INTERNATIONAL ATOMIC ENERGY AGENCY
UNITED NATIONS EDUCATIONAL, SCIENTIFIC AND CULTURAL ORGANIZATION
INTERNATIONAL CENTRE FOR THEORETICAL PHYSICS
I.C.T.P., P.O. BOX 586, 34100 TRIESTE, ITALY, CABLE CENTRATOM TRIESTE

H4.SMR/854-14

College on Computational Physics

15 May - 9 June 1995

An Introduction to PVM (Parallel Virtual Machine)

F. Massaioli

Università "la Sapienza"
Rome, Italy

An Introduction to PVM (Parallel Virtual Machine)

Describing PVM Version 3.1

Robert Manchek

University of Tennessee

Computer Science Department

August 4–5, 1993

Agenda

Wednesday

9:00 – 12:00 Lecture

- Introduction
- Overview of PVM
- Using PVM
- PVM Programming Interface
- Writing PVM Applications

13:00 – 17:00 Lab

- Running PVM
- Building, running example programs
- [Writing an application]

Thursday

9:00 – 12:00 Lecture

- Writing PVM Applications cont'd
- Advanced Topics
- Debugging
- PVM Implementation

13:00 – 17:00 Lab

- Writing an application

PVM Team

Carolyn Aebischer

Weicheng Jiang

Robert Manchek

Keith Moore

University of Tennessee

Adam Beguelin

Carnegie Mellon University,

Pittsburgh Supercomputer Center

Jack Dongarra

University of Tennessee,

Oak Ridge National Laboratory

Al Geist

Oak Ridge National Laboratory

Vaidy Sunderam

Emory University

Parallel Virtual Machine

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

- It's a poor man's supercomputer allowing one to exploit the aggregate power of unused workstations during off–hours.
- It'a a metacomputer allowing multiple supercomputers distributed around the world to be linked together to achieve Grand Challenge performance.
- It's an educational tool allowing colleges without access to parallel computers to teach parallel programming courses.

Overview of PVM

Major Features 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 users' PVMs 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

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 architectures (serial / parallel), different operating systems

- Network

  Different network types, 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 BSDI	Alliant FX/8
DEC Alpha/OSF-1	BBN TC2000
DEC Microvax	Convex
DECstation	Cray YMP and C90
DG Aviion	IBM 3090
HP 9000/300	Intel Paragon
HP 9000/700	Intel iPSC/2
IBM RS/6000	Intel iPSC/860
IBM/RT	Kendall Square KSR-1
NeXT	Sequent Symmetry
Silicon Graphics IRIS	Stardent Titan
Sun 3	Thinking Machines CM-2, CM-5
Sun 4, Sparc	

- Version 3 portable to non-Unix machines and multiprocessors
- iPSC/860, Paragon, CM-5 ports running and available, KSR-1 port in progress
- VMS port soon ?

Evolution of PVM

- Version 1 written during summer of 1989 at ORNL
  Proof of concept – never released

- Version 2 written March 1991 at UTK
  Intended as substrate for HeNCE
  Stable, robust version – first released publicly

- Version 3 written September 1992 – February 1993
  Second release, 3.1, has been available 3 months.
  Version 3.2 TBR August 93

The PVM System

System is composed of:

- Pvmd daemon program

  Runs on each host of virtual machine

  Provides inter–host point of contact

  Authenticates tasks

  Execs processes on host

  Provides fault detection

  Mainly a message router, but is also
  a source and sink of messages

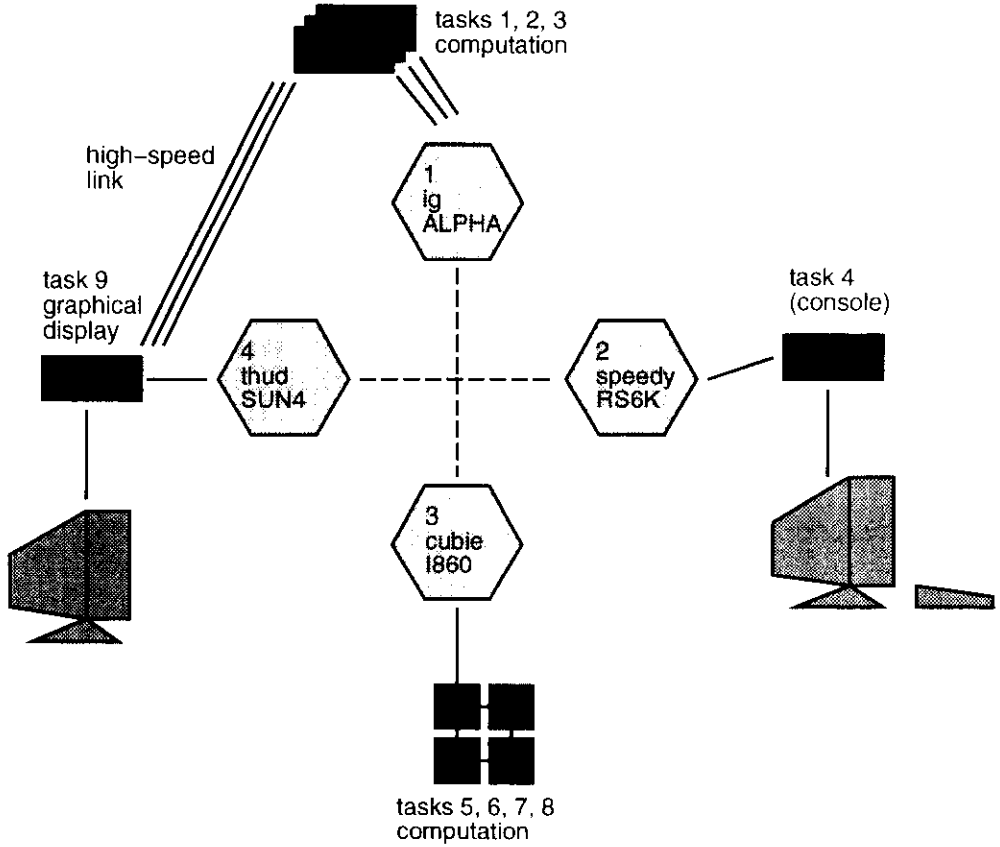
  More robust than application components

The PVM System

System is composed of:

- Libpvm programming library
  - Linked with each application component (program)
  - Kept as simple as possible
  - Functions are low-level PVM "syscalls"
- Application components
  - Written by user in PVM message-passing calls
  - Are executed as PVM "tasks"

PVM In Operation



Changes from v2 to v3

- Pvmds have fault detection / recovery,
  Can build fault-tolerant applications
- Console not built into pvmd; is a normal task
- Portable to multiprocessors
- Dynamic configuration of virtual machine;
  able to add and delete hosts while running
- Multiple message buffers available in libpvm
  Messages can be saved, forwarded
- Vsnd( ) interface removed
  Routing is done automatically
- New libpvm function names
  Won't collide with other libraries

Internal Changes in v3

- Task identified by a single integer (TID)
  Allows more efficient implementation
  Can build symbolic/group libraries on top
- Pvmd-pvmd communication uses UDP
  lost-packet retry based on host-host RTT
- Scalable to larger number of hosts / reduced centrality
  Each pvmd can assign TIDs autonomously
  Require a master host to manage host table
- Modular communication layers / easier to
  replace with custom transport mechansims

How to Obtain PVM

Source code, User's guide, Examples, and related material are published on netlib, a software repository with sites at ORNL and ATT.

To get started, send email to netlib:

```
% mail netlib@ornl.gov
Subject: send index from pvm3
```

Instructions will be mailed back automatically.

Also available via anonymous FTP from:

```
netlib2.CS.UTK.EDU
```

Questions and problems can be addressed to:

```
pvm@msr.epm.ornl.gov
```


Using PVM

Building PVM

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

`pvmd3`	The pvm daemon
`pvm`	The console program
`libpvm3.a`	The C programming library
`libfpvm3.a`	The FORTRAN library

  and installs in pvm3/lib/ARCH
- Copy to 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/src`	PVM system source

- It can be installed in a shared directory:

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

  or:

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

- pvm3/bin directory is generally private

Programming Concepts

Host	A physical machine, e.g. Unix workstation 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/pvmd3
Message	An ordered list of data sent between tasks
Group	An ordered list of tasks assigned a symbolic name Each task has a unique index in the group Any task may be in zero or more groups

Pvmd Hostfile

- You can use a hostfile to specify a configuration or set per–host parameters
- Each host listed in hostfile is automatically added unless preceded by an ampersand (&)
- Hosts entered one per line, name followed by options

		DEFAULTS
`lo=`	Different loginname	same
`pw`	Pvmd asks for password	don't ask; use rsh
`dx=`	Special location of pvmd	`pvm3/lib/pvmd`
`ep=`	Special a.out search path	`pvm3/bin/%`
`ms`	Requires manual startup	don't

- Can specify default settings using host "*"
- Example:

`% = ARCH`

```
# this is a comment line
ig
#honk
thud ep=appl1:pvm3/bin/%
# remaining hosts don't trust us
* pw
cobra
& viper lo=rjm
```

Starting PVM

Console automatically starts a pvmd if needed:

```
ig% pvm
pvm> conf
1 host, 1 data format
        HOST     DTID     ARCH     SPEED
          ig    40000    ALPHA         1
```

Pvmd can be started by hand to supply a hostfile or passwords:

```
ig% echo "thud pw" > H
ig% ~/pvm3/lib/pvmd H
Password (thud:manchek):
^Z
Suspended
ig% bg
[1]   /leather/homes/manchek/pvm3/lib/pvmd H &
ig% pvm
pvmd already running
pvm> conf
2 hosts, 2 data formats
        HOST     DTID     ARCH     SPEED
          ig    40000    ALPHA         1
        thud    80000     SUN4         1
```


Adding Hosts

The add and delete commands reconfigure the machine:

```
pvm> conf
2 hosts, 2 data formats
        HOST     DTID     ARCH     SPEED
          ig    40000    ALPHA         1
       speedy   80000     RS6K         1
pvm> add thud honk
2 successful
        HOST     DTID
        thud    c0000
        honk   100000
pvm> delete speedy
1 successful
        HOST   STATUS
      speedy   deleted
pvm> conf
3 hosts, 2 data formats
        HOST     DTID     ARCH     SPEED
          ig    40000    ALPHA         1
        thud    c0000     SUN4         1
        honk   100000    ALPHA         1
```

- If a password is needed to add a host, be sure the master pvmd is still running in the foreground! (Use a different window for the console)

Programming in PVM

Libpvm

PVM provides C (or C++) and FORTRAN programmers with a library of about 70 functions

C applications:

```
#include <pvm3.h>
```

Compiling:

```
cc -I$PVM_ROOT/include myprog.c -L$PVM_ROOT/lib -lpvm3
```

FORTRAN applications:

Header file in include/fpvm3.h

Compiling and linking more system-dependent than for C

Need to link with -lfpvm3 and -lpvm3

Programs using group library need to link with `-lgpvm3`

Extra libraries are required on some hosts.

Look at pvm3/console/ARCH/Makefile for hints

PVM 3.x C Routines (1/2)

Process Control

```
int   cc = pvm_spawn( char *aout, char **argv, int flag,
                      char *where, int cnt,  int *tids )
int   cc = pvm_kill( int tid )
void     pvm_exit()
int  tid = pvm_mytid()
int  tid = pvm_parent()
int stat = pvm_pstat(int tid )
int stat = pvm_mstat(char *host )

int   cc = pvm_config(int *host, int *arch,
                      struct hostinfo **hostp )
int   cc = pvm_addhosts(char **hosts, int cnt, int *st )
int   cc = pvm_delhosts(char **hosts, int cnt, int *st )
```

Dynamic Process Groups

```
int cc   = pvm_bcast(     char *group, int msgtag )
int inum = pvm_joingroup( char *group )
int cc   = pvm_lvgroup(   char *group )
int tid  = pvm_gettid(    char *group, int inum )
int inum = pvm_getinst(   char *group, int tid )
int cc   = pvm_barrier(   char *group, int cnt )
int size = pvm_gsize(     char *group )
```

Message Buffers

```
int buf    = pvm_mkbuf( int encoding )
int cc     = pvm_freebuf( int buf )
int buf    = pvm_getsbuf()
int buf    = pvm_getrbuf()
int oldbuf = pvm_setsbuf( int buf )
int oldbuf = pvm_setrbuf( int buf )
int buf    = pvm_initsend( int encoding )
```

Error Handling

```
int info = pvm_perror(char *msg
int info = pvm_serror( int how )
```


PVM 3.x C Routines (2/2)

Message Passing

```
int cc = pvm_advise( route )

int cc = pvm_pkbyte(  char   *cp, int cnt, int std )
int cc = pvm_pkcplx(  float  *xp, int cnt, int std )
int cc = pvm_pkdcplx( double *dp, int cnt, int std )
int cc = pvm_pkdouble(double *dp, int cnt, int std )
int cc = pvm_pkfloat( float  *fp, int cnt, int std )
int cc = pvm_pkint(   int    *np, int cnt, int std )
int cc = pvm_pklong(  long   *np, int cnt, int std )
int cc = pvm_pkshort( short  *np, int cnt, int std )
int cc = pvm_pkstr(   char   *cp)

int cc = pvm_send(  int tid, int msgtag )
int cc = pvm_mcast( int tids, int ntask, int msgtag )

int buf = pvm_probe(int tid, int msgtag )
int buf = pvm_nrecv(int tid, int msgtag )
int buf = pvm_recv( int tid, int msgtag )
int cc  = pvm_bufinfo(int buf, int *len,
                      int *msgtag, int *tid )

int cc = pvm_upkbyte(  char   *cp, int cnt, int std )
int cc = pvm_upkcplx(  float  *xp, int cnt, int std )
int cc = pvm_upkdcplx( double *dp, int cnt, int std )
int cc = pvm_upkdouble(double *dp, int cnt, int std )
int cc = pvm_upkfloat( float  *fp, int cnt, int std )
int cc = pvm_upkint(   int    *np, int cnt, int std )
int cc = pvm_upklong(  long   *np, int cnt, int std )
int cc = pvm_upkshort( short  *np, int cnt, int std )
int cc = pvm_upkstr(   char   *cp )
```

PVM 3.x FORTRAN Routines (1/2)

Process Control

```
call pvmfspawn(task, flag, where, ntask,tids, info)
call pvmfkill(tid, info)
call pvmfexit(info)
call pvmfmytid(tid)
call pvmfparent(tid)
call pvmfpstat(tid, pstat)
call pvmfmstat(host, mstat)

call pvmfconfig(nhost, narch, info)
call pvmfaddhost(host, info)
call pvmfdelhost(host, info)

call pvmfnotify( who, about, data, info )
```

Dynamic Process Groups

```
call pvmfjoingroup(group, inum)
call pvmflvgroup(  group, info)
call pvmfwhois(    group, inum, tid)
call pvmfgetinst(  group, tid, inum)
call pvmfbarrier(  group, cnt, info)
```

Multiple Msg Buffers

```
call pvmfmkbuf(encoding, info)
call pvmffreebuf(buf, info)
call pvmfgetsbuf(buf)
call pvmfgetrbuf(buf)
call pvmfsetsbuf(buf, oldbuf)
call pvmfsetrbuf(buf, oldbuf)
call pvmfinitsend(encoding, info)
```

Error Handling

```
call pvmfperror(msg, info)
call pvmfserror(how, info)
```


PVM 3.x FORTRAN Routines (2/2)

Message Passing

```
call pvmfpack( what, xp, nitem, stride, info )

call pvmfsend( tid, msgtag, info)
call pvmfmcast( ntask, tids, msgtag, info )

call pvmfnrecv( tid, msgtag, buf)
call pvmfrecv(  tid, msgtag, buf)
call pvmfbufinfo(buf, bytes, msgtag, tid, info)

call pvmfunpack( what, xp, nitem, stride, info )
```

what options

STRING	REAL4
BYTE1	COMPLEX8
INTEGER2	REAL8
INTEGER4	COMPLEX16

Error Handling

PVM C functions return status value

In FORTRAN status is passed back in an extra parameter

Generally, a status of >= 0 is successful,
negative status indicates an error.

When an error occurs, libpvm automatically prints a message indicating the task ID, function and error, e.g.:

```
libpvm [t40001]: pvm_sendsig(): Bad Parameter
```

Automatic error reporting can be disabled by calling `pvm_serror()`

Error reports can be generated manually by calling `pvm_perror()`

Messages

PVM provides functions for composing messages containing mixed data types

Any C data type can be packed in a message

Example:

```
send_job(worker, name, wd, ht, coeffs)
    int worker;
    char *name;
    int wd, ht;
    double *coeffs;
{
    pvm_initsend(PvmDataDefault);
    pvm_pkstr(name);
    pvm_pkint(&wd, 1, 1);
    pvm_pkint(&ht, 1, 1);
    pvm_pkdouble(coeffs, 6);
    pvm_send(worker, 12);
}
```

Matching calls can unpack the data on the receiving end.

Packing and unpacking structures, lists, etc.
requires writing functions.

Messages, cont'd

Each message has an encoding, either:

- Native data format of host that packed it (RAW)
- Externalized (XDR)

Determined by param to `pvm_initsend()` or `pvm_mkbuf()`
`PvmDataDefault` or `PvmDataRaw`

- RAW format packs and unpacks faster
- XDR format can be passed between unlike hosts.

Task receiving message can decode XDR + 1 other format, may not be able to read message.

Task can forward message without reading it, however.

In-place packing is like RAW but data remains in place until sent. Only descriptors of the data are packed.

- Faster for large, dense data
- Uses less memory (for buffers)
- Has side-effect: message is a "template".

Messages, cont'd

Once composed, a message can be:

- Sent to another task
- Multicast to a list of tasks
- Broadcast to a group (if using the group library)

Any number of times without repacking

Send operation is non-blocking:

- Returns ASAP
- Does not imply receipt

Messages are downloaded to receiving task and put in a queue to be accepted.

Recv operation downloads messages until matching one in queue, removes and returns it

Messages sent between any two tasks A and B will arrive in the order they are sent

Recv picks messages by sender and/or code can pull messages from queue out of order

Group Library

An optional library is available to facilitate writing applications by providing naming structure.

Tasks can enroll in names groups and can be addressed by the group name and an index

Functions are provided to allow tasks to:

Join, leave groups

Lookup a group member by group name and index or TID

Broadcast to a group

Perform barrier synchronization within a group

Writing PVM Applications

Hello World

Simple example composed of two programs

Program "hello.c", the main program:

```
#include <pvm3.h>

main()
{
    int tid;
    char reply[30];

    printf("I'm t%x\n", pvm_mytid());
    if (pvm_spawn("hello2", (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 hello2\n");
    pvm_exit();
}
```

Program "hello2.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();
}
```


Pllizing Applications

How should you Parallelize your application?

As a sort of "Network Assembly Language", PVM leaves the decisions up to you.

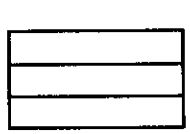
What to put in "component" tasks

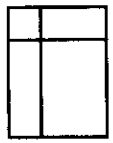
What messages to pass between tasks

How many tasks to have, where to run them

Some applications divide up naturally into, f.e. computation and interactive components.

Look for separable calculations:





Speedup depends on

$$\frac{\text{communication}}{\text{computation}} \text{ and } \frac{\text{cpu speed}}{\text{network speed}} \text{ ratios}$$

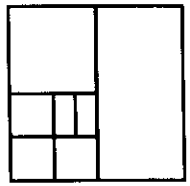
Divide and Conquer

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:

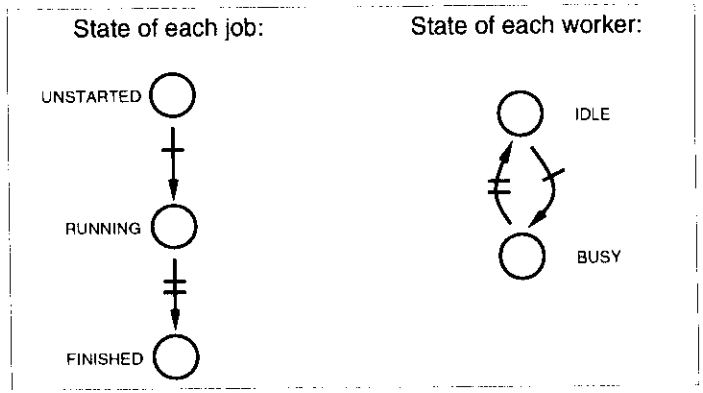


Using a pool of workers may be faster than actually creating new tasks

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, cont'd

BOT can be modified in many ways

Adjust sizes of jobs

- Slower 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

Q manager library or system needed

Fast Message Routing

Default – task–task messages routed through PVMDs

Can use `pvm_advise()` to control routing

Advise parameter is one of:

`PvmDontRoute`	Task won't attempt to route directly and will refuse requests
`PvmAllowDirect`	Task won't attempt to route directly but will grant requests
`PvmRouteDirect`	Task will attempt to route directly (request may be refused)

Routing policy only applies to future messages and doesn't affect current connections

Once direct route established between tasks, both will use the route.

Successful routing also depends on max. # open files

Using `pvm_getfds()`

PVM tasks are usually either computing or blocked on recv

May need to service input from other sources

Can get a list of FDs in use by libpvm for use
with e.g. `select()`

Xep example uses getfds to get input from workers
and X events at the same time

This is simple with default message routing,
more complex when using direct routing – FD set changes

This function is experimental

More Messages – `recvf()`

Recv means:

`pvm_nrecv(), pvm_probe(), pvm_recv()`

Recv uses "matching function" to pick a message from receive queue.

Default function picks by sender and/or code
Specifying either as –1 indicates wildcard

Use `pvm_recvf()` to install different matching function

Recv calls function for each message in queue

match(int mid, int tid, int code)

mid	Message buffer ID
tid	TID passed to recv
code	Code passed to recv

Match function returns preference to recv:

<0	Error condition; stop
0	Don't pick this message
1	Pick this message; stop now
>1	Pick highest preference over all

Using `notify()`

Task using notify can request a message when an exceptional condition occurs:

A (different) task exits or fails

A pvmd (host) is deleted or fails

[Not Implemented in version 3.1] A new pvmd (host) is added to the machine

Task can specify message code to be used

Message body contains data specific to the event

`PvmTaskExit`	TID of one task
`PvmHostDelete`	pvmd–TID of one host
`PvmHostAdd`	[NIY] count of new hosts and pvmd–TIDs

For example:

```
int tids[2], x;

pvm_spawn(..., 2, tids)
pvm_notify(PvmTaskExit, 666, 2, tids);
recv(-1, 666);
pvm_upkint(&x, 1, 1);
printf("eek, t%x exited\n", x);
```

TaskExit and HostDelete reply even if task or host already dead (avoids a race).

Bag–of–Tasks Revisited

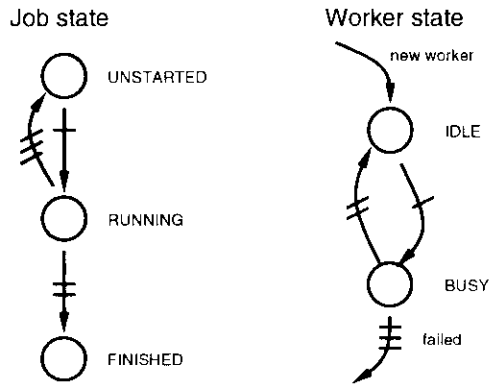
Notify can be used to build a failure–tolerant BOT

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:



Debugging PVM Applications

Task run by hand can be started under a debugger

Modify program slightly – add `PvmTaskDebug` to the `pvm_spawn()` flags parameter

From console, use –? flag with spawn

Pvmd runs spawned program in debugger

Use the following steps when debugging:

- Run program as a single task – eliminate computational errors
- Run program in parallel on a single host – eliminate errors in message passing
- Run program on a few workstations to look for problems with synchronization and deadlock

The Debugger Script

- During normal spawn, pvmd execs:

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

- When PvmTaskDebug is set, pvmd execs:

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

- The debugger script supplied starts an xterm process connected to a debugger process (dbx) debugging the a.out

- The script can be modified to do something else. The following one–liner times each spawned a.out:

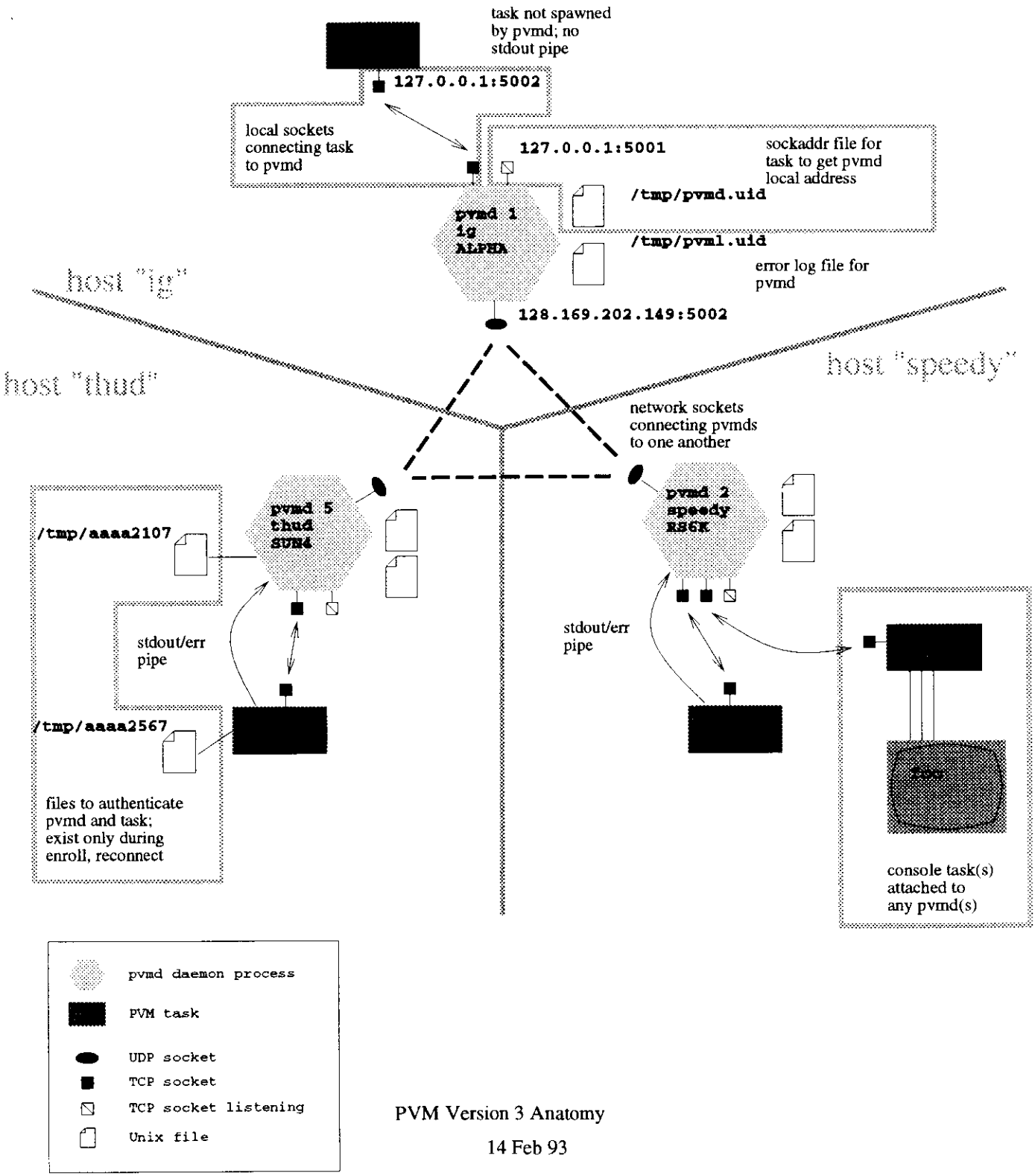
```
/bin/time "$@" 2>> $HOME/time.out
```

- This is an area of ongoing research...
  - Changing name of debugger script
  - Coordinated debugging of multiple tasks
  - Needs to scale better

PVM Implementation

Details of version 3 implementation

Potentially useful to consider when writing applications

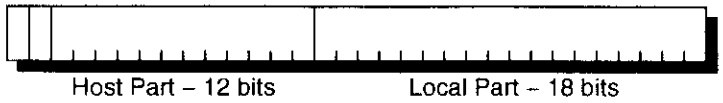


PVM Version 3 Anatomy

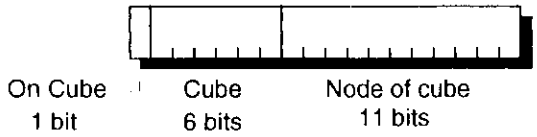
14 Feb 93

TID – Task Identifier

- Task Identifier is 32–bit integer:

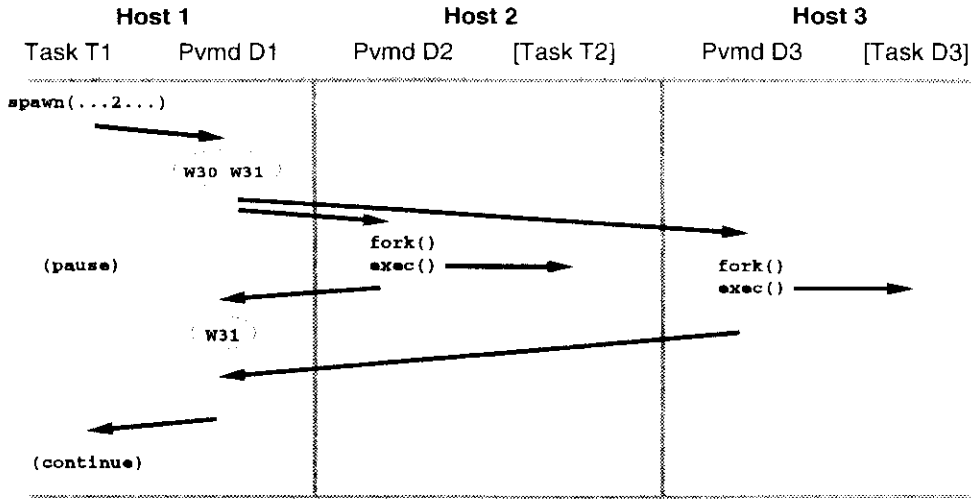


- Host part determines "owning" host of tid
- Host numbers synchronized by global host table
- Local part defined by each pvmd – key to MPP ports
- Routing a message is easy
  Route to correct pvmd, forwards to task
- Local part can be subdivided, e.g.:



Wait Contexts

- Wait contexts save state
- Pvmd can't block while waiting
- Wait context has unique ID
- Waits are "peered" for multi–host operations

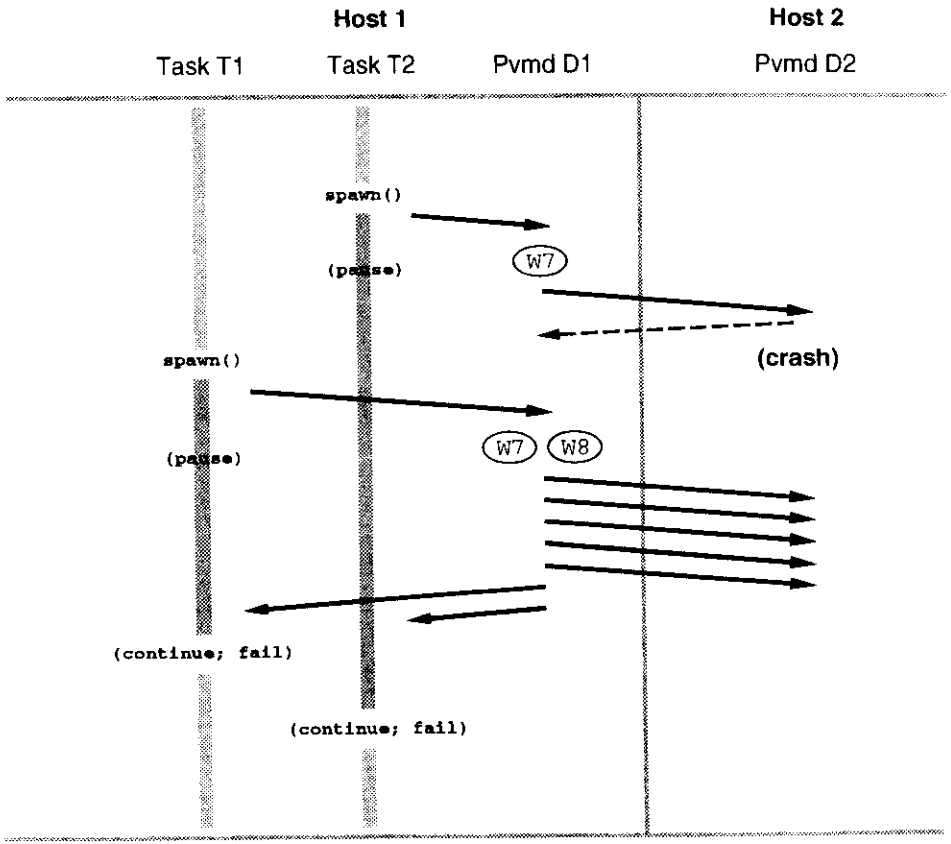


Fault Detection

- Pvmd recovers automatically,
  Application must do own recovery
- Model: dead host stays dead
  Host can be readded later
- Driven by pvmd–pvmd message system
- Triggered by retry timeout
- Libpvm operations won't hang
- Task can request to be notified

Fault Recovery

Message graph showing fault recovery:



TCP vs. UDP

What protocols are available?

- UDP
  - Datagrams of limited length
  - Unreliable delivery
  - No circuit required
- TCP
  - Stream of unbounded length
  - Reliable
  - Requires a circuit
- Others ?
  - Not readily available.

Protocols Used

- Pvmd–Pvmd uses UDP
  - Must scale to 100s of hosts
  - Don't need high bandwidth
  - D–D messages are "packet–like'
  - Want to do fault detection
  - TCP may still be the right choice ?

can set up connections with no N**2 end–end communication

- Task–Pvmd uses TCP [unlike v2]
  - UDP isn't reliable even within a host, and
  - Model: Task can't be interrupted
- Task–Task uses TCP
  - Tests showed UDP can reach or surpass TCP performance, but...
  - This beats up the hosts
  - Don't want to reinvent TCP
  - Should encourage clueful vendors

Changes in V.3.2

Majority of changes in version 3.2 are under the hood

A few changes affect the API

- `pvm_advise()`, `pvm_setdebug()`, `pvm_serror()`

  will be combined into a single function:

  `pvm_setopt(int what, int val)`

  What is one of:

  `PvmSetRoute`

  `PvmSetDebug`

  `PvmSetAutoErr`

  and possibly:

  `PvmSetFragSize`

  `PvmGetFragSize`

- `struct hostinfo` loses the `hi_mtu` field

  because it wasn't useful

- Libfpvm functions `pvmfconfig()`, `pvmftasks()` have more parameters

  Are now called nhosts (or ntasks) times,

  return all fields of struct hostinfo and struct taskinfo

Work In Progress

Work going into next version

- More scalability – 100s of hosts

  Parallel startup

- Better debugging support

  Libpvm generates trace messages

- Better high–speed network utilization

  FDDI fiber – DEC Alpha

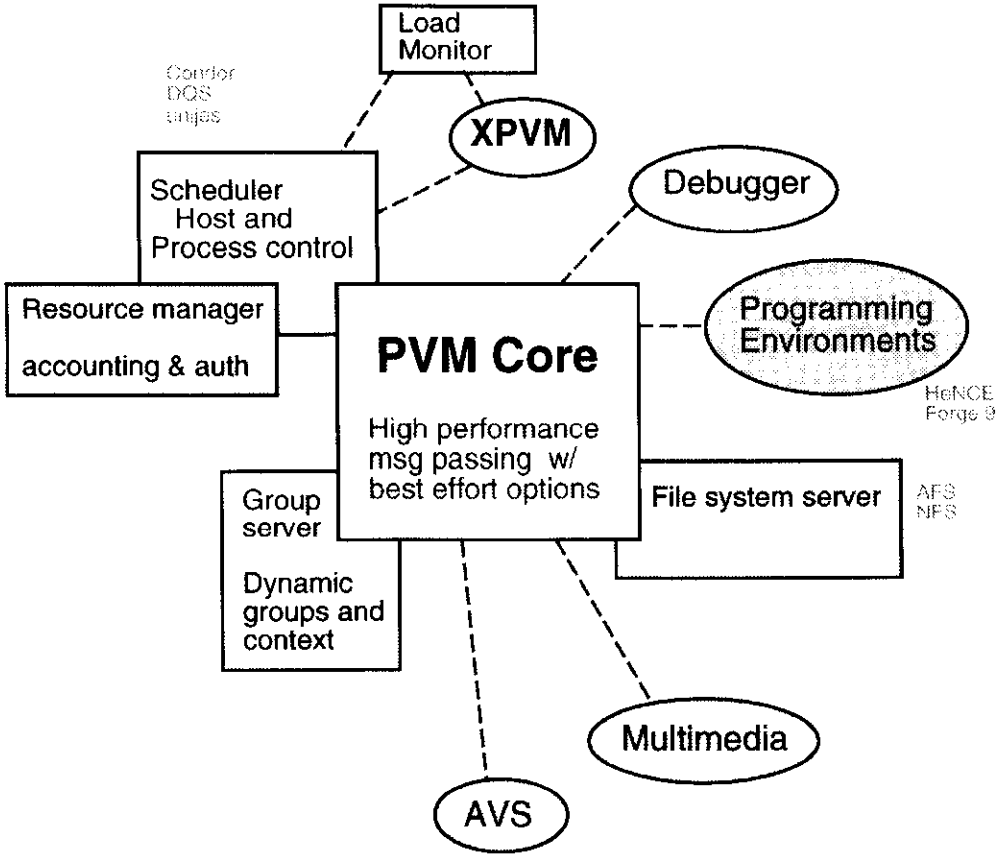
- Better console program

  Quoting, command aliases and history

Other work

- Load balancing
- Automatic application fault tolerance
- X–Window console (Tk–based ?)
- TCL–based shell

Future: Big Picture



References

Internet RFCs

768 Postel, J. B. User Datagram Protocol

791 Postel, J. B. Internet Protocol

793 Postel, J. B. Transmission Control Protocol

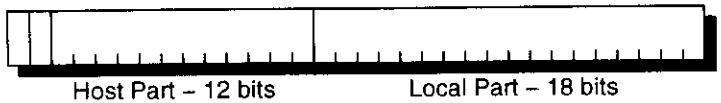
1014 Sun Microsystems, Inc. XDR: External Data Representation standard

1323 Jacobson, V.; Braden, R.; Borman, D. TCP Extensions for High Performance

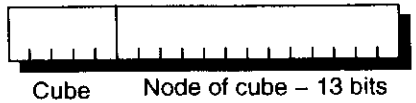
PVM 3.0 User's Guide and Reference Manual

Version 3 Implementation

- Task Identifier is 32–bit integer:



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



The PVM System

System is composed of:

- Pvmd daemon program

  Runs on each host of virtual machine

  Provides inter–host point of contact

  Authenticates tasks

  Execs processes on host

  Provides fault detection

  Mainly a message router, but is also
  a source and sink of messages

  More robust than application components

The PVM System

System is composed of:

- Libpvm programming library

  Linked with each application component (program)

  Kept as simple as possible

  Functions are low–level PVM "syscalls"

- Application components

  Written by user in PVM message–passing calls

  Are executed as PVM "tasks"