Enhancements to PVM's BEOLIN Architecture

Paul L. Springer[1]

California Institute of Technology, Jet Propulsion Laboratory, 4800 Oak Grove Drive, Pasadena CA 91109, USA

Abstract. Version 3.4.3 of PVM had previously been enhanced by the addition of a new architecture, BEOLIN, which allowed a PVM user to abstract a Beowulf class computer with a private network to appear as a single system, visible to the outside world, which could spawn tasks on different internal nodes. This new enhancement to PVM handles the case where each node on the Beowulf system may be composed of multiple processors. In this case, the software will, at the user's request, spawn multiple jobs to each node, to be run on the individual processors.

1 Introduction

The BEOLIN architecture support was added to PVM in version 3.4.3. The motivation for this addition was the limitation PVM had before that time in dealing with Beowulf clusters. Prior to that version, a PVM user of a heterogeneous system that included a Beowulf cluster was unable to treat that cluster as a single system image. If the Beowulf nodes were on an internal network, they were invisible to the rest of the PVM system. The only way the nodes could be used was to treat them as individual computers, provided they were visible to the entire PVM system. The disadvantage to this approach was that the user's application was forced to issue PVM *addhost* and *spawn* commands to each individual node. Furthermore, each node incurs the overhead of running the PVM daemon.

The special architecture support that had been added on earlier versions of PVM for individual parallel machines, such as IBM's SP2, Intel's Paragon, and others, provided the inspiration to add the BEOLIN architecture support, to handle Beowulf clusters running Linux. By means of the BEOLIN support, PVM can spawn tasks to a single machine target for the Beowulf cluster. That target knows about, but hides the details of the machine, such as the numbers of nodes available, and the individual IP address of each node. The target handles the details of spawning the tasks onto individual Beowulf nodes for execution.

Since that time hardware enhancements have been made to Beowulf systems, and one such enhancement is the availability of multiple processors per node, where those processors share the node's IP address. The initial BEOLIN release could only handle a single processor per node. This paper describes a new enhancement that supports multiple processors per node.

No paper was presented describing the original BEOLIN release, and so information on that release will be included in this paper to provide the necessary background.

2 Design

Of the Massively Parallel Processor (MPP) architectures previously supported, BEOLIN is most similar to the SP2MPI port, which supported the SP2, using MPI as the underlying message passing protocol. BEOLIN, however, uses sockets instead of MPI.

The following description of the initiation of tasks onto a BEOLIN machine assumes the cluster has a front-end: a node that is visible to the rest of the PVM system, and can itself communicate to the individual nodes that comprise the BEOLIN computer. PVM can begin running on the cluster in different ways, for example by a user logging on to the front-end and running the pvm command on that front-end, or by running a pvm application on an external computer that issues a *pvm_addhosts* command, targeting the cluster. (See figure 1.)

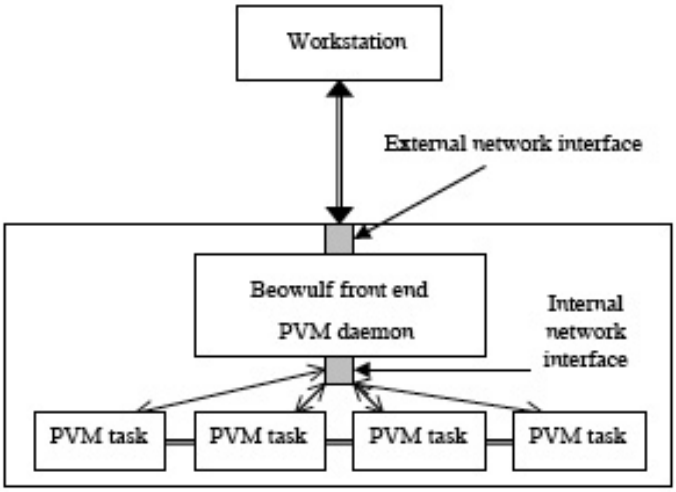

Fig. 1. Beowulf Block Diagram

When PVM is initiated on the cluster, the first thing that happens is that the PVM daemon (pvmd) begins running on the front-end. The daemon reads an environment variable to discover the names of the nodes in the cluster, and forms a node pool for later use. The variable in question specifies the initial order of node allocations, as well as the number of tasks that can be executed on a given node. Subsequent spawning requests cause the daemon to allocate nodes from the pool and initiate the requested tasks onto them. As tasks complete, their corresponding nodes are freed back into the pool for future use.

At the time the pvmd daemon on the front-end initiates a task on a cluster node, it forks a copy of itself, with both child and parent running on the front-end. The child's standard output and error are connected to the just-started task. Each child process so generated is visible to pvmd, and the pvm monitor will show the child as a task with the suffix ".host" appended to its name. No copy of the daemon runs on the targeted node.

When the remote task begins execution, it examines a shared file space (/tmps) for a file name beginning with "pvmd" and ending with the user id. That file contains the necessary addressing information for the task to make a connection with the pvmd daemon. PVM messages and commands are relayed through the resulting connection.

3 Installing and Using the BEOLIN Port of PVM

As of version 3.4.3, BEOLIN is a defined architecture for PVM. However, to force the build process to use the BEOLIN files, the environment variable *PVM_ARCH* must be set to the value *BEOLIN*–don't rely on the *pvmgetarch* command for this. If this is not done properly, the build will probably produce a plain LINUX version instead. Once the architecture variable is properly set, build PVM as described in Chapter 9 of [1].

Before the resulting BEOLIN build can be run, a shared file space with the name /tmps must be set up in such a way that any file in this subdirectory is accessible by the front-end as well as the cluster nodes on which tasks may be run by PVM.

In the previous version of BEOLIN, there was a requirement that the nodes had to be able to connect to the front-end using the address returned by the *gethostname()* Linux call on the front-end. In the current version that requirement has been relaxed by the use of an existing command line parameter for PVM. When starting either pvm or pvmd, using the command line parameter *-n* $<$ *hostname* $>$ tells the BEOLIN code that pvmd should write the IP address of $<$ *hostname* $>$ in the /tmps file it creates, informing the cluster nodes what address they should use in making their connection to the daemon.

The BEOLIN daemon code looks for the environment variable *PROC_LIST* when it starts up on the front-end. This should be defined in the environment used by pvmd when it is running, and is typically set in the .cshrc (or equivalent) file. The value of the variable should be set to a colon separated list of the names of the cluster nodes available for pvmd to use. If multiple processors exist on the node, and it is desirable to use them when spawning tasks, the node name can appear multiple times in *PROC_LIST*. For example, a line in the .cshrc file that read *setenv PROC_LIST n0:n1:n1:n2:n3* would allow pvmd to spawn 1 task onto node n0, 2 onto n1, 1 onto n2 and 1 onto n3.

Note that the user can not designate which processor in a node a task is to be run on. The assignment of the specific processor is left up to the O/S. If more nodes are required by the spawn request than there are slots available in the node pool, PVM will return an "Out of Resources" error.

If all messages between cluster nodes are forced to pass through the pvmd on the front-end, the job will not scale well, and the front-end will become a communication bottleneck. To avoid this situation, it is strongly recommended that PVM's direct message routing be used. This can be accomplished by the application calling the *pvm_setopt()* routine, with the *PVMRouteDirect* parameter. This forces messages from one node to another to go directly, instead of being routed through the daemon.

3.1 Restrictions and Limitations

The BEOLIN version of the PVM monitor or pvmd can be run on the front-end node, but starting either one on one of the other nodes of the cluster has not been well tested, and is advised against.

When the application calls *pvm_addhosts()* from an external computer, to add the cluster, it should only specify the front-end node of the cluster as an argument. The command should not be used to add the other cluster nodes to the virtual machine. Similarly, the *add* command of the PVM monitor should not be used to add individual nodes. Avoid including individual node names in the PVM hostfile. The purpose of this BEOLIN port is to treat the entire cluster as a single machine, to be addressed only by the name of the front-end.

PVM uses its TID (task ID) word to uniquely identify each task running on the virtual machine. To distinguish different tasks running on the same node, the BEOLIN code uses the three bit partition field in the TID. This limits the maximum number of child tasks able to run on a single cluster node to eight.

Even if all the cluster nodes have a direct connection to the external network, this BEOLIN port can still be used to treat the cluster as a single PVM machine. In this particular case, any node can be arbitrarily designated as the front-end.

4 Internals

The bulk of the BEOLIN code is in the pvmdmimd.c file in the BEOLIN source code subdirectory in the PVM package. This section will give an overview of the BEOLIN program code.

4.1 Initialization

When the pvmd daemon starts up on the cluster front-end, the main() routine in pvmd.c calls *mpp_init()*, passing it the argc and argv parameters, in order to initialize the BEOLIN part of PVM. The *mpp_init()* routine first parses the environment variable *PROC_LIST*, and calls *gethostbyname()* for each entry in the list, storing the resulting addressing information internally. The number of times each node is referenced in the list is tracked, and the reference number is stored in the *partNum[]* array.

Several arrays are created in the BEOLIN initialization process, with entries in the arrays corresponding to entries in *PROC_LIST*. The *nodepart* array holds

the partition number for the entry, the *nodeaddr* array contains the IP address, the *nodeconn* array is initialized to 0, and the *nodelist* array carries the name as it appears in *PROC_LIST*.

4.2 Spawning

When the cluster's pvmd receives a command to start the spawned tasks on its nodes (by means of a call to *pvm_spawn()*, for example), control is passed to the BEOLIN routine *mpp_load()*. The first thing *mpp_load()* does is to check to see if there are enough node slots in its free node pool, by calling *mpp_new()*. If there are enough slots, *mpp_new()* allocates and returns the set of slots that will be used to spawn the tasks; otherwise it will report an error. As part of the allocation process, *mpp_new()* generates an identifier called ptype that is unique to the set of nodes allocated for this set of tasks, and puts this set of nodes into its busynodes list, marking each node with the ptype value.

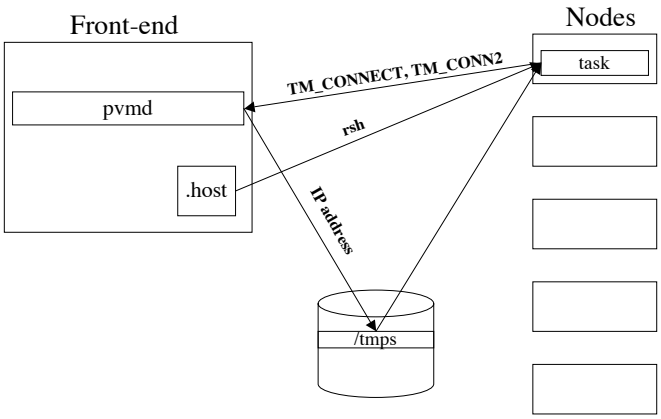

Fig. 2. Task Spawning

Once the node slots are allocated, *mpp_load()* iterates a set of actions for each of the new tasks, whereby it sets up a task structure for that task, and calls *forkexec()* in the main body of PVM to actually start the task running by means of an rsh command, on the appropriate node. Before *forkexec()* runs the rsh command, it first forks to produce the previously described "host" task. The "host" task then executes the rsh command. (See figure 2.)

When first started on the target node, each spawned task must connect with its corresponding "host" task on the front-end. This action is triggered by the

first PVM call the spawned task makes, and the action takes place inside the routine *pvmbeatask()*, contained in the lpvm.c module which is part of the PVM code linked to the application. To open a socket connection with pvmd, *pvmbeatask()* calls the version of the *mksocs()* routine in the lpvm.c module. The connection is made by searching for a file name with the pattern pvmd.*userid*, in the /tmps file space shared by the front-end and the nodes. The file is read and the information within it is used to make the connection.

Once the socket connection is made, the spawned task sends a *TM_CONNECT* message to the pvmd. The pvmd's routine *tm_connect()* sends an acknowledgment back to the task, which then causes the task to respond with a *TM_CONN2* message back to the pvmd. The pvmd code enters the *tm_conn2()* routine, which in turn calls the BEOLIN *mpp_conn()* code to determine which of the task spawn requests this current connection process matches. When the match is made, the corresponding entry in the BEOLIN *nodeconn* array is set true.

4.3 Communication

When the application wants to send a message from one node to another, the routine *pvm_send()* is called. Once the message is constructed, *mroute()* is called to determine the routing and send the message. It first determines whether the direct routing option has been set, allowing the message to bypass the pvmd. If so, and if this is the first time the destination has been requested by this task, it opens a socket to the destination task, and makes an entry (ttpcb) in the task process control block list. The entry includes the socket information, to be used the next time a message is sent to the same destination.

4.4 Task termination

Task termination is triggered by the application's call to *pvm_exit()*, done for each spawned task that is part of the application. When the pvmd receives the exit request, it calls *task_free()*, which in turn calls BEOLIN's *mpp_free()* routine, passing each task as the argument. The *mpp_free()* routine first finds a match for this task, based on node number and partition number, and then shuts down the "host" task corresponding to the spawned task that was passed, sets a flag to indicate that this task is done executing, and returns. The pvmd code later calls BEOLIN's *mpp_output()* routine, as it does on a regular basis. As *mpp_output()* cycles through each task that has been spawned and is still alive, it checks to see if the "host" task corresponding to this task is still alive. If not, it then takes responsibility for shutting down the spawned task, and returns its nodes to the free pool.

5 Performance

In allowing the use of multiple processors on a node, this latest enchancement to the BEOLIN architecture focuses on increasing the computing power available to

a PVM application. It is appropriate, then, to evaluate this version by choosing a benchmark that is computationally intensive, rather than one that relies on communication speed. One benchmark that meets this criteria is based on the POV-Ray program[2]. POV-Ray is a multi-platform program that creates an image of a scene using a ray tracing technique. It uses as its input a POV format text file that describes the objects in the scene as well as the camera and lighting and other effects.

POV-Ray was not originally written to run under PVM, but there is a patch available to do this. The patch was downloaded from [3], and applied to version 3.5 of POV-Ray. As patched, the program runs in a master-slave configuration, with the work parceled out to the slaves whenever a slave has completed its previous work. The number of slaves is configurable at startup.

For the purposes of benchmarking, a standard benchmark.pov file is available as input to the program, as well as a standard initialization file[4]. A large number of both sequential and parallel benchmark numbers have been published. The benchmark was run on a Beowulf cluster composed of 800 MHz dual Pentium III nodes. PVM was configured to run on 2, 4, 8 and 16 nodes, using both 1 and 2 processors per node. The results can be seen in the accompanying figures.

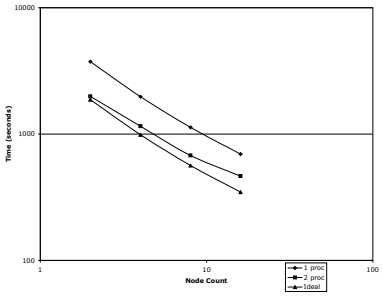

Fig. 3. BEOLIN Performance

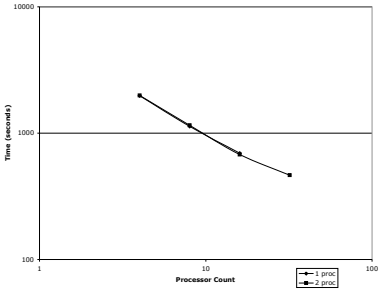

Fig. 4. Comparative Performance

Figure 3 shows the improvement in computing power enabled by the new BEOLIN version. The upper trace shows the benchmark times with the old limitation of running just 1 processor per node. In the middle trace, both processors on the node are put to use, and the run times are faster, close to the ideal of being twice as fast (as shown in the bottom trace). The ideal might be missed not because of the BEOLIN implementation, but because the problem itself doesn't scale well. The true explanation can best be determined by looking at figure 4. The two processors per node trace is shown as before. The trace labeled "1 proc" indicates how well the problem scales when using the same number of *total processors* (one per node) as the two processors per node case. It can be seen that run times are identical whether the processors used are on the same node

or not. This indicates that the slight lack of scaling shown in figure 3 is not the result of using two processors on the same node.

As shown in figure 4, performance falls off slightly when 32 processors are used, in the two processors per node case. The computer used did not have 32 nodes, so the equivalent time for the 1 processor per node case could not be measured. But the benchmark produces a number of statistics, including load balancing information, and analysis of that showed that the run time was slower in this case because work was not evenly distributed, and near the end of the run only a few nodes had work to do.

6 Future Work

The largest number of nodes used so far with this port of PVM has been 31. The software needs to be run on much larger machines in order to characterize its performance at the high end. It is unclear what bottlenecks and limitations exist, and how they will manifest themselves, when one pvmd daemon is controlling a large number of tasks. Changes in the pvmd code to accommodate very large systems may be necessary. One particular area to examine is the way that a host task is forked off on the front-end for each spawned task. This will not scale as the number of tasks becomes very large. To support larger number of tasks, it will be necessary to change the way this is done, or perhaps to spawn limited numbers of additional pvmds on the other nodes.

7 Conclusion

The BEOLIN architecture described here offers new capabilities to the PVM applications programmer. A Linux PC cluster can now be added to the virtual machine, and parallel tasks spawned onto the cluster, even when the individual nodes are on a private network. Clusters with multiple processors on a single node can have tasks spawned to each processor on the node.[1]

References

1. Geist, Al, et al: PVM: Parallel Virtual Machine. The MIT Press, Cambridge, Massachusetts (1996)
2. "POV-RAY - The Persistence of Vision Raytracer." 25 Mar. 2005. Persistence of Vision Raytracer Pty. Ltd. 3 Jul. 2005. $< http://www.povray.org >$
3. "PVM patch for POV-RAY." 2003. SourceForge. 3 Jul. 2005. $< http://pvmpov.sourceforge.net >$
4. "Haveland-Robinson Associates - Home Page." 5 Mar. 2004. Haveland-Robinson Associates. 3 Jul. 2005. $< http://haveland.com/povbench >$

[1] This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. The funding for this research was provided for by the Defense Advanced Research Projects Agency under task order number NM0715612, under the NASA prime contract number NAS7-03001.