Introduction to PVM

A 1 Day Course

Slides

Martin Preston

Manchester and North HPC Training & Education Centre

Introduction to PVM

Course Introduction

Course Outline

- We will cover:
  - A short reminder of the basic concepts of message passing as a parallel programming paradigm.
  - The concept of executing your program on a distributed virtual machine (VM).
  - How PVM attempts to provide the illusion of a single computing resource.
  - How an application programmer creates programs on the VM.
  - How a user interacts with these programs on the VM.
  - How messages are passed between programs executing on the machine.

Cont.

- How collections of programs may be managed easily by the application programmer.
- How PVM provides some higher level parallel programming features.
- An example of how a sample application might be constructed using PVM, a ray tracer.
- A summary of the features of PVM, and what else you might want to use.

Introduction to Message Passing

Parallel Programming

- Why Parallel Programming?
- Certain classes of problems are either:
  - Too large for available serial architectures.
  - Take too long to execute on serial machines.
  - Don't handle large loads well, e.g., a database server which operates well with 10 users, but when 100 people use it the performance suffers.

The promise of Parallel programs

- So, as a programmer, we are forced to turn to parallel machines as a possible solution to our problems.
- Parallel processing claims to be
  - Cheaper, in terms of Price/Performance.
  - Faster than equivalently expensive uniprocessor machines.
  - Scalable : the performance of a particular program may be improved by execution on a large machine.
  - Reliable? In theory if processors fail we can simply use others.
  - Handle bigger problems.
- But to use parallel machine we must employ *parallel programming*.

What is Parallel Programming?

- Concurrent operation of elements within a system
- Low-level hardware has been parallel for many years. e.g., overlapped I/O, multi-tasking. Traditionally this parallelism is hidden from the user.
- Machines which allow the user to take advantage of parallelism are normally referred to as *parallel machines*, to distinguish from conventional architectures.

How do we take advantage of parallel machines?

- We must identify potential parallelism in *applications*.
- If we can find sections of computer code which can be executed at the same time as other sections, *without changing the results generated by that program*, then we have found potential parallelism.
- Before we can discuss how we exploit this parallelism we review the machines.

Parallel Machines

- A wide variety of parallel architectures exist.
- Fortunately there exists a useful taxonomy which we can employ to categorise them (Flynns taxonomy).
- This taxonomy categorises machines dependent on how each handles instructions (multiple programs), and data.

Flynns Taxonomy

- The taxonomy breaks into :
  - SISD - Single Instruction/ Single Data. This category corresponds to conventional serial architectures.
  - MISD - Multiple Instruction/ Single Data. Here the machine lets several processors execute different instructions on *one* data stream in a pipeline fashion.
  - SIMD - Single Instruction/ Multiple Data. Here a single program is executed on several pieces of data simultaneously.
  - MIMD - Multiple Executions being executed on separate data simultaneously.
- This abstract taxonomy provides a general indication of the capabilities and programming style required for a particular machine, but more information is usually needed.

Other Considerations

- Grain Size : The size of processes which we execute on processors affect how we can use the machine.
- Interconnection : How processors communicate with one another.
- Coupling : How processors communicate with memory.
- Programming Mechanism. Whilst a wide variety of abstract parallel programming styles exist (one of which is the principal topic of this course), not all are useful on all machines.

Parallel Programming

- As computer science has yet to develop compilers which automatically use parallel machines well, we must program in special ways.
- Various paradigms exist, but we are interested in the *message passing* technique.

Message Passing

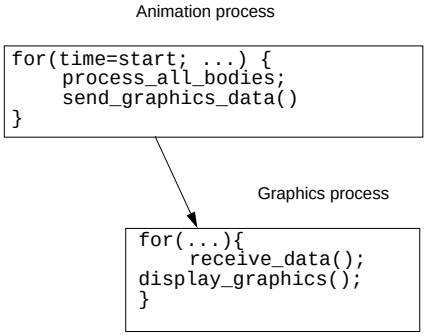
- For message passing to be a viable means of exploiting parallelism we conventionally employ it on MIMD machines.
- The *application* is split into a number of *programs*. Each program operates ‘independently’, usually on different processors.
- The logic of the application is maintained by coordinating the component programs through the exchange of messages.
- The maintenance of this underlying logic, which controls how the application works is the responsibility of the *programmer*, not the machine.
- This makes this form of programming hard!

An Example

- Computer animation:

```
for(time = start; time++; time<end)
{
     process_all_bodies;
     display_bodies;
}
```

- Both steps are time consuming, they could be organised as two seperate programs:



Facilities of MP Libraries

- Application programmers don't want to deal with the messy aspects of getting processors to communicate.
- So they use message passing libraries which provide:
  - the ability to create processes on remote machines
  - the ability to monitor the state of these remote processes
  - routines which enable messages to be sent reliably from program to program, without the programmer needing to know *how* this is achieved.

MP Implementations

- There are a large range of message passing libraries in use today, on a wide range of architectures.
- A programmer must choose between them, though they all perform similar functions, and so porting is not very difficult.
- We will concentrate on one popular library, PVM.

Introduction to the Parallel Virtual Machine (PVM)

The Goals of PVM

- PVM has been developed over many years by a collection of academics at various US institutions who were interested in research on distributed computing.
- To help their work they developed an MP library which was primarily intended for use on a cluster of workstations, which the library should make look like an MIMD parallel computer.
- This means many people use PVM on networks of normal workstations, but can take code from these environments and execute them on expensive PM's.

History

- Version 1 of PVM was written during summer of 1989 at Oak Ridge National Labs (ORNL). Used as proof-of-concept and never generally released.
- Version 2 written March 1991 at UTK. Intended for use in HeNCE. Was a stable and robust version, and was released publicly.
- Version 3 written September 1992- Feb93
  - 3.1 (April 93) Improved task-to-task routing
  - 3.2 (Aug 93) Improved support for PM's
  - 3.3 (Jun 94) More shared & distributed memory support.
  - 3.3.3 (Aug 94) Current version

Philosophy of PVM

- PVM is intended to present the user with a view of a *VIRTUAL MACHINE*.
- This is composed of many physical machines, the complexities of dealing with which are hidden by use of the PVM libraries.
- By writing applications to execute on an abstract virtual machine the user should be able to take code and run it in a wide variety of environments.
- To encourage this PVM has been ported to many machines!

PVM Environments

- Currently supported machines:

80386 with BSD	Alliant FX/8
DEC Alpha OSF/1	BBN TC2000
DEC MicroVax	Convex
HP 9000/300	Cray YMP
HP 9000/700	Cray C90
IBM RS/6000	Cray 6400
IBM/RT	IBM 3090
NeXT	Intel Paragon
SIlicon Graphics Iris	Intel iPSC/2
SUN 3	Kendall Square Research 1
SUN 4, Sparc	Sequent Symmetry
	Stardent Titan
	Thinking Machines CM-2 & CM-5

- This list is continually being extended as PVM development team is quite active.

Major Features of PVM

- Easy to install
- Easy to configure
- Multiple users can each use PVM simultaneously
- Easy to write programs (as easy as any MP!)
- C & Fortran supported
- Package is small
- Multiple applications from one user can execute.

Heterogeneity

- PVM supports heterogeneity at 3 levels:
- Application: Subtasks can exploit the architecture best suited to their application.
- Machine: Computers with different formats and OS's can collaborate in a single VM.
- Network: Different types of networks can be used, FDDI, Ethernet, Token Ring....

The Drawbacks

- We said that the user interacts with a virtual machine. Unfortunately the desire of users to exploit the facilities of particular machines mean this isn't exactly true.
- The VM is virtual only in-so-much as communication is abstracted.
- The programmer will still need to physically develop code for particular machines, and deal with multiple compilers, different windowing systems, editors etc.
- This can be either viewed as a good or bad thing!
- We can now discuss the basics of how PVM works, and what it offers.

Fundamentals of PVM

The Virtual Machine

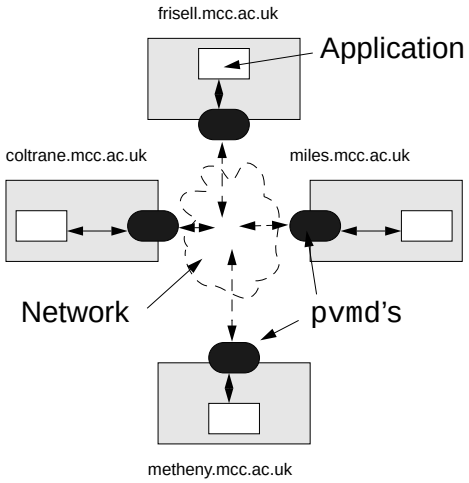
- The user describes a collection of machines which are contactable through networks, which together constitute the virtual machine.
- The user needs to supply:
  - Names (usually DNS registered) for the component machines.
  - Information about how PVM can spawn tasks on it, e.g., does the user need to manually type in a password on that computer.
  - Some indication of the power of the machine, e.g., you want to be able to tell PVM that your Cray is more powerful than your PC running a PD Unix.
- Given this information the computer can create the PVM system.

How PVM Works

- When PVM is started it examines the VM in which it is to operate, and creates a process on each machine, this is called the PVM demon, or simply `pvmd`.
- PVM provides a library of functions that the application programmer calls. Each function has some particular effect in the VM.
- However all this library really provides is a convenient way of asking the local `pvmd` to perform some work.
- The `pvmd` then acts as the virtual machine.

Cont.

- A sample virtual machine might be presented as:



- Note that all communication is via `pvmd`'s

What does pvmd do?

- `pvmd` is responsible for:
  - providing inter-host point of contact.
  - Authenticates task, i.e., provides all the security features we want to ensure multiple PVM users don't interfere with each other.
  - Executes processes on machines.
  - Provides fault detection.
  - Routes messages not from or intended for its host.
  - Transmits messages from its application to a destination.
  - Receives messages from other pvmd's, and buffers it until the destination application can handle it.
  - Is designed to be more robust than applications.

Describing the VM using a hostfile

- How do we tell the PVM system which computers we want to be part of the VM?
- The most common, but not the only, way of doing this is via a hostfile, which is read by PVM.
- For example:

```
#This is a sample PVM virtual hostfile
miles.man.ac.uk
frisell.mac.ac.uk
metheny.man.ac.uk
# each of the above machines can be used to run processes on
scofield.man.ac.uk pw
# but scofield requires the user to enter a password first
```


Cont.

- Each host listed in the file is automatically added to the VM unless prefixed with an &
- Hosts are entered one per line, with the name followed by options:

Option (Description)	Defaults
`lo = <> (Different login name)`	`same`
`pw (Pvmd asks for password)`	`don't ask, use rsh`
`dx = <> (Special location of pvmd)`	`$PVM_ROOT/lib/pvmd`
`ep = <> (Special a.out search path)`	`$PVM_ROOT/bin/`
`ms (Requires manual startup of pvmd)`	`don't`

- You can specify default settings for subseqent hosts with name *
- Comments preceded with #

Using rsh

- To create remote processes using rsh you must create a .rhosts file on the remote machine.
- Contains the list of machines which you will allow people to create jobs from, and the users you trust, e.g.,

```
miles.mcc.ac.uk frisell
```

- The remote rsh demon looks at this file before allowing you to create jobs.
- *This is a security risk!*
- To test it works, create a .rhosts on remote, then, on local type

```
rsh remote ls
```

- If you get a listing it worked!

Cont.

- Sometimes this simple example will work, but PVM will fail.
- The most common cause is I/O in your login scripts.
- I/O only makes sense if you have a console/window attached to a job. If PVM is creating jobs remotely for you, then it has nowhere to send messages to you.
- Make sure your .cshrc/.bashrc etc. files don't contain I/O.
- If necessary put those commands in your .login file, as this won't be executed by rsh.

Programming in PVM

Starting a PVM Program

- A programmer writes an ordinary program, which links in the PVM library, which it makes calls to for message passing and remote process control.
- However, before this program can be executed the *user* must start the `pvmd` on the local machine.
- When the application starts executing locally the first PVM function call initiates a connection to the *local* pvmd.

The VM Console

- PVM works to make the collection of machines function as a virtual machine.
- In common with most computers it also provides a console to this virtual machine.
- The console program allows users to execute commands which affect the VM, and perform some simple functions without writing programs.
- The console program is started by typing

```
$ pvm [-n master] [hostfile]
pvm> _
```

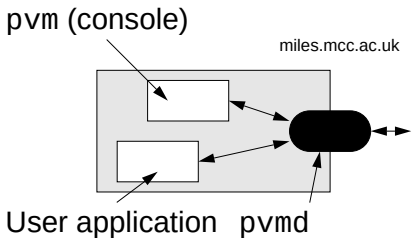
- The -n option is used to position the master pvmd on a remote host.

Console Details

- When started pvm looks to see if `pvmd` is running, and if not starts one!
- The user executes commands by name, a useful selection of which include:
  - add <name> : Add a new host to the VM
  - delete <name> : Delete a host from the VM
  - halt : Shuts down the console, AND the pvmd and other apps.
  - quit : Only shuts down the console.
  - jobs : Lists any jobs executing on the VM
  - spawn : Starts a PVM application
  - help : List commands which the console accepts

Cont.

- It is important to emphasise that the console is *called* pvm, but is only another application, it doesn't actually add any extra functionality!



- The console is normally used as a convenient way of starting pvmd with a particular hostfile.

The PVM A.P.I.

- We can now discuss how the users application interacts with the VM.
- It achieves this by calling functions in libpvm, which call the pvmd to do the work.
- Functions/Libraries are defined for C (& C++) and Fortran.
- Individual applications may use programs which have been written in C & Fortran, and expect them to be able to communicate easily across the VM.
- On the VM all programs are identified by an integer which is provided by the PVM libraries (actually pvmd).

Cont.

- A program finds its name by calling:

```
int tid = pvm_mytid(void);
or
call pvmfmytid(tid)
```

- There are two things worthy of note about this
  - All PVM functions are named pvm_* (from C) or pvmf* (from Fortran). This makes reading code easier!
  - When we describe a PVM function we will present the C & Fortran specs. They usually differ slightly, but perform the same operation.
- This function is normally the first thing any program calls, as it is important for a program to know its name!

Leaving the VM

- The last thing any program will call is

```
int info = pvm_exit(void);
call pvmfexit(info)
```

- This removes the program from the VM, *BUT* the program will continue to execute on that machine!
- Usually such a program will do a little local processing afterwards, perhaps to free some memory, and will then call

```
exit();
```

- Only then will the program actually stop!
- We have now described the two most important functions....but what do the programs do between starting and finishing?

Creating programs on the VM

- One of the most important tasks a program can do is create other programs!
- This is achieved by calling:

```
int numt = pvm_spawn(char *task,
    char **argv,int flag,
    char *where, int ntasks, int *tids)
call pvmfspawn(task, flag, where,
                ntasks, tids, numt);
```

- The parameters for which mean
  - task : Name of program which we wish to create.
  - argv : List of parameters we wish to pass to this new program, as per normal UNIX argv
  - where : Hostname , or empty string
  - ntasks : Number of programs with this name to create
  - tids : Place to put the tids of new procs.

Cont.

- The Flag parameter is a way of telling the VM how to deal with the new process. It is a bit mask which is constructed from a collection of operations.
  - PvmTaskDefault : PVM Chooses where to spawn processes
  - PvmTaskHost: `where` argument is a hostname
  - PvmTaskArch : `where` argument is a particular architecture to execute on, but we don't care which physical machine.
  - PvmTaskDebug: Starts a debugger, and runs task under it
  - PvmTaskTrace : Trace data will be generated for post-mortem debugging!
  - PvmMppFront: Starts task on MPP front end.
  - PvmHostCompl: Complements host set in where.

An Example

- To clarify this, the following example starts a single program called worker, on a machine called `miles.mcc.ac.uk` (the binary for which must be compiled and ready on that machine)

```
int worker_tid;

number = pvm_spawn("worker",
        (char **) 0,
        PvmTaskHost | PvmTaskDebug,
        "miles.mcc.ac.uk",
        1, &worker_tid);
```

- The variable number is set to the number of tasks which have been created : In this case 1 means success, 0 failure.
- Note that we also wish to run a debugger on `miles.mcc`, and run the worker in that.
- The tid of the child is placed in worker_tid

Exercise

- We have now explained
  - How PVM operates across a collection of workstations by pvmd's.
  - The sort of functionality PVM applies to the application writer.
  - How a user & program start PVM ready for use.
  - How a program creates other programs across the virtual machine.
- To reinforce this we now set a very simple exercise:

Write a program which uses PVM to start 3 child processes, each of which prints a hello message to the screen, and then exits.

Message Passing in PVM

What do we do with the VM?

- Once we have created tasks on the VM we need to be able to coordinate their efforts to be able to take advantage of parallelism.
- We do this by using a parallel programming technique called *message passing*.
- As previously discussed the programs communicate by sending data between themselves.

MP Communication Styles

- MP libraries provide various forms of communication:
  - One process addressing a collection of processes.
  - One process sending a message to another single named process.
  - One process tries to perform a global reduction operation on a collection of processes.
- PVM is no exception, and supports all these forms of communication.

The PVM MP Procedure

- Message passing in PVM is a 3 step process:

1 Create a buffer on the source process which will hold the data to be sent in any subsequent messages.

2 Place data in the buffer.

3 Initiate a send to a named group or process.

- We will discuss each of these steps in turn.

Managing Buffers

- The process of actually transferring data to another process, i.e., passing a message, may be a very involved process.
- To simplify the user programs view of this data is usually placed in buffers, and then the pvmd transfers the data from those buffers across the network.
- In PVM there is only ever one active send buffer.
- However multiple buffers may exist, and the library allows the program to switch the active one between them.

Creating a buffer

- The program asks the PVM API to make and manage buffers, and the program only deals with them in terms of their id.
- To create a buffer the program calls

```
int bufid = pvm_mkbuf(int encoding);
```

- The encoding parameter is intended to allow applications to speed up message transfer.
- Physical computers use specific ways of encoding data into memory.
- Unfortunately different processors use different ways of encoding this!
- Normally this doesn't bother us, but if we want to transfer binary data between machines it does!

Encoding Data

- PVM handles this problem by allowing the user to choose whether some intermediary encoding will be necessary in a particular message transfer.
- PVM uses SUN's XDR library to create a machine independent data format if you request it.
- Settings for the encoding option are:
  - PvmDataDefault: Use XDR by default, as the local library cannot know in advance where you are going to send the data.
  - PvmDataRaw: No encoding, so make sure you are sending to a like machine.
  - PvmDataInPlace: Not only is there no encoding, but the data isn't even going to be physically copied into the buffer. More on this later.

Using buffers

- Once we have created a buffer we need to tell PVM to use it,

```
int oldbuf = pvm_setsbuf(int bufid);
```

- This sets the active send buffer (hence the s!)
- We can switch between buffers using this function, using oldbuf to tell us the buffer we are switching from.
- When we wish to free the resources used by the buffer we call:

```
int info = pvm_freebuf(int bufid);
```

Preparing to send a message

- Note that PVM, when started, automatically creates a buffer for sending.
- However we still need to initialise it using:

```
int bufid = pvm_initsend(int encoding);
```

- Once either we have created a new buffer and set it to active, or initialised the default buffer, we are prepared to send a message.

Packing Data

- Any sends merely transfer data from the current send buffer.
- Consequently we must be able to place our application data in that buffer.
- This is achieved using packing functions.
- To place a single variable in the buffer the program calls a pack function, which places the data in the buffer, and returns a status to tell us whether it is successful.
- We never know how the packing function did its work (as it is dependent on the encoding for a particular buffer), and we don't need to know where the data actually went to.

Packing Functions

- The list of packing functions are

Function	Data
`pvm_pkbyte(char *cp, .. );`	Byte
`pvm_pkcplx(float *xp, ..);`	Float
`pvm_packdcplx(double *zp,..);`	Double
`pvm_pkdouble(double *dp, ..);`	Double
`pvm_pkfloat(float *fp, ..);`	Float
`pvm_pkint(int *np, .. );`	Integer
`pvm_pklong(long *np, ..);`	Long int
`pvm_pkshort(short *np, ..);`	Short
`pvm_pkstr(char *cp);`	Text
`pvm_packf(const char *fmt, <>)`	Arbitrary

- The last parameters of most functions are

```
, int nitem, int stride);
```

- To allow multiple items of the same type to be packed.

Cont.

- The last function packs a string/data line of the same format printf() accepts into the buffer.
- A single Fortran function handles all this,

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

  - xp : First item of array to be packed
  - nitem: Number of items (including length for strings)
  - what: data type

STRING	0	REAL4	4
BYTE1	1	COMPLEX8	5
INTEGER2	2	REAL8	6
INTEGER4	3	COMPLEX16	7

  - info: Set to number of items encoded.

Packing data in place

- Most PVM programs spend a lot of time packing data for sends.
- This may be an expensive operation, as the memory has to be physically copied.
- Things can be speeded up by describing the message buffer as using `PvmDataInPlace` encoding.
- Then, when data is packed, a pointer is placed in the buffer to indicate where the actual data is held.
- Must be careful to not delete the data before the data gets sent!

Sending Data

- Having created a buffer, and placed the data in it, we can now send the data to another process.
- The PVM program sends a particular buffer with
  - a destination, which is the tid of the process the message is intended for.
  - a msgtag, which is a label which is placed on the message, and which is presumably meaningful to the application.
- The msgtag allows different types of messages to be labeled. For example, warning messages might have a different flag to work messages.

Initiating a send

- Now, finally the program sends the data by calling

```
int info = pvm_send(int tid, int msgtag)
```

- If the program wishes to send to a collection of other processes it would call a multicast version of send:

```
int info = pvm_mcast(int *tids,
                     int ntask,
                     int msgtag);
```

- Here the array of integers, tids, of length ntask, is used to send messages to all of these.

An Example

- The following code excerpt sends a message containing two pieces of data to process # 1, with msgtag 2

```
char *text = “Hello out there”;
int count = 3;
int new_buffer;

new_buffer = pvm_mkbuf(PvmDataDefault);
pvm_setsbuf(new_buffer);
pvm_pkstr(text);
pvm_pkint(&count, 1, 0);
pvm_send(1,2);
```

- We wish to use XDR encoding, and use a new buffer to keep the message.
- We have now covered sending messages across PVM, at least in outline.

Receiving Messages

- Not surprisingly receiving messages in PVM is also a 3 step process:

1 Prepare a buffer to receive incoming messages.

2 Receive the message into that buffer.

3 Unpack the data from that message.

- Again we will go through each step in turn.

Preparing the receive buffer

- In PVM there is only ever one active send buffer, and one active receive buffer.
- Receive buffers are constructed using pvm_mkbuf(..), and then selected using

```
int oldbuf = pvm_setrbuf(int bufid);
```

- Then all incoming receives will be placed in that buffer.

Receiving a message

- There are 3 ways of receiving a message in PVM, all of which involve asking for a particular msgtag, or wildcard:
  - Your program will then wait until a message has arrived which matches this.
  - Receive the message if possible, otherwise carry on processing, but setting a flag to indicate the success of the operation. This is often called *non-blocking receive*.
  - Wait for a particular message, but if it hasn't arrived in some specified time operate as though we called a non-blocking receive.
- and one useful sundry routine:
  - ask to see if there is a message with a particular msgtag ready to be received.

pvmd's role

- All these functions rely on there being a difference between messages having reached a processor, and the message being placed in the application receive buffer.
- This is possible because all messages are actually directed to the relevant pvmd.
- pvmd is always able to receive messages, and performs some buffering to allow some delays between the message arriving and the application asking for it!

Blocking Receipt

- When a program wishes to receive a message into the currently active receive buffer, and wishes to timeout until it has finished this operation, it calls

```
int bufid = pvm_recv(int tid, int msgtag)
call pvmfrecv(tid, msgtag, bufid)
```

- Note that we can wait for particular messages from a particular sender.
- To use a looser wait we specify -1 in either tid or msgtag, as a wildcard for the relevant data.
- When this function has completed the relevant message will be placed in the active receive buffer.

Non-blocking Receipt

- If we want to receive a message if it is on the local pvmd, but not wait if it isn't there, we call

```
int bufid = pvm_nrecv(int tid, int msgtag)
call pvmfnrecv(tid, msgtag, bufid)
```

- If the message was there, then after the function completes the message will be in the active receive buffer, AND bufid will be set to the id of this buffer.
- Otherwise bufid will be set to 0, and the message won't be in the buffer.

Timed Receipt

- As a halfway house between waiting forever, and not-waiting at all, PVM provides a timed receipt function.
- The program supplies a UNIX timeval struct to indicate how long it is prepared to wait:

```
int bufid = pvm_trecv(int tid, int msgtag,
                      struct timeval *tmout)
call pvmftrecv(tid, msgtag, sec, usec, bufid)
```

- If the receipt was unsuccessful (a timeout occurred) then bufid will be set to 0, otherwise it functions as if pvm_recv() was called.

Asking pvmd

- As an additional useful feature PVM provides a function which lets the application ask the local pvmd if a message has arrived.

```
int bufid = pvm_probe(int tid, int msgtag)
call pvmfprobe(tid,msgtag,bufid)
```

- If the message isn't there it sets bufid to 0, otherwise it is set to the bufid which *would* receive the message.
- Note that the message isn't transferred into the active receive buffer!

Unpacking Messages

- Once the message has been received the program must unpack its contents into local variables.
- Again this is performed by calling unpack routines, which handle any necessary decoding which may be necessary.

Cont.

- The list of unpacking functions are

Function	Data
`pvm_upkbyte(char *cp, .. );`	Byte
`pvm_upkcplx(float *xp, ..);`	Float
`pvm_upackdcplx(double *zp,..);`	Double
`pvm_upkdouble(double *dp, ..);`	Double
`pvm_upkfloat(float *fp, ..);`	Float
`pvm_upkint(int *np, .. );`	Integer
`pvm_upklong(long *np, ..);`	Long int
`pvm_upkshort(short *np, ..);`	Short
`pvm_upkstr(char *cp);`	Text
`pvm_upackf(const char *fmt, <>)`	Arbitrary

- The last parameters of most functions are nitems and stride again.
- A single Fortran function performs all this:

```
call pvmfunpack(what,xp,item,stride,info);
```


Transferring structures

- It is normal in PVM applications for structures to be passed around from program to program.
- To ease the coding effort it is common to write a pack and unpack function specific to this structure, as once written it reduces the potential for errors.

Misc M.P.

- We have covered most of the functionality of message passing in PVM.
- However there are a couple of routines which relate to MP, but aren't critical to its use.
- Sending arrays:

```
int info = pvm_psend(int tid, int msgtag,
                     void *vp, int cnt, int type);
call pvmfpsend(tid,msgtag,xp,cnt,type,info)
```

- In C the what argument can be one of a list of pre-defined types, in Fortran the same values are used as for packing.
- and receiving them

```
int info = pvm_precv(int tid,int msgtag, void *vp,
                     int cnt,int *rtid, int *rtag,
                     int *rcnt);
call pvmfprecv(int,msgtag,xp,cnt,type,rtid,rtag,
               rcnt,info)
```

Cont.

- Querying the contents of buffers:

```
int info = pvm_bufinfo(int bufid, int *bytes,
                       int *msgtag, int *tid);
call pvmfbufinfo(bufid,bytes,msgtag,tid,info)
```

- Bytes is set to the length of the message currently in the named buffer, and tid the source.
- Asking what the active send and receive buffers are:

```
int bufid = pvmget{s|r}buf(void)
call pvmfget{s|r}buf(bufid)
```

- Modifying the way we can wait on messages reaching the local receive buffer.

```
int(*old)() = pvmrecvf(int(*new) (int buf,
                               int tid, int tag))
```

- This can only be done in C.

Exercise

- Now, to reinforce how PVM programs send messages, we want to write an application which ....

Groups

The need for groups

- Many MP applications require the ability to send messages to large groups of processes.
- We could implement this in the application, as repeated send calls, but most environments have more efficient ways of sending out one message to n destinations.
- In PVM we can simply do this by calling pvm_mcast() with an array of tids.
- Is this sufficient?....no!

Dynamic Groups

- The PVM mcast function only works when the sender knows the tids of *all* the recipients.
- What happens if a processor wishes to join an existing collection? It could
  - send a message to one processor responsible for keeping an up-to-date list of all processors within each collection
  - it could send messages to each member of the collection saying “hello, i’m a new member”.
- Both of these approaches are slow, and fraught with difficulties....what happens if two processors try to join at the same time?
- The application programmer could manage all this, but its a lot of work!

PVM’s Groups

- Instead PVM provides a group abstraction, which handles all these issues itself, leaving the application programmer to work on the application!
- Internally the groups implementation handles all the hassle of ensuring dynamic groups can be reliably and efficiently available.
- At any time any processor can join a group, and when a processor sends a message to the group it will reach all member of the group.

Joining a PVM group

- A program joins a PVM group by calling the function

```
int inum = pvm_joingroup(char *group);
call pvmfjoingroup(group, inum)
```

- Note that
  - the text group name indicates which group we wish to join. If a group of that name does not exist it is created, and this process enrolled in it.
  - the inum parameter indicates the instance number of this process in that group.
- The inum parameter may change if you join, leave and then rejoin the group.
- It has no relation to the tid of the calling process!

Leaving the group

- Similarly a program leaves a group by calling

```
int info = pvm_lvgroup(char *group);
call pvmflvgroup(group, info)
```

- The group name is needed because a program can be a member of an arbitrary number of groups.
- The leave group function will block until all the relevant processing has been performed.

Group Operations

- There are several operations a program can perform on a group.
- Ask for the tid of a particular inum in a group:

```
int tid = pvm_gettid(char *group, int inum)
call pvmfgettid(group,inum,tid)
```

- and the reverse operation:

```
int inum = pvm_getinst(char *group, int tid);
call pvmfgetinst(group,tid,inum)
```

- We can also find the number of processes in a particular group

```
int size = pvm_gsize(char *group);
call pvmfgsize(group, size)
```


Sending messages

- Any process, whether its a member of a group or not, can send a message to all the members of a group.

```
int info = pvm_bcast(char *group, int msgtag)
call pvmfbcast(group,msgtag,info)
```

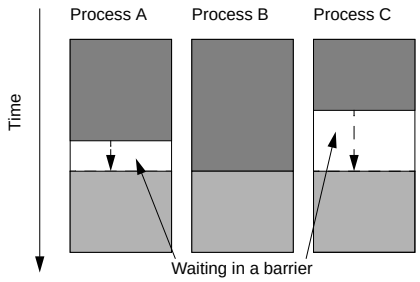
- The message is sent to all the processes which are in the group when the bcast is called.
- If a process joins the group while the broadcast being made it may not receive the message.

Controlling Groups

- We often wish to use groups as a way of controlling the application.
- For example in a master/slave application we might wish to have all the slaves in a group.
- One of the most important things we can do with groups is synchronise them.
- In PVM we can perform synchronisation operations by the barrier function.

Barriers

- A program, which is a member of a particular group, checks into the current barrier operating for that group.
  - if a program checks into a barrier for a group it isn't a member of an error has occurred.
  - only one barrier is active for each group.
- Barriers are effectively blocking functions, which will only return once a certain number of programs have entered the barrier.



Cont.

- A program enters the barrier for a particular group by calling:

```
int info = pvm_barrier(char *group, int count)
call pvmfbarrier(group,count,info)
```

- This barrier will wait until count members of a group have called pvm_barrier.
- Normally count is set to the number of processes that are in the group, but the program must specify this explicitly!
- Note that it is possible for different programs to each be waiting for different numbers of processors to join the barrier.
- The calling function counts as 1 member.

Global Reduction

- In writing parallel programs the author often has to code repetitive operations.
- To aid the developer PVM provides inbuilt support for one major class of common operations : *global reduction*.
- If n processors have some data on each, and we wish to process all the data to determine some value, and place this value on a single processor, then this is global reduction.
- We could implement all this ourselves, but PVM does half of what we need for us.

Cont.

- Global reduction operates as follows:
  - All the processors which contain data we wish to use, join a group.
  - They *all* call the pvm_reduce function.
  - This function call specifies the location of the data, the number of local data items, the datatype of the array, and the function which will be performed on it.
  - The function call also tells the operation the *group instance number* that we want the result to be placed on, the root.
  - The data is then processed, and transferred to the root. The actual data transfer is all handles by the pvm_reduce function.
- In order to make this a useful operation the pvm_reduce function allows the user to supply their own data operation function.

pvm_reduce()

- The function call is

```
int info = pvm_reduce(void (*func)(),
                      void *data,
                      int count,
                      int datatype,
                      int msgtag, char *group,
                      int root)
call pvmfreduce(func,data,count,datatype, msgtag,
                      group,root,info)
```

  - func is the function it will call on the data
  - data: pointer to the start address of an array of count values.
  - datatype: The data in the array, specified either by a constant (Fortran) or defined name (C).
  - msgtag : The tag of the message sent ot the root
  - root : Group instance number of process the data will be sent to
  - group : Name of group.

An Example

- Imagine an example in which a collection of workers were each performing a minimisation function on particular parametric areas of a function.
- After they have all finished we want to find the local minimum each has found, find the minimum of that set, and pass the result to the master.
- We first have to write the local function that each processor performs.
- In our case we assume that each worker has an array of parameter values, each entry contains a local minima. (For simplicities sake we'll use a 1D function)

Cont.

- So each worker has

```
float minimas[NUMBER_MINIMAS];
```

- We wish to find the minimum value in this list. As this is such a common operation PVM provides a minimum function : `PvmMin()`.
- So each worker would be able to use this to find the local minimum.
- Each worker also needs to know where the local result needs to be sent to. This is just an instance number in the group. In this case we do:

```
parent_tid = pvm_parent();
root = pvm_getinst(MY_GROUP, parent_tid);
```

- We are, of course, assuming that the master created all the workers.

Cont.

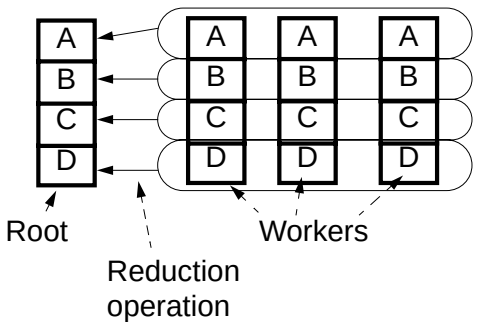
- So each worker, AND THE PARENT, calls the function

```
info = pvm_reduce(PvmMin, minimas,
                  NUMBER_MINIMAS, PVM_FLOAT,
                  1, MY_GROUP, root);
```

- Note that we are using 1 as the msgtag, though in a real application you would use something more meaningful.
- After the root calls this function its version of the minimas array will be overwritten with the result of the reduce operation over the group.
- The reduction operators are performed element-wise on the data.

Element-wise reduction

- In our example, then, the minimsation works like:



- This means that, if each worker has a collection of local minimas, the root still has some processing to perform on its new set of minimas.

Warning!

- pvm_reduce() *does not block*!
- This means that if a task calls pvm_reduce() and then leaves the group before the root calls the reduction operation an error will occur!
- In cases where this sort of thing could occur it is normal to use barriers to safeguard against it.
- However, provided the programmer is aware of the risks of a catastrophe the reduction operations are a useful tool.

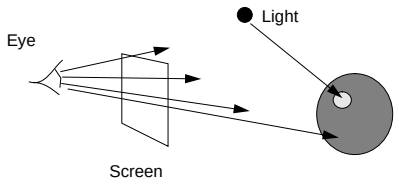
Exercise

- To reinforce the use of groups within PVM we now set a simple exercise using them!
-

Case Study

Ray Tracing

- We now wish to present a more complete PVM example. For a case study we pick ray-tracing as our application.
- The situation looks like:



- With complicated scenes this process is extremely expensive, often requiring hours of computation to produce a single image.
- Consequently ray tracing is a popular target for parallelisation.

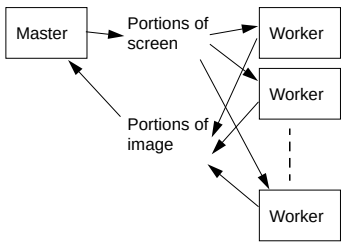
Cont.

- Assuming we already have a clear serial implementation of the ray tracing application (which may or may not be easy to produce) we can choose our parallelisation strategy.
- Fundamentally the processing involved for determining the colour of a pixel can be broken down into:
  - Determining whether the ray hits an object.
  - Calculating the portion of the colour dependent on the position of each light source.
  - Calculating the colour due to reflection
  - and the colour due to refraction through the intersecting surface.

- The time complexity of determining the colour of a pixel is entirely dependent on the intricacies of the route its ray traverses.
- The unpredictability indicates that work farming is a promising tactic, and so PVM is an ideal library to implement this.
- This means that we use two classes of processes:
  - The master, which doles out work to,
  - the workers.
- We are going to have to write two separate programs then, the master and the worker, each of which operates on the virtual machine.
- We now have to decide what the master passes to its workers.

Work packets

- We can pass out portions of the screen.
- The architecture of the application might look like:



- So, disregarding the setup costs, the two programs will form a logically straightforward application.

Creating the virtual machine

- Before we can run any PVM programs it is necessary to first define the VM.
- For the purposes of this example we can introduce one of the more advanced features of the hostfile, the ability to indicate the strength of different processors.
- This will enable us to tailor the execution of a parallel ray trace to the abilities of the VM.

```
miles.mcc.ac.uk sp = 1000
scofield.mcc.ac.uk sp = 5000
frisell.mcc.ac.uk sp = 2500
```

- Values are in the range 1-1000000, with 1000 the default.

Implementing the master

- The master program does the following:

```
split screen into n square blocks;
num_received = 0;
num_togo = n;

initialise workers;

while(num_received < n)
{
  id = receive(message);
  if(message=image)
  {
    place received image on screen;
    num_received--;
  }

  if(message=request)
  {
    send(id,next_block);
    num_togo--;
  }
}

shut down workers;
finish.
```


Initialisation

- In many parallel applications it is necessary to create sub-tasks, and then provide them with enough information to begin work.
- In our sample ray tracer we can achieve this by spawning tasks, joining them all into a group, then broadcasting the scene details to that group.

```
int workers[NUM_WORKERS];

info = pvm_spawn("worker",(char **)argc,
                 PvmTaskDefault,0,NUM_WORKERS,
                 workers);
mynum = pvm_joingroup("renderer");

/* Pack scene into current send buffer */

/* Then wait for all workers to join barrier */

pvm_barrier("renderer",NUM_WORKERS+1);

info=pvm_bcast("renderer",SCENE);
```


Receiving Requests

- Now we enter a loop which services the workers.
- We differentiate the different types of work request by their msgtag. First we handle any outstanding work requests

```
if(pvm_nrecv(-1, WORK_REQUEST))
{
    /* Unpack name of sender */
    /* Pack next work unit into active send buffer
    pvm_send(tid, WORK_PACKET);
}
```

- If there are any requests for work outstanding this receives the first one, unpacks it (within which the destination is encoded), and sends off the next scanline.

Cont.

- The procedure for receiving completed scanlines is very similar:

```
if(pvm_nrecv(-1, FINISHED_LINE))
{
    /* Unpack finished line */
    /* Place in screen */
}
```

- Note that, as communication is error free in PVM, there is no need to acknowledge receipt.
- Finally, when the screen is finished:

```
/* Pack 'die' message into send buffer */
pvm_bcast("renderer", WORK_PACKET);
pvm_lvgroup("renderer");
pvm_exit();
```

Implementing the worker

- The worker program looks like:

```
initialise self;
finish_looping = 0;
while(!finish_looping)
{
  send(master,request);
  receive(tag);

  if(tag=work)
  {
    process relevant portion of
        image screen;
    send(master,image);
  }
  if(tag=exit)
    finish_looping=1;
}

tidy up;
```

- Note that the worker asks the master for work, and then receives a message. It has to check the tag to see whether it is a die message.

Initialisation

- The worker is created by the master program. Note that the NUM_WORKERS was a command line argument to the master, and this has been passed onto the worker.Its initialisation process is:

```
myname = pvm_mytid();
parent = pvm_parent();

pvm_joingroup("renderer");

pvm_barrier("renderer",NUM_WORKERS+1);

bufid = pvm_recv(parent,SCENE);

/* Unpack the scene */
```

- The creation of the scene is a somewhat involved process, but is ignored for the purposes of this case study.

Work Cycle

- We ask for a work packet

```
/* Pack myname into the current send buffer */

int info = pvm_send(parent,WORK_REQUEST);
info = pvm_recv(parent,WORK_PACKET);

/* Unpack work packet */

if(..scan line to be rendered..)
{
    /* Perform work */
    /* Pack line into local send buffer */
    pvm_send(PARENT, FINISHED_LINE);
}
```

- But if the packet is a die message we exit the loop, and call

```
pvm_lvgroup("renderer");
pvm_exit();
```


Implementation Issues

- This, then, is the framework for a very simple parallel ray tracer.
- It performs some coarse load balancing by placing more processes on faster processors.

Conclusions

The features of PVM

- PVM provides
  - A *portable* platform for the construction of parallel programs.
  - A robust message passing channel.
  - Some fault tolerance.
  - A higher level interface to the virtual machine.
  - A well documented, well supported, popular application development tool.
- PVM, has during its lifetime, provided a very useful parallel programming tool, and will continue to do so for some time.
- However it is important to be aware of a recent international standard for message passing programs, MPI, which describes standard features MP libraries should provide.

MPI

The Message Passing Interface

- The standards group intended MPI to
  - Provide source code compatibility between machines and widely differing architectures.
  - Allow efficient implementations by providing some commonly used higher level features (e.g., global reduction operations).
  - Support heterogeneous architectures painlessly (from the programmers point of view).
- Most current MPI implementations are proof-of-concept or research oriented packages.
- Bindings are specified for C & Fortran.

MPI Basics

- Messages are composed of two types of data:
  - Basic types (MPI_CHARs. MPI_FLOATs) which are defined.
  - Derived types, which the user can construct. A similar effect can be achieved by writing a pack and unpack routine for structures in PVM, but this is easier to police.
- Derived types are defined in terms of the basic inbuilt types.
- The packing which the user must perform in PVM happens implicitly within an MPI implementation.

Communication

- There are 4 types of communication modes:

Sender mode	Mode notes
Synchronous send	Only completes when messages received by destination.
Buffered Send	Will complete independently of the recipient.
Standard Send	May be either synchronous or buffered
Ready send	The recipient *must* already have posted a receive call, otherwise behaviour is undetermined!

- Sends and receives may be blocking or non-blocking.

Cont.

- MPI specifies similar reliability to PVM, and PVM programs can be ported with reasonable ease.
- MPI was developed by experienced MP programmers. So many commonly used features have been placed within the standards, potentially allowing optimal coding.

Drawbacks

- MPI suffers from two principal problems *at the moment*:
  - It does not specify a standard way of describing the virtual machine upon which an application operates. This in stark contrast to the considerable effort expended by PVM.
  - MPI implementations tend to be proof-of-concept and are not maintained or documented as well as PVM. (Hopefully this will improve!)
- Therefore the PVM programmer should be aware that MPI is likely to gain in popularity in the future, though he or she may not need to program in it now.

Course Summary

- This course has covered:
  - The basic concept of a message passing library
  - The concept of the virtual machine
  - The basics of how PVM provides the illusion of a virtual machine (pvmd)
  - The philosophy of programming in PVM
  - How programs are created on the VM
  - How messages are sent and received by programs operating on the VM
  - How PVM provides dynamic program groups.
  - Some of the higher level features groups allow, barriers & global reduction.
  - A sample application framework.