PVM: Parallel Virtual Machine

David W. Walker
Mathematical Sciences Section
Oak Ridge National Laboratory
Oak Ridge, TN 37831-6367

walker@msr.epm.ornl.gov

WEB: http://www.epm.ornl.gov/~walker/

Presented at
CERN School of Computing, Arles, France
August 24, 1995

Parallel Virtual Machine

- PVM enables a collection of different computing systems to be viewed as a single parallel virtual machine
- MIMD process model
- Distributed memory model
- PVM transparently handles message passing, data conversion, and task scheduling
- Tasks (or processes) interact by message passing
- Message passing
  - is done by calls to a message passing library
  - uses system controlled buffers
  - is typed
- Point-to-point and collective message passing operations are provided
- PVM has support for process groups

PVM Components

The main components of PVM are:

- **The PVM daemon**. This runs on every host. It controls the PVM resources on each host and mediates interactions with other hosts.
- **The PVM console**. This is a user interface that allows the user to interactively configure the parallel virtual machine.
- **The PVM group server**. This manages process groups within PVM
- **The PVM library**. A set of routines for managing tasks and process groups, and for performing message passing.

The PVM Console

The PVM console allows a user to interactively start, query, and modify the virtual machine.

- Creating and modifying the VM
  - add hostname
  - delete hostname
  - Or put names of hosts into a hostfile and start pvm console:

    ```
    pvm hostfile
    ```
- Querying the VM
  - `conf` to get details of current VM
  - `jobs` to get list of jobs currently running
  - `mstat` to get status of a host

The PVM Console (continued)

- Leaving and exiting
  - **`quit`** exits console but leaves PVM jobs running
  - **`halt`** kills all PVM process and shuts down PVM
- Job control
  - **`kill`** is used to kill a PVM process
  - **`reset`** kills all PVM jobs except console(s)
  - **`spawn`** starts a PVM application

PVM Integration Example

```
program master
include 'fpvm3.h'
parameter (npmax=100, pi=3.141592654)
integer tids(npmax), bufid, atid, atag, alen
print *, 'Give number of processes < ',npmax
read *, nprocs
npts = nprocs + 1
deltax = pi/npts
call pvmfspawn ('worker', PvmTaskDefault, '', nprocs, tids, numt)
sum = sin(0.5*deltax)
do i=1,nprocs
   call pvmfinitsend (PvmDataDefault, bufid)
   call pvmfpack (integer4, i, 1, 1, info)
   call pvmfpack (real4, deltax, 1, 1, info)
   call pvmfsend (tids(i), 10, info)
end do
do i=1,nprocs
   call pvmfprecv (-1, 10, psum, 1, real4,
                   atid, atag, alen, info)
   sum = sum + psum
end do
print *,'The integral is ',sum*deltax
call pvmfexit (info)
stop
end
```

PVM Integration Example

The worker program

```
program worker
include 'fpvm3.h'
integer bufid, parent
call pvmfparent (parent)
call pvmfrecv (parent, 10, bufid)
call pvmfunpack (integer4, i, 1, 1, info)
call pvmfunpack (real4, deltax, 1, 1, info)
x = (i-0.5)*deltax
psum = sin(x)
call pvmfpsend (parent, 10, psum, 1, real4, info)
call pvmfexit (info)
stop
end
```

PVM Integration Example 2

- In the previous program each process computes just one point. This is not likely to be very efficient or flexible since it takes longer to spawn the processes and communicate between the master and the workers than it would to just do the work on one process.
- In the next example each process is given a set of points to handle.
- This increases the *granularity*, or work per processor, of the problem
- Given M things to distribute over N_p processes we evaluate $n = \lceil M/N_p \rceil$. Each process handles n things until we run out
- Example, $M = 1000$ and $N_p = 18$, then $n = 56$. So 17 processes get 56 things and 1 process gets 48 $(1000 = 56 \times 17 + 48)$
- Example, $M = 100$ and $N_p = 18$, then $n = 6$. So 16 processes get 6 things, 1 process gets 4, and 1 process gets 0 $(100 = 6 \times 16 + 4)$

PVM Integration Example 2

```
program master
include 'fpvm3.h'
parameter (npmax=100, pi=3.141592654)
integer tids(npmax), bufid, atid, atag, alen
print *, 'Give number of worker processes < ',npmax
read *, nprocs
print *,'Give number of points'
read *, npts
deltax = pi/npts
if (nprocs .gt. 0) call pvmfspawn ('int-worker2',
               PvmTaskDefault, '', nprocs, tids, numt)
nlocal = (npts-1)/(nprocs+1) + 1
sum = 0.0
nend = min(nlocal,npts)
do i=1,nend
   x = (i-0.5)*deltax
   sum = sum + sin(x)
end do
```

Continued on next slide…

PVM Integration Example 2

...continued from previous slide

```
nbeg = nlocal + 1
do i=1,nprocs
   nend = min(nbeg+nlocal, npts)
   call pvmfinitsend (PvmDataDefault, bufid)
   call pvmfpack (integer4, nbeg, 1, 1, info)
   call pvmfpack (integer4, nend, 1, 1, info)
   call pvmfpack (real4, deltax, 1, 1, info)
   call pvmfsend (tids(i), 10, info)
   nbeg = nend + 1
end do
do i=1,nprocs
   call pvmfprecv (-1, 10, psum, 1, real4,
                       atid, atag, alen, info)
   sum = sum + psum
end do
print *,'The integral is ',sum*deltax
call pvmfexit (info)
stop
end
```

PVM Integration Example 2

The worker program

```
program worker
include 'fpvm3.h'
integer parent, bufid
call pvmfparent (parent)
call pvmfrecv (parent, 10, bufid)
call pvmfunpack (integer4, nbeg, 1, 1, info)
call pvmfunpack (integer4, nend, 1, 1, info)
call pvmfunpack (real4, deltax, 1, 1, info)
psum = 0.0
do i=nbeg,nend
   x = (i-0.5)*deltax
   psum = psum + sin(x)
end do
call pvmfpsend (parent, 10, psum, 1, real4, info)
call pvmfexit (info)
stop
end
```

Using PVM Process Groups

- Having separate programs for the master and worker processes can be inconvenient
- If master and worker are doing similar work we can combine them into a Single-Program, Multiple Data (SPMD) program
- Processes can figure out what part of the problem to solve based on their group instance number.

PVM Integration Example 3

Program divides into three parts. The first part spawns process and communicates initial data.

```
program groups
include 'fpvm3.h'
parameter (npmax=100, pi=3.141592654)
integer tids(npmax), bufid, atag, atid, alen, parent
call pvmfjoingroup ('agroup', inum)
if (inum .eq. 0) then
   print *, 'Give number of processes < ',npmax
   read *, nprocs
   print *, 'Give number of points'
   read *, npts
   call pvmfspawn ('int-spmd', PvmTaskDefault, '',
                  nprocs, tids, numt)
   call pvmfinitsend (PvmDataDefault, bufid)
   call pvmfpack (integer4, npts, 1, 1, info)
   call pvmfpack (integer4, nprocs, 1, 1, info)
   call pvmfmcast (nprocs, tids, 10, info)
else
   call pvmfparent (parent)
   call pvmfrecv (parent, 10, bufid)
   call pvmfunpack (integer4, npts, 1, 1, info)
   call pvmfunpack (integer4, nprocs, 1, 1, info)
end if
call pvmfbarrier ('agroup', nprocs+1, info)
```

PVM Integration Example 3

In the second part each process evaluates its contribution to the integral

```
nlocal = (npts-1)/(nprocs+1) + 1
nbeg   = inum*nlocal + 1
nend   = min (nbeg+nlocal-1, npts)
deltax = pi/npts
psum   = 0.0
do i=nbeg,nend
   x = (i-0.5)*deltax
   psum = psum + sin(x)
end do
```

To be continued…

PVM Integration Example 3

In the third part each process sends its contribution to the parent, which sums the contributions and outputs the result

```
if (inum .eq. 0) then
   sum = psum
   do i=1,nprocs
      call pvmfprecv (-1, 20, psum, 1, real4,
                      atid, atag, alen, info)
      sum = sum + psum
   end do
   print *,'The integral is ',sum*deltax
else
   call pvmfpsend (parent, 20, psum, 1, real4, info)
end if
call pvmfexit (info)
stop
end
```

PVM Integration Example 3

An alternative way of summing the contributions is the use PVM's reduce function

```
call pvmfreduce (PvmSum, psum, 1, real4, 20,
                 'agroup', 0, info)
if (inum .eq. 0) then
   sum = psum
   print *,'The integral is ',sum*deltax
end if
call pvmfbarrier ('agroup', nprocs+1, info)
call pvmfexit (info)
stop
end
```

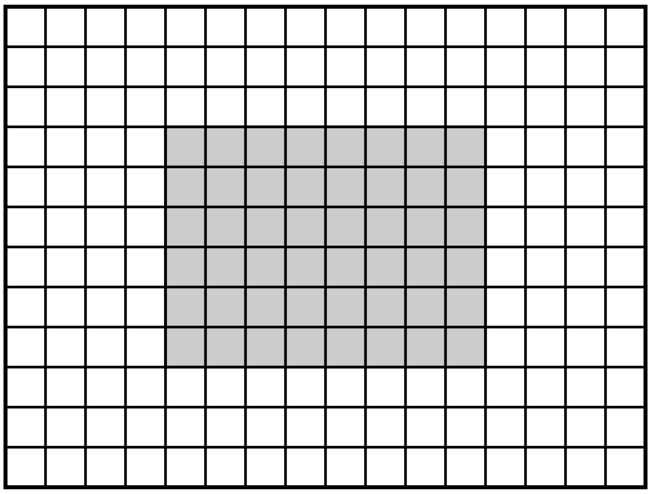
- Note use of barrier to prevent any process from leaving the group before the reduction is completed.
- Remember to declare `PvmSum` as external.

Laplace Solver

- Solution of 2D Laplace equation

$$\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} = 0$$

on rectangular domain with $\phi = 0$ on boundary, and $\phi = 1$ on an interior rectangular region

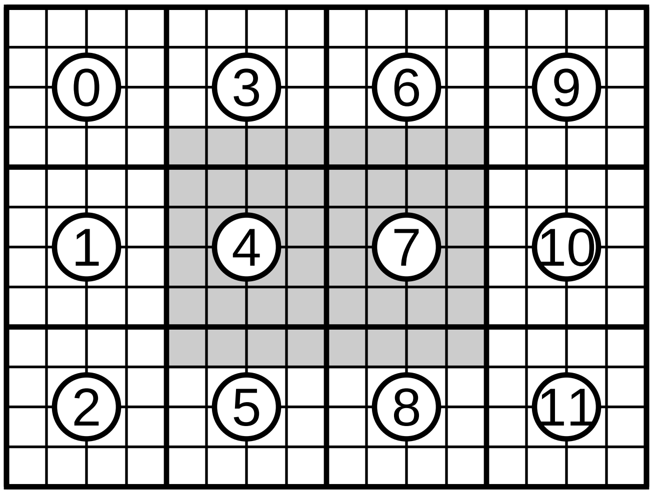


- Solution by Jacobi iteration;

$$\phi^n_{i,j} = \frac{1}{4}\left(\phi^{n-1}_{i-1,j} + \phi^{n-1}_{i+1,j} + \phi^{n-1}_{i,j-1} + \phi^{n-1}_{i,j+1}\right)$$

Data Distribution

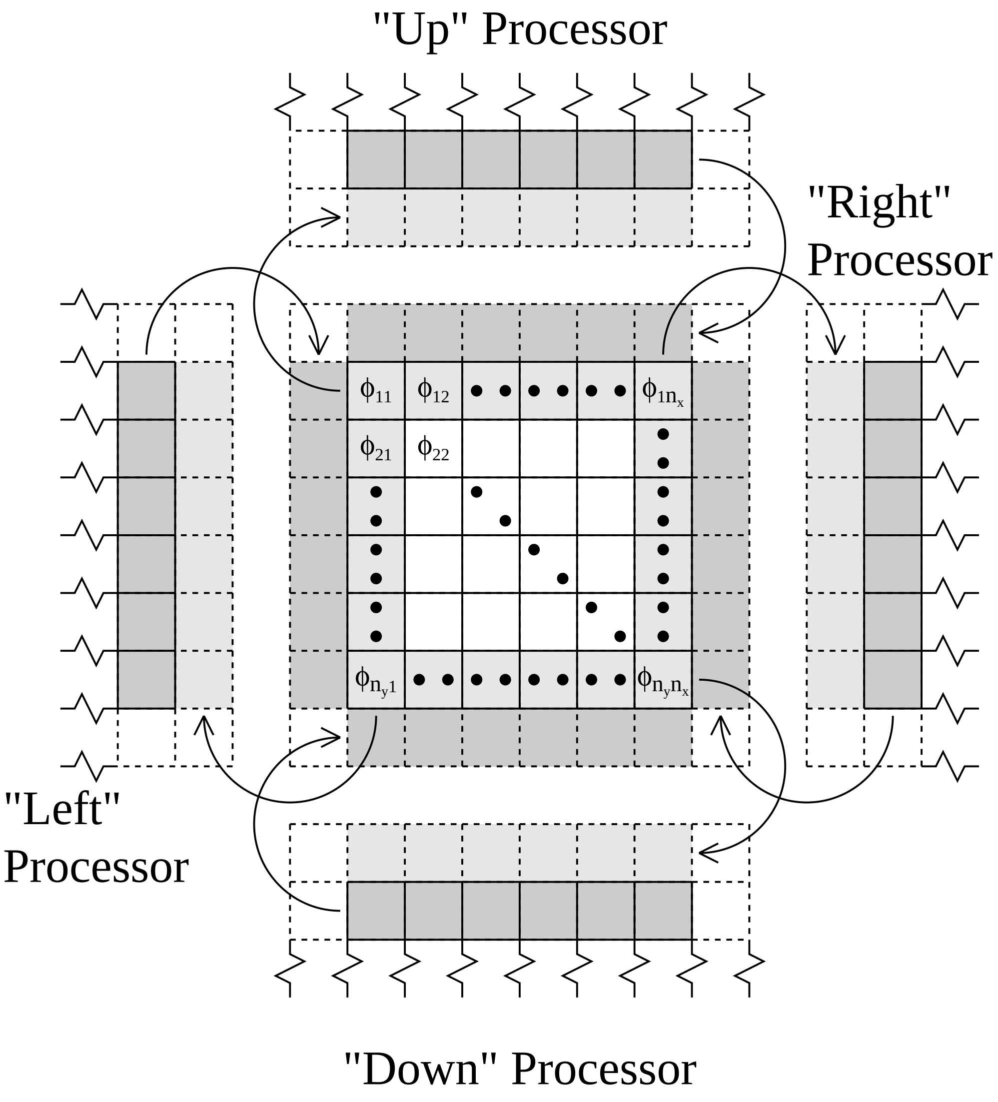
- Use spatial block data distribution over 2D process mesh.
- Make use of PVM groups



Communication

- Each process needs to know boundary values of its neighbors to updates its own boundary values
- Each process must communicate with its left, right, up and down neighbors at the start of each iteration
- Data received are stored in edges of ϕ array

Communication of Ghost Values



Skeleton Laplace Code

1. Initialize group and send parameters to processes. Do barrier
2. Initialize grid `phi`. Use logical array, `mask`, to identify boundaries ($\phi = 0$) and fixed interior region ($\phi = 1$).
3. Identify neighboring processes
4. For each iteration
   (a) Copy `phi` array to `oldphi` array
   (b) Exchange boundary data
   (c) Update `phi` array
5. Output final `phi` array
6. Do barrier, exit PVM and stop

Identifying Neighbors

PVM has no explicit support for process topologies so we have to to translation between group instance number (`inum`) and location in process grid, (`ipy, ipx`).

```
ipx = inum/npy
ipy = mod (inum, npy)
numx = mod (ipx-1+npx, npx)
numy = ipy
num = numy + numx*npy
call pvmfgettid ('agroup', num, tidl)
numx = mod (ipx+1, npx)
num = numy + numx*npy
call pvmfgettid ('agroup', num, tidr)
numx = ipx
numy = mod (ipy-1+npy, npy)
num = numy + numx*npy
call pvmfgettid ('agroup', num, tidu)
numy = mod (ipy+1, npy)
num = numy + numx*npy
call pvmfgettid ('agroup', num, tidd)
```

The Main Loop

The main iterative loop is as follows

```
 do iter=1,maxiter
    do j=0,nptsx+1
       do i=0,nptsy+1
          oldphi(i,j) = phi(i,j)
       end do
    end do
    call commlr (oldphi, nptsx, nptsy, tidl, tidr)
    call commud (oldphi, nptsx, nptsy, tidu, tidd)
    do j=1,nptsx
       do i=1,nptsy
          if (mask(i,j)) phi(i,j) =
#                0.25*(oldphi(i-1,j) + oldphi(i+1,j) +
#                      oldphi(i,j-1) + oldphi(i,j+1))
       end do
    end do
 end do
```

Communicating to Left and Right

1. Send `oldphi(1:nptsy,1)` to left. Receive from right into `oldphi(1:nptys,nptsx+1)`
2. Send `oldphi(1:nptsy,nptsx)` to right. Receive from left into `oldphi(1:nptsy,0)`
3. Data sent are contiguous in memory

```
call pvmfinitsend (PvmDataDefault, bufid)
call pvmfpack (real4, oldphi(1,1), nptsy, 1, info)
call pvmfsend (tidl, 30, info)
call pvmfrecv (tidr, 30, bufid)
call pvmfunpack (real4, oldphi(1,nptsx+1), nptsy, 1, info)

call pvmfinitsend (PvmDataDefault, bufid)
call pvmfpack (real4, oldphi(1,nptsx), nptsy, 1, info)
call pvmfsend (tidr, 31, info)
call pvmfrecv (tidl, 31, bufid)
call pvmfunpack (real4, oldphi(1,0), nptsy, 1, info)
```

Communicating to Down and Up

1. Send `oldphi(1,1:nptsx)` upwards. Receive from downwards into `oldphi(nptys+1,1:nptsx)`
2. Send `oldphi(nptsy,1:nptsx)` downwards. Receive from upwards into `oldphi(0,1:nptsx)`
3. Data sent are at fixed stride in memory. The stride is given by the leading dimension of `oldphi`, which is `nymax+2`

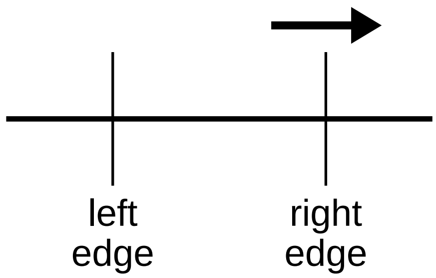
```
call pvmfinitsend (PvmDataDefault, bufid)
call pvmfpack (real4, oldphi(1,1), nptsx, nymax+2, info)
call pvmfsend (tidu, 40, info)
call pvmfrecv (tidd, 40, bufid)
call pvmfunpack (real4, oldphi(nptsy+1,1), nptsx, nymax+2, info)

call pvmfinitsend (PvmDataDefault, bufid)
call pvmfpack (real4, oldphi(nptsy,1), nptsx, nymax+2, info)
call pvmfsend (tidd, 41, info)
call pvmfrecv (tidu, 41, bufid)
call pvmfunpack (real4, oldphi(0,1), nptsx, nymax+2, info)
```

Particle Migration Example

- **Problem:**

  Particles move on a domain that is block distributed. Particle data must be moved between neighboring process subdomains when they cross subdomain boundaries.

- Since each process subdomain has 6 surfaces (for 3D), the migration can be done in 6 communications. We shall just consider the migration across the righthand boundary.



- Assume particles stored in a structure

```
struct Pstruct {double x[3]; double v[3]; int k;}
     particle[100];
```

Migrating The Particles

```
count = 0;
bufid = pvm_initsend (PvmDataDefault);
count = 0;
for(i=0;i<nparticles;i++)
  if (particle[i].x[0] > right_edge) {
     info = pvm_pkdouble (&(particle[i].x[0]), 3, 1);
     info = pvm_pkdouble (&(particle[i].v[0]), 3, 1);
     info = pvm_pkint    (&(particle[i].k),    1, 1);
     count += 1;
  }
info  = pvm_send (tidr, tag);
bufid = pvm_recv (tidl, tag);
info  = pvm_bufinfo (bufid, &bytes, &atag, &asrc);
nrecv = bytes/sizeof(Pstruct);
for(i=0;i<nrecv;i++){
  unpack into local data structures
}
```

Summary of PVM Examples

- We have looked at examples making use of
  - Master/worker process model
  - SPMD process model based on process groups
- Processes can join and leave a group asynchronously – can cause problems
- When using process groups it is important to use barriers to define the section of code during which all processes must be in the group.
- Lack of support for process topologies inconvenient
- PVM pack routines useful for composing messages of mixed datatype and structures.