PVM and MPI

Jack Dongarra
University of Tennessee
and
Oak Ridge National Laboratory

- **PVM Parallel Virtual Machine**
  - **XPVM**
- **MPI Message Passing Interface**
  - **Standards activity**

Heterogeneous Network Computing

The goal of the project is to develop parallel programming tools to assist computational scientist in solving problems on a network of heterogeneous computers.

- **Programmer explicitly specifies parallelism.**
- **High latency + Low bandwidth + Fast processor => Coarse Grain**

- **PVM Low level - Message Passing**
  - **Designed by a small team of people**
  - **Scientific computing**
  - **Heterogeneous computing**
- **MPI Low level - Message Passing**
  - **Designed by community**
  - **Not only scientific computing**

University of Tennessee

Oak Ridge National Lab

Carnegie Mellon University

Pittsburgh Supercomputer Center

Emory University

Carolyn Aebischer

Adam Beguelin

Al Geist

Jack Dongarra

Weicheng Jiang

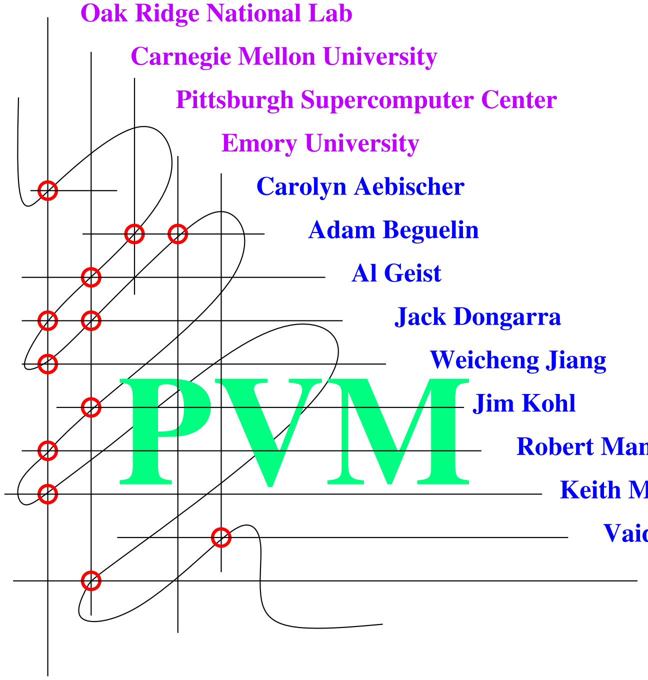
Jim Kohl

Robert Manchek

Keith Moore

Vaidy Sunderam

Honbo Zhou



What is PVM?

- Library (Fortran and C)
  - Message send/receive routines.
  - Message creation (packing) and unpacking routines.
  - Task create and control routines.
  - Others.
- Runs on heterogeneous collections of machines.
  - Workstations.
  - Small scale (shared memory) SMPs.
  - Large scale parallel machines (Paragon, T3D, SP-2)
- Virtual parallel machine: collection of machines used as single parallel computer.
- Console: Interactive PVM control program.
  - Manage VM – add/delete hosts.
  - Start/kill tasks running on the VM.
  - Trace task behavior.

Hello World

Simple example 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("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();
}
```


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:

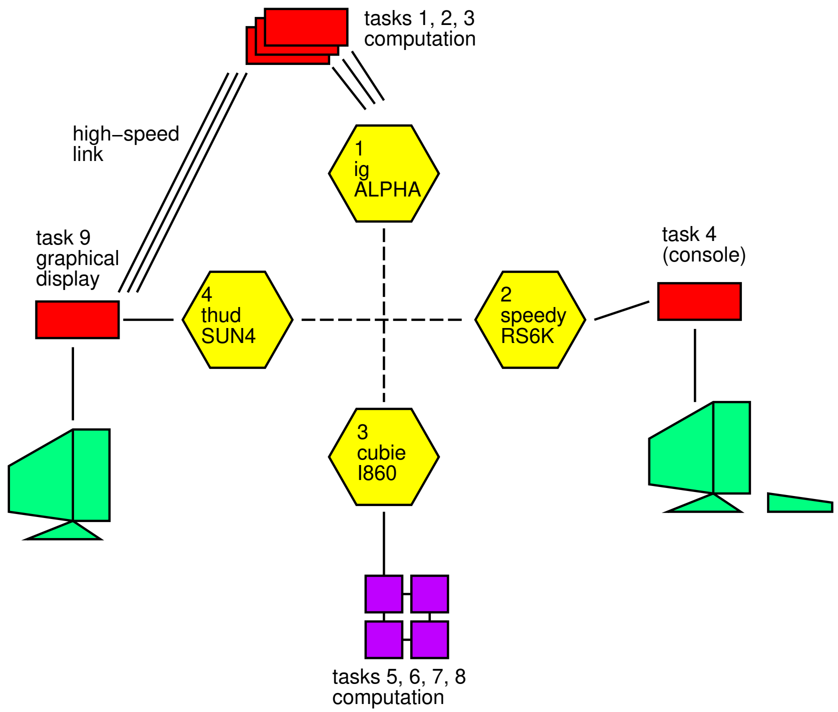
80386/486 with BSDI	Alliant FX/8
DEC Alpha/OSF−1	BBN TC2000
DEC Microvax	Convex
DECstation	Cray YMP and C90
DG Aviion	IBM 3090
HP 9000/300	Intel Paragon
HP 9000/700	Intel iPSC/2
IBM RS/6000	Intel iPSC/860
IBM/RT	Kendall Square KSR−1
NeXT	Sequent Symmetry
Silicon Graphics IRIS	Stardent Titan
Sun 3	Thinking 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

PVM In Operation



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

The PVM System

System is composed of:

- Pvmd daemon program

  Runs on each host of virtual machine

  Provides inter–host point of contact

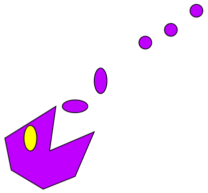
  Authenticates tasks

  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"

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

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.

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

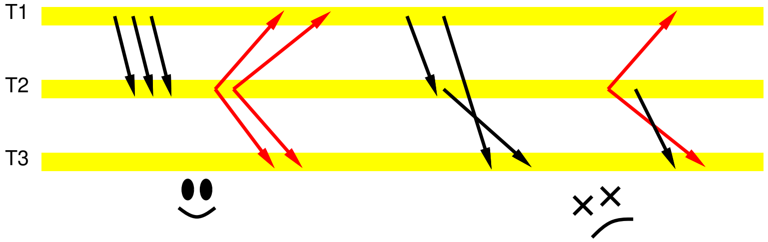
Recv picks messages by sender and/or code can pull messages from queue out of order

Message Order

PVM preserves the order or delivery of two messages sent from one task to another

But it doesn't preserve global logical order

Currently, multicasts [`pvm_mcast()`] may cross regular direct sends [`pvm_send()`]



Receiving a message is separate from delivering it

Messages can be received in any order by picking on TID or Code

Use different message code if you must ensure order

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

Pllizing Applications

How should you Parallelize your application?

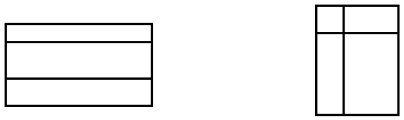
As a sort of "Network assembly language",
PVM leaves the decisions up to you.

- What to put in "component" tasks
- What messages to pass between tasks
- How many tasks to use, where to run them

Some applications divide up naturally into, f.e.:

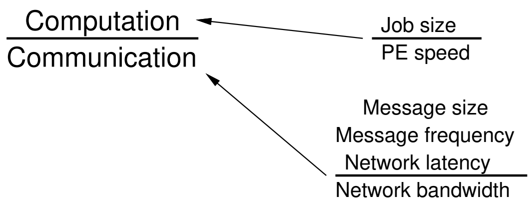
- Computation and interactive components.
- Independent worker tasks

Look for separable calculations:



Pllizing Applications

Parallel speedup is determined by ratio of:



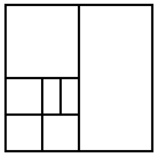
Divide and Conquer

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:

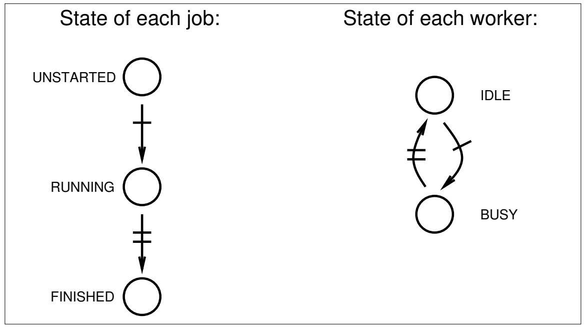


Using a pool of workers may be faster than actually creating new tasks

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

PVM

- From Netlib, XNetlib, or FTP get:
  - pvm3.3.4.tar.z.uu
- uudecode pvm3.3.8.tar.z.uu
- uncompress pvm3.3.8.tar.Z
- tar pvm3.3.8.tar
- cd pvm3
- make
- set path = (. $HOME/pvm3/lib/SUN4)
- pvm
  - add ens gin thud.cs.utk.edu
  - from another window start program
  - $HOME/pvm3/bin/SUN4/xxx

```
      PROGRAM INPROD
*
*  INPROD performs a parallel inner product.
*
*     .. External Subroutines ..
      EXTERNAL PVMFMYTID, PVMFPARENT, PVMFSPAWN, PVMFEXIT, PVMFINITSEND
      EXTERNAL PVMFPACK, PVMFSEND, PVMFRECV, PVMFUNPACK
*
*     .. External Functions ..
      REAL SDOT, SINPROD
      EXTERNAL SDOT, SINPROD
*
*     .. Intrinsic Functions ..
      INTRINSIC MOD
*
*     .. Parameters ..
      INTEGER MAXN
      PARAMETER( MAXN = 8000 )
      INCLUDE 'fpvm3.h'
*
*     .. Scalars ..
      INTEGER N, MYTID, MYPROC, NPROCS, IBUF, IERR
      INTEGER I, J
      REAL SIP, PIP
      CHARACTER*1 ERRCHAR
      LOGICAL ERRCHK
*
*     .. Arrays ..
      INTEGER TIDS(0:63)
      REAL X(MAXN), Y(MAXN)
*
*     Enroll in PVM and get my and the master process' task ID number
*
      CALL PVMFMYTID(MYTID)
      CALL PVMFPARENT(TIDS(0))
*
*     If I need to spawn other processes (I am master process)
*
      IF (TIDS(0) .EQ. PVMNOPARENT) THEN
*
*        Get starting information
*
         WRITE(*,*) 'How many processes should participate (1-64)?'
         READ(*,*) NPROCS
         WRITE(*,2000) MAXN
         READ(*,*) N
         WRITE(*,*) 'Do you wish to perform error checking?'
         READ(*,*) ERRCHAR
         ERRCHK = (ERRCHAR.EQ.'Y' .OR. ERRCHAR.EQ.'y')
         MYPROC = 0
         TIDS(0) = MYTID
*
         DO 10 I = 1, NPROCS-1
*
```



```
*           Spawn process and check for error
*
            CALL PVMFSPAWN('sinprod', 0, 'anywhere', 1, TIDS(I), IERR)
            IF (IERR .NE. 1) THEN
               WRITE(*,*) 'ERROR, could not spawn process #',I,
     $                    '.  Dying . . .'
               CALL PVMFEXIT(IERR)
               STOP
            END IF
*
*           Send out startup info
*
            CALL PVMFINITSEND(PVMDEFAULT, IBUF)
            CALL PVMFPACK(INTEGER4, N, 1, 1, IERR)
            CALL PVMFPACK(INTEGER4, I, 1, 1, IERR)
            CALL PVMFPACK(INTEGER4, NPROCS, 1, 1, IERR)
            CALL PVMFPACK(INTEGER4, ERRCHK, 1, 1, IERR)
            CALL PVMFSEND(TIDS(I), 0, IERR)
10       CONTINUE
*
*        All other processes should check in with spawning process
*
         DO 20 I = 1, NPROCS-1
*
*           Recv slave task ID from process number I
*
            CALL PVMFRECV(TIDS(I), 1, IBUF)
*
*           Unpack and make sure received ID agrees with IDs returned by spawn
*
            CALL PVMFUNPACK(INTEGER4, J, 1, 1, IERR)
            IF (J .EQ. TIDS(I)) THEN
               WRITE(*,1000) MYTID, I, TIDS(I)
            ELSE
               WRITE(*,*) 'Data corruption during checkin: tid, '//
     $                    'should be =', J, TIDS(1)
               WRITE(*,*) 'Error during checkin phase, aborting run'
               CALL PVMFEXIT(IERR)
               STOP
            END IF
20       CONTINUE
*
*     If I am a process who was spawned by master process
*
      ELSE
*
*        Receive startup info
*
         CALL PVMFRECV(TIDS(0), 0, IBUF)
         CALL PVMFUNPACK(INTEGER4, N, 1, 1, IERR)
         CALL PVMFUNPACK(INTEGER4, MYPROC, 1, 1, IERR)
         CALL PVMFUNPACK(INTEGER4, NPROCS, 1, 1, IERR)
         CALL PVMFUNPACK(INTEGER4, ERRCHK, 1, 1, IERR)
*
```

```
*       Send my task ID back as acknowledgement that process started correctly
*
        CALL PVMFINITSEND(0, IBUF)
        CALL PVMFPACK(INTEGER4, MYTID, 1, 1, IERR)
        CALL PVMFSEND(TIDS(0), 1, IERR)
      END IF
*
*     Everybody generates same X & Y
*
      CALL SGENMAT(N, 1, X, N, TIDS(0), N, NPROCS, N/NPROCS)
      CALL SGENMAT(N, 1, Y, N,  N/NPROCS, TIDS(0), TIDS(0)/NPROCS,
     $            NPROCS*N)
*
*     Compute the inner product in parallel
*
      PIP = SINPROD(N, X, Y, MYPROC, NPROCS, TIDS)
*
*     Perform error checking
*
      IF (ERRCHK) THEN
*
*       Compute the inner product sequentially
*
        SIP = SDOT(N, X, 1, Y, 1)
        IF (MYPROC .EQ. 0)
     $      WRITE(*,*) '<x,y> is sequential inner product, <x^y^>'//
     $                 ' is parallel product'
        WRITE(*,*) '|<x,y> - <x^,y^>| = ', ABS(SIP - PIP)
      END IF
*
      IF (MYPROC .EQ. 0) WRITE(*,*) 'DONE INPROD'
      CALL PVMFEXIT(IERR)
*
1000  FORMAT(I10,' Successfully spawned process #',I2,', TID =',I10)
2000  FORMAT('Enter the length of vectors to multiply (1 -'I7,'):')
      STOP
*
*     End program INPROD
*
      END
*
      REAL FUNCTION SINPROD(N, X, Y, MYPROC, NPROCS, TIDS)
*
*  PVM example routine written by Clint Whaley on 6/28/93
*
*     .. Scalar Arguments ..
      INTEGER N, MYPROC, NPROCS
*
*     .. Array Arguments ..
      INTEGER TIDS(0:*)
      REAL X(*), Y(*)
*
*  Purpose
*  =======
```



```
*  Returns the inner product of <y,x>, which is computed in parallel.
*
*  Arguments
*  =========
*
*  N         (input) INTEGER
*            Number of elements in X and Y.
*
*  X         (input) REAL, dimension (N)
*            1st vector to multiply.
*
*  Y         (input) REAL, dimension (N)
*            2nd vector to multiply.
*
*  MYPROC    (input) INTEGER
*            Integer ID between 0 (master) and NPROCS-1
*
*  NPROCS    (input) INTEGER
*            The number of processes working on the problem.
*
*  TIDS      (input) INTEGER, dimension (NPROCS)
*            Array of task IDs (assigned by PVM) of processes.
*
*  ======================================================================
*
*     .. External Subroutines ..
      EXTERNAL PVMFINITSEND, PVMFPACK, PVMFSEND, PVMFRECV
      EXTERNAL PVMFUNPACK, PVMFMCAST
*
*     .. External Functions ..
      REAL SDOT
      EXTERNAL SDOT
*
*     .. Intrinsic Functions ..
      INTRINSIC MOD
*
*     .. Parameters ..
      INCLUDE 'fpvm3.h'
*
*     .. Local Scalars ..
      INTEGER LN, IG, IERR, K
      REAL STMP, STMP2
*
*     Executable Statements
*
*     LN is the number of elements to be multiplied locally.  Process 0
*     handles any extra rows; IG is global index into X & Y.
*
      LN = N / NPROCS
      IF (MYPROC .EQ. 0) THEN
        LN = LN + MOD(N, NPROCS)
        IG = 1
      ELSE
        IG = LN*MYPROC + MOD(N, NPROCS) + 1
```

```
      END IF
      STMP = SDOT(LN, X(IG), 1, Y(IG), 1)
*
*     If I am not master, send my part of inner product to master,
*     and get back global answer.
*
      IF (MYPROC .NE. 0) THEN
         CALL PVMFINITSEND(PVMDEFAULT, IERR)
         CALL PVMFPACK(REAL4, STMP, 1, 1, IERR)
         CALL PVMFSEND(TIDS(0), 100, IERR)
         CALL PVMFRECV(TIDS(0), 101, IERR)
         CALL PVMFUNPACK(REAL4, STMP, 1, 1, IERR)
*
*     If I am master, recv parts of inner product, sum up, and broadcast answer
*
      ELSE
         DO 10 K = 1, NPROCS-1
            CALL PVMFRECV(-1, 100, IERR)
            CALL PVMFUNPACK(REAL4, STMP2, 1, 1, IERR)
            STMP = STMP + STMP2
10       CONTINUE
         CALL PVMFINITSEND(PVMDEFAULT, IERR)
         CALL PVMFPACK(REAL4, STMP, 1, 1, IERR)
         CALL PVMFMCAST(NPROCS-1, TIDS(1), 101, IERR)
      END IF
*
      SINPROD = STMP
      RETURN
*
*     END OF SINPROD
*
      END
*
*
```


XPVM Console

Graphical PVM console which will allow you to:

- Manage the virtual machine – add/delete hosts, reset
- Spawn and kill tasks
- Visualize an application running

XPVM graphically displays trace data generated by the pvmds and tasks allowing you to see:

- Spawn calls
- Task-task messages
- Task-task signals

Trace data is generated by the system automatically, can also be generated by the application

Tracing is normally switched off for maximum performance but can be enabled dynamically without recompiling or even restarting the application

XPVM Trace Display

XPVM displays a per-host or per-task graph showing program activity

The graph can be zoomed in or out to the desired level of detail

Status of each component determines color of bar graph

Communication (messages or control) is shown as an arc between components

XPVM will be extended with other functions

Planned PVM 3 Research

- **Visualization**

  XPVM GUI provides network and application views.
  Use for monitoring, debugging, tuning applications.

- **Fault Tolerance.**

  Task migration across VM
  Checkpoint restart capabilities for applications.

- **Multimedia**

  Extend PVM capabilities to include distributed multimedia applications by adding 'best effort' real–time communication.

- **Performance.**

  Focus on portable high performance message passing core
  Exploit high–speed networks and protocols when available
  Network multicast and multiple transport layers
  Use shared–memory when appropriate for data movement
  Investigate lightweight thread implementations of core.

Other Research Areas

Scalability

Fault Tolerance

Distributed Shared–memory

Heterogeneous Load Balancing

Resource Management – for example DQS

High speed, low latency network interfaces

Wide–area distributed computing