Using PVM on Multiprocessor Machines

Weicheng Jiang

May 21, 1993

1 Why PVM?

We started porting PVM to massively parallel machines a few months ago, right now it is (or will be) available on the following platforms:

- Intel iPSC/860 and Paragon
- CM-5 (?)
- IBM SP-1 (?)
- Cray MPP (?)
- nCUBE/2 (?)
- Meiko (?)
- T.node (?)

On a workstation cluster PVM plays the role of a distributed operating system. It allows the programmer to start processes on any machine connected to the network and pass messages between them. The reason for putting PVM on parallel machines is to give the user a uniform, vendor-independent programming interface. We have all come to appreciate the convenience of having Unix on just about every workstation, and for an emerging field such as parallel computing the importance of a standard interface could not be overstated. To deliver on this promise we have to maintain compatibility between different versions of PVM, unfortunately PVM itself changes so rapidly that this remains an elusive goal.

2 Overhead

PVM offers a portable programming environment across multiple platforms, but at what price? On a MPP machine PVM tasks run on the nodes like regular processes. PVM gets in the way only when the tasks request for services, mainly to send and receive messages. The user has to pack the data into a buffer before they are sent and unpacked them at the receiving end. This overhead is small compared to the time it takes to send a message. To prove this we measured the costs of preparing and sending a message in PVM versus the roundtrip times on an Intel iPSC/860.

Table 1: Timing for node-to-node PVM send/recv on iPSC/860.

Size (bytes)	Pack (μs)	Send (μs)	Echo (μs)
100	26.5	82	686
1K	72	84	1037
10K	519	96	4295
100K	4944	104	36460

In these measurements the send buffer was initialized to PvmDataRaw, there was no encoding involved, and the fragment size was 128K so the data were copied into the buffer in one piece. Therefore the packing time in the first colume basically reflects the memory access time. As we can see the additional cost of packing increases with the message size but remains small compared to the message travel time.

To improve the performance of a PVM MPP application, the node program should call pvm_send() as soon as the data becomes available, so (hopefully) when the other guy calls pvm_recv() the message will already be in its buffer. Also the Raw encoding (no encoding) format should be used whenever possible.

3 Implementation

One major advantage of MPP machines is the fast interconnects between nodes. PVM exploits this capability by passing task-to-task messages directly, without the interference of the pvmd.

The "address" of a task is encoded in the task ID, as illustrated below:

```
 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1
+-+-+---------------------+-+----------+--------------------+
|s|g|       host index    |n|  instance |   node # (2048)      |
+-+-+---------------------+-+----------+--------------------+
```

The 'n' bit is used to distinguish between node tasks (running on the compute nodes) and host tasks (running on the service nodes). The node number is the address of the node, but "instance" can have different meanings on different machines. On the Paragon, for example, processes running on the same node are identified by their "process types", which are basically local process IDs. On the iPSC/860, the nodes are not time-shared, but PVM can manage multiple cubes, so this field stores the cube ID.

PVM send calls the native nonblocking ("incomplete", in Intel terminology) send function on the system. The task can then go on to do other things while the message is being sent. When the task calls pvm_initsend() or pvm_freebuf(), PVM will check to see if the send operation has completed.

4 Porting PVM Applications

Some MPP machines do not provide full Unix support on the nodes. For example processes running on the nodes of an iPSC/860 have no Unix process IDs and they cannot receive Unix signals. There is a similar problem for the CM-5.

If a node process forks, the behavior of the new process is machine-dependent. In any event it would not be allowed to become a new PVM task.

The pvm_spawn() function always creates tasks on the compute nodes. Processes running on the service nodes (and only these processes) can enroll in PVM by calling pvm_mytid().

On some machines (e.g., iPSC/860), only one process is allowed on each node, so the total number of PVM tasks cannot exceed the number of nodes available.

5 Mixed Environment

When PVM is used on a workstation, there is usually one (or a few at the most) task(s) and one PVM Daemon (pvmd). The issue there is how to keep pvmd quiet most of the time so the tasks can get some work done on the CPU. On a parallel machine, however, the pvmd runs on the front-end and it has to provide services to a large number of tasks. Care must be taken to reduce the traffic going through pvmd. One way to avoid a bottleneck is to divide your tasks into groups, tasks in one group can be assigned to a multiprocessor machine while those in the next group can be assigned to a cluster of workstation, and so forth. Tasks in the same group can communicate more frequently. Another remedy is to open up a task-to-task TCP socket to get the data out, but this won't work if your MPP machine does not support sockets on the nodes.

Load balancing is another area that needs some work. PVM can figure out the number of nodes available on a multiprocessor machine, but the user should also be allowed to specify in the host file the effective speed rating (defined as: (single-processor speed) $\times$ (number of processors)) of the machine. In the current implementation tasks are divided evenly among all hosts, and no consideration is given to the relative speed of individual host. So if you start PVM on your desktop, adding the superfast MPP, and then spawn 1000 tasks, 500 of them will be spawned on your workstation while the other 500 get spawned on the MPP, unless you specify explicitly the target host name or architecture.