An Introduction to PVM Programming

1. Introduction to PVM
2. Beginning Programming
3. Compiling and Running Your Program
4. Communication Between Tasks
5. Dynamic Process Groups
6. Load Balancing
7. Performance Considerations
8. Conclusion

Introduction to PVM

PVM is a message passing system that enables a network of Unix computers to be used as a single distributed memory parallel computer. This network is referred to as the virtual machine.

PVM can be used at several levels. At the highest level, the *transparent* mode, tasks are automatically executed on the most appropriate computer. In the *architecture-dependent* mode, the user specifies which type of computer is to run a particular task. In *low-level* mode, the user may specify a particular computer to execute a task. In all of these modes, PVM takes care of necessary data conversions from computer to computer as well as low-level communication issues.

PVM is a very flexible message passing system. It supports the most general form of MIMD parallel computation. This allows the use of any programming paradigm--all neccessary control structures can be implemented with suitable PVM constructs.

Beginning Programming

There are a few structures that are common to all PVM programs written in C. Every PVM program should include the PVM header file. This contains needed information about the PVM programming interface. This is done by putting
`#include "pvm3.h"` at the beginning of a C program or #include "fpvm3.h" in a Fortran program.

The first PVM function called by a program, commonly
`info=pvm_mytid()`, enrolls the process in PVM. `info` is an integer returned by the function. Like all PVM functions, `pvm_mytid()` will return a negative number if an error occurs. Programs should check for these errors, and respond appropriately. When a PVM program is done, it should call
`pvm_exit()`.

In order to write a parallel program, tasks must be executed on different processors. This is accomplished by calling `pvm_spawn()`. The man pages explain this function in detail. Here is an example of a typical call to
`pvm_spawn()`.

```
numt=pvm_spawn("my_task", NULL, PvmTaskDefault, 0, n_task, tids)
```

This spawns n_task copies of the program "my_task" on the computers that PVM chooses. The actual number of tasks started is returned to numt. The task id of each task that is spawned is returned in the integer array tids. Every PVM process has a unique integer that identifies it--this is its task id.

PVM has many useful information functions built into it. Some examples are pvm_parent(), pvm_config(), pvm_tasks(), etc... These can give your programs access to the types of information available at the pvm console. Details can be found in the man pages.

You should look at the source code for some of the sample programs now, to see what PVM programs look like.

Compiling and Running Your Program

In order to run your programs, you must compile them. On my machine,
```
cc -L~/pvm3/lib/ALPHA foo.c -lpvm3 -o foo
```
will compile a program called foo.c. You will have to change the name ALPHA to the architecture name of your computer. After compiling, you must put the executible file in the directory ~/pvm3/bin/ARCH. Also, you need to compile the program separately for every architecture in your virtual machine. If you use dynamic groups, you must also add -lgpvm3 to the compile command.

Your executible file can then be run. To do this, you should first run PVM. After PVM is running, your executible may be run from the Unix command line, like any other program.

Communication Between Tasks

Once you have learned how to spawn tasks and compile your programs, you are ready to do some actual parallel programming. To do this, you must learn how to let different tasks communicate with each other. In PVM, task-to-task communication is done with message passing.

When you are ready to send a message from task A to task B, task A must first call pvm_initsend(). This clears the default send buffer and specifies the message encoding. `bufid=pvm_initsend(PvmDataDefault)` is a typical use of this. After initialization, the sending task must then pack all of the data it wishes to send into the sending buffer. This is done with calls to the pvm_pack() family of functions. `pvm_packf()` is a printf-like function for packing multiple types of data. There are also functions for packing arrays of a single type of data, such as `pvm_pkdbl()` for packing doubles. The man pages have all the details on the different packing functions under `pvm_pack()`.

After the data have been packed into the sending buffer, the message is ready to be sent. This is accomplished with a call to `pvm_send()`. `info=pvm_send(tid, msgtag)` will send the data in the sending buffer to the process with the task id of tid. It tags the message with the integer value msgtag. A message tag is useful for telling the receiving task what kind of data it is receiving. For example, a message tag of 5 might mean add the numbers in the message, while a tag of 10 might mean multiply them. `pvm_mcast()` is a similar function. It does the same thing as `pvm_send()`, except it takes an array of tids instead of just one. This is useful when you want to send the same message to a set of tasks.

The receiving task makes a call to `pvm_recv()` to receive a message. `bufid=pvm_recv(tid, msgtag)` will wait for a message from task tid with tag msgtag to arrive, and will receive it when it does. A -1 can be specified for either the tid or msgtag, and will match anything. PVM 3.3 adds `pvm_trecv()`, which has the added ability to time out after a specified length of time. `pvm_nrecv()` can also be useful. This does a non-blocking receive--if there is a suitable message it is received, but if there isn't, the task does not wait. Sometimes `pvm_probe()` can be helpful as well. This will tell if a message has arrived, but takes no further action.

When a task has received a message, it must unpack the data from the receiving buffer. `pvm_unpack()` accomplishes this in the same manner that `pvm_pack()` uses to pack the data in. All data must be unpacked in exactly the same order as it was packed. Note that C structures must be packed element by element.

PVM 3.3 adds the function `pvm_psend()` for convenience. This packs and sends data in one call. PVM also includes routines for manipulating send and receive buffers directly. See the man pages on `pvm_mkbuff()` and `pvm_freebuf()` if you are interested in this.

Dynamic Process Groups

Dynamic process groups can be used when a set of tasks performs the either the same function or a group of closely related functions. Users are able to give a name to a set of PVM processes, which are all given a group instance number in addition to their tid.

When a task calls `inum=pvm_joingroup("group_name")`, it will be added to the group "group_name". If no such group exists, it will be created. The group instance number is returned in `inum`. This will be 0 for the first group member, and the lowest available integer for the rest of the group members. A task may belong to more than one group at a time. To leave a group, a tasks makes a call to `pvm_lvgroup()`. Be careful--if tasks leave the group without replacements joining, there will be gaps in the instance numbers.

There are group information functions. `pvm_getinst(), pvm_gettid()` , and `pvm_gsize()` return a process's group instance, tid, and group size, respectively. Other useful group functions are `pvm_bcast()` and `pvm_barrier()`. `pvm_bcast()` is very similar to `pvm_mcast()`, but instead of sending a message to an array of tids, it sends it to all members of a group. `pvm_barrier()` is used for synchronization. A task that calls `pvm_barrier()` will stop until all the members of its group call `pvm_barrier()` as well.

There are other group functions. See the PVM 3.3 man pages for more details. `pvm_gather()`, `pvm_scatter()`, and `pvm_reduce()` are some examples of group functions you may find useful.

Load Balancing

Load balancing is very important for applications. Making sure that each host is doing its fair share of work can be a real performance enhancer.

The simplest method is *static* load balancing. In this method, the problem is divided up and tasks assigned to processors only once. The partitioning may occur before the job starts, or as an early step in the application. The size and number of tasks can be varied depending on the processing power of a given machine. On a lightly loaded network, this scheme can be quite effective.

When computational loads vary, a more sophisticated *dynamic* method of load balancing is required. The most popular method is called the *Pool of Tasks* paradigm. This is typically implemented as a master/slave program where the master manages a set of tasks. It sends slaves jobs to do as they become idle. This method is used in the sample program xep supplied with the distribution. This method is not suited for applications which require task to task communication, since tasks will start and stop at arbitrary times. In this case, a third method may be used. At some predetermined time, all the processes stop; the work loads are then reexamined and redistributed as needed. Variations of these methods are possible for specific applications.

Performance Considerations

PVM places no limits on the programming paradigm a PVM user may choose. However, there are some performance considerations that should be taken into account.

The first is task granularity. This typically refers to the ratio of the number of bytes received by a process to the number of floating point operations it does. Increasing the granularity will speed up the application, but the tradeoff is a reduction in the available parallelism.

The total number of messages sent is also a consideration. As a rule of thumb, sending a small number of large messages takes less time than sending a large number of small messages. This is not always the case, though. Some applications can overlap sending small messages with other computation. The optimal number of messages is application specific.

Some applications are suited to functional parallelism, others to data parallelism. In functional parallelism, different machines do different tasks based on their specific capabilities. For example, a supercomputer may solve part of a problem suited to vectorization, a multiprocessor may solve another part suited to parallelization, and a graphics workstation may be used to visualize the data in real time. In data parallelism, the data is distributed to all of the tasks in the virtual machine. Operations (often quite similar) are then performed on each set of data and information is passed between processes until the probelm is solved. PVM programs may also use a mixture of both models to fully exploit the strengths of different machines.

Network considerations are also important if you are using a cluster of machines. Different computers will have different processing power. Even if they are all the same model of workstation, there can be sizable performance differences due to multitasking of other users. Network latencies can cause problems. Also, the processing power available may change dynamically based on machine loads. To combat these problems, some form of load balancing should be used.

Conclusion

Hopefully this tutorial has shown you the basics of programming with PVM and demonstrated some of its capabilities. For more information, you should consult the user's guide, as well as the man pages. Both are available online. They can give you the details you need to write real PVM applications.