Parallel Processing using PVM on a Linux Cluster

Thomas K. Gederberg

CENG 6532 Fall 2007

What is PVM?

- PVM (Parallel Virtual Machine) is a "software system that permits a heterogeneous collection of Unix computers networked together to be viewed by a user's program as a single parallel computer".[1]
- Since the distributed computers have no shared memory, the PVM system uses Message Passing for data communication among the processors.
- PVM and MPI (Message Passing Interface) are the two common message passing API's for distributed computing.
- MPI, the newer of the two API's, was designed for homogenous distributed computers and offers higher performance.
- PVM, however, sacrifices performance for flexibility. PVM was designed to run over a heterogeneous network of distributed computers. PVM also allows for fault tolerance – it allows for the detection of, and reconfiguration for, node failures.
- PVM includes both C and Fortran libraries.
- PVM was developed by *Oak Ridge National Laboratory*, the *University of Tennessee*, *Emory University*, and *Carnegie Mellon University*.

Installing PVM in Linux

- PVM is easy to install on a machine running Linux and can be installed in the users home directory (does not need root or administrator privileges).
- Obtain the PVM source code (latest version is 3.4.5) from the PVM website: http://www.csm.ornl.gov/pvm/pvm_home.html
- Unzip and untar the package (pvm3.4.5.tgz) in your home directory – this will create a directory called pvm3
- Modify your startup script (.bashrc, .cshrc, etc.) to define:

  $PVM_ROOT = $HOME/pvm3

  $PVM_ARCH = LINUX
- cd to the pvm3 directory
- Type make
- The *makefile* will build pvm (the PVM console), pvmd3 (the pvm daemon), libpvm3.a (PVM C/C++ library), libfpvm3.a (PVM Fortran library), and libgpvm3.a (PVM group library) and places all of these files in the $PVM_ROOT/lib/LINUX directory.
- The *makefile* will also build pvmgs (PVM group server) and place it in the $PVM_ROOT/bin/LINUX directory.

Configuring the Cluster

- The Linux cluster consists of three machines running Linux (Ubuntu 7.04) connected via a wireless router.
- Recommend using SSH (Secure Shell) rather than RSH (Remote Shell) as the communication protocol. Generate a public/private key on the main machine and copy the key to the .ssh directory on each of the remote machines. This will allow for passwordless communication.
- Install PVM on each machine.
- In the ~/.pvm3 directory on each machine, create a .rhosts file that lists the name of each of the machines in the cluster (one name per line). The /etc/host file on each machine should list the machine name and its IP address.

Overview of Common PVM Routines

- Process Control

int tid = pvm_mytid()

- The routine pvm_mytid() returns the TID (task identifier) of this process and enrolls the process into PVM if this is the first PVM call.

int info = pvm_exit()

- The routine pvm_exit() tells the PVM daemon that this process is leaving PVM. Typically, pvm_exit is called before exiting the C program.

int numt = pvm_spawn(char *task, char **argv, int flag, char* where, int ntask, int *tids)

- The routine pvm_spawn() starts up *ntask* copies of the executable *task* on the virtual machine. *argv* is a pointer to an array of arguments to *task*. The *flag* argument is used to specify options such as *where* the tasks should be spawned, whether to start a debugger, or whether to generate trace data.

Overview of Common PVM Routines (continued)

- Information

int tid = pvm_parent()

- The routine pvm_parent() returns the TID of the process that spawned this task or the value of PvmNoParent if not created by pvm_spawn (i.e., tid will equal PvmNoParent if this process is the parent process).

int dtid = pvm_tidtohost(int tid)

- The routine pvm_tidtohost() returns the TID of the daemon running on the same host as tid – useful for determining on which host a given task is running.

int info = pvm_config(int *nhost, int *narch, struct pvmhostinfo **hostp)

- The routine pvm_config() returns information about the virtual machine including the number of hosts, *nhost*, and the number of different architectures, *narch*. *hostp* is a pointer to a user declared array of *pvmhostinfo* structures.

Overview of Common PVM Routines (continued)

- Message Passing

  Sending a message consists of three steps:

  - the send buffer must be initialized
  - the message must be “packed” into this buffer
  - the completed message is sent

- Initializing the Send Buffer

  int tid = pvm_initsend(int encoding)

  - The routine pvm_initsend() clears any current send buffer and creates a new one for packing a message. The encoding scheme used is set by *encoding*. XDR (the *External Data Representation* standard) encoding scheme is used by default.

Overview of Common PVM Routines (continued)

- Message Passing (continued)
- Packing the 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 data into a single message. In each routine, the first argument is a pointer to the item in the array, *nitem* is the number of items in the array to pack, and *stride* is the stride to use when packing.

```
int info = pvm_pkbyte( char *cp, int nitem, int stride )
int info = pvm_pkcplx( float *xp, int nitem, int stride )
int info = pvm_pkdcplx( double *zp, 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 )
```

Overview of Common PVM Routines (continued)

- Message Passing (continued)
- Sending and Receiving Data

int info = pvm_send(int tid, int msgtag)

- The routine pvm_send() labels the message with an integer identifier *msgtag* and sends it immediately to the process with the task identifier of *tid*.

int bufid = pvm_recv(int tid, int msgtag)

- The routine pvm_recv is a blocking receive that will wait until a message with label *msgtag* has arrived from the process with the task identifier of *tid*. A value of -1 in *msgtag* or *tid* matches anything (wildcard).

int bufid = pvm_nrecv(int tid, int msgtag)

- The routine pvm_nrecv is a non-blocking receive. If the message has not yet arrived, pvm_nrecv returns *bufid* = 0. This routine can therefore be called multiple times until the message has arrived, while performing useful work between calls.

Overview of Common PVM Routines (continued)

- Message Passing (continued)

- Unpacking the Data
  - Each of the following routines unpacks an array of the given data from the active receive buffer. They can be called multiple times to unpack data from a single message. In each routine, the first argument is a pointer to the item in the array, *nitem* is the number of items in the array to unpack, and *stride* is the stride to use when unpacking.

```
int info = pvm_upkbyte( char *cp, int nitem, int stride )
int info = pvm_upkcplx( float *xp, int nitem, int stride )
int info = pvm_upkdcplx( double *zp, 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 )
```

Example PVM Program (psdot.c)

- The example program, psdot.c, performs a parallel dot product of two vectors, each containing 8000 elements.
- The program divides up the work of the dot product into three nearly equal pieces (one piece to be executed by the master process executing on the controlling computer and the other two pieces executing by worker processes on the two remote computers).
- Each worker will get N/P vector elements to process where N = 8000 (the size of each vector) and P = 3 (the number of processors). The master will keep N/P + mod(N/P) vector elements for itself to process. Therefore the task is split up as follows:
  - Worker #1 has 8000/3 = 2666 elements
  - Worker #2 has 8000/3 = 2666 elements
  - Master has 8000/3 + mod(8000/3) = 2666 + 2 = 2668 elements
  - Total is 2666 + 2666 + 2668 = 8000

Example PVM Program (continued)

- After determining how to split up the work, the program spawns two copies of itself on each of the two worker computers.
- Worker #1 receives 2666 elements (elements 2668 through 5333) of vectors *X* and *Y* and Worker #2 receives 2666 elements (elements 5334 through 7999) of vectors *X* and *Y*. The Master keeps elements 0 through 2667 of *X* and *Y* for itself.
- Each Worker will compute the dot product of its own subvectors and send the result back to the Master. The Master will compute the dot product of its subvector and add the result to the dot products received from each of the two Workers to produce the total dot product.
- To verify that the parallel dot product was computed correctly, the Master program also computes the total dot product alone for comparison.

psdot.c source code

```
#include "pvm3.h“                          /* pvm3 header file */
#include <stdio.h>
#include <stdlib.h>

#define N 8000                             /* size of the vectors X and Y */
#define P 3                                /* number of processors */

double dot(int n, double x[], double y[]);
void randvec(double *x, int seed);

int main()
{
  double x[N], y[N], mdot, wdot, sdot;
  int tids[N], mytid, tid, numt, status, istart, proc, mn, wn;

  /* pvm_mytid() enrolls the process into PVM on its first call
     and generates a unique task id if this process was not
     created by pvm_spawn. */

  mytid = pvm_mytid();
```

psdot.c source code (page 2 of 6)

```
/* Check to see if I am the master process.  If I am the master process,
     then I need to spawn other processes */

  if (tids[0] == PvmNoParent) {

    tids[0] = mytid;

    randvec(x, mytid);                  /* Randomly generate x and y using mytid
    randvec(y, 2*mytid-1);                  and 2*mytid-1 as seeds */

    wn = N/P;                           /* # of elements to be processed by each worker */
    mn = wn + N % P;                    /* # of elements to be processed by the master */

    istart = mn;                        /* starting index for elements to be processed by
                                            the workers.  istart starts at mn since the
                                            the master will process the 0..mn-1 elements */

    printf("\nNumber of elements in each vector = %i\n", N);
    printf("Number of processors = %i\n", P);
    printf("Number of elements for each worker (wn) = %i\n", wn);
    printf("Number of elements for the master (mn)  = %i\n\n", mn);
```

psdot.c source code (page 3 of 6)

```
/* Loop over all worker processes such that proc = 1,2,...,P-1 = worker #.
    Process P is the master. */

 for (proc = 1; proc < P; proc++) {                /* Spawn worker processes */
   numt = pvm_spawn("psdot",(char **)0,PvmTaskDefault,"",1,&tid);
   if (numt != 1) {
     printf("ERROR: could not spawn process # %i.  Dying...\n", proc);
     pvm_exit();           return 1;
   } else {
     tids[proc] = tid;                       /* save task id of spawned task */
     printf("Spawned worker %i with task id = %i\n", proc, tid);
   }

   /* Send messages to Workers.  Values returned (status) should be ≥ 0) */

   status = pvm_initsend(PvmDataDefault);    /* use XDR encoding */
   status = pvm_pkint(&wn,1,1);              /* pack wn */
   status = pvm_pkdouble(&x[istart],wn,1);   /* pack wn elements of x[] starting at x[istart] */
   status = pvm_pkdouble(&y[istart],wn,1);   /* pack wn elements of y[] starting at y[istart] */
   status = pvm_send(tids[proc],0);          /* send the package to worker with task id
                                                of tids[proc] */
   istart += wn;                             /* advance istart wn elements for the next worker */
 }    /* end for (proc = 1; proc < P; proc++) */
```

psdot.c source code (page 4 of 6)

```
mdot = dot(mn,x,y);                    /* Compute master's part of the dot product */
printf("\nMaster computed partial dot product of %f\n", mdot);

/* Receive the dot products from each of the workers and add to get the
   total dot product */

for (proc = 1; proc < P; proc++) {
  status = pvm_recv(-1,1);                    /* receive a package from a worker */
  status = pvm_upkdouble(&wdot,1,1);          /* unpack the workers dot product */
  printf("Worker returned partial dot product of %f\n", wdot);
  mdot = mdot + wdot;                         /* add workers dot product to masters dot product */
}

/* Print out the result */

printf("\nParallel result for <x,y>   = %f\n", mdot);

/* Now computer the dot product sequentially for comparison */

sdot = dot(N,x,y);
printf("Sequential result for <x,y> = %f\n", sdot);
}
```

psdot.c source code (page 5 of 6)

```
else {

    /*************************** Worker Processing ****************************/

    status = pvm_recv(tids[0],0);                 /* receive message from master */
    status = pvm_upkint(&wn,1,1);                 /* unpack integer (1 item, stride = 1) */
    status = pvm_upkdouble(x,wn,1);               /* unpack subvector x (wn items, stride = 1) */
    status = pvm_upkdouble(y,wn,1);               /* unpack subvector y (wn items, stride = 1) */

    /* Compute local dot product and send it to master */

    wdot = dot(wn,x,y);

    status = pvm_initsend(PvmDataDefault);        /* use XDR encoding */
    status = pvm_pkdouble(&wdot,1,1);             /* pack the local dot product */
    status = pvm_send(tids[0],1);                 /* send the package */
  }
  pvm_exit();
  return 0;
}                                                 /***** end of main program ******/
```

psdot.c source code (page 6 of 6)

```
/*************** dot function ******************/

double dot(int n, double x[], double y[])
{
  int i;
  double temp = 0.0;
  for (i = 0; i < n; i++) {
    temp += x[i] * y[i];
  }
  return temp;
}
```

```
/*************** randvec function ***************/

void randvec(double x[], int seed)
{
  int i, sign;
  srand(seed);
  for (i = 0; i < N; i++) {
     if (rand()%2 == 0)
        sign = 1;
     else
        sign = -1;
     x[i] = sign*(double)rand()/(2.0e+6);
  }
}
```

Compiling/Linking psdot.c

- The psdot object code must be linked with the pvm3 C library:

  gcc psdot.c –lpvm3 –o psdot

- PVM assumes that the executable resides in $HOME/pvm3/bin/LINUX.

- The executable psdot should be placed in $HOME/pvm3/bin/LINUX on all machines in the cluster.

Running psdot.c

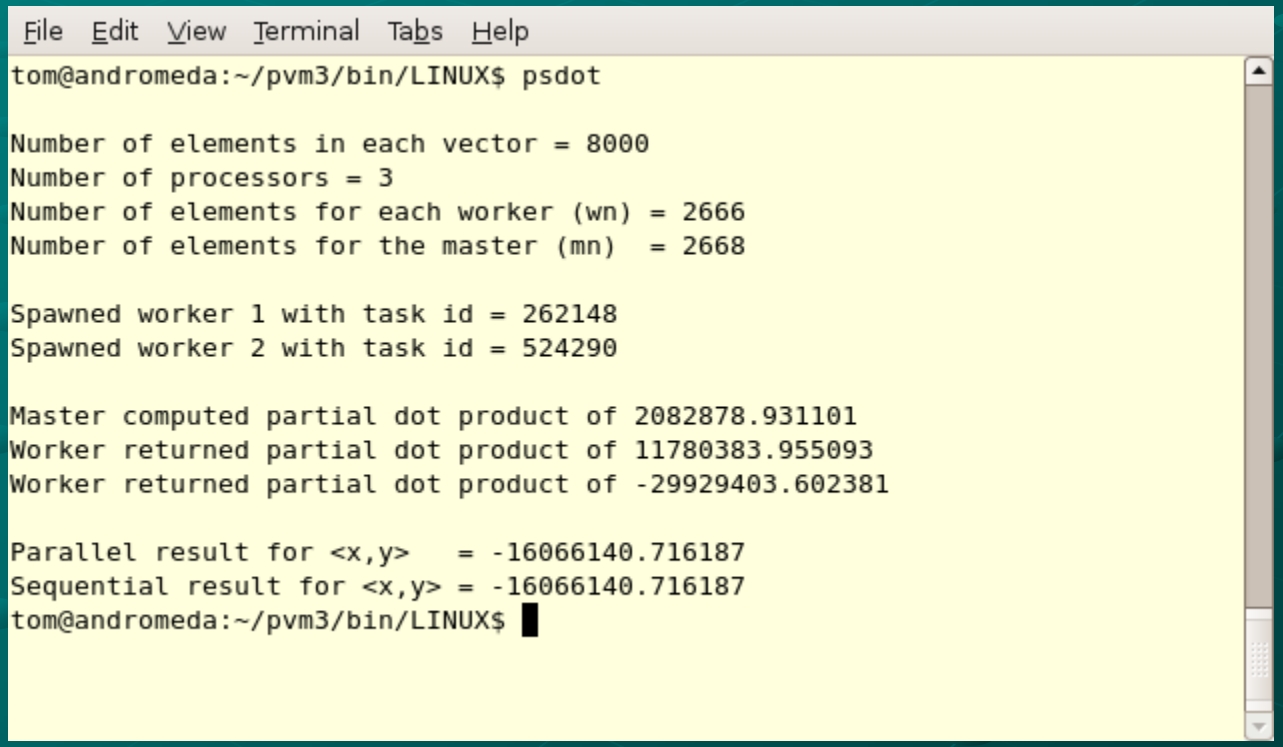
- Start the PVM daemon on the host and add the two remote machines (*orion* and *taurus*).
- Quit the PVM console (daemon still running).

```
File  Edit  View  Terminal  Tabs  Help
tom@andromeda:~/pvm3/bin/LINUX$ pvm
pvm> add orion
add orion
1 successful
                    HOST     DTID
                   orion    80000
pvm> add taurus
add taurus
1 successful
                    HOST     DTID
                  taurus    c0000
pvm> conf
conf
3 hosts, 1 data format
                    HOST     DTID     ARCH   SPEED       DSIG
                andromeda    40000    LINUX    1000 0x00408841
                    orion    80000    LINUX    1000 0x00408841
                   taurus    c0000    LINUX    1000 0x00408841
pvm> quit
quit

Console: exit handler called
pvmd still running.
tom@andromeda:~/pvm3/bin/LINUX$
```

Running psdot.c

- Execute the psdot program.



Conclusions

- PVM is a free, well documented, and easy to install message passing API for distributed computing.
- PVM is well suited for learning parallel programming.

References

1. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam, PVM: Parallel Virtual Machine – A User's Guide and Tutorial for Networked Parallel Computing: The MIT Press; Cambridge, Massachusetts: 1994.
2. A. Geist, J. Kohl, P. Papadopoulos, PVM and MPI: a Comparison of Features, May 30, 1996, http://www.csm.ornl.gov/pvm/