A PVM Tutorial

From: A SURVEY OF COMPUTATIONAL PHYSICS
by RH Landau, MJ Paez, and CC Bordeianu.
Copyright Princeton University Press, Princeton, 2007.
Electronic Materials copyright: R Landau, Oregon State Univ, 2007; MJ Paez, Univ Antioquia, 2007; and CC Bordeianu, Univ Bucharest, 2007.
Support by National Science Foundation.

This tutorial is based on the Web tutorial of Hans Kowallik.

We recommend using MPI rather than PVM because it is more modern, more common, and somewhat higher-level. However, we are told that some users either prefer or have only PVM, and so we discuss it briefly here. PVM is a software system that allows you to combine a number of computers which are connected over a network into a *Parallel Virtual Machine*. This machine can consist of computers with different architectures, running different flavors of the Unix/Linux operating systems, and can still be treated as if it were a single parallel machine.

CONFIGURING PVM

Before using PVM, a user has to complete a number of configuration tasks. These task depend on the way PVM was installed on your system and the characteristics of your system. Instead of trying to cover all the different systems we will assume that:

1. You are running under a Unix/Linux operating system.
2. Your system has shared home directories, that is, you see the same set of files regardless of which computer you log onto.
3. PVM is already installed on all the machines you want to use.
4. PVM libraries are in the `/usr/lib` or `/usr/local/lib` directories, and that your compiler searches these automatically.
5. PVM *include files* in are the `/usr/include` directory, and that your compiler searches here automatically.

You can still use PVM if one or more of these are not true for you, however there are a several of things you have to do differently:

1. Find out which computers are available. You can use any computer on which you have an account, and on which PVM is installed.
2. Edit or create the file `.rhosts` in your home directory. This file must have an entry for every computer you want to use. The entry is in the form of the name of the computer and your login name on that machine:

   ```
   ucs.orst.edu kowallih
   daphy.physics.orst.edu hans
   goophy.physics.orst.edu hans
   mango.physics.orst.edu hans
   banana.physics.orst.edu hans
   coconut.physics.orst.edu hans
   papaya.physics.orst.edu hans
   ```

   If your login name is the same on all machines, then you can leave the field with the login name blank, but it doesn't hurt to put it in.
3. Set environment variables. If you use csh or tcsh, then add to your `.cshrc` file:

   ```
   setenv PVM_ROOT /usr/local/pvm3
   setenv PVM_ARCH `$PVN_ROOT/lib/pvmgetarch`
   setenv XPVM_ROOT /usr/local/pvm3/xpvm
   set path=($path $PVN_ROOT/lib)
   set path=($path $PVN_ROOT/lib/$PVN_ARCH)
   ```

   If there is no `pvm3` directory in `/usr/local`, then you have to change the first entry to whatever directory holds these files.
4. Create directories for your executables:

   ```
   > mkdir $HOME/pvm3/bin/PVM_ARCH
   ```

   where `PVM_ARCH` is the PVM code for the architecture. Although this step is not necessary for PVM to work, it will make your life much easier if you are going to use PVM on computers with different architectures. You can find the code for each computer's via the PVM function `pvmgetarch`.

Different System Configurations

No shared home directories In this case you have to repeat many of the steps described under general configuration on all the machines you want to use. Specifically, you must create the `.rhosts` file and add the environment variables to your `.cshrc` file.

PVM is not installed In this case you have to install it yourself or find someone to do it for you. In § we give some hints on how to install PVM in `/usr/lib` or `/usr/local/lib`. If you plan to use PVM frequently, you might want to put the libraries into these default directories. Otherwise you have to tell the compiler explicitly where it can find them:

```
> cc -o master master.c -lpvm3 -Lpath_to_your_libraries
```

PVM include files are not in /usr/include Again the best thing to do is put them there, otherwise compile with,

```
> cc -o master master.c -lpvm3 -Ipath_to_your_include_files
```

Notice the first character in `lpvm3` is a small `l` as in library, while the first character in `Ipath_to_your_include_files` is a capital `I` as in include.

THE PVM CONSOLE

The PVM console is the interface between the parallel virtual machine and the user. You use it to start and stop processes, to display information about the state of the virtual machine, and most importantly, to start and stop PVM on local and remote machines.

Step 1: Starting PVM Log into one of the computers you want to include in PVM and enter:

```
> pvm                                                Start PVM from command line
```

If PVM is properly installed, it will start and respond with its prompt:

```
pvm>                                                 The PVM prompt
```

Congratulations, you just created a parallel virtual machine of one physical machine. Of course this is rather useless, so let's extend our system.

Step 2: Adding hosts This is done from the PVM prompt:

```
pvm> add hostname                                    Add a host
```

where `hostname` is the name of the computer you want to add. This will start PVM on the specified hosts and, if successful, will produce a message such as:

```
1 successful
HOST DTID
banana 140000
```

You can continue to add additional hosts as desired.

Step 3: Checking your configuration You display the configuration of your parallel virtual machine from the PVM prompt:

```
pvm> conf
```

This will give you information about the hosts configured, their PVM identification number and their architecture.

Step 4: Deleting hosts Sometimes it is necessary to remove hosts from the virtual machine to test or debug a program:

```
pvm> delete hostname
```

where `hostname` is the name of the computer you want to delete.

Step 5: Leaving the console If you are done with setting up your virtual machine, and if you don't need any of the other functions of the console, you close the console but keep PVM running:

```
pvm> quit                                   Close console, not PVM
```

Step 6: Stopping PVM To stop PVM after your parallel program has finished, enter the PVM console, and then from the PVM prompt:

```
> pvm                                       From Unix shell
pvm> halt                                   From PVM prompt
```

This stops PVM on all the machines and kills all programs running under PVM. This is the best and easiest way to stop PVM.

FIRST PVM PROGRAM: MASTER-SLAVE COMMUNICATIONS

Problem: Write a program that determines the names and the local times of all the physical machines in the virtual machine, and prints that information to standard output.

Finally we can write and run our a PVM program. The most straightforward model for writing parallel programs using a message-passing systems such as PVM is with a *master* process and a *slave* process. The master is started by the user on one machine only. It then starts and controls processes on the other machines (slaves) that perform the work. The master's work includes:

- Determining which physical machines are part of the virtual machine.
- Starting a slave process on every physical machine to be used.
- Collecting the results which are sent back by the slaves.
- Printing the results to standard output.

C versions of the master and slave programs are given in Lsts. 1 and 2.

Listing 1 The PVM master **PVMcommunMaster.c** showing communication.

```
/* PVM master for simple communication; starts slave, get's t's */
#include <stdio.h>
#include <pvm3.h>
main() {
struct pvmhostinfo *hostp;
int result, check, i, nhost, narch, stid;
char buf[64];
pvm_setopt(PvmRoute, PvmRouteDirect);        // communication channel
gethostname(buf, 20);                   // get master's name
printf("The master process runs on %s \n", buf);
// get & display parallel machine configuration
pvm_config( &nhost, &narch, &hostp );        // get configuration
printf("I found following hosts in your virtual machine\n");
for (i = 0; i < nhost; i++)
{    printf("\t%s\n", hostp[i].hi_name);  }
for (i=0; i<nhost; i++)              // spawn processes
{  check=pvm_spawn("answer", 0,PvmTaskHost,hostp[i].hi_name, 1, &stid);
   if (!check) printf("Couldn't start on %s\n", hostp[i].hi_name); }
result=0;
while (result<nhost)
{  pvm_recv(-1, 2);          /* wait for reply message */
   pvm_upkstr(buf);          /* unpack message */
   printf("%s\n", buf);          /* print contents */
   result++;  }
pvm_exit;               /* we are done */
}
```

Each slaves' work include:

- Determining the name of the machine on which it is running.
- Determining the wall time on this machine.
- Sending a message with this information back to the master.

Listing 2 The PVM slave **PVMcommunSlave.c** showing communication.

```
/* PVM slave, returns machine name & local time */
#include <stdio.h>
#include <pvm3.h>
#include <time.h>
main() {
time_t now;
char name[12], buf[60];
int ptid;
ptid = pvm_parent();          /* the ID of the master process */
pvm_setopt(PvmRoute, PvmRouteDirect);
gethostname(name, 64);       /* find name of machine */
now=time(NULL);           /* get time */
strcpy(buf, name);       /* put name into string */
strcat(buf, "'s time is ");
strcat(buf, ctime(&now));    /* add time to string */
pvm_initsend(PvmDataDefault);    /* allocate message buffer */
pvm_pkstr(buf);           /* pack string into buffer */
pvm_send(ptid, 2);       /* send buffer to master */
pvm_exit;            /* slave is done and exits */
}
```

Let's do it. Compile the source code for a slave processes:

```
> cc -o answer PVMcommunSlave.c -lpvm3
```
Compile C PVM program

For Fortran users, the programs are `PVMbugMstr` and `PVMbugsSlave`, and we have placed all needed commands for setup and execution in a Makefile on the CD: ℭ𝔇

```
# Makefile for MSTR/WRKR program –– using PVM 3.3
# PVM's "architecture" classification  -- tailor to your system
```

```
ARCH     = $(PVM_ARCH)
# Location and names of PVM files  -- tailor to your system
PVMLOC   = /usr/local/pvm3
PVMLIB   = -L$(PVMLOC)/lib/$(ARCH) -lfpvm3 -lpvm3
# Name and options for FORTRAN compiler -- tailor to your system
FC       = f77
FFLAGS   = -O -I$(PVMLOC)/include
    all:    PVMbugMstr PVMbugsSlave
PVMbugMstr:    PVMbugMstr.o
    $(FC) -o $(@) $(FFLAGS) PVMbugMstr.o $(PVMLIB)
PVMbugsSlave:    PVMbugsSlave.o
    $(FC) -o $(@) $(FFLAGS) PVMbugsSlave.o $(PVMLIB)
strp:
    strip PVMbugMstr PVMbugsSlave
clean:
    rm -f *.o core *.lst
cleanall:
    rm -f PVMbugMstr PVMbugsSlave *.o core *.lst
```

Note, it is important that you call the executable `answer` because this is the name of the program that the master process tries to start. If you are using computers of different architectures, then you run this Makefile on one machine of every architecture, and immediately copy the executable into the architecture-specific directories you created in your configuration step.

Compilation Compile the source codes to obtain an executable `master`:

```
> cc -o master PVMcommunMaster.c -lpvm3
```

From Unix shell

Fortunately, you have to perform this compilation only once, and that is on the machine where you want to run the master.

Execution Now that you have: installed and configured PVM, created your parallel machine, compiled all programs, and put them into their places, all that is left is to start the master process from the unix prompt and get results:

```
> master
```

Start execution from Unix shell

```
The master process runs on mango
I found the following hosts in your virtual machine
    mango
    goophy
    daphy
    coconut
mango's time is Fri May 10 13:10:50 2007
daphy's time is Fri May 10 13:15:42 2007
coconut's time is Fri May 10 13:15:46 2007
goophy's time is Fri May 10 13:17:01 2007
```

Warning! At some point PVM may get confused. In those case it's a good idea to stop PVM and start it again. Sometimes in order to restart PVM, you may have to change to the directory `/tmp` and remove some of the PVM files you have created there (you can issue a long list command `ls -l` to see the names, owners, and creation times of files, and remove the files with the `rm` command).

Compiling Slave Programs On Different Machines

Creating all the necessary slave executables requires the following steps:

1. Compile a slave process on ARCH1 from the source directory:
   ```
   > cc -o answer PVMcommunSlave.c -lpvm3                Compile on architecture_1
   ```
   This creates the program `answer` in the current directory. Copy the executable into the directory for architecture_1's PVM executables:
   ```
   > cp answer $HOME/pvm3/bin/ARCH1
   ```
2. Compile a slave process on architecture_2 from the source directory:
   ```
   > cc -o answer PVMcommunSlave.c -lpvm3                Compile on architecture_2
   ```
   This creates the program `answer` in the current directory. Copy the executable into the directory for architecture_2's PVM executables:
   ```
   > cp answer $HOME/pvm3/bin/ARCH2
   ```
   This leaves us with two architecture-specific programs with the same name in different directories where PVM will find them.

THE BIFURCATION MAP; TRIVIALLY PARALLEL

The logistics map (§ **??**) is one of many chaotic systems whose study would be nearly impossible without the extensive use of computational resources. It is described by the simple equation,

$$x_{n+1} = \mu x_n (1 - x_n). \tag{0.1}$$

This produces the bifurcation diagram in Fig. 1 with the procedure outlined in § **??**, and repeated here:

1. Start at $\mu = 1.0$.
2. Pick an arbitrary starting value for x_0.
3. Use this x_i to calculate the next value x_{i+1} in the sequence (0.1).
4. Repeat the cycle 200 times to eliminate transient behaviors.
5. Repeat the cycle another 200 times, but now save the x values.
6. Increase μ by a small amount, say 0.01, and repeat the process from step 2 until $\mu = 4$ is reached.

The fact that the initial value for x is arbitrary explains why this problem is perfect for parallel processing. This means that the calculations for different μ values are independent and so can be run on different processors without any message passing. Again we will use the master and slave model for this problem.

The basic tasks of the **master** are trivial:

- Determine the configuration of the virtual machine.
- Start the slave processes on all the physical machines.
- Send the general parameters to slaves.
- Split the μ range into parts.
- Send the ranges of μ values to the slaves.
- Continue the computation until all μ values are covered.
- Wait for the slaves to finish their calculations.
- Tell the slaves to shut down.

Some points require additional discussion. First, each slave process also needs three parameters to perform its work. It has to know how long to wait to avoid transients, how many x values to calculate, and in how many subranges it should divide the μ range it is working on. Instead of building these values into the slave program, we make it easy to modify the program by having the master send these parameters to the slaves.

Second, we need to decide the number of subdivisions n into which we divide the μ range. This directly affects the performance of our parallel program; if we make n too large, then too much time is spent communicating with the slaves, rather than having the slaves busy working; if we make n too large, then all the processors may have to wait idly a long time for the last process to finish. The best value for n depends on the amount of overhead connected to starting a new task, the computing time required for each task, and the number of physical machines in your virtual machine. In this tutorial you should try different n values and see how it affects the total time.

Lst. 3 gives a PVM master program in C for creating a bifurcation plot.

Listing 3 The PVM master program **PVMbugsMaster.c** for creating a bifurcation plot.

```
/* master program for bifurcation diagram of logistic map*/
#include <stdio.h>
#include <pvm3.h>
#define min 1                   /* minimum for m */
#define max 4                   /* maximum for m */
#define step 0.1                /* m range for slave */
#define nstep 100.0             /* number of steps for slave */
#define skip 200                /* # results to skip */
#define count 300               /* # results to save */
main() {
struct pvmhostinfo *hostp;
int bufid, check, dum, i, nhost, narch, ptid, stid;
char name[64];
double buf[5], m;
ptid = pvm_mytid();          /* get PVM ID number */
pvm_config( &nhost, &narch, &hostp );   /* configure virtual machine */
gethostname(name, 64);
printf("The master process runs on %s \n", name);
printf("I found the following hosts in your virtual machine\n");
for (i = 0; i < nhost; i++)
{ printf("\t%s\n", hostp[i].hi_name); }
printf("\nStarting slaves\n");
for (i=0; i<nhost; i++)         /* start slaves on all hosts */
{  check=pvm_spawn("mapslave", 0,PvmTaskHost,hostp[i].hi_name, 1, &stid);
   if (!check)
   {  printf("Couldn't start process on %s\n", hostp[i].hi_name);
      nhost--;
      } }
pvm_setopt(PvmRoute, PvmRouteDirect);
buf[2]=nstep;                /* parameters for slaves */
buf[3]=skip;
buf[4]=count;
for(m=min; m<=max; m+=step)     /* m parameter for slaves */
{  printf("%f\n", m);               /* some feedback */
   bufid=pvm_recv(-1, 2);               /* slave is ready */
   pvm_bufinfo(bufid, &dum, &dum, &stid);    /* which machine? */
   buf[0]=m;                      /* min and max m */
   if((m+step)<max) buf[1]=m+step;
   else buf[1]=max;
   pvm_initsend(PvmDataDefault);         /* send parameters */
   pvm_pkdouble(buf, 5, 1);          /* to slave */
   pvm_send(stid, 1);
}
for (i=0; i<nhost; i++)             /* wait for final results */
{  bufid=pvm_recv(-1, 2);           /* wait for message */
   pvm_bufinfo(bufid, &dum, &dum, &stid);    /* which machine */
   pvm_initsend(PvmDataDefault);         /* tell slave to shut down */
   pvm_send(stid, 0);
}
pvm_exit;
}
```

The **slave** program is nearly identical to the program used for calculating the bifurcation map sequentially. It:

1. Informs the master that it is ready to work.
2. Receives μ values `m_min, m_max, steps, skip`, and `count`.
3. Sets `m = m_min`.
4. Calculates and skips successive `y` values.
5. Calculates and saves another `count y` values.
6. Increases `m` by `(m_max-m_min)/steps`.
7. Goes to step 4 and repeats the whole process until `m = m_max`.
8. Goes to step 1 and starts over until the master says to shut down.

If you compare this to our sequential bug program, you will notice that all we had to do is add in message passing. However, we also had to be careful to have a different name for the output file on each slave. The source code for a slave is given in Lst. 4

Listing 4 The PVM slave program **PVMbugsSlave.c** for creating a bifurcation plot.

```
/* slave program for bifurcation plot of logistics map */
#include <stdio.h>
#include <pvm3.h>
FILE *output;                        /* internal file name */
main() {
double m, sent[5], new, old;
int bufid, dum, ptid, type, x, xskip, xcount;
char name[30], tmp[10], tmp2[10];
ptid = pvm_parent();
pvm_setopt(PvmRoute, PvmRouteDirect);  /* tell master we're ready */
pvm_initsend(PvmDataDefault);
pvm_send(ptid, 2);
gethostname(tmp2, 10);
do                                  /* wait for news from master */
{  bufid=pvm_recv(ptid, -1);              /* any message from master */
   pvm_bufinfo(bufid, &dum, &type, &dum);   /* kind of message? */
   if (type)                               /* more work arrived */
   {  pvm_upkdouble(sent, 5, 1);
      xskip=sent[3];                  /* skip transients */
      xcount=sent[4];                 /* # points to record */
      strcpy(name, tmp2);             /* create unique file name */
      sprintf(tmp, "%f", sent[1]);
      strcat(name, tmp);
      strcat(name, ".dat");
      output=fopen(name, "w");
      for (m=sent[0]; m<=sent[1]; m+=(sent[1]-sent[0])/sent[2])
      { old=0.5;                     /* arbitrary starting value */
        for (x=1; x<=xskip; x++) old=m*old*(1-old);     /* rm transients */
        fprintf(output, "%f\t%f\n", m, old);
        for (x=1; x<=xcount; x++)           /* record xcount points */
         {  new=m*old*(1-old);                 /* avoid some doubles */
            if (new != old) fprintf(output, "%f\t%f\n", m, new);
            old=new;
         } }
      fclose (output);
      pvm_initsend(PvmDataDefault);      /* tell master we're ready */
      pvm_send(ptid, 2);
   }
} while(type);
pvm_exit;                       /* type=0 means we are done */
}
```

Results: A Parallel Plot of Bifurcation Map

In Fig. 1 we show the bifurcation plot created in parallel, with different colors used to indicate work done by different machines. If you count how many areas have a specific color, you will notice that different machines completed a different number of tasks:

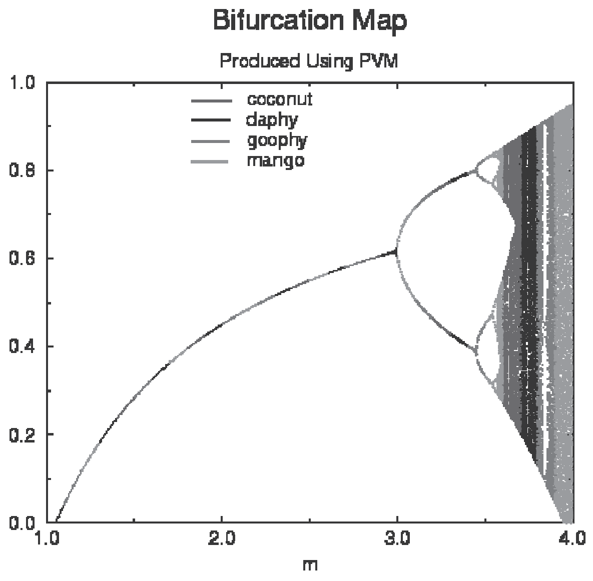

Figure 1 A bifurcation diagram for the logistics map constructed with PVM. The regions of different shadings

Machine	**Architecture**	**Number of Tasks**
daphy	DEC alpha 200	9
goophy	DEC alpha 3000	8
mango	IBM RS6000	7
coconut	IBM RS6000	6

Although this table seems to indicate that coconut has two thirds the speed of daphy, this is not a controlled experiment. Because these are multi-user machines, other users might have used one or more of the machines at the same time, and additional tasks such as webserving, fileserving, mail were also occurring.

MONTE CARLO INTEGRATION, TRIVIAL PARALLELIZATION

In Chap. **??** we performed a Monte Carlo integration of the area of a geometric figure residing within a square, such as Fig. 2. The computation requires you to pick random points within the square and count the total number of points that fall within the figure. The area is just the ratio of the points inside the figure to the total number of points times the known area of the box. Here we made the borders of the figure analytic functions for each quadrant:

1. upper right circle of $r = 1/2$, center (0.5, 0.5), $y = -\sqrt{-x^2 + x} + 0.5$
2. lower right quadratic, $y = 1.2x^2 - 0.3$
3. lower left ellipse, $y = -\sqrt{(1 - x^2/0.25)0.09}$
4. upper left circle with $r = 1/2$, center (0,0), $y = \sqrt{0.25 - x^2}$

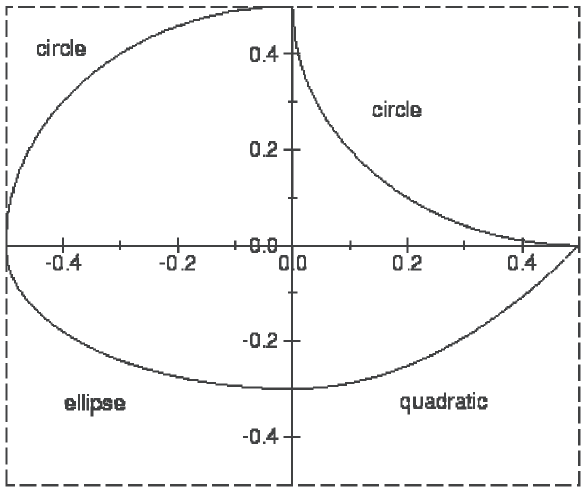

Figure 2 The figure whose area we compute via a parallel Monte-Carlo integration.

We use different machines for each quadrant, and if available, more than one machine for each quadrant. For example, for quadrant 1:

1. Pick two random numbers (x, y) in the range 0 to 1/2.
2. Use the border relation to decide if point is within figure.
3. If inside, increase a variable by one.

The master program:

1. Determines the configuration of the virtual machine.
2. Starts as many different slave processes as there are physical machines.
3. If a slave is done, collects the result and checks if there is another task to be calculated for this quadrant.
4. If further tasks are required of a slave, tells it to continue.
5. If no further task is required, starts a slave for the quadrant in which there remains the largest number of unfinished tasks.
6. Continues with step 3 until all tasks for all quadrant are completed.

Again we split up the work in small pieces in order to be flexible and to provide better load balance in case some machines are much faster than others (when a fast machine finishes its assignment, it helps out a slower one). Let's say we want to sample a total of one million points. We do this by sampling 250,000 points in each quadrant, with 50,000 points assigned to each slave. A program for the master is given in Lst. 5.

Listing 5 The PVM master program **PVMmonteMaster.c** for Monte Carlo integration.

```
/* parallel Monte Carlo integration master */
#include <stdio.h>
#include <pvm3.h>
#define task 100            /* number of tasks per quadrant */
```

```
main() {
struct pvmhostinfo *hostp;
int bufid, check, dum, htid, nhost, narch, ptid, stid, type;
int back[1], done[5], i, j, q, k, min_t, min_q, result;
char name[20], name2[20], tmp[2];
double area;
ptid = pvm_mytid();           /* get your PVM ID number */
pvm_config( &nhost, &narch, &hostp );   /* config of virtual machine */
gethostname(name, 20);
printf("The master process runs on %s \n", name);
printf("I found following hosts in your virtual machine\n");
for (i = 0; i < nhost; i++)
{ printf("\t%s\n", hostp[i].hi_name);  }
printf("\nStarting slaves\n");
for (i=0; i<4; i++) done[i]=0;       /* reset some counters */
i=0;
j=0;
result=0;
do
{  i++;                    /* start slaves 1,2,3,4,1,... */
   if(i==5) i=1;              /* until all machines have a */
   strcpy(name, "monteslave");           /* job running */
   sprintf(tmp, "%i", i);
   strcat(name, tmp);
   check=pvm_spawn(name, 0,PvmTaskHost,hostp[j].hi_name, 1, &stid);
   if (!check)
   {  printf("Couldn't start process on %s\n", hostp[j].hi_name);
      nhost--;      }
   else
   {  printf("started slave for quadrant %i on %s\n", i, hostp[j].hi_name)
       ;
      done[i]++;     }
   j++;
} while ((j<nhost) && (j<4*task));
for(i=j; i<4*task; i++)          /* wait for slaves to finish to */
{  bufid=pvm_recv(-1, -1);       /* any machine any message*/
   pvm_bufinfo(bufid, &dum, &type, &stid);  /* from which quadrant? */
   pvm_upkint(back, 1, 1);           /* the result of the task */
   result+=back[0];
   if (done[type]<task)          /* there are still open tasks */
   {   pvm_initsend(PvmDataDefault);     /* tell slave to continue */
       pvm_send(stid, 1);
       done[type]++;      }
   else                  /* no open tasks, start new slave */
   {  printf("quadrant %i is done\n", type);
      htid=pvm_tidtohost(stid);          /* find host of this slave */
      for (k=0; k<nhost; k++)
      { if (htid==hostp[k].hi_tid) strcpy(name2, hostp[k].hi_name); }
      pvm_initsend(PvmDataDefault);      /* tell slave to shut down */
      pvm_send(stid, 0);
      min_t=done[1];                 /* find quadrant with most */
      min_q=1;                  /* open tasks - this way */
      for (k=2; k<5; k++)            /* the fastest machine */
      { if (done[k]<min_t)          /* new quadrant or + slow */
          {  min_t=done[k];           /* after it's done */
             min_q=k;  }   }
      strcpy(name, "monteslave");        /* which slave to start */
      sprintf(tmp, "%i", min_q);
      strcat(name, tmp);
      pvm_spawn(name, 0,PvmTaskHost, name2, 1, &stid);
      printf("started slave for quad %i on %s\n", min_q, name2);
      done[min_q]++;
   }  }
for(i=0; i<nhost; i++)           /* wait for last tasks to end */
 { bufid=pvm_recv(-1, -1);    /* any machine/ message*/
   pvm_bufinfo(bufid, &dum, &dum, &stid);    /* where from */
   pvm_upkint(back, 1, 1);
   result+=back[0];
   pvm_initsend(PvmDataDefault);         /* tell slave to shut down */
   pvm_send(stid, 0);
}
area=result;                     /* calculate final result */
printf("the area is %f\n", area/(task*4*50000));
pvm_exit;
}
```

Even though we need a different slave programs for each quadrant, they are simple and differ by only a few of lines. Each slave:

1. Generates two random numbers in the appropriate quadrant.
2. Uses the border relation for this quadrant to check if the point is inside the figure.
3. If point is inside the figure, increases a counter by one.
4. Goes back to step 1 until a set number of points has been checked.
5. Sends the value of the counter back to the master.
6. Asks the master if there are more points to check for this quadrant.
7. If more points are needed, reset all variables and start over.

The only things you have to change for each slave are the range of the random numbers in order for the points to fall in the right quadrant and the relation describing the border of the figure for this quadrant. A C program for the quadrant 1 slave is given in Lst. 6.

Listing 6 The PVM slave program **PVMmonteSlave1.c** for quadrant-1 Monte Carlo integration.

```
/* parallel Monte Carlo integration, Quadrant 1 Slave*/
#include <stdio.h>
#include <math.h>
#include <pvm3.h>
#define steps 50000
#define xmin 0.0
#define xmax 0.5
#define ymin 0.0
#define ymax 0.5
double f(double x)          /* border function   */
{ return(-sqrt(-x*x+x)+0.5); }
main() {
int i, ptid, bufid, dum, type, send[1];
double x, y;
srand48(pvm_mytid());     /* seed  random generator */
ptid = pvm_parent();
do
{  send[0]=0;
   for (i=1; i<=steps; i++)
```

```
    {  x= drand48()*(xmax+xmin);         /* random points in the */
       y= drand48()*(ymax+ymin);         /* quadrant */
       if (f(x)<=y) send[0]++;           /* point is inside the figure */
    }
    pvm_initsend(PvmDataDefault);        /* send result back to master */
    pvm_pkint(send, 1, 1);
    pvm_send(ptid, 1);
    bufid=pvm_recv(ptid, -1);            /* any message from master */
    pvm_bufinfo(bufid, &dum, &type, &dum);   /* more work ? */
}while(type);

pvm_exit;                     /* type=0 means we are done */
}
```

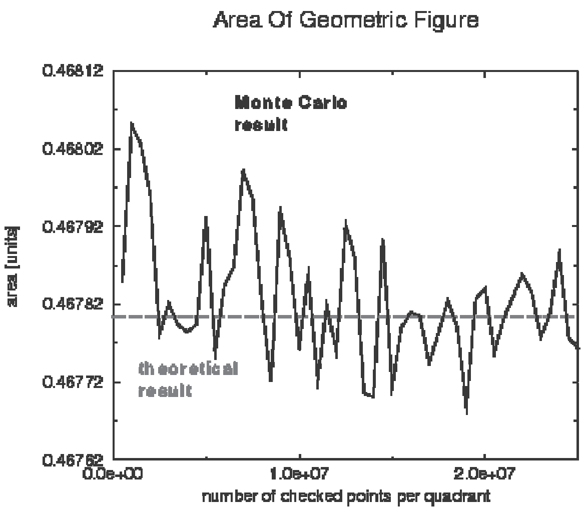

Figure 3 The area of a geometric figure calculated in parallel via Monte Carlo integration in comparison with the exact answer.

The results of the computation are shown in Fig. 3. We used Maple to find the analytic result for the area, `area=0.4678097245`, which means that the figure fills approximately 47% of the unit square. To compare this result to the Monte Carlo one, we ran the parallel program several times increasing the number of points we checked from only five hundred thousand per quadrant to twenty-five million points per quadrant. As it is typical for Monte Carlo methods, the numerical result oscillates around the true value, but with an amplitude that decreases as the number of points increases.

INSTALLING PVM HINTS

The best way to install PVM is to get a nice system administrator to do it. This saves you work and makes it possible for other people on your system to use PVM. However, if you are the nice system administrator, your system administrator is not nice, or you can't find him because he is on his well-earned vacation in the Caribbeans, then here are the things you have to do.

1. Download the software package for `pvm` and `xpvm` from

   ```
   http://www.netlib.org/pvm3/
   http://www.netlib.org/pvm3/xpvm/
   http://www.csm.ornl.gov/pvm/
   ```

2. Find a location on your computer for PVM. If you are the system administrator installing PVM so that all users can use it, then `/usr/local/pvm3` is a natural choice. Otherwise create a directory `pvm3` in your home directory and put the downloaded software in there.
3. Unpack and build PVM. PVM is distributed in various packed and compressed formats (indicated by the file name extension). For example, if your file name is `pvm3.3.10.tar.gz`. then you have to do the following:
   a. `gunzip pvm3.3.10.tar.gz`
   b. `tar -xvf pvm3.3.10.tar`
   c. Set the environment variable `PVM_ROOT` to the directory where you put the PVM software by adding to your `.cshrc` file:

      ```
      setenv PVM_ROOT $HOME/pvm3            One way
      setenv PVM_ROOT /usr/local/pvm3       Another way
      ```

   d. If you use `csh`:

      ```
      PVM_ROOT=$HOME/pvm3
      PVM_DPATH=$PVN_ROOT/lib/pvmd
      export PVM_ROOT PVM_DPATH
      ```

   e. or, if you use `sh` or `ksh`, include in your `.profile` file:

      ```
      PVM_ROOT=/usr/local/pvm3
      PVM_DPATH=$PVN_ROOT/lib/pvmd
      export PVM_ROOT PVM_DPATH
      ```

   f. Enter `make` in the `PVM_ROOT` directory. This will build the libraries and binaries required to run PVM. If everything compiles correctly, you are ready to go.
   g. Move the files. If you have root access, then you should copy the PVM libraries `libfpvm3.a, libgpvm3.a, libpvm3.a` from `pvm3/lib/$PVN_ARCH` into `/usr/lib`, where the compiler can find them. For the same reason, put the include files `fpvm3.h, pvm3.h, pvmsdpro.h, pvmtev.h` into `/usr/include`.

PVM COMMAND REFERENCE

Sending & Receiving Messages

Routine Name	Operation
pvm_barrier	Blocks calling process until all processes in a group calls
pvm_bcast	Broadcasts data in active message buffer
pvm_bufinfo	Returns information about message buffer
pvm_freebuf	Disposes of a message buffer
pvm_getrbuf	Returns message buffer identifier for active receive buffer
pvm_getsbuf	Returns message buffer identifier for active send buffer
pvm_initsend	Clears default send buffer & specify message encoding
pvm_mcast	Multicasts data in active message buffer
pvm_mkbuf	Creates a new message buffer
pvm_nrecv	Non-blocking receive
pvm_pack	Packs active message buffer with proper data arrays
pvm_precv	Receives message directly into buffer
pvm_probe	Checks if message has arrived
pvm_psend	Packs and send data in one call
pvm_recv	Receives message
pvm_send	Immediately sends data in active message buffer
pvm_setrbuf	Switches active receive buffer and saves previous buffer
pvm_setsbuf	Switche active send buffer
pvm_trecv	Receive with timeout
pvm_unpack	Unpack active message buffer into proper data arrays

Controlling Virtual Machine

Routine Name	Operation
pvm_addhosts	Adds hosts to VM
pvm_catchout	Catches output from child tasks
pvm_config	Returns information on present VM configuration
pvm_delhosts	Deletes hosts from VM
pvm_exit	Tells `pvmd` that leaving PVM
pvm_getopt	Returns value of `libpvm` options
pvm_gettid	Returns `tid` of process
pvm_halt	Shuts down entire PVM system
pvm_hostsync	Get time-of-day clock from PVM host
pvm_kill	Terminates specified PVM process
pvm_mstat	Returns status of a host in VM
pvm_mytid	Returns the `tid` of calling process
pvm_notify	Requests notification of PVM event
pvm_parent	Returns `tid` of parent process
pvm_perror	Prints message describing last error
pvm_pstat	Returns status of specified process
pvm_setopt	Sets `libpvm` options
pvm_spawn	Starts new PVM process
pvm_start_pvmd	Starts new PVM daemon
pvm_tasks	Returns information about tasks on VM
pvm_tidtohost	Returns host of specified process

Advanced Features & Group Operations

Routine Name	Operation
pvm_archcode	Returns data code for an architecture name
pvm_delete	Deletes data from `pvmd` database
pvm_gather	Gathers messages from group
pvm_getfds	Gets file descriptors in use by PVM
pvm_getinst	Returns instance number of a process
pvm_getmwid	Gets wait ID of a message
pvm_gettmask	Gets trace mask of task or its children
pvm_gsize	Returns number of members in named group
pvm_insert	Stores data in `pvmd` database
pvm_joingroup	Enrolls calling process in named group
pvm_lookup	Retrieves data from `pvmd` database
pvm_lvgroup	Unenrolls calling process from `named` group
pvm_recvf	Redefines comparison function that accepts messages
pvm_reduce	Performs reduction operation over group members
pvm_reg_hoster	Registers task as PVM slave starter
pvm_reg_rm	Registers task as PVM resource manager
pvm_reg_tasker	Registers task as PVM task starter
pvm_scatter	Sends data to each member of group
pvm_sendsig	Sends signal to another PVM process
pvm_setmwid	Sets wait ID of message
pvm_settmask	Sets trace mask of task or its children

After starting the PVM console by typing `pvm` at the unix prompt, you can use the following commands:

Available Commands for the PVM Console

add hostname	Add computer specified by hostname to virtual machine
alias	Defines or lists command aliases (same as unix command)
conf	Returns detils of present configuration of virtual machine
delete hostname	Removes computer hostname from virtual machine (parallel tasks still running on machine will be stopped)
echo arg	Returns arg (expands alias).
halt	Kills all PVM processes, & shuts down PVM (a proper end)
help arg	Returns description of command; list of command if no arg
id	Prints console task ID
jobs	Lists jobs running on virtual machine
kill tid	Kills process tid; `kill -c tid` also kills children processes
mstat hostname	Shows status of the computer hostname
ps	Lists processes running on the virtual machine; options:
-a	lists processes all computers, default = local
-hhost	lists processes on `host`
-nhost	lists processes with task ID `host`
-l	use long output format
-x	list all processes including console nulls
pstat tid	Shows status of task specified by `tid`
reset	Kills all PVM processes except console and PVM daemons
setenv	Displays or sets environment variables (same as Unix)
sig	Don't use if you are not really sure what this does
spawn arg	Starts program `arg`; takes following options:
-count	count = number of tasks to start
-host	start task on `host`
-ARCH	start task on hosts of type `ARCH`
-?	enable debugging
->	redirect task output to console
-> file	redirect task output to `file`
->> file	append task output to `file`
-@	trace this job, display trace information on console
-@file	trace this job, write trace information to file
trace	Sets or displays trace event mask
unalias arg	Removes defined alias `arg`
version	Returns version of PVM being used

A. GEIST, A, A. BEGUELIN, JACK DONGARRA, WEICHENG JIANG, ROBERT MANCHEK, AND VAIDY SUNDERAM (1994), *PVM: Parallel Virtual Machine A User's Guide and Tutorial for Networked Parallel Computing*, Oak Ridge National Laboratory, Oak Ridge, TN.