Microvax	Convex
DECstation	Cray YMP, C90, T3D
DG Avion	IBM 3090
HP 9000/300	Intel Paragon
HP 9000 PA-RISC	Intel iPSC/2
IBM RS/6000	Intel iPSC/860
IBM RT	Kendall Square KSR-1
NeXT	MasPar MP-1, MP-2
Silicon Graphics IRIS	Sequent Symmetry
Multiprocessor SGI	Stardent Tritan
Sun 3	Thinking Machines CM-2, CM-5
Sun 4, Sparc	

- Portable to
  - Monoprocessor machines
  - Shared memory mutliprocessor machines
  - Parallel mutliprocessors

Evolution of PVM

- Version 1 written during summer 1989
  Oak Ridge National Laboratory (ORNL)
  Proof of concept – never releases
  Vaidy Sunderam & Robert Manchek
- Version 2 written March 1991
  University of Tennessee, Knoxville (UTK)
  Stable, robust version – First released publicly
- Version 3 written September 1992/February 1993
  Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, Vaidy Sunderam
  PVM 3 (version 3.3.10, dec. 1995)

Structure

PVM consists of

- A communication library: `libpvm3.a`
- A support software, pvmd3, a copy of which must be running, for each user, on each machine in the cluster
- A console management, `pvm`
- An additional library, `libfpvm3.a`
- An additional library, `libgpvm3.a` used for group management

Compiling a PVM application:

```
cc -o task myprog.c libpvm3.a
f77 -o task myprog.f libfpvm3.a libpvm3.a
```

For groups add `libgpvm3.a` before `libpvm3.a`

Structure: the daemons

The `pvmd3` daemon program

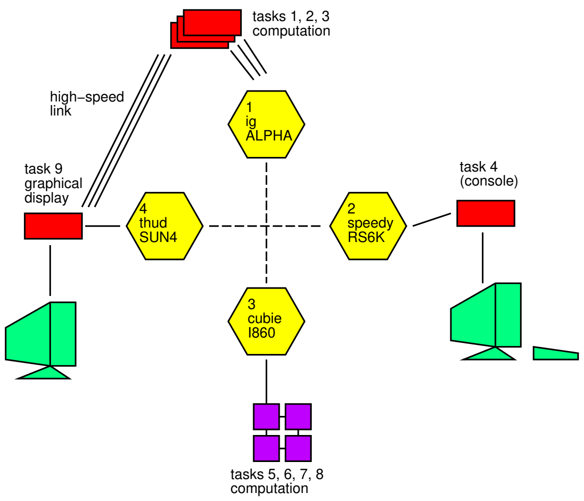
- Runs on each host of virtual machine
- Provides inter-host point of contact
- Creates and terminates tasks
- Maintains the database of running processes
- Provides fault detection
- Synchronizes processes with *events* and *barriers*
- Delivers messages
- Mainly a message router, but is also a source and sink of messages
- More robust than application components

Structure: the library

The communication library handles:

- Data conversions (when needed)
- Buffer packing and unpacking
- Communication with the local daemon

Structure of a virtual machine



How to get it

- Only one person at an organization needs to get and install PVM for everyone
- Anyone may install PVM (does not require special privileges)
- *Netlib*: software distribution
- Four ways:
  1. Send an email to `netlib@ornl.gov`
     with the message `send index from pvm3`
  2. Use *xnetlib*
     - A X-Window interface to *netlib*
     - Available by email to `netlib@ornl.gov`
       with the message `send xnetlib.shar from xnetlib`
  3. Use anonymous ftp on `ftp.netlib.org`
  4. Use WWW on URL
     - `http://www.netlib.org:/pvm3/`

USING PVM

Table of Contents

Building PVM

- To build PVM, chdir to top of distribution (pvm3) and type `make`
- This builds

`pvmd/pvmd3`	The PVM daemon
`pvm`	The console program
`libpvm3.a`	The C programming library
`lifpvm3.a`	The Fortran library

and installs them in `pvm3/lib/ARCH`

- Copy to/install on other machines if necessary

Installing PVM

- By default, PVM is installed in `$HOME/pvm3/`

`pvm3/lib`	pvmd, libraries, scripts
`pvm3/include`	Header files
`pvm3/bin`	Application binaries
`pvm3/doc`	Documentation
`pvm3/man`	Manual pages
`pvm3/examples`	Examples
`pvm3/conf`	Configuration files
`pvm3/src`	PVM sources

- It can be installed in a shared directory:

```
setenv PVM_ROOT /usr/local/pvm3
```

or

```
mkdir $HOME/pvm3
foreach d (lib include doc man examples conf src)
   ln -s /usr/local/pvm3/$d $HOME/pvm3/$d
end
mkdir $HOME/pvm3/bin
```

- `pvm3/bin/` directory is generally private

Programming concepts

Host A physical machine, e.g.

- A Unix workstation
- An hypercube

Virtual machine A meta-machine composed of one or more hosts

Process A program, data, stack, etc., e.g.

- A Unix process
- A Node program

Task A PVM process – The smallest unit of computation

TID The unique (per virtual machine) identifier associated with each task

PVMD The PVM daemon, e.g. `pvm3/lib/ARCH/pvmd`

Message An ordered list of data sent between tasks

Group An ordered list of tasks assigned a symbolic name
Each task has an unique index in the group
Any task may be in zero or more groups

Starting a parallel virtual machine

Three ways to initialize PVM:

1. Use the `pvm` console
2. Use `xpvm`, a X-Window interface to the `pvm` console
3. Start the `pvmd` daemon

The pvm console

- Allows interactive adding and deleting of hosts to the virtual machine
- Allows interactive starting and killing of PVM processes
- Starts by the `pvm` command
  Prints the prompt `pvm>`

- Accepts the commands :

  help to get informations about the interactive commands

  conf lists the configuration of the virtual machine (host name, pvmd task ID, architecture type...)

  add <**host-name-list**> adds these hosts to the virtual machine

  delete <**host-name-list**> deletes these hosts. PVM processes still running on these hosts are lost

```
farm4-giga% pvm
pvm> conf
1 host, 1 data format
                    HOST     DTID     ARCH   SPEED
              farm4-giga    40000    ALPHA    1000
pvm> add farm5-giga
1 successful
                    HOST     DTID
              farm5-giga    80000
pvm> conf
2 hosts, 1 data format
                    HOST     DTID     ARCH   SPEED
              farm4-giga    40000    ALPHA    1000
              farm5-giga    80000    ALPHA    1000
pvm> delete farm5-giga
1 successful
                    HOST  STATUS
              farm5-giga  deleted
pvm> conf
1 host, 1 data format
                    HOST     DTID     ARCH   SPEED
              farm4-giga    40000    ALPHA    1000
```


spawn starts a PVM application

kill can be used to terminate any PVM process

ps lists the processes currently on the virtual machine (location, task ID, parent's task ID)

pstat shows status of a single PVM process

```
pvm> spawn psdot
1 successful
tc0001
pvm> ps
                    HOST      TID   FLAG 0x COMMAND
pvm> ps -a
                    HOST      TID   FLAG 0x COMMAND
              farm5-giga    c0001     6/c,f psdot
pvm> ps -al
      HOST      TID     PTID    PID   FLAG 0x COMMAND
farm5-giga    c0001   (cons)  19331     6/c,f psdot
pvm> pstat c0001
t   c0001  run
```

- Spawn a PVM task:
  - If `pvm3/bin/$PVM_ARCH/task` exists
    - ▹ Spawn from the PVM console
    - ▹ Spawn from the shell
      `pvm3/bin/$PVM_ARCH/task`
  - From the shell:
    `/where/the/binaries/are/task`

reset kills all PVM processes except console and resets all the internal PVM tables and message queues

quit exits console leaving daemons and PVM jobs running

halt kills all PVM processes including console and then shut down PVM

```
pvm> quit

pvmd still running.
farm4-giga% pvm
pvmd already running.
pvm> conf
2 hosts, 1 data format
                    HOST     DTID     ARCH   SPEED
              farm4-giga    40000    ALPHA    1000
              farm5-giga    c0000    ALPHA    1000
pvm> halt
libpvm [t40003]: mxfer() EOF on pvmd sock
farm4-giga%
```

- The console may be started and stopped multiple times on any of the hosts on which PVM is running

Configuring a virtual machine

- The user may start `pvmd` with a hostfile then start `pvm` if desired
- The first daemon then starts to replicate itself on all the nodes
- The hostfile defines a configuration
- A basic hostfile looks like

```
# configuration used for my run
ms2
ms3
mpp.lifl.fr
```

- Example:

```
mpp.lifl.fr% pvmd mypvmhostfile
```


- Several options can be specified

lo=userid allows the user to specify an alternate login name for this host
(default: same, i.e. `$USER`)

so=pw will cause PVM to prompt for the password on this host
(default: don't ask; use `rsh`)

dx=location_of_pvmd specifies a location other than the default
(default: `pvm3/lib/pvmd`)

ep=paths_to_user_executables allows the user to specify a series of paths to search down to find the requested files to spawn on this host
The default value is `$HOME/pvm3/bin/%`
`%` = `$PVM_ARCH`

- Example of configuration file

```
# Comment lines start with #
ms2.ens.lifl.fr
mpp.lifl.fr   dx=/usr/local/pvm3/lib/MASPAR/pvmd

# set default option for following host with *
* ep=~/pvmdir/bin/SUN4
budweiser.lifl.fr
lutece.lifl.fr

# replace default options with new values
* lo=phm  ep=/usr/local/pvm/bin/%
corona.lifl.f
adelscott.lifl.fr

# machine for adding later are specified with &
# these only need listing if options are required
& ms1.ens.lifl.fr ep=~/pvm3/bin/SUN4:~/bin
& watou.lifl.fr dx=/usr/local/pvm3/lib/DEC/pvmd
```


XPVM

- Programming environment for PVM
- X-Window interface to PVM
- Allows:
  - Debugging
  - Messages tracing
  - Performance analysis

Tools

- Several tools are available with PVM
- In the directory `pvm3/lib`
- `pvmgetarch` returns the architecture of the current host
  May be used to initialize environment
- `aimk`, architecture independent `make`

```
#       Make wrapper for multiple arch. builds.
#
#       Automatically sets PVM_ARCH and PVM_ROOT for make to use.
#
#       Action depends on makefile locations:
#       1.  If $PVM_ARCH/Makefile or $PVM_ARCH/makefile exists,
#               chdir to $PVM_ARCH and exec make there.
#
#       2.  Else if ./Makefile.aimk exists,
#               chdir to $PVM_ARCH and exec make
#               with -f $PVM_ROOT/conf/$PVM_ARCH.def
#               and -f ../Makefile.aimk PVM_ARCH=$PVM_ARCH
#
#       3.  Else will simply exec make in cwd.
#
#       usage:
#           aimk [-here] [-nomk] [ make args ... ]
#
```


PROGRAMMING WITH PVM, THE USER INTERFACE

Table of Contents

Introduction

- In PVM, all the tasks are identified by an integer supplied by the local daemon: the *tid*
- All the PVM routines are written in C
- C++ applications can linked to the PVM library
- Fortran application can call these routines through a Fortran interface (translates arguments: reference → value, takes into account naming convention to call C from Fortran...)
- For all functions, a negative value is returned in case of errors
- PVM functions begins with `pvm_` in C and with `pvmf` in Fortran

Process management primitives

- `int tid = pvm_mytid (void)`
  `call pvmfmytid (tid)`
  - Enrolls this process into PVM on its first call and generates a unique `tid`
  - It returns the `tid` of this process and can be call multiple times
  - `pvm_mytid()` **must** be call before any other calls in an application
  - If PVM has not been started before, it returns an error code
  - Example in C:

```
if (me = pvm_mytid () < 0)
    printf ("error in enrolling\n") ;
else
    printf ("success, I am %d\n", me) ;
```

  - Example in Fortran:

```
CALL PVMFMYTID (ME)
IF (ME .LT. 0) THEN
    WRITE (*,*) 'error in enrolling'
ELSE
    WRITE (*,*) 'success, I am ', ME
ENDIF
```


- `int info = pvm_exit (void)`
  `call pvmfexit (info)`
  - Tells the local daemon that the process in leaving PVM
  - Does not kill the process, which can continue to perform task just like any other Unix process
  - It should be called by all PVM processes before they stop or exit for good
  - It must be called by processes that were not started with `pvm_spawn`;
    otherwise PVM will not know if the process has crashed or exited normally
  - Example in C:

```
/* program done */
pvm_exit () ;
exit () ;
```

  - Example in Fortran:

```
C     **program done**
      CALL PVMEXIT (INFO)
      STOP
```

- `int numt = pvm_spawn (char *task, char **argv, int flag, char *where, int ntask, int *tids)`
  `call pvmfspawn (task, flag, where, ntask, tids, numt)`
  - It starts up `ntask` copies of an executable file `task` on the virtual machine:
    `pvm3/bin/$PVMARCH/task argv`
  - `argv` is a pointer to a `NULL` terminated array of arguments to `task`
  - The `flag` argument is used to specify options, and is sum of
    - `PvmTaskDefault` **0** PVM chooses where to spawn processes (in a cyclic fashion) (`where` is ignored)
    - `PvmTaskHost` **1** the `where` argument specifies a particular host to spawn on
    - `PvMTaskArch` **2** the `where` argument specifies a type of architecture to spawn on
    - `PvmTaskDebug` **4** starts these processes up under debugger
    - `PvmTaskTrace` **8** the PVM calls in these processes will generate trace data
    - `PvmMppFront` **16** starts the processes on MPP front-end
    - `PvmHostCompl` **32** uses complement host set
  - On return, `numt` is set to the number of tasks successfully spawned or an error code if no tasks could be started
  - It returns a vector of the spawned tasks' `tids`

- On multiprocessors, restrictions apply
- Examples in C

```
numtr = pvm_spawn ("host", 0, PvmTaskHost,
                   "ms3", 1, &tid[0]) ;
numtr = pvm_spawn ("host", 0,
                   PvmTaskHost+PvmTaskDebug,
                   "ms3", 1, &tid[0]) ;
numtr = pvm_spawn ("node", 0, PvmTaskArch,
                   "SUN4", 1, &tid[i]) ;
numtr = pvm_spawn ("FEM1", args, 0, 0, 16, tids) ;
numtr = pvm_spawn ("pde", 0, PvmTaskHost,
                   "paragon.lifl.fr", 512, tids) ;
```

- Examples in Fortran

```
CALL PVMFSPAWN ("host", PVMTASKHOST,
                "ms3", 1, TID, NUMTR)
CALL PVMFSPAWN ("host",
                PVMTASKHOST+PVMTASKDEBUG,
                "ms3", 1, TID, NUMTR)
CALL PVMFSPAWN ("node", PVMTASKARCH,
                "SUN4", 1, TID, NUMTR)
CALL PVMFSPAWN ("FEM1" 0, 0, 16, TIDS, NUMTR)
CALL PVMFSPAWN ("pde" PVMTASKHOST,
                "paragon.lifl.fr", 512,
                TIDS, NUMTR)
```



```
      program firstpvm
      include 'fpvm3.h'
      integer nprocs, info, tids (0:20), mytid
      call pvmfmytid (mytid)
      call pvmfparent (tids(0))
      if (tids (0) .lt. 0) then
C***  In the master code ***
         tids(0) = mytid
         read (*,*), nprocs
         call pvmfspawn ('a.out', PVMTASKDEFAULT,
    &                    '*', nproc-1, tids(1), numt)
         ....
      else
C***  In the slave code ***
         ....
      end if
      call pvmfexit (info)
      end
```


- `int info = pvm_kill (int tid)`
  `call pvmfkill (tid, info)`
  - Kills some other PVM task identified by `tid`
  - On Unix system, sends a terminate (SIGKILL) signal to the PVM process identified by `tid`
  - It is not designed to kill the calling task (which could be accomplished by calling `pvm_exit()` followed by `exit()`)
  - A value less than zero for `info` indicates an error

Information

- `int tid = pvm_parent (void)`
  `call pvmfparent (tid)`
  - Returns the `tid` of the process that spawned this task
  - Or the value `PvmNoParent` if not created by `pvm_spawn()`
  - Example

```
      CALL PVMFPARENT (PTID)
      IF (PTID .EQ. PvmNoParent) THEN
C     I have been spawned from the pvm console
C     or from the shell
```


- `int status = pvm_pstat (int tid)`
  - Returns the status of a PVM task identified by `tid`
  - `status` is `PvmOk` if the task is running
    `PvmNoTask` if not
    `PvmBadParam` if the `tid` is bad
  - Example

```
ptid = pvm_parent () ;
status = pvm_stat (ptid) ;
```

- `int status = pvm_mstat (char *host)`
  - Returns `PvmOk` if the machine `host` is running
    `PvmHostFail` if `host` is unreachable
    `PvmNoHost`if `host` is not in the virtual machine
  - Can be useful when implementing application level fault tolerance
  - Example

```
status = pvm_mstat ("ms2.ens.lifl.fr") ;
status = pvm_mstat ("ms2") ;
```


- `int info = pvm_config (int *nhost, int *narch, struct hostinfo **hostp)`
  - Returns information about the virtual machine
  - Information is similar to that available from the `conf` command of the pvm console
  - Returns the number of hosts (pvmd) in `nhost`
  - Returns the number of different data formats being used in `narch`
  - Returns a pointer to an array of structures which contain information about each host/pvmd

```
struct hostinfo {
   int  hi_tid ;       /* task ID */
   char *hi_name ;     /* host machine */
   char *hi_arch ;     /* architecture of the host
                                          machine */
   int hi_speed ;      /* relative speed */
};
```

- Example

```
struct hostinfo *hi ;
int nh, na ;
pvm_config (&nh, &na, &hi) ;
```


- `int info = pvm_tasks (int which, int *ntask, struct taskinfo **taskp)`
  - Returns information about the PVM tasks running on the virtual machine
  - Information is similar to that available by the `ps` command of the pvm console
  - The integer `which` specifies the tasks to return information about:
    - ▷ 0,which means all tasks
    - ▷ A pvm daemon tid, which means tasks running on that host
    - ▷ A task tid, which means just the given task
  - The number of tasks is returned in `ntask`
  - `taskp` is a pointer to an array of `taskinfo` structures of size `ntask`

```
struct taskinfo {
    int  ti_tid ;    /* task ID */
    int  ti_ptid ;   /* parent tid */
    int  ti_host ;   /* pvmd tid */
    int  ti_flag ;   /* status flag */
    char *ti_a_out ; /* task executable name */
};
```

  - The flag indicates if the task is waiting for a message, waiting for the pvmd, or running
  - `a.out` only for “spawned” tasks

- Example

```
/* info for the tasks on the current host */
me = pvm_mytid () ;
(void) pvm_tasks (me, &nb, &taskp) ;
(void) pvm_tasks (taskp[0]. ti_host, &nb, &taskp) ;
```


Dynamic configuration

Add or delete hosts on a parallel virtual machine

- C interface
  `int info = pvm_addhosts (char **hosts, int nhost, int *infos)`
  `int info = pvm_delhosts (char **hosts, int nhost, int *infos)`
  - Add or delete a set of `nhost` hosts in the virtual machine
  - `hosts` is an array of pointers to character strings containing the names of the machine to be added or deleted
  - The argument `infos` is an array of length `nhost` that contains the status code for each host being added or deleted It allows the user to check if only one of a set of hosts caused a problem rather than trying to add or delete the entire set of hosts again
  - Expensive operation that require the synchronization of the entire virtual machine

- Fortran interface
  `call pvmfaddhost (host, info)`
  `call pvmfdelhost (host, info)`
  - Add or delete a the specified host in the virtual machine
  - `host` is character string containing the name of the machine to be added or deleted
  - The argument `info` contains the status code for the host being added or deleted
  - Expensive operation that require the synchronization of the entire virtual machine

- Example in C

```
static char * hosts [] = {
   "mpp.lifl.fr",
   "alpha.lifl.fr",
} ;
int status [2], info, i ;

info = pvm_addhosts (hosts, 2, status) ;
if (info < 0)
  for (i=0; i<2; i++)
    if (status [i] < 0)
      printf ("error (%d) in adding %s\n",
              status [i], hosts [i]) ;
```

- Example in Fortran

```
CALL PVMFADDHOST ('mpp.lifl.fr', INFO)
IF (INFO .LT. 0) THEN
  WRITE (*,*) 'Error in adding mpp.lifl.fr', INFO
ENDIF
CALL PVMFADDHOST ('alpha.lifl.fr', INFO)
IF (INFO .LT. 0) THEN
  WRITE (*,*) 'Error in adding alpha.lifl.fr', INFO
ENDIF
```

Signaling

- `int info = pvm_sendsig (int tid, int signum)`
  - Sends a Unix signal `signum` to the PVM task identified by `tid`
  - Example, on Unix hosts

    ```
    pvm_kill (tid) ;
    ```

    is equivalent to

    ```
    pvm_sendsig (tid, SIGKILL) ;
    ```


- `int info = pvm_notify (int what, int msgtag, int ntask, int *tids)`
  - Causes a message with a given `msgtag` to be sent to a set of tasks specified by `ntask` and `tids` when some event happen in the virtual machine
  - The argument `what` specifies the event type:

    `PvmTaskExit` notify if a task exits

    `PvmHostDelete` notify if a host is deleted (or fails)

    `PvmHostAdd` notify if a host is added
  - Message body contains data specific to the event

    `PvmTaskExit` Tid of the task

    `PvmHost Delete` pvmd-tid of the host

    `PvmHostAdd` count of the new hosts and pvmd-tid
  - Example: A console that updates its `hostinfo` information on each addition/deletion of a host in the virtual machine
    - ▹ The console has an initial state of the virtual machine
    - ▹ This console will receive a message every time a host is deleted or added
    - ▹ Based on the `msgtag` of his message, updates the state of the virtual machine

Error messages

- `int info = pvm_perror (char *msg)`
  `call pvmperror (msg, info)`
  - Prints the error status of the last PVM call
  - It is similar to the Unix `perror()` routine
  - Example in C

    ```
    if (pvm_send (tid, msgtag) < 0)
        pvm_perror ("Step 6");
    ```
  - Example in Fortran

    ```
    CALL PVMFSEND (TID, MSGTAG, IERR)
    IF (IERR .LT. 0) THEN
       PVMFPERROR ("Step 6", INFO);
    ```


- `int oldval = pvm_setopt (int what, int val)`
  `call pvmfsetopt (what, val, oldval)`
  - Sets options of the PVM system
  - `what = PvmAutoErr` automatic error message printing on or off for subsequent PVM call by this process
  - `val` defines whether detection is to be turned on (1), or off (0)
  - When `val` value is 2, the process prints the error message and then dies
  - All `stdout` and `stderr` messages are placed in the file `/tmp/pvml.<uid>` on the master pvmd's host
  - Example:

    ```
    CALL PVMFSETOPT (PVMAUTOERR, 1, INFO)
    ```

- `int info = pvm_catchout (FILE *ff)`
  `call pvmfcatchout (yesno, info)`
  - Causes the characters printed to `stdin` and `stdout` by the sapwned task to be redirect to the file `ff`
  - In Fortran, allows (`yesno=1`) or not (`yesno=0`) output to be redirect to the parent's task `stdout`

Message passing

- Sending a message is composed of three steps
  1. A send buffer must be initialized
  2. The message must be "packed" in this buffer
  3. The entire buffer is
     - Send to another task, or
     - Multicast to a list of tasks, or
     - Broadcast to a group (if using the group library)

     Any number of times without repacking
- Send operation is non-blocking:
  - Returns ASAP
  - Does not implies receipt
- Messages sent between any two tasks A and B will arrive in the order they are sent

- A message is received by
  1. Calling either
     - a blocking, or
     - a non-blocking

     receive routine
  2. "Unpacking" each of the packed items from the buffer
- The received routines can be set to accept
  - Any message, or
  - Any message from a specified source, or
  - Any message with a specified message tag, or
  - Only messages with a given message tag from a given source
- PVM also allows users to define their own receive contexts

Message buffers

- Message buffer routines are required only if the user wishes to manage multiple message buffers

  At any given moment, there is only one *active* send buffer and one *active* received buffer

  The packing, sending, receiving, and unpacking routines only affect the *active* buffer
- `int bufid = pvm_initsend (int encoding)`
  `call pvmfinitsend (encoding, bufid)`
  - Basic buffer manipulation
  - Clears the send buffer and prepares it for packing a new message
  - Specifies the encoding method used for packing messages:

    `PvmDataDefault` XDR encoding is used if the virtual machine configuration is heterogeneous, otherwise no encoding is done

    `PvmDataRaw` No encoding is done

    `PvmDataInPlace` Data left in place. Buffer only contains size and pointers to the items to be send

- `int bufid = pvm_mkbuf (int encoding)`
  - Creates a new empty send buffer
  - `encoding` is one of `PvmDataDefault`, `PvmDataRaw`, `PvmDataInPlace`
  - Returns the buffer ID
- `int info = pvm_freebuf (int bufid)`
  - Disposes the buffer `bufid`
  - Should be done after a message has been sent and is no longer needed
  - Then call `pvm_mkbuf()` to create a buffer for a new message if required
  - Neither of these is required when using `pvm_initsend()`
- `int bufid = pvm_getsbuf (void)`
  - Returns the active send buffer ID
- `int bufid = pvm_getrbuf (void)`
  - Returns the active receive buffer ID

- `int oldbuf = pvm_setsbuf (int bufid)`
  - Sets the active send buffer to `bufid`
  - Saves the state of the previous buffer
  - Returns the previous active buffer ID
- `int oldbuf = pvm_setrbuf (int bufid)`
  - Sets the active receive buffer to `bufid`
  - Saves the state of the previous buffer
  - Returns the previous active buffer ID
- If `bufid` is set to 0 in `pvm_set?buf()`, then the present buffer is save and there is no active buffer

  This can be used to save the present state of an application's messages

  So that a math library or graphical interface which also used PVM messages will not interfere with the state of the application's buffers
- Example, forward a message without repacking it

```
oldid = pvm_setsbuf (0) ;
bufid = pvm_recv (src, tag) ;
info  = pvm_send (dst, tag) ;
info  = pvm_setrbuf (oldid) ;
```


Packing data

- Each of the following routines packs an array of the given data type into the active send buffer
- They can be called multiple times to pack a single message
- Thus, a message can contain several arrays each of different data type
- An application must unpack the messages exactly like they were packed

C interface

- C structures must be passed by packing their individual elements
- The arguments for each of the routines of the C interface are
  - a pointer to the first item to be packed
  - `nitem` which is the total number of items to pack from this array
  - `stride` which is the stride to use when packing

- `pvm_pkstr()` is an exception; its packs a NULL terminated character string and so does not need `nitem` nor `stride`

```
int info = pvm_pkbyte   (char   *cp, int nitem, int stride)
int info = pvm_pkcplx   (float  *fp, int nitem, int stride)
int info = pvm_pkdcplx  (double *dp, int nitem, int stride)
int info = pvm_pkdouble (double *dp, int nitem, int stride)
int info = pvm_pkfloat  (float  *fp, int nitem, int stride)
int info = pvm_pkint    (int    *np, int nitem, int stride)
int info = pvm_pklong   (long   *np, int nitem, int stride)
int info = pvm_pkshort  (short  *np, int nitem, int stride)
int info = pvm_pkstr    (char   *cp)
```

Fortran interface

- call pvmfpack (datatype, data, item, stride, info)
  - datatype is one of

STRING	0	REAL4	4
BYTE1	1	REAL8	6
INTEGER2	2	COMPLEX8	5
INTEGER4	3	COMPLEX16	7

  - data is the first item to be packed
  - nitem is the total number of item to be packed
  - stride is the stride to use when packing

- Example/exercise: consider a $N \times N$ matrix M
  - Two PVM tasks want to exchange
    - ▷ The matrix
    - ▷ The rth row of the matrix
    - ▷ The cth column of the matrix
    - ▷ The diagonal of the matrix
    - ▷ The r_l to r_u rows of the matrix
    - ▷ The c_l to c_u columns of the matrix

Unpacking data

- The following routines unpack (multiple) data types from the active receive buffer
- In an application, they should match their corresponding pack routines in
  - Types
  - Number of items
- C interface

```
int info = pvm_upkbyte   (char   *cp, int nitem, int stride)
int info = pvm_upkcplx   (float  *fp, int nitem, int stride)
int info = pvm_upkdcplx  (double *dp, int nitem, int stride)
int info = pvm_upkdouble (double *dp, int nitem, int stride)
int info = pvm_upkfloat  (float  *fp, int nitem, int stride)
int info = pvm_upkint    (int    *np, int nitem, int stride)
int info = pvm_upklong   (long   *np, int nitem, int stride)
int info = pvm_upkshort  (short  *np, int nitem, int stride)
int info = pvm_upkstr    (char   *cp)
```

- Fortran interface

  call pvmfunpack (datatype, data, nitems, stride, info)

Sending and receiving data

- int info = pvm_send (int tid, int msgtag)
  call pvmfsend(tid, mstag, info)
  - Labels the message with an integer identifier msgtag
  - Sends it immediately to the process tid
- int info = pvm_mcast (int *tids, int ntask, int msgtag)
  call pvmfmcast (ntask, tids, mstag, info)
  - Labels the message with an integer identifier msgtag
  - Broadcasts the message to the ntask tasks specified in the array tids

- `int bufid = pvm_nrecv (int tid, int msgtag)`
  `call pvmfnrecv (tid, mstag, info)`
  - Non-bloking receive routine
  - If the requested message has not arrived, returns 0
  - Can be called multiple times for the same message to check
  - If a message with label `msgtag` has arrived from `tid`, places this message in a new active receive buffer which is created
  - The previous active receive buffer is cleared, unless it has been saved
  - A value of -1 in `msgtag` or `tid` matches anything (wildcard)
- `int bufid = pvm_recv (int tid, int msgtag)`
  `call pvmfrecv (tid, mstag, info)`
  - Blocking receive routine
  - It will wait until a message with label `msgtag` has arrived from `tid`
  - Otherwise acts like `pvm_nrecv()`

- Example:

```
C     ** sender **
C        send a int N and an array of N reals
      CALL PVMFINITSEND (PVMDEFAULT, IBUF)
      CALL PVMFPACK (INTEGER4, N, 1, 1, IERR)
      CALL PVMFPACK (REAL8, Y(1), N, 1, IERR)
      CALL PVMFSEND (TIDS(I), METAG, IERR)

C     ** receiver **
      CALL PVMFRECV (-1, METAG, IBUF)
      CALL PVMFUNPACK (INTEGER4, N, 1, 1, IERR)
      CALL PVMFUNPACK (REAL8, Z(1), N, 1, IERR)
```


- `int info = pvm_bufinfo (int bufid, int *bytes, int *msgtag, int *tid)`
  `call pvmfbufinfo (bufid, bytes, mstag, tid, info)`
  - Returns informations about the message in the buffer `bufid`:
    - ▷ Actual message tag
    - ▷ Actual source tid
    - ▷ Length in bytes
  - It can be used to determine the label and source of a message that was received with wildcards
  - Example

```
bufid = pvm_recv (-1, -1) ;
info = pvm_bufinfo (bufid, &bytes, &type, &source) ;
printf ("message from %d, tag %d\n", source, type) ;
```


- `int bufid = pvm_probe (int tid, int msgtag)`
  `call pvmfprobe (tid, mstag, bufid)`
  - Checks if a message with label `msgtag` has arrived from `tid`
  - `tid` and `msgtag` may be wildcarded by the value -1
  - If the requested message has not arrived, returns 0
  - Otherwise returns the buffer ID for the message, but does not “received” it
  - After a message has arrived, `pvm_recv()` must be call before the message can be unpacked
  - In addition, `pvm_bufinfo()` can be called with the returned buffer ID to determine information about the message before receiving it
  - Example

```
tid = pvm_parent () ;
msgtag = 4 ;
arrived = pvm_probe (tid, msgtag) ;
if (arrived)
  info = pvm_bufinfo (arrived, &len, &tag, &tid) ;
else
  /* go to other computing */
```

- `int (*old)() = pvm_recvf (`
  `int (*new)(int buf, int tid, int tag))`
  - No Fortran interface
  - Modifies the receive context used by the receive functions
  - Can be used to extend PVM
  - The default receive context is to match on source and message tag
  - This can be modified by any user defined comparison function
  - The matching function returns preference to `pvm_recv()`

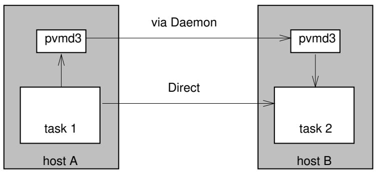
    < 0 return immediately with this error code
    0 do not pick this message
    1 pick this message and stop now
    > 1 pick this highest preference after scanning them all

- Example, implementing probe with `pvm_recvf()`

```
#include "pvm3.h"

static foundit = 0 ;

static int foo_match (int mid, int tid, int code)
{
  int t, c, cc ;

  if ((cc = pvm_bufinfo (mid, (int *) 0, &c, &t)) < 0)
    return cc;
  if ((tid == -1 || tid == t)
  && (code == -1 || code == c))
    foundit = 1 ;
  return 0 ;
}

int probe (int src, int code)
{
  int (*omatch)(), cc ;

  omatch = pvm_recvf (foo_match) ;
  foundit = 0 ;
  if ((cc = pvm_nrecv (src, code) < 0)
    return cc ;
  pvm_recvf (omatch) ;
  return foundit ;
}
```


Communication realization

- `int oldval = pvm_setopt (int what, int val)`
  `call pvmfsetopt (what, val, oldval)`
  - Sets options of the PVM system
  - `what = PvmRoute` advises PVM to set up direct task-to-task links for all subsequent communications or not
  - By default, PVM routes messages via daemons (UDP protocol)
  - Direct task-to-task communication are possible (using TCP)



  - `val` specifies one of

    `PvmDontRoute` Doesn't allow direct links to this task
    `PvmAllowDirect` Allows but doesn't requests direct links (It is the default)
    `PvmRouteDirect` Requests direct links

	Route	Dont	Allow
Route	direct	—	direct
Dont	—	—	—
Allow	direct	—	—

- Once a direct route established between tasks, both will use the route
- Once a link is established, it remains until the application finishes
- If a link can not be established
  - ▹ Because one of the two tasks has requested `PvmDontRoute`
  - ▹ Or because no resources are available

  The default route though the PVM daemons is used
- `pvm_setopt (PvmRoute...)` can be called multiple times to selectively establish direct links
- But it is typically set only once near the beginning of each task
- Performance of direct task-to-task is better
- Draw backs are
  - ▹ Lack of scalability
  - ▹ Startup time

Dynamic process groups

- Dynamic process group functions are build on the top of the core PVM routines
- Needs to link with a separate library: `libgpvm3.a`
- The pvmd does not perform the group functions
- A group server handles these functions; it is automatically started when a group function is first invoked
- New feature in PVM (since version 3.0) → experimental, feedback is encouraged
- Debate:
  - Static or dynamic grouping
  - Only tasks in a group can call group functions
  - Efficiency/reliability
- Any PVM task may
  - Join or leave any group at any time
  - Without having to inform any other task
- Tasks can broadcast messages to groups of which they are not a member
- Broadcast message are not send back to the sender

- `int inum = pvm_joingroup (char *group)`
  `call pvmfjoingroup (group, inum)`
  `int info = pvm_lvgroup (char *group)`
  `call pvmflvgroup (group, info)`
  - Allows a task to join a user name group
  - The first call to `pvm_joingroup()`creates a group with name group
  - A task can join multiple group
  - `pvm_joingroup()` returns the instance number (`inum`) of the process in this group
  - Instance numbers run from 0 to the number of group member minus 1
  - Instance numbers are recycled so a task joining a group will get the lowest available instance number
  - So there is no guarantee that the instance numbers are in a continuous set

- `int tid = pvm_gettid (char *group, int inum)`
  `call pvmfgettid (group, inum, tid)`
  - Returns the `tid` of the process with a given group name and instance number
- `int inum = pvm_getinst (char *group, int tid)`
  `call pvmfgetinst (group, tid, inum)`
  - Returns the instance number of the task `tid` in the specified group
- `int size = pvm_gsize (char *group)`
  call pvmfgsize (group, size)—
  - Returns the number of members in the specified group

- `int info = pvm_barrier (char *group, int count)`
  `call pvmfbarrier (group, count, info)`
  - The calling process blocks until `count` member of the group have call `pvm_barrier()`
  - In general, `count` should be the total number of members of the group
  - The logical function of the routine is to provide a group synchronization
  - `count` is required because of the dynamic aspect of PVM groups
  - The calling process must be a member of the group
  - All the calling processes must set the same `count`
  - If `count` is set to -1, PVM uses the value of `pvm_gsize()`. Useful if the group is not changing.
  - Example

```
inum = pvm_joingroup ("worker") ;
...
info = pvm_barrier ("worker", pvm_gsize ("worker")) ;
```

- `int info = pvm_bcast (char *group, int msgtag)`
  `call pvm_bcast (group, msgtag, info)`
  - Labels the message with the identifier `msgtag`
  - Broadcasts the message to all the tasks in the specified group
  - "All the tasks" is defined to be those tasks the group server thinks are in the group when the routine is called
  - Example

```
info = pvm_initsend (PvmDataRaw) ;
info = pvm_pkint (array, 10, 1) ;
msgtag = 5 ;
pvm_bcast ("worker", msgtag) ;
```


Fortran interface

- Most of the C functions are available in Fortran
- Fortran subroutines begin with `pvmf` instead of `pvm_`
- Status returns by C functions is passed back in an extra parameter
- Must link with the *additional* library: `libfpvm3.a`

WRITING APPLICATIONS

Table of Contents

Hello world in C

- A simple application composed of two programs
- Program hello.c, the main program

```
#include <pvm3.h>

main ()
{
   int tid ;
   char reply [100] ;

   printf ("I'm t%x\n", pvm_mytid ()) ;
   if (pvm_spawn("hello_other", (char**)0,
                 0, "", 1, &tid) == 1)
   {
      pvm_recv (-1, -1) ;
      pvm_bufinfo (pvm_getrbuf (), (int*)0,
                   (int*)0, &tid) ;
      pvm_upkstr(reply) ;
      printf (From t%x: %s\n", tid, reply) ;
   } else
      printf ("Can't start hello_other\n") ;
   pvm_exit () ;
}
```


- Program hello_other.c, the auxiliary program

```
#include <pvm3.h>

main ()
{
   int ptid ; /* parent's tid */
   char buf [100] ;

   ptid = pvm_parent () ;
   strcpy (buf, "Hello world from ") ;
   gethostname (buf+strlen(buf), 64) ;
   pvm_initsend (PvmDataDefault) ;
   pvm_pkstr (buf) ;
   pvm_send (ptid, 1) ;
   pvm_exit () ;
}
```


Hello world in Fortran

- A simple application composed of two programs
- Program hello.f, the main program

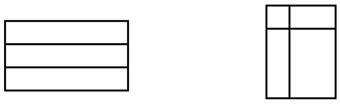
```
      PROGRAM HELLO
      INCLUDE 'fpvm3.h'
*
      INTEGER mytid, tid
      INTEGER bufid, info
      CHARACTER*100 reply1, reply2
*
      CALL PVMFMYTID (mytid)
      WRITE (*,*) 'I m t', mytid
      CALL PVMFSPAWN ('hello_other', 0, '*',
    &                 1, tid, info)
      IF (info .EQ. 1) THEN
        CALL PVMFRECV (-1, -1, info)
        CALL PVMFGETRBUF (bufid)
        CALL PVMFBUFINFO (bufid, 0, 0, tid)
        CALL PVMFUNPACK (STRING, reply1, 1, 1, info)
        CALL PVMFUNPACK (STRING, reply2, 1, 1, info)
        WRITE (*,*) 'From t', tid, ':', reply1, reply2
      ELSE
        WRITE (*,*) 'Can t start hello_other'
      ENDIF
      CALL PVMFEXIT (info)
      END
```


- Program hello_other.f, the auxiliary program

```
      PROGRAM HELLO_OTHER
      INCLUDE 'fpvm3.h'
*
      INTEGER ptid
      INTEGER bufid, info
      CHARACTER*100 buf
*
      CALL PVMFPARENT (ptid)
      CALL GETHOSTNAME (buf, 64)
      CALL PVMFINITSEND (PvmDataDefault)
      CALL PVMFPACK (STRING, 'Hello From ', 1, 1, info)
      CALL PVMFPACK (STRING, buf, 1, 1, info)
      CALL PVMFSEND (ptid, 1, info)
      CALL PVMFEXIT (info)
      END
```

Parallelizing applications

- How should you parallelize your application ?
- As a sort of "Network assembly language",
  PVM leaves the decision up to you
  - What to put in "component" tasks
  - What messages to pass between tasks
  - How many tasks to have, where to run them
- Some applications divide up naturally
- Look for separable calculations:

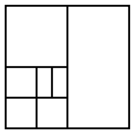


- Speedup depends on

$$\frac{\text{communication}}{\text{computation}} \text{ and } \frac{\text{cpuspeed}}{\text{networkspeed}} \text{ ratios}$$

Divide and Conquer

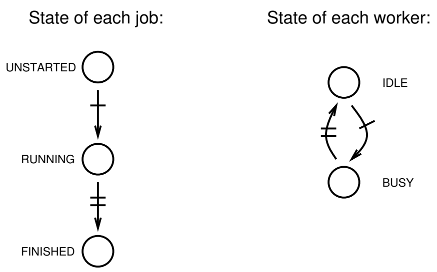
- Dynamically discovers a tree of calculations
- Calculation parameters are handed to worker
- Worker may do the work or delegate half to another worker
- The calculation might proceed:



- Using a pool of workers may be faster than actually creating new tasks

Bag of Tasks

- Simple model that provides load balancing
- Given a list of jobs and a set of workers, how to schedule all the jobs?




```
foreach job J
   J.state <- UNSTARTED
foreach worker W
   W.state <- IDLE
forever do
   while (E(J:J.state=UNSTARTED) & E(W:W.state=IDLE))
      J.state <- RUNNING
      W.job   <- J
      W.state <- BUSY
   if (!E(W:W.state=BUSY))
      break
   wait for reply from worker (W)
   W.job.state <- FINISHED
   W.state     <- IDLE
```


- Bag of Tasks (BoT) can be modified in many ways
- Adjust size of jobs
  - Smaller workers get smaller jobs
  - Start bigger jobs first
- BoT can also be used when workers must communicate between themselves
  - Have to take this into account when scheduling
  - May need a special constraint system
- Refinements can improve performance greatly but may make the code very convoluted
- Queue manager library or system needed

The master/slave model

- PVM is not restricted to this model
- Useful programming paradigm and simple to illustrate
- The master calls `pvm_mytid()` to
  - Allow it to use the PVM system, and
  - Enable interprocessor communications
- It then call `pvm_spawn()` to execute a given number of slave programs on other machines
- Each slave program must also call `pvm_mytid()` to enable processor communications
- Subsequently, `pvm_send()` and `pvm_recv()` are used to pass messages between processes
- When finished, all the PVM programs should call `pvm_exit()` to
  - Allow PVM to disconnect any sockets to the processes, and
  - Allow PVM to keep track of which processes are running

- Example: the master code `master.c`

```
#include "pvm3.h"
#define SLAVENAME "slave"

main()
{
  int mytid ;                   /* my task id */
  int tids[32] ;                /* slave task ids */
  int n, nproc, i, who, msgtype ;
  float data[100], result[32] ;

  /* enroll in pvm */
  mytid = pvm_mytid () ;

  /* start up slave tasks */
  puts ("How many slave programs (1-32)?") ;
  scanf ("%d", &nproc) ;

  pvm_spawn (SLAVENAME, (char**)0, 0, "", nproc, tids) ;

  /* Begin User Program */
  n = 100 ;
  /* initialize_data (data, n) ; */
  for (i=0; i<n; i++){
    data[i] = 1 ;
  }
```


- Master code, cont'd

```
  /* 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 () ;
}
```


- Example: the slave code `slave.c`

```
main () {
  int mytid ;                 /* my task id */
  int tids[32] ;              /* task ids   */
  int n, me, i, nproc, master, msgtype ;
  float data[100], result, 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 () ;
}
```

- Slave code, cont'd

```
float work (int me; int n; float *data; int *tids; int nproc)
{
  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) ;
}
```


The SPMD model

- There is only a single program
- There is no master program directing the computation
- Sometimes called *hostless* program
- There is still the issue of getting all the processes initially started
- In the following example, the user starts the first copy of the program
- By checking `pvm_parent()`, this copy can determine that it was not spawn by a PVM task
- It then spawns multiple copies of itself and pass them the array of tids
- At this point each copy is equal and can work on its partition of the data, in collaboration with the other processes

- Example, the SPMD code: `spmd.c`

```
#include "pvm3.h"
#define NPROC 4

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 () ;

  /* find out if I am parent or child */
  tids[0] = pvm_parent () ;
  if (tids[0] < 0) {      /* then I am the parent */
    tids[0] = mytid ; me = 0 ;
    /* start up copies of myself */
    pvm_spawn ("spmd", (char**)0, 0, "", NPROC-1, &tids[1]) ;

    /* multicast tids array to children */
    pvm_initsend (PvmDataDefault) ;
    pvm_pkint (tids, NPROC, 1) ;
    pvm_mcast (&tids[1], NPROC-1, 0) ;
  } else {    /* I am a child */
    /* receive tids array */
    pvm_recv (tids[0], 0) ;
    pvm_upkint (tids, NPROC, 1) ;
    for (i=1; i<NPROC ; i++)
      if (mytid == tids[i]) { me = i ; break ; }
  }
/* all NPROC tasks are equal now */
```


- The spmd code, cont'd

```
/*-----------------------------------------------------------
 * all NPROC tasks are equal now
 * and can address each other by tids[0] thru tids[NPROC-1]
 * for each process me => process number [0-(NPROC-1)]
 *-----------------------------------------------------------
 */
 printf ("me = %d mytid = %d\n", me, mytid) ;
 dowork (me, tids, NPROC) ;

 /* program finished exit pvm */
 pvm_exit () ;
 exit (1) ;
 }
```

- The spmd code, cont'd

```
/* Simple example passes a token around a ring */

dowork (int me; int *tids; int nproc)
{
  int token ;
  int dest ;
  int count  = 1 ;
  int stride = 1 ;
  int msgtag = 4 ;

  if (me == 0) {
    token = tids[0] ;
    pvm_initsend (PvmDataDefault) ;
    pvm_pkint (&token, count, stride) ;
    pvm_send (tids[me+1], msgtag) ;
    pvm_recv (tids[nproc-1], msgtag) ;
    printf ("token ring done\n") ;
  } else {
    pvm_recv (tids[me-1], msgtag) ;
    pvm_upkint (&token, count, stride) ;
    pvm_initsend (PvmDataDefault) ;
    pvm_pkint (&token, count, stride) ;
    dest = (me == nproc-1)? tids[0] : tids[me+1] ;
    pvm_send (dest, msgtag) ;
  }
}
```


DEBUGGING METHODS

Table of Contents

Debugging a PVM program

- Debugging parallel programs is much more difficult than debugging serial programs
  - There are more processes running simultaneously
  - Their interaction can occur error
- *Deadlock*: a programming error has caused all the processes to be waiting on messages
- *Resource failure*: due to an error, processes have spawned extra processes
- All PVM routines returns an error condition if some error has been detected during their execution
- Two routines `pvm_perror()` and `pvm_setopt (PvmAutoErr...)` are provided to allow the detection and printing of any PVM error

- Spawned PVM tasks can also be started under a debugger
  - By setting the `flag` option to include `PvmTaskDebug` in `pvm_spawn()`
  - By setting the `-?` option of the `spawn` instruction under the console
- PVM will execute the shell script `pvm3/lib/debugger`:

  `pvm3/lib/debugger pvm3/bin/ARCH/a.out arg1 arg2 ...`

- As supplied, this script
  - Starts an `xterm` window on the host PVM was started
  - Spawns the task under a debugger in this window
- The shell script can be modified to include a favorite debugger or to do something else.

  The following one-liner times each spawned `a.out`:

  `/bin/time "$@" 2>> $HOME/time.out`

- Diagnostic print statement sent to either `stdout` or `stderr` from a spawned task will not appear on the user's screen

  Use `pvm_catchout ()` to do it
- All these prints are routed to a single log file

  `/tmp/pvml.<uid>`

  on the host where PVM was started

Three steps in debugging PVM programs

1. Run, if possible, the program on a single process

   Debug as any other serial program

   Catch error unrelated to parallelism
2. Run the program using 2-4 processes on a single machine

   Check the communication syntax and logic (tag...)
3. Run the same program on a 2-4 processes across several machines

   Check for synchronization errors produced by the networks delay

IMPLEMENTATION DETAILS

Table of Contents

Network protocols

- On Unix systems, mainly two protocols used:
  - TCP
  - UDP
- Protocols use **sockets**: points of contact to a process
- Three possible implementations of communications:
  1. Point to point TCP
  2. Dynamic TCP sockets
  3. Daemon mediated communications

Point to point TCP TCP sockets are established between every pair of nodes when the program is initiated

Referred to as a TCP socket crossbar

Its run into trouble when scaling up to large number of nodes

No used in PVM

Dynamic TCP sockets Establish a socket between two communicating nodes at run time when they first communicate with each other

Scales better on large set of nodes

But first communication is significantly slower than subsequent ones.

This startup cost must be amortized by following communications

Used in PVM if the programmer specifies

```
pvm_setopt (PvmRoute, PvmRouteDirect) ;
```

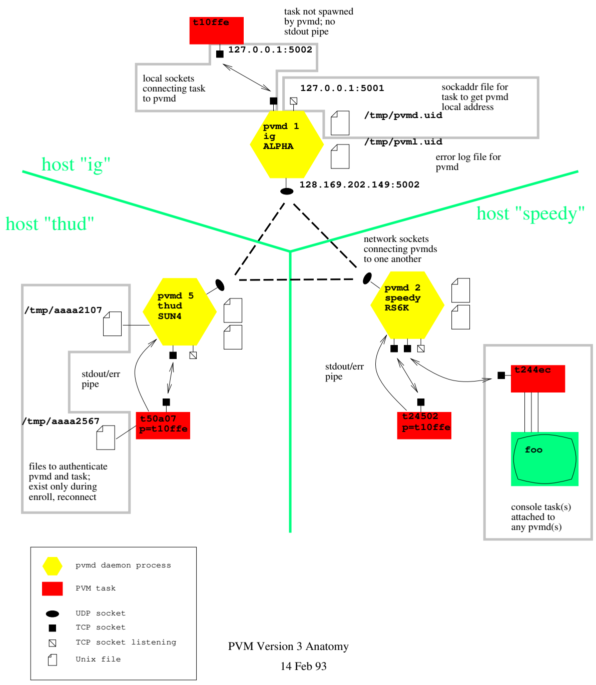
Daemon mediated communications Use UDP communication

A socket is attached to each daemon (pvmd)

The processes communicate to their local daemons which forward the communication to another daemon (or to the local process)

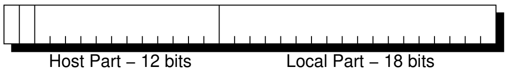
That is the default communication mode in PVM

PVM at work

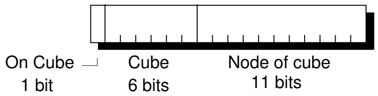


Tid–Task identifier

- Task identifier is a 32-bit integer



- Host part determines "owning" host of tid
- Host numbers synchronized by global host table
- Local part defined by each `pvmd`, it is the key to MPP ports
- Routing a message is easy
  Route to the `pvmd`, forward to task
- Local part can be subdivided, e.g.:



Data conversion

PVM version 2

- PVM insures the needed conversions if the different hosts use different data representation (IEEE, IBM..., big/little endian...)
- PVM converts data using XDR
- If all the host have the same data representation, there will be no conversion (even if there is different types of host)
- Else
  - The majority representation is chosen as a "standard"
  - The data will be send on this standard on the network
  - The hosts using a different representation realize local conversions
  - If two "minority" hosts communicate, there will be 2 conversions
- Alternatives
  1. The sender converts the data into the format of the receiver
     The sender has to know the format of each of the other hosts
  2. In order to avoid to code this information, all the data are converted into a format independent of the hosts and of the network
     The receiver converts the data from that format to its own format

PVM version 3

- The fact that the virtual machine is homogeneous may be corrupted by the dynamic adding of a new host in the virtual machine
- Depending on the user declaration
  - Either PVM 3 always uses XDR
  - Or no conversions are done

PROS AND CONS

Table of Contents

Pros of PVM

- The PVM programming model is clean
- PVM is free and easily obtained via electronic mail
- PVM is easy to install, learn, and use
- PVM is small and portable across Unix compatible machines and some multiprocessors
- Fortran is supported
- The source codes are available
- A graphical environment is available:
  - *HeNCE*, Heterogeneous Network Computing Environment
  - Automatic generation of PVM code
  - Programmer specifies parallelism by drawing a directed acyclic graph that represents the logical structure of the program
  - It is impossible to describe some algorithms

Cons of PVM

- Low level package. PVM lacks high level facilities such as
  - Fault tolerance
  - Process migration...
- Distributed I/O is not supported
- No support for
  - Shared memory
  - Data mapping
  - Data distribution
  - Data replication
- Security relies entirely on Unix based security techniques
  No support for data encryption for example
- No load balancing
  But a load balancing package is under development

- Scalability issues
  - Broadcast are realized in a linear fashion
    Hardware multicast is not used
  - Local communications are linearized though the daemon process
  - On a large virtual machine (>10, 20 hosts): multiplication of the daemon processes
- Basic trace/debugger facilities
- No performance monitoring
- Support is (was ?) a problem since PVM is a public domain
  `comp.parallel.pvm` news group

CONCLUSIONS

Table of Contents

New features of PVM 3.4

(EuroPVM 95, Lyon, Sepember 1995)

- Context
  - Problem:
    - ▹ Use of a Math Lib using PVM in a PVM application.
    - ▹ A math routine may intercept a user message !
  - Solution : communication context
    - ▹ Additional tag associated with messages
    - ▹ Generated by the system at execution time
- Data encoding
  - Was XDR
  - Will be RMR: Receiver Makes Right
  - Avantages:
    - ▹ At most one conversion on any message
    - ▹ XDR only defined for 32 bit words
  - Drawbacks:
    - ▹ Each receiver must be able to decode every other format
- Buffered Tracing

  ⇒ Update of XPVM

- Name service
  - Goal
    - ▹ Two PVM programs start independently
    - ▹ And discover information about each other
  - Use of a “name server”
    - ▹ Database managed by PVM
- Static groups
  - Presently : dynamic group
    - ▹ Members can join and leave asynchronously
    - ▹ Cost of performance: PVM uses a central group server ⇒ Bottleneck
  - Add static groups
    - ▹ Members of the group can never change once the group is formed
    - ▹ Created synchronously
      - ◇ List of tids

- MPI/PVM and PVM/MPI
  - MPI Forum: Standard specification for message-passing
  - PVM: de facto standard for distributed computing
  - Question: will MPI and PVM merge?
  - The PVM development team will not change the PVM API
    - ▹ Obligation to the PVM community
    - ▹ PVM API has evolved based on needs and requests from users
    - ▹ PVM API simple and easy to learn and use
  - MPI API?
    - ▹ Good design for MPP
    - ▹ For network of workstations: lacks features (process control...)

Work in progress

Work going into next version

- More scalability–100 of hosts
  Parallel startup
- Better debugging support
  `libpvm` generates trace messages
- Better high-speed network utilization
  FDDI fiber–DEC Alpha
- Better console program
  Quoting, command aliases and history

Other works

- Load balancing
- Automatic application fault tolerance
- X-Window console
- TCL–based shell

Other software

P4

Express

ISIS

MPI Message Passing Interface

EXERCISES

Table of Contents

Add/delete host

A added/deleted host console (page 45)

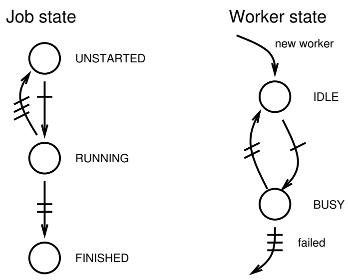
Matrix

Communication of matrix rows and columns (page 57)

Failure-tolerant BoT

Bag of Tasks revisited

- pvm_notify() can be used to build a failure-tolerant BoT
- If a worker fails, the algorithm would normally block waiting for a response
- The notify message would be received instead of the response, allowing the master to re-schedule
- Can use HostAdd to respond to new resources
- The state diagrams are modified like so:



Gather

For 3 distributed arrays, compute

```
a = b [p]
```