Scyld Beowulf Series 29

User’s Guide

Scyld Software

Scyld Beowulf Series 29: User's Guide

by Scyld Software
29cz Edition
Published April 2005

Table of Contents

Preface

Welcome to the Scyld Beowulf Cluster Operating System User's Guide. This manual is for someone who will use a Beowulf cluster to run applications. This manual covers the basics of Beowulf parallel computing - what a Beowulf is, what you can do with it, and how you can use it. Topics covered include basic Beowulf architecture, unique features of the Scyld Beowulf Operating System, navigating the Scyld environment, how to run programs, and how to monitor their performance.

What's *not* covered here is how to install, configure, or administer your Beowulf cluster. If you have not yet built your cluster or installed the Scyld Beowulf Operating System, you need to refer to the *Installation Guide*. If you are looking for information on administering your cluster, you will need to read the *Administrator's Guide*. This manual is for the user accessing a Beowulf System that has already been configured.

What's also not covered is a complete guide to using the Linux operating system, on which the Scyld software is based. Some of the basics you need to know are presented here. However, if you've never used Linux or UNIX before, it is suggested that you check out a book or online resources about the subject. A good source of information are books by *O'Reilly and Associates*[1].

This manual will not teach you to write programs for a Beowulf cluster. You will find information about developing applications in the *Programmer's Guide*.

With this manual, you will learn the basic functionality of the utilities needed to start being productive with a Scyld Beowulf cluster.

Feedback

We welcome any reports on errors or problems that you may find. We also would like your suggestions on improving this document. Please direct all comments and problems to: <`support@scyld.com`>.

When writing your e-mail, please be as specific as possible, especially with errors in the text. Include the chapter and section information. Also, mention in which version of the manual you found the error. This version is 29cz, published April 2005.

Notes

1. http://www.oreilly.com

Chapter 1. Scyld Beowulf Overview

The Scyld Beowulf Cluster Operating System is a Linux-based software solution for high performance computing systems. It solves many of the problems long associated with Linux Beowulf-class cluster computing while simultaneously reducing the costs of system installation, administration and maintenance. With the Scyld Beowulf Cluster Operating System, the cluster is presented to the user as a single, large scale parallel computer.

This chapter serves as an introduction to both the Scyld Beowulf Cluster Operating System and this document. It presents background information on Beowulf clusters and then delves into the specifics of a Scyld Beowulf. By the end of this chapter, you will understand what a Beowulf cluster is and how it can be used. You will also understand the differences between the standard Beowulf architecture and a Scyld Beowulf. A high-level technical summary of Scyld Beowulf and its major software components is also presented in this chapter with further details provided throughout the Scyld Beowulf Series 29.

What is a Beowulf Cluster?

The term *Beowulf* refers to a multi-computer architecture designed for executing parallel computations. A *Beowulf cluster* is a parallel computer system conforming to the Beowulf architecture, which consists of a collection of commodity computers referred to as *nodes* connected via a private network running an open source operating system. Each node, typically running Linux, has its own processor(s), memory storage and I/O interfaces. The nodes communicate with each other through a private network, such as Ethernet or Myrinet, using standard network adapters. The nodes usually do not contain any custom hardware components and are trivially reproducible.

One of these nodes, designated as the *master node*, is usually attached to both the private network and the public network, and is the cluster's administration console. The remaining nodes are commonly referred to as *compute nodes* or *slave nodes*. The master node is responsible for controlling the entire cluster and for serving parallel jobs and their required files to the compute nodes. In most cases, the compute nodes in a Beowulf cluster are configured and controlled by the master node. Typically these compute nodes do not have keyboards nor monitors and are accessed soley through the master node. To the cluster's master node, the compute nodes are simply treated as additional processor and memory resources available for its use.

In conclusion, Beowulf is a technology of networking Linux computers together to create a parallel, virtual supercomputer. The collection as a whole is known as a Beowulf cluster. While early Linux-based Beowulf clusters provided a cost effective hardware alternative to the supercomputers of the past for executing high performance computing applications, the original software implementations for Linux Beowulf clusters were not without their problems. The Scyld Beowulf Cluster Operating System addresses and solves many of these problems.

A Brief Beowulf History

The type of *cluster computer* architecture described in the previous section has been around for a while in various forms. This original concept for a *network of workstations* (NOW) simply used a group of normal processors connected through a typical office network. These workstations used a small amount of special software to harness their idle cycles. This concept is depicted in the figure shown below.

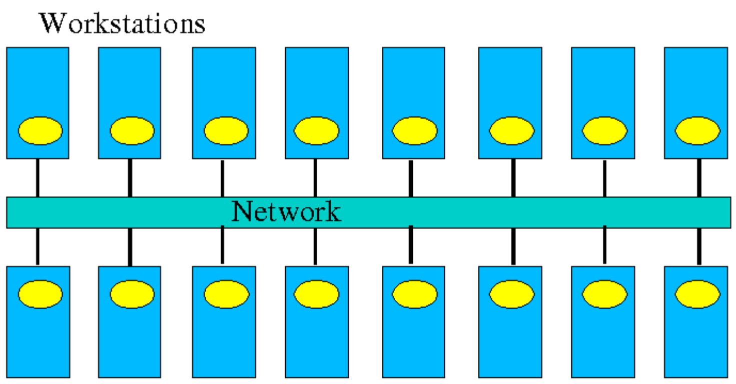


Figure 1-1. Network-of-Workstations Architecture

The NOW concept evolved to the *Pile-of-PCs* architecture with one master PC connected to the public network and the remaining PCs in the cluster connected to each other, and the master through a private network. Over time, this concept solidified into the Beowulf architecture as described in the previous section and depicted in the figure shown below.

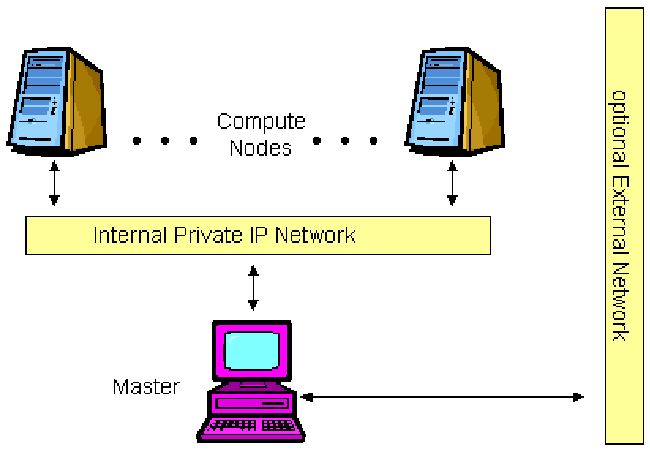


Figure 1-2. A Basic Beowulf Cluster

For a cluster to be correctly termed a *Beowulf*, it must adhere to the *Beowulf philosophy*. This philosophy has three main components:

1. scalable performance
2. the nodes must be using commodity off-the-shelf (COTS) hardware
3. the nodes must be running open source software, typically Linux

Using commodity hardware allows Beowulf clusters to take advantage of the economies of scale in the larger computing markets. In this way, Beowulf clusters can always take advantage of the fastest processors developed for high-end workstations, the fastest networks developed for backbone network providers, and so on. The progress of Beowulf clustering technology is not governed by any one company's development whims, resources or schedule.

First Generation Beowulf Clusters

The original Beowulf software environments were implemented as downloadable add-ons to commercially available Linux distributions. These Linux distributions included all of the software needed for a networked workstation: the kernel, various Linux utilities and many add-on packages. The downloadable Beowulf add-ons included several programming environments and development libraries as individually installable packages.

With this first generation Beowulf scheme, every node in the cluster required a full Linux installation and was responsible for running its own copy of the kernel. This requirement combined with the lack of a uniform, commercially supported distribution created many administrative headaches for the maintainers of Beowulf-class clusters. For this reason, early Beowulf systems tended to be deployed by the software application developers themselves and required detailed knowledge to install and use. The Scyld Beowulf Cluster Operating System distribution reduces and/or eliminates these and other problems associated with the original Beowulf-class clusters.

Scyld Beowulf

The Scyld Beowulf Cluster Operating System provides a software solution designed specifically for Beowulf clusters. The Scyld distribution streamlines the process of configuring, administering, running and maintaining a Beowulf-class cluster computer. It was developed with the goal of providing the operating system software infrastructure for commercial production cluster solutions.

Scyld software was designed with the differences between master and compute nodes in mind, and only runs appropriate parts of the operating system on each component. Instead of having a collection of computers each running its own fully installed operating system, Scyld creates one large distributed computer. The user of a Scyld Beowulf cluster will never login to one of the compute nodes, nor worry about which compute node is which. To the user, the master node *is* the computer and the compute nodes appear merely as attached processors capable of providing more computing power. With a Scyld Beowulf, the cluster appears to the user as a single computer:

- the compute nodes appear as attached processor and memory resources
- all jobs start on the master and are migrated to the compute nodes at runtime
- all compute nodes are managed/administered collectively via the master

The Scyld Beowulf architecture simplifies cluster integration and setup, minimal and easy system administration, increases reliability, and seamless cluster scalability. In addition to its technical advances, Scyld Beowulf provides a standard, stable, commercially supported platform for deploying advanced clustering systems.

Top Level Features of Scyld Beowulf

The following list summarizes the top level features available with Scyld Beowulf:

Easy Installation

The Scyld Beowulf installation procedure is identical to a standard Linux server installation with one additional dialog screen for configuring the network interfaces on the master node. See the *Scyld Beowulf Installation Guide* for full details.

Install Once, Execute Everywhere

A full installation of Scyld Beowulf is only required on a single node in the cluster, the master node. Compute nodes are provisioned from the master node during their boot-up procedure and dynamically cache any additional parts of the system during process migration.

Single System Image

Scyld Beowulf makes a cluster *act and feel* like a single, multi-processor, parallel computer. The master node maintains and presents to the user a single process space for the entire cluster. See the section on the *BProc: Beowulf Distributed Process Space* in the *System Design Description* chapter in the *Scyld Beowulf Administrator's Guide*.

Execution Time Process Migration

Scyld Beowulf stores applications on the master node. At execution time, processes are migrated from the master node to the compute nodes. This approach virtually eliminates the risk of *version skew* and means hard disks are not required for the compute nodes. See the section on the *BProc: Beowulf Distributed Process Space* in the *System Design Description* chapter in the *Scyld Beowulf Administrator's Guide*.

Seamless Cluster Scalability

Scyld Beowulf seamlessly supports the dynamic addition and deletion of compute nodes to / from the cluster without modification to existing source code and / or configuration files. See the *BeoSetup* chapter in the *Scyld Beowulf Administrator's Guide*.

Administration Tools

Scyld Beowulf includes simplified tools for performing cluster administration and maintenance. Both graphical user interface (GUI) and command line interface (CLI) tools are supplied. See the *Scyld Beowulf Administrator's Guide*.

Web-based Administration Tools

Scyld Beowulf includes web-based tools for remote administration and monitoring of the cluster and job execution. See the *Administrator's Guide* for more information.

Batch Queuing Tools

Scyld Beowulf includes a robust batch queuing / job scheduling system. *BBQ*, the Scyld Beowulf Batch Queuing system, includes command-line, GUI and web-based interfaces. See the chapter on *Running Programs* in this guide and the chapter on *Job Batching* in the *Administrator's Guide*.

Additional Features

Additional features include support for cluster power management (Wake-on-LAN, Power On/Off), both runtime and development support for MPI and PVM, and support for the LFS, NFS3 and PVFS file systems. This information is covered in various places throughout the Scyld Beowulf documentation set.

Fully Supported

Scyld Beowulf is fully supported by Scyld Software.

Scyld Beowulf Technical Summary

Scyld Beowulf presents a more uniform system image of the entire cluster to both users and applications through extensions to the kernel. A guiding principle of these extensions is to have little increase in kernel size and complexity and more importantly, negligible impact on individual processor performance. In addition to its enhanced Linux kernel, the Scyld Beowulf distribution includes improved libraries and utilities specifically designed for high performance computing applications. Generally speaking, more detailed information on the various topics discussed in this section can be found in other chapters of this document and in the *Administrator's Guide*.

Beowulf Process Space Migration Technology

Scyld Beowulf is able to provide a single system image through its use of *BProc*, the Beowulf process space management kernel enhancement. *BProc* enables the processes running on cluster compute nodes to be visible and manageable on the master node. All processes appear in the master node's process table. Processes start on the master node and are migrated to the appropriate compute node by *BProc*. Process parent-child relationships and UNIX job control information are both maintained with migrated jobs. The `stdout` and `stderr` stream from jobs is redirected back to the master through the network. The *BProc* mechanism is one of the primary features that makes Scyld Beowulf different from traditional Beowulf clusters. For more information, see the *System Design Description* chapter in the *Scyld Beowulf Administrator's Guide*.

Compute Node Boot Procedure

The compute nodes in a Scyld Beowulf cluster boot using a two-stage procedure. Compute nodes begin their boot process using a local, minimal *stage 1* boot image, after which they contact the master node to obtain their final *stage 2* boot image. Stage 1 boot images contain a minimal Linux kernel with just enough functionality to configure a reliable TCP/IP socket connection between the compute node and the master node.

Once the stage 1 image is booted, the compute node attempts to communicate with the master to obtain its required runtime files and complete its initialization procedure. After the master node validates the compute node's Ethernet address and verifies the node is *officially* part of the cluster, it replies back to the compute node with the its IP address and a fully functional stage 2 kernel. Further information on the cluster boot procedure can be found in both the *System Design Description* and the *Booting the Cluster* chapters in the *Scyld Beowulf Administrator's Guide*.

Compute Node Categories

Each compute node in the cluster is classified into one of three categories by the master node: *unknown*, *ignored* or *configured*. An *unknown* node is one not formally recognized by the cluster as being either a *configured* node or an *ignored* node. When bringing a new compute node online, or after replacing an existing node's network interface card, the node will be classified as *unknown*. *Ignored* nodes are typically nodes that for one reason or another you'd like the master node to simply ignore. They are not considered part of the cluster and will not receive a response from the master during their boot process. A *configured* node is one that is listed in the cluster configuration file using the `node` tag. These are nodes that are formally part of the cluster and recognized as such by the master node. When running jobs on your cluster, these are the nodes actually used as computational resources by the master. For more information on node categories, see the *System Design Description* chapter in the *Scyld Beowulf Administrator's Guide*.

Compute Node States

For each of the *configured* nodes in the cluster, *BProc* maintains the current condition of the node. This piece of information, known as the node's *state*, is always one of the following values: `down`, `unavailable`, `error`, `up`, `reboot`, `halt`, `pwroff` or `boot`. Each state is described below.

`down`

Node is not communicating with the master and its previous state was either `down`, `up`, `error`, `unavailable` or `boot`

`unavailable`

Node has been marked unavailable or off-line by the cluster administrator; typically used when performing maintenance activities

`error`

Node encountered an error during its initialization; this state may also be set manually by the cluster administrator

`up`

Node completed its initialization without error; node is online and operating normally. *This is the only state in which end users may use the node.*

`reboot`

Node has been commanded to reboot itself; node will remain in this state until it reaches the `boot` state as described below

`halt`

Node has been commanded to halt itself; node will remain in this state until it is reset or powered back on, and reaches the `boot` state as described below

`pwroff`

Node has been commanded to power itself off; node will remain in this state until it is powered back on and reaches the `boot` state as described below

`boot`

Node has completed its *phase 2* boot but is still initializing; after the node finishes booting, its next state will be either `up` or `error`

More information on the *node states* can be found in the *System Design Description* chapter in the *Administrator's Guide*.

Major Software Components

The following is a list of the major software components distributed with the Scyld Beowulf Cluster Operating System. For more information, see the relevant sections in the Scyld Beowulf Series 29: the *Installation Guide*, *Administrator's Guide*, the *User's Guide*, the *Reference Guide*, and the *Programmer's Guide*.

bproc

The Beowulf process migration technology; an integral part of Scyld Beowulf

beosetup

A GUI interface for configuring the cluster

beostatus

A GUI interface for monitoring cluster status

beostat	A text-based tool for monitoring cluster status
beoboot	A set of utilities for booting the compute nodes
beofdisk	A utility for remote partitioning of hard disks on the compute nodes
beoserv	The beoboot server; it responds to compute nodes and serves the boot image
bpmaster	The bproc master daemon; it only runs on the master node
bpslave	The bproc compute daemon; it runs on each of the compute nodes
bpstat	A bproc utility; it reports status information for all nodes in the cluster
bpctl	A bproc utility; a command line mechanism for controlling the nodes
bpsh	A bproc utility; a replacement utility for **rsh** (remote shell)
bpcp	A bproc utility; a mechanism for copying files between nodes, similar to **rcp** (remote copy).
MPI	The Message Passing Interface; optimized for use with Scyld Beowulf
PVM	The Parallel Virtual Machine; optimized for use with Scyld Beowulf
mpprun	A parallel job creation package for Scyld Beowulf
bbq	The Beowulf Batch Queue system; a cluster enhanced version of **atq**
beoqstat	The GUI BBQ tool; a GUI interface for viewing and deleting your batch jobs

beoweb

The web-based cluster administration and monitoring tool package

Typical Applications of Scyld Beowulf

Beowulf clustering provides a great solution for anyone executing jobs that involve a large number of computations and large amounts of data, such as image rendering. For example, some of the special effects used in the movie *Titanic* were done using clusters. Beowulf clustering is ideal for both large monolithic parallel jobs but also for running many normal sized jobs, many times such as in monte carlo type analysis. Examples of Beowulf applications include Finite Element Analysis for mechanical system modeling, seismic data analysis, computational fluid dynamics, financial analysis, genome research, computational drug development, etc.

These types of jobs can be performed many times faster running on a Scyld Beowulf cluster as compared to running on a single computer. The increase in speed depends on the application itself, the number of nodes in the cluster, and the type of equipment used in the cluster. All of these items can be easily tailored and optimized to suit the needs of your specific application.

More and more applications are being developed for both business and commercial applications. Many companies are handling more data than ever before and need increasing computational power to handle it efficiently. In many cases, these needs are being fulfilled using Beowulf clusters. The following are some examples of applications already being performed using Beowulf clusters.

Computationally Intensive Activities

Optimization problems, stock trend analysis, complex pattern matching, medical research, genetics research, image rendering

Scientific Computing / Research

Engineering / simulations, 3D modeling, finite element analysis, fluid dynamics, PCB / ASIC routing

Large-Scale Data Processing

Data mining, complex data searches and results generation, manipulating large amounts of data, data archival and sorting

Web / Internet Uses

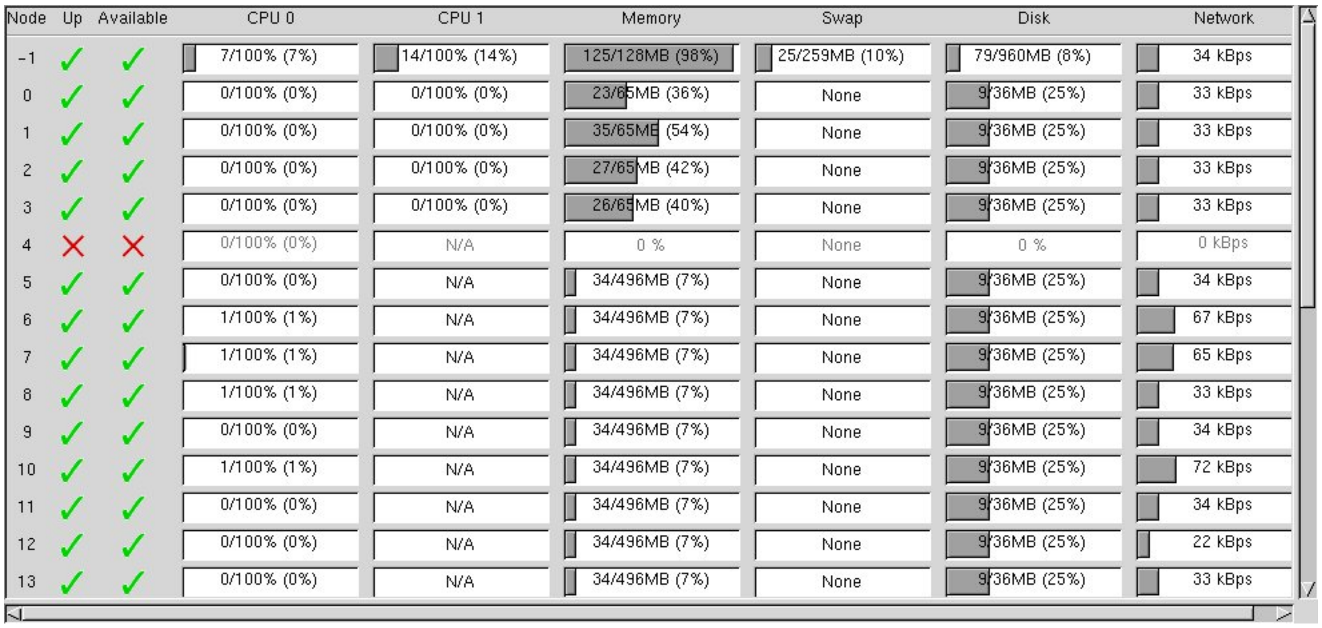
Web farms, application serving, transaction serving, calculating serving, data serving

Chapter 2. Interacting with the System

Verifying that the Cluster is Up and Running

Before you interact with a cluster, you might first want to make sure that the cluster has compute nodes that are up and running. Unlike traditional Beowulf clusters, Scyld Beowulf provides consistent reporting at multiple levels about the availability of the nodes.

The **beostatus** tool is the best way to get an idea of the status of the cluster, including which nodes are up. The **beostatus** tool has many ways of showing you the status of individual nodes. The default way is to show up as an X window. This happens when you log in and can be done by simply typing **beostatus** at the command prompt. These different ways are documented later in this chapter.



Node	Up	Available	CPU 0	CPU 1	Memory	Swap	Disk	Network
-1	✓	✓	7/100% (7%)	14/100% (14%)	125/128MB (98%)	25/259MB (10%)	79/960MB (8%)	34 kBps
0	✓	✓	0/100% (0%)	0/100% (0%)	23/65MB (36%)	None	9/36MB (25%)	33 kBps
1	✓	✓	0/100% (0%)	0/100% (0%)	35/65MB (54%)	None	9/36MB (25%)	33 kBps
2	✓	✓	0/100% (0%)	0/100% (0%)	27/65MB (42%)	None	9/36MB (25%)	33 kBps
3	✓	✓	0/100% (0%)	0/100% (0%)	26/65MB (40%)	None	9/36MB (25%)	33 kBps
4	✗	✗	0/100% (0%)	N/A	0 %	None	0 %	0 kBps
5	✓	✓	0/100% (0%)	N/A	34/496MB (7%)	None	9/36MB (25%)	34 kBps
6	✓	✓	1/100% (1%)	N/A	34/496MB (7%)	None	9/36MB (25%)	67 kBps
7	✓	✓	1/100% (1%)	N/A	34/496MB (7%)	None	9/36MB (25%)	65 kBps
8	✓	✓	1/100% (1%)	N/A	34/496MB (7%)	None	9/36MB (25%)	33 kBps
9	✓	✓	0/100% (0%)	N/A	34/496MB (7%)	None	9/36MB (25%)	34 kBps
10	✓	✓	1/100% (1%)	N/A	34/496MB (7%)	None	9/36MB (25%)	72 kBps
11	✓	✓	0/100% (0%)	N/A	34/496MB (7%)	None	9/36MB (25%)	34 kBps
12	✓	✓	0/100% (0%)	N/A	34/496MB (7%)	None	9/36MB (25%)	22 kBps
13	✓	✓	0/100% (0%)	N/A	34/496MB (7%)	None	9/36MB (25%)	33 kBps

In order to actually use a Scyld Beowulf cluster, you must have at least one node up. As shown in the screenshot above, all of the nodes that are up have a green check mark on their row. If there are less nodes up than you think there should be, or some say error, you should contact your systems administrator so that they can fix the problem.

Another command that can be used is **bpstat**. When run without any options, it prints out a listing of all the nodes and their current stats. If you are using **bpstat** instead of looking for the green checkmark, you will want to check that the node's state is set to `up`.

Issuing Commands

Master Node

When you log into the cluster, you are actually logging into the Master Node. As such, any commands that you type directly on the command line will execute on the master node. The only exception to that is when you use the special commands for interacting with the compute nodes.

Compute Node

Bpsh is a utility for running jobs on the compute nodes. It is primarily intended for running utilities and maintenance tasks, rather than for parallel programs. Bpsh is a replacement for the traditional UNIX utility “rsh”, used to run a job on a remote computer. Like **rsh**, the arguments to **bpsh** are the node to run the command on, and the command to run. **Bpsh**, doesn’t allow you to get an interactive shell on the remote node like **rsh** does, however it does allow you to run a command on more than one node without having to type the command once for each node.

The typical use of **bpsh** is to run a command or utility program on a single node or a range of nodes. For example, if you wanted to check for the existence of a file in the `/tmp` directory of node 3 called “output”, from the master node you would simply run the command:

```
bpsh 3 ls /tmp/output
```

and the output would appear on the master node in the terminal on which you issued the command. A range of nodes can also be specified for bpsh to operate on. To run the same command on nodes 3, 4, and 5 simultaneously, you would issue the command:

```
bpsh 3,4,5  ls /tmp/output
```

The -a flag is used to indicate to **bpsh** that you wish to run on all available nodes. So, the command:

```
bpsh -a  ls /tmp/output
```

would produce output for every node currently active in your cluster. If you have a large cluster, this output may be hard to read. For instance, if your cluster had 64 nodes, and on half of them the file /tmp/output existed, and you ran the command above you would get back the text “/tmp/output” 32 times and the text “ls: /tmp/output: no such file or directory” 32 times. The lines of output would be sorted by the speed at which the compute nodes responded. This makes it impossible to determine if the file existed on a particular node or not. **bpsh** has a number of options for formatting its output to make it more useful for the user. The **-L** option makes bpsh wait for a full line from a node before it prints out the line. This keeps you from having half a line from node 0 printed, with a line from node 1 tacked onto the end, then followed by the rest of the line from node 0. The **-p** option prefixes each line of output with the node number of the line that produced it (this forces the functionality for **-L** as well). The **-s** option forces the output of each node to be printed in sorted numerical order, ie. all the output for 0 will appear before any of the output for node 1. A divider can be added between the output of each node by adding a **-d** option. Using **-d** causes the functionality of **-s** to be used even if **-s** isn’t specified. So, the command:

```
bpsh -A -d -p  ls /tmp/output
```

when run on an 8 node cluster would produce output like this:

```
0       ------------------------------------------------------------------
  0: ls: /tmp/output: No such file or directory
1       ------------------------------------------------------------------
  1: ls: /tmp/output: No such file or directory
```

which makes it clear which nodes do and do not have the designated file.

Bpsh provides a convenient yet powerful interface for manipulating all or a subset of the cluster nodes simultaneously. **Bpsh** maintains the flexibility of being able to access a node individually, but removes the requirement of accessing each compute node individually when a collective operation is desired. The complete reference to all the options available for the bpsh command can be found in the *Reference Guide*.

bpsh and Shell Interaction

Some advanced users like to use special shell features such as piping and input and output redirection. Getting this functionality to work with *bproc* can sometimes be tricky, so this section is aimed at explaining what is required to get this functionality to work as you'd like. In all the examples below, the program running will be **cmda**. If it is piped to anything, it will be piped to **cmdb**, if an input file is used, that file will be `/tmp/input` and if an output file is used, it will be `/tmp/output`, and the node used will always be node 0.

The easy case is running a command on the compute node, and doing something with its output on the master, or giving it input from the master. Here are a few examples:

```
bpsh 0 cmda | cmdb
bpsh 0 cmda > /tmp/output
bpsh 0 cmda < /tmp/input
```

A bit trickier thing to do is to run the command on the compute node and do something with its input or output on that compute node. There are two ways of doing that. The first way requires that all the programs you run be on the compute node. For this to work, you would first have to copy the cmda and cmdb executable binaries to the compute node. After copying the executables, here is an example of how you could execute them:

```
bpsh 0 sh -c "cmda | cmdb"
bpsh 0 sh -c "cmda > /tmp/output"
bpsh 0 sh -c "cmda < /tmp/input"
```

The other way does not require any of the programs to be on the compute node, but wastes a lot of network bandwidth as it takes the output, sends it to the master node, then right back to the compute node. Here is how it would work:

```
bpsh 0 cmda | bpsh 0 cmdb
bpsh 0 cmda | bpsh 0 dd of=/tmp/output
bpsh 0 cat /tmp/input | bpsh 0 cmda
```

The final thing to do, is to run a command on the master and do something with its input or output on the compute nodes.

```
cmda | bpsh 0 cmdb
cmda | bpsh 0 dd of=/tmp/output
bpsh 0 cat /tmp/input | cmda
```

Copying Data to Compute Nodes

There are a few different ways to get data from the master node to the compute node. The easiest way is NFS shared files. By default, all files in `/home` are shared to all compute nodes via NFS. This includes all files that are in your home directory. In order to access the file on the compute node, you need to open the file and it will be there, even though its really accessing the file that is stored on the master node.

Another method for getting data to a compute node is to use **bpcp**. This command works much like the standard UNIX command **cp** in that you pass it a file to copy as one argument and the destination as the next argument. However, any file listed, the source, destination, or both, can be prepended with a node number and a `:` to specify that the file in question is on that node. So, if you wanted to copy the file `/tmp/foo` to the same location on node 1, you would do:

```
bpcp /tmp/foo 1:/tmp/foo
```

The third method for transfering data is to do it programatically. This is a bit more complex, so will only be described conceptually here. If you are using an MPI job, you can have your rank 0 process on the master node read in the data, then use MPI's message passing abilities to send the data over to a compute node. If you are instead writing a program that uses BProc functions directly, you can read the data while you are on the master node, then when you move over to the compute node, you should still be able to access the data you read in while on the master node.

Monitoring and Signaling Processes

top, ps, and kill

One of the features that Scyld Beowulf has that traditional beowulf doesn't is a shared Process Id (pid) space. This allows you to see and control jobs that are running on the compute nodes from the master node using standard UNIX tools such as **top**, **ps**, and **kill**.

Scyld Beowulf adds a tool called **bpstat** that makes it easy to see what nodes the processes are actually running on. **bpstat** has two options that help us know what nodes a process is running on.

The first option is -p. Using **bpstat -p** lists all of the processes that are currently being send to compute nodes and gives their pid as well as the node they are running on and the node they started on.

```
[user@cluster root]# bpstat -p
PID     Node
6301    0
6302    1
6303    0
6304    2
6305    1
6313    2
6314    3
6321    3
```

The PID column tells us what the process id is. The Node column indicates which node it is running on.

The second option is -P. With this option, **bpstat** takes the output of **ps** and outputs it again with a new column at the beginning indicating which node the process is running on. Here is some sample output from **ps**:

```
[user@cluster root]$ ps xf
  PID TTY      STAT   TIME COMMAND
 6503 pts/2    S      0:00 bash
 6665 pts/2    R      0:00 ps xf
 6471 pts/3    S      0:00 bash
 6538 pts/3    S      0:00 /bin/sh /usr/bin/linpack
 6553 pts/3    S      0:00  \_ /bin/sh /usr/bin/mpirun -np 5 /tmp/xhpl
 6654 pts/3    R      0:03      \_ /tmp/xhpl -p4pg /tmp/PI6553 -p4wd /tmp
 6655 pts/3    S      0:00          \_ /tmp/xhpl -p4pg /tmp/PI6553 -p4wd /tmp
 6656 pts/3    RW     0:01          \_ [xhpl]
 6658 pts/3    SW     0:00          |   \_ [xhpl]
 6657 pts/3    RW     0:01          \_ [xhpl]
 6660 pts/3    SW     0:00          |   \_ [xhpl]
 6659 pts/3    RW     0:01          \_ [xhpl]
 6662 pts/3    SW     0:00          |   \_ [xhpl]
 6661 pts/3    SW     0:00          \_ [xhpl]
 6663 pts/3    SW     0:00              \_ [xhpl]
```

And here is the same output, after being run through **bpstat -P**:

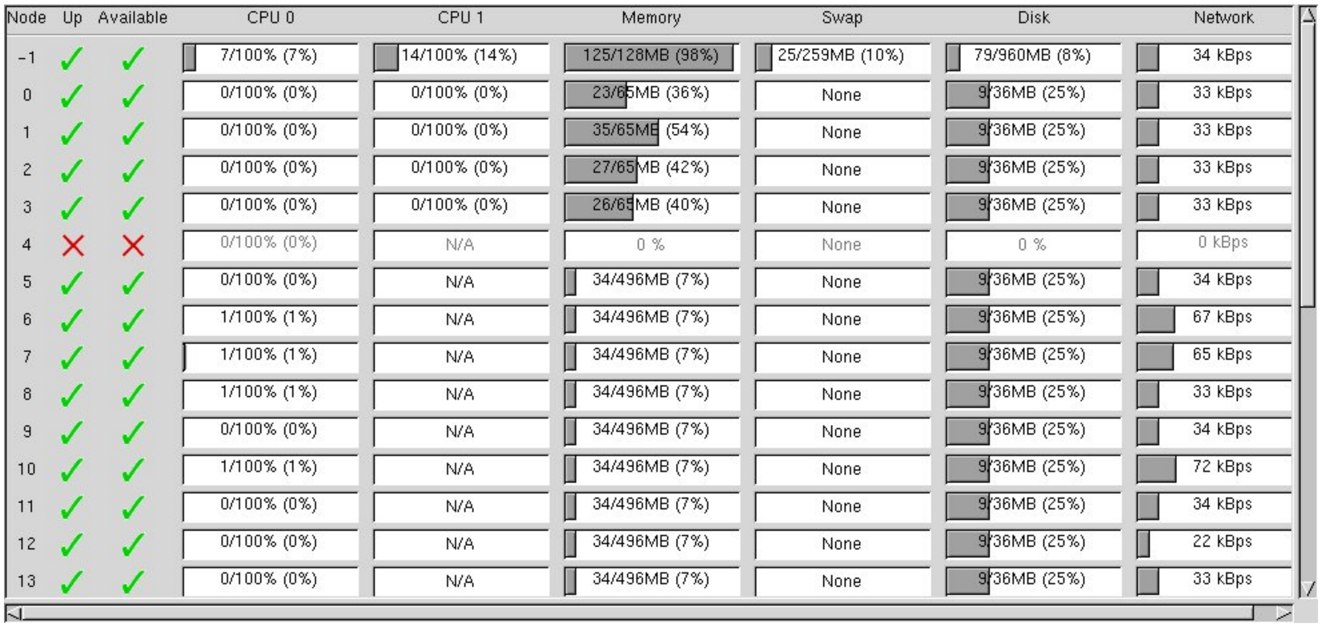
```
[user@cluster root]$ ps xf|bpstat -P
NODE      PID TTY      STAT   TIME COMMAND
         6503 pts/2    S      0:00 bash
         6666 pts/2    R      0:00 ps xf
         6667 pts/2    R      0:00 bpstat -P
         6471 pts/3    S      0:00 bash
         6538 pts/3    S      0:00 /bin/sh /usr/bin/linpack
         6553 pts/3    S      0:00  \_ /bin/sh /usr/bin/mpirun -np 5 /tmp/xhpl
         6654 pts/3    R      0:06      \_ /tmp/xhpl -p4pg /tmp/PI6553 -p4wd /tmp
         6655 pts/3    S      0:00          \_ /tmp/xhpl -p4pg /tmp/PI6553 -p4wd /tmp
0        6656 pts/3    RW     0:06          \_ [xhpl]
0        6658 pts/3    SW     0:00          |   \_ [xhpl]
1        6657 pts/3    RW     0:06          \_ [xhpl]
1        6660 pts/3    SW     0:00          |   \_ [xhpl]
2        6659 pts/3    RW     0:06          \_ [xhpl]
2        6662 pts/3    SW     0:00          |   \_ [xhpl]
3        6661 pts/3    SW     0:00          \_ [xhpl]
3        6663 pts/3    SW     0:00              \_ [xhpl]
```

Monitoring Node Status

Scyld Beowulf includes a graphical based tool as well as command line tools for monitoring state and performance information for each node on the cluster. The graphical based tool provides continuous updates, while the command line tools provide snapshots only.

Beostatus GUI Tool

The Beowulf status monitor **beostatus** provides a graphical view of the node state, processor utilization, memory and disk usage, and network performance. Each line in the **beostatus** display reports information about a single node. Once running, **beostatus** is non-interactive; the user simply monitors the reported information. However, at startup, there are a number of command line options which can be used to modify the default behaviour of **beostatus**.

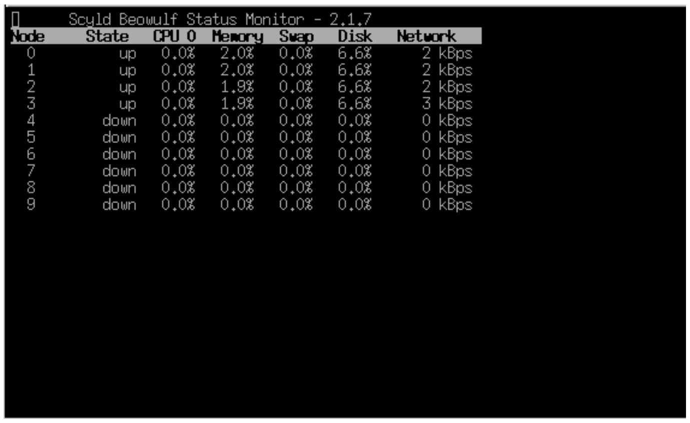


Node	Up	Available	CPU 0	CPU 1	Memory	Swap	Disk	Network
-1	✓	✓	7/100% (7%)	14/100% (14%)	125/128MB (98%)	25/259MB (10%)	79/960MB (8%)	34 kBps
0	✓	✓	0/100% (0%)	0/100% (0%)	23/65MB (36%)	None	9/36MB (25%)	33 kBps
1	✓	✓	0/100% (0%)	0/100% (0%)	35/65MB (54%)	None	9/36MB (25%)	33 kBps
2	✓	✓	0/100% (0%)	0/100% (0%)	27/65MB (42%)	None	9/36MB (25%)	33 kBps
3	✓	✓	0/100% (0%)	0/100% (0%)	26/65MB (40%)	None	9/36MB (25%)	33 kBps
4	✗	✗	0/100% (0%)	N/A	0 %	None	0 %	0 kBps
5	✓	✓	0/100% (0%)	N/A	34/496MB (7%)	None	9/36MB (25%)	34 kBps
6	✓	✓	1/100% (1%)	N/A	34/496MB (7%)	None	9/36MB (25%)	67 kBps
7	✓	✓	1/100% (1%)	N/A	34/496MB (7%)	None	9/36MB (25%)	65 kBps
8	✓	✓	1/100% (1%)	N/A	34/496MB (7%)	None	9/36MB (25%)	33 kBps
9	✓	✓	0/100% (0%)	N/A	34/496MB (7%)	None	9/36MB (25%)	34 kBps
10	✓	✓	1/100% (1%)	N/A	34/496MB (7%)	None	9/36MB (25%)	72 kBps
11	✓	✓	0/100% (0%)	N/A	34/496MB (7%)	None	9/36MB (25%)	34 kBps
12	✓	✓	0/100% (0%)	N/A	34/496MB (7%)	None	9/36MB (25%)	22 kBps
13	✓	✓	0/100% (0%)	N/A	34/496MB (7%)	None	9/36MB (25%)	33 kBps

Figure 2-1. Screenshot of Beostatus in Gnome/X Mode

The -u flag can be used to change the update interval that **beostatus** uses. The default interval is four seconds. Keep in mind when using this option that each update produces additional loading on the master node, each compute node, and the interconnection network, and if updates are too frequent they can affect the overall system performance.

Beostatus can also be used in settings where an X-windows tool is undesirable or impractical, such as accessing your beowulf master node through a slow remote network connection. The -c flag causes beostatus to display the same information, but to display it using the curses mode text output in the window in which the command is run. Curses allows a program to have precise control of the position of the cursor in a text-only window. The values displayed in the window will be updated just as they would with the graphical display.



```
        Scyld Beowulf Status Monitor - 2.1.7
Node      State   CPU 0  Memory  Swap   Disk    Network
  0          up   0.0%   2.0%   0.0%   6.6%     2 kBps
  1          up   0.0%   2.0%   0.0%   6.6%     2 kBps
  2          up   0.0%   1.9%   0.0%   6.6%     2 kBps
  3          up   0.0%   1.9%   0.0%   6.6%     3 kBps
  4        down   0.0%   0.0%   0.0%   0.0%     0 kBps
  5        down   0.0%   0.0%   0.0%   0.0%     0 kBps
  6        down   0.0%   0.0%   0.0%   0.0%     0 kBps
  7        down   0.0%   0.0%   0.0%   0.0%     0 kBps
  8        down   0.0%   0.0%   0.0%   0.0%     0 kBps
  9        down   0.0%   0.0%   0.0%   0.0%     0 kBps
```

Figure 2-2. Beostatus in Curses Mode

bpstat Command Line Tool

The **bpstat** command can be used to show a snapshot of the cluster, but doesn't continually update over time. When run without any arguments, it prints out a list of all the nodes in the cluster including some information about them that includes its current status. A node has to be in the 'up' state before you can do anything with it.

```
[user@cluster root]$ bpstat
Node(s)                        Status       Mode       User       Group
4-9                            down         ---------- root       root
0-3                            up           ---x--x--x root       root
```

beostat Command Line Tool

If you wish to see the raw data for the status on the compute nodes, you can use the **beostat** command. The use of this command is not detailed in this guide, however a list of all its options can be found in the *Reference Guide*.

Chapter 3. Running Programs

In this section, we'll look at how to run serial and parallel jobs on your Beowulf cluster, and how to monitor the status of the cluster once your applications are running.

First we will see how to manually run a simple non-cluster-aware program on a single computational node. Then we will continue with the concept of mapping multi-process jobs on to the cluster computational nodes. We will also look at how to run non-cluster aware program across multiple nodes, an MPI program, a PVM program, and other types of cluster aware programs.

Introduction

You may be wondering what the difference is between executing a program on a Scyld Beowulf cluster and executing a program on a stand-alone computer. First lets review how things work on a stand-alone computer, then we'll talk about how the Scyld Beowulf cluster works.

Program Execution Description on a Stand-alone Computer

On a stand-alone computer running Linux, Unix, and most other operating systems, executing a program is a very simple process. For example, to list the files on the present working directory, you type **ls** followed by <return>. This will provide on your screen a list of all files and sub-directories contained in the directory you were originally in when you typed **ls**.

So what actually happened? First off, there is a program running, called a shell, which prints the prompt and interprets your commands. When you type **ls** and hit <return>, the shell reads the command **ls** and executes the program stored as `/bin/ls`. The program **ls** collects and prints the list of files in the current directory to the standard output stream which happens to be your screen.

What's Different About Program Execution on a Scyld Beowulf Cluster?

So what is program execution like on a Scyld Beowulf cluster? A Scyld cluster is different than simply a group of stand-alone machines connected by a network. In the Scyld Beowulf configuration, only the Master node is similar to a fully installed system you are traditionally used to. The compute nodes have only the minimal system necessary to support an application initiated from a master node.

On the master node of the cluster, since it contains a full installation, **ls** works exactly as it would on any other stand alone computer system. When you type **ls** on the master node of the cluster, the **ls** program gets executed and returns the files as contained on the master node just like when you would execute **ls** on any other UNIX system.

On the compute nodes program execution is very different. Remember that on a Scyld Beowulf there are no resident applications on the compute nodes of the cluster. They only reside on the Master Node. So how do you run programs on a compute node? On a Scyld cluster processes are migrated at execution time from the Master Node to the Compute nodes. Take **ls** for example. To execute **ls** on Compute Node number 1, you simply type **bpsh 1 ls** followed by <return>. The result of **ls** shows up on your screen on the master node. How this works is via Scyld's **BProc** software. A full description of **bproc** and **bpsh** and how they function are covered both in this chapter and in more detail in the *Administrator's Guide*.

Traditional Beowulf Cluster - Description

Remember, a job on a Beowulf cluster is actually a collection of processes running on the compute nodes. In traditional clusters of computers, and even on earlier Beowulf clusters, getting all these processes started and running together was a complicated task. Starting a job consisted of each of the following tasks:

- Ensure that the user has an account on all the target nodes, either manually or via script
- Ensure that the user can spawn jobs on all the target nodes. This typically entails configuring a ‘hosts.allow‘ file on each machine, creating a specialized PAM module (a Linux authentication mechanism), or by creating a server daemon on each node to spawn jobs on the user’s behalf.
- Get a copy of the program binary to each node, either manually, with a script, or through a network file system.
- Ensure that each node has available identical copies of all libraries needed to run the program.
- Provide knowledge of the state of the system to the application manually, through a configuration file, or through some add-on scheduling software.

Scyld Beowulf Cluster - Description

With a Scyld Beowulf cluster, most of these steps are removed. Jobs are started on the master node and are migrated out to the compute nodes using BProc. By using a cluster architecture where jobs may only be initiated from the master node using BProc:

- Users no longer need accounts on remote nodes.
- Users no longer need authorization to spawn jobs on remote nodes.
- Neither binaries nor libraries need to be available on the remote system.
- The BProc system provides a consistent view of all jobs running on the system.

With all these complications removed, it simply becomes a matter of letting BProc know about your job when you start it, and the other requirements associated with running an application on a cluster as described go away!

There are two ways to indicate to BProc that you are about to launch a job that should execute on the node processors. One way deals specifically with launching parallel programs (for example, MPI jobs), and the other way deals with launching any other kind of program. Both methods are covered in this chapter.

Executing Programs That Aren’t Parallelized

Starting and Migrating Programs to Compute Nodes (bpsh)

There are no executable programs on the filesystem of the computational nodes. Thus, there’s no getty, login or shell on the compute nodes by default. By now you are probably wondering how does one actually run a program on a computation node?

Hopefully you are familiar with the commands **rsh** or **ssh**. Both of these commands allow you to run a program on a remote machine and in the absence of any command it will start a shell program on a remote machine. While neither of these will work on Scyld Beowulf (since there are no binaries on the compute nodes) there exists a similar command called **bpsh**. **bpsh** allows you to run an arbitrary program on a compute node. Here’s an example:

```
> bpsh 2 ls -FC /
dev/ etc/ home/ lib/ lost+found/ proc/ sbin/ scratch/ tmp/ usr/
```

Here we see the standard **ls** command running on node 2. However, you can see from the output that there is not a `/bin` directory. So how did **ls** run exactly? Well, it actually started running on the master node executed up to right before the first line of *main()* and then memory mapped the entire program out to node 2. It then completed running on node 2 forwarding the output back to the master node.

This was not a special version of **ls**. This will work with any program. All three standard I/O streams will by default get forwarded back to the master. Often programs will stop working if you try to run them in the background as they start thinking they might need to read standard input. Hence when you plan on running a program in the background on a compute node you should use the **-n** flag which will close standard input at invocation.

One thing to understand with the default compute node setup is that it is not possible to run shell scripts on the compute nodes. While it's perfectly possible to run a shell like **/bin/bash** on the compute nodes, none of the expected executables will be found on the compute nodes. It is possible to copy the shell utilities (like **ls**) over to the compute node. However, it is not recommended. Shell scripts should be run on the master and modified to include any **bpsh** commands required to affect the nodes.

Copying information to the Compute Nodes for Your Program (bpcp)

You may need to get data to and from the computational nodes. The **bpcp** command has the same syntax as **rcp** or **scp**, and is similar to the plain UNIX **cp**.

Before we get to some examples we should note that by default the `/home` directories of a user are NFS exported to each of the nodes. This is an easy way to be able to read *small* configuration files on the compute nodes. However, the NFS serving of the master has limited capacity, so don't try to read large files on multiple nodes using NFS from the server. At the very least, it will turn the network and the NFS server of the master node into a major bottleneck. It it also possible to overload the NFS server to the point that it will shutdown its NFS services. (This normally shouldn't happen, but you can't say we didn't warn you.)

Here are some examples of its usage. Let's copy a data file called `f001.dat` from the current directory to the `/tmp` directory on node 6.

```
> bpcp f001.dat 6:/tmp
```

The default directory on the compute node is the current directory on the master. Be aware that the current directory on the compute node may either not exist or already be NFS mounted from the master.

```
> cd /tmp
> bpcp f002.dat 2:
```

Here we have copied the file `/tmp/f002.dat` to the `/tmp` directory on node 2.

You can copy directly from node to node without any data passing between the nodes.

```
> bpcp 2:/tmp/f002.dat 3:/tmp
```

With this command we copied a file directly between node 2 and node 3. The contents of the file never passed though the master during this transaction.

An Introduction to Parallel Programming APIs

What does it mean to run in parallel? Most programmers are familiar with sequential programs. Simple programs like "Hello World" and basic sorting programs are typically sequential programs. That is, the program has a beginning, an execution sequence, and an end. At any time while it is running, it is only executing at a single point. A thread is similar to a sequential program in that is also has a beginning, a sequence, and an end. Also, at any time while it is running, there is a single point of execution. The big difference is that a thread is not a stand-alone program; it runs within a program. The concept of threads becomes important when a program has multiple threads running at the same time and performing different tasks. To run in parallel means that more than one thread of execution is running at the same time often on different processors or in the case of a cluster, different computers. There are a few things required to make this work and be useful. First, somehow the program has to get to the different computers and get started. Second, at some point the data has to be exchanged between the processes.

In the simplest case, you run the same single process program with different input parameters on all the nodes and gather the results at the end of the run. This idea of using a cluster to get faster results of the same non-parallel program with different input is called *parametric* execution.

However, you can imagine a much more complicated example. Say you are running a simulation, where each process represents some number of elements in the system. Every few time steps all the elements need to exchange data across boundaries to synchronize the simulation. This is where we start to see the need for a *message passing interface*.

To solve these two problems of program startup and message passing, you can develop your own code using POSIX interfaces or you could depend on an existing parallel application programming interface (API) for solving these issues. Scyld recommends using a standard parallel API called MPI (Message Passing Interface).

MPI - A Brief Description

MPI (Message Passing Interface) is currently the most popular API for writing parallel programs. The MPI standard doesn't specify many of the details of exactly how it should be implemented. This is very nice for system vendors (like Scyld) as they can change the way MPI programs run without adversely affecting the output of the program. The Scyld Beowulf product includes MPICH - a freely available implementation of the MPI standard. MPICH is a development project managed by Argonne National Laboratory and Mississippi State University (visit *the MPICH web site*[1] for more information).

With MPI, the same program is automatically started a number of times and is allowed to ask two questions. How many of us (size) are there, and which one am I (rank)? Then, a big set of conditionals are evaluated to determine what each process is going to do. You are also allowed to send and receive messages from other processes.

The advantages of MPI is that the programmer:

- doesn't have to worry about how the program gets started on all the machines.
- has a simplified interface to deal with interprocess messages.
- doesn't have to worry about mapping processes to nodes.
- abstracts out the details of the network so it can be implemented over very different kinds of hardware allowing your code to be more portable.

PVM - A Brief Description

Parallel Virtual Machine (PVM) was an earlier parallel programming interface. It was not a specification like MPI but a single set of source code distributed on the Internet. PVM reveals much more about the details of starting your job on remote nodes. That is, it fails to abstract out those implementation details as well as MPI does. It's also considered deprecated by most, but there are existing applications written for it. In general, we advise against writing new programs in PVM, but there may be existing application written in PVM you need to run. Also in particular cases, some of the unique features of PVM, may suggest its use.

Others

As mentioned earlier, one can develop their own parallel API by using various UNIX and TCP/IP standards. Just in terms of starting a remote program, there are programs written:

- using the rexec function call.
- to use the rexec or rsh program to invoke a sub-program.
- to use Remote Procedure Call (RPC).
- to invoke another sub-program using the inetd super server.

The problem with all of these is that there are a lot of implementation details to deal with. What are the network addresses? What is the path to the program? What is the account name on each of the computers? How is one going to load balance the cluster. Since Scyld Beowulf is a bit different and doesn't have binaries on all the nodes, your mileage may vary using these methods.

Briefly, we can say that Scyld has some experience with getting **rexec()** calls to work and one can simply substitute **bpsh** for **rsh** for programs that use **rsh**, but that we recommend you write your parallel code in MPI.

Mapping Jobs - How do Jobs get "Mapped" to the Compute Nodes?

To run programs specifically designed to execute in parallel across a cluster, things are a slightly more complex. When one wants to run a program across multiple machines, a little more information is required. The minimal information is: How many nodes do you want to run on?

The easy way to do this is by setting the environment variable **NP**. Here's an example:

```
> NP=4 ./a.out
```

This will run the MPI program a.out, which is located in the current directory, with four processes. What isn't specified is where these processes will run. That's the job of the *mapper*.

We divide the job of scheduling between what we call *Mapping* and *Batching* (also known as *Queuing*). Mapping is the act of deciding which node each process will run on. While it seems awfully simple, as various requirements are added, it can get complex. The mapper scans available resources at job submission time to decide which processors to run on. *Batching* is the queuing of jobs until the mapped resources become available. Scyld Beowulf includes a mapping API documented in the Programmer's Guide that describes how to write your own mapper.

The default mapper's behavior can be controlled by setting the following environment variables:

- `NP` - The number of processes requested. Not the number of processors. As in the example above, NP=4 ./a.out will run the MPI program a.out with four processes.
- `ALL_CPUS` - Set the number of processes to the number of CPUs available to the current user. If in the example above, ALL_CPUS=1 ./a.out would run the MPI program a.out on all available CPUs.
- `ALL_LOCAL` - Run every process on the master node. (For debugging purposes.)

- `NO_LOCAL` - Don't run any processes on the master node.
- `EXCLUDE` - A colon delimited list of nodes that should be avoided during the node assignment.
- `BEOWULF_JOB_MAP` - A colon delimited list of nodes. The first node listed will be the first process (MPI Rank 0) and so on.

There is also a small program called **beomap** that will give you a peek at what the mapping would have been for a job run right now by the current user with the current environment. It's quite useful in shell scripts or to learn about how these environment variables work.

Here are some more examples:

```
> NP=4 beomap
-1:1:2:3
> NO_LOCAL=1 NP=4 beomap
0:1:2:3
```

Running Serial Programs in Parallel (mpprun and beorun)

For jobs that are not "MPI-aware" or "PVM-aware", but need to be started in parallel, the utilities *mpprun* and *beorun* are provided. More sophisticated than *bpsh*, *mpprun* and *beorun* can automatically select ranges of nodes on which to start your program. They can also run tasks on the master node, and can determine the number of CPUs on a node and start a copy on each CPU. *mpprun* and *beorun* are very similar, and have similar parameters, but differ in that *mpprun* runs programs sequentially on the selected prcessors, while *beorun* runs programs concurrently on the selected processors.

mpprun

Mpprun is intended for applications rather than utilities and runs them sequentially on the selected nodes. The basic syntax of how to use *mpprun* is as follows:

```
mpprun [options] prog-name arg1 arg2 ....
```

The *prog-name* is the program you wish to run; it need not to be a parallel program. The *arg* arguments are the arguments that should be passed to each copy of the program you are running.

The complete list of options that can be passed to *mpprun* is documented in the *Reference Guide*. Options exist to control the number of processors to start copies of the program on, to start one copy on each node in the cluster, to start one copy on each CPU in the cluster, to force all jobs to run on the master node, or to prevent any jobs from running on the master node. The most interesting of the options is the *--map* or *--beowulf_job_map* option. The map option allows you to specify the specific nodes on which copies of a program should be run as a colon separated list. This argument, if specified, is used by the mapping software to override the optimally selected resources that it would otherwise use.

Some examples of using *mpprun* are shown below:

```
mpprun -np 16 app infile outfile
```

The command above runs 16 tasks of program app.

```
mpprun -np 16 --exclude 2:3 app infile outfile
```

Runs 16 tasks of program app on any available nodes except nodes 2 and 3.

```
mpprun --beowulf_job_map 4:2:1:5 app infile outfile
```

Runs 4 tasks of program app with task 0 on node 4, task 1 on node 2, task 2 on node 1 and task 3 on node 5.

beorun

Beorun is intended to run applications rather than utilities and runs them concurrently on the selected nodes. The basic syntax of how to use *beorun* is as follows:

```
beorun [options] prog-name arg1 arg2 ....
```

The *prog-name* is the program you wish to run; it need not to be a parallel program. The *arg* arguments are the arguments that should be passed to each copy of the program you are running.

The complete list of options that can be passed to *beorun* is documented in the *Reference Guide*. Options exist to control the number of processors to start copies of the program on, to start one copy on each node in the cluster, to start one copy on each CPU in the cluster, to force all jobs to run on the master node, or to prevent any jobs from running on the master node. The most interesting of the options is the *--map* or *--beowulf_job_map* option. The map option allows you to specify the specific nodes on which copies of a program should be run as a colon separated list. This argument, if specified, is used by the mapping software to override the optimally selected resources that it would otherwise use.

Some examples of using *beorun* are shown below:

```
beorun -np 16 app infile outfile
```

The command above runs 16 tasks of program app.

```
beorun -np 16 --exclude 2:3 app infile outfile
```

Runs 16 tasks of program app on any available nodes except nodes 2 and 3.

```
beorun --beowulf_job_map 4:2:1:5 app infile outfile
```

Runs 4 tasks of program app with task 0 on node 4, task 1 on node 2, task 2 on node 1 and task 3 on node 5.

Running MPI-Aware Programs

So what do we mean by MPI-aware programs? Programs that have been written to the MPI specification and linked with the Scyld MPICH library are MPI-aware programs.

Direct Execution

Let’s assume we have a compiled MPI program called **my-mpi-prog**. Let us also assume that the user is running the Bourne shell (**/bin/sh** or **/bin/bash**). Different shells have different semantics for setting environment variables. Details about how to build MPI programs can be found in the *Programmer’s Guide*. Let’s apply the environment variables we learned about in the mapping section to running **my-mpi-prog**. First let's run a four process job using the copy of **my-mpi-prog** that is in the current directory.

```
NP=4 ./my-mpi-prog
```

Another way of saying the same thing would be:

```
NP=4
export NP
./my-mpi-prog
```

Notice the user didn't have to say which node to run on or how to get the program there. That's taken care of by the mapper which always runs jobs on the nodes with the lowest CPU utilization. However, the user can affect where the program runs using some of the other environment variables. Here's another example:

```
NP=6 NO_LOCAL=1 EXCLUDE=2:4:5 ./my-mpi-prog
```

In this example we are starting six processes (NP=6), not running any of them on the master node (NO_LOCAL=1) or nodes 2, 4, or 5 (EXCLUDE=2:4:5).

mpirun

Almost all implementations of MPI have a **mpirun** program. So you may choose to start your MPI jobs using that command.

mpirun can be used exactly like *mpprun*, and supports all the same features as *mpprun* does. However, *mpirun* has some additional features available specifically targeted at MPI programs.

Using mpirun

All of the options available via environment variables through direct execution are available as flags to *mpirun*. For example, the *mpirun* command:

```
mpirun -np 16 mpiprog arg1 arg2
```

is equivalent to running the commands (in Bourne shell):

```
export NP=16
mpiprog arg1 arg2
```

as long as mpiprog is a properly compiled MPI job (see the *Programmer's Guide* for details on creating MPI programs).

Setting Mapping Parameters from Within a Program

A program can be designed to set all the required parameters itself. This option makes it possible to create programs in which the parallel execution is completely transparent. However, it should be noted that this will only work on Scyld Beowulf, while the rest of your MPI program should work on any MPI platform. Use of this feature differs from the in-line approach in that all options that need to be set on the command line can be set from within the program, and this feature may only be used with programs specifically designed to take advantage of it, rather than any arbitrary MPI program. However, this option makes it possible to produce turn-key application and parallel library functions in which the parallelism is completely hidden. More details in the use of this option are provided in the *Programmer's Guide*, but a brief example of the necessary program source code to invoke *mpirun* with the **-np 16** option from within a program is shown below.

Example 3-1. MPI Programming Example

```
/* Standard MPI include file */
#include <mpi.h>
/* Scyld mpirun include file */
#include <mpirun.h>
main(int argc, char **argv) {
        setenv("NP","16",1); // set up mpirun env vars
        MPI_Init(&argc,&argv);
        MPI_Finalize();
}
```

Running with Myrinet

The Scyld MPI library has Ethernet support built-in. If your application was compiled with this library and you now want to run under Myrinet, then there are at least two ways to fix this problem. First, you can modify your environment so your application will use the Myrinet version of the MPI library. To do this, simply type the following command before running your program:

```
[user1@cluster]$ LD_PRELOAD=/usr/lib/libgmpi.so.1
```

Alternatively, you can have your system administrator replace the Ethernet version of the MPI library with the Myrinet version.

Running PVM-Aware Programs

PVM applications are programs that have been written to use the *Parallel Virtual Machine* software system, an *application programming interface* (API) for writing parallel applications. This software system, designed to enable a collection of heterogeneous computers to be used as a coherent and flexible concurrent computational resource, has been specifically tailored to take advantage of the technologies used in a Scyld Beowulf. Therefore, a PVM-aware program is one that has been written to the PVM specification and linked against the Scyld PVM library. A complete discussion of how to configure a cluster to use PVM is beyond the scope of this document, however a brief introduction is provided below (with the assumption that the reader has some background knowledge on using PVM).

The master PVM daemon can be started on the master node using the PVM console: **pvm**. You can then use the console's **add** command to add compute nodes to your virtual machine. The syntax for the command is as follows: **add .#**, replacing the '#' character with the node's assigned number in the cluster (as listed by **beosetup** or **bpstat**. Alternately, you can start the PVM console by specifying a hostfile name on the command line. This file should contain the .# names of the compute nodes you want as part of the virtual machine. As with standard PVM, this method automatically spawns PVM slave daemons to the appropriate compute nodes in the cluster. From within the PVM console, use the **conf** command to list your virtual machine's configuration. You should see a separate line for each node being used. Once your virtual machine has been configured, you simply run your PVM applications like you normally would.

Other Programs that are Parallelized But Do Not Use MPI or PVM

Programs written for use on other types of clusters may involve various levels of change to function on Scyld Beowulf. Scripts or programs that invoke **rsh** and/or **rcp** can instead call **bpsh** and **bpcp**, respectively. Also, **beomap** may be used by any script to load balance programs that are to be dispatched to the compute nodes. Details, on how to port applications to Scyld Beowulf may be found in the *Programmer's Guide*.

Batching Jobs

The Beowulf Batch Queuing (bbq) Package

The **bbq** package provides a simple job batch spooler. It is based on the standard **at** package and has been enhanced to handle the Scyld cluster environment. It is installed by default during the normal installation procedure. **bbq** generates a list of requested processor resources based on the same variables that **mpirun**, and **mpprun** use, and releases submitted jobs as the requested resources become available to keep the cluster at a predetermined level of utilization. This 'level of utilization' will be referred to as the *load average* in the remainder of this discussion.

With **bbq**, users can schedule programs to run at specific times or just submit programs to run as cluster processor resources are available. The **bbq** package consists of three programs, the daemon **atd**, a job submission tool **at**, and a queue viewer **bbq**. The Scyld enhancements are supported only on the *b* queue. Other standard **at** queues are available, but only the *b* queue has knowledge of the cluster resources. As a convenience, the command **batch** has been aliased to **at -q b** which uses the *b* queue for job submission.

User job output will be captured as text and sent in an email to the user. Both standard out and standard error are captured and sent to username@localhost. To forward the mail to your regular email account, create a *.forward* text file in your home directory on the master node containing the full email address of where you wish to send the mail. Note that you can have multiple entries in this file, and mail will be sent to each of them.

Submitting Jobs

Users may submit jobs by typing at the **batch** prompt or redirecting a file on the batch command line. For the interactive batch prompt, simply type **batch time/date**, the prompt *at>* will then be displayed. Type each command at the prompt followed by a carriage return, then terminate the sequence with a *^d* character. **batch** will then parse any time and date parameters, display the time to execute the sequence, and then exit. To use an input file, simply create a text file with the commands you wish to execute and use the **-f** option or the < pipe symbol. For example, **batch noon tomorrow -f jobfile** , or **batch noon tomorrow < myfile** . Note, that there is no functional difference between the two methods.

When entering commands interactively or through a file, it is recommended that the first line be a comment identifying the job. Comments are started with the # character, and are not processed. **bbq** displays the first 38 characters of user input, and so will display the comment and following characters up to the 38 character limit. This should assist in identifying the job later.

The time specification used by **bbq** allows for a wide range of string types. Users can specify time in friendly terms like **now**, **noon**, **midnight**, **today**, and do simple arithmetic expressions such as **now + 3hours**, **4pm + 2days**. Some examples, **batch 06:32 2001-06-04**, same as **batch 6:32am jun 4 2001**, **batch noon sun**, **batch tomorrow**. When only the time is specified and the time is greater than the current time, a date of today is assumed. When the time is in the past, tomorrow is assumed. When only the date is specified the time is assumed to be the current time. When both the time and a date are given the time must be specified first. Any unrecognized formats or expressions in the time specification will be answered with a *Garbled time* error message.

When submitting jobs, users should be aware of how their environment is set and not mix programs with different resource requests in one job. For example, a job with the following lines,

```
NP=3
myprogram1
```

```
NP=4
myprogram2
```

would confuse the *bash* shell as **bbq** would place both **NP** strings in the environment. *bash* uses the first string and ignores the second. Likewise, if the user has **NP** already in the environment before the **batch** command is run, the first NP is used. To remove any confusion, users should not have any MPI variables set in their default environment, but instead set them in the job file, or as part of the **mpirun** command line. You can use the shell **unset** *variablename* command to remove any existing ones. See the next section for a list of all MPI environment variables.

Job Processing

Once a job is submitted, a text file will be created in the */var/spool/at* directory containing the user's name, group, environment, home directory, and the command sequence. The time to execute and the queue name will be encoded in the file name. **bbq** relies on the standard **at** processing to determine the execution time for each job. When that time arrives, **bbq** uses the process request mapped against the available processor resources to determine whether or not to run the job. Jobs will pend in the queue until each of the processors requested falls below the load average selected by the cluster administrator. The load average value is set to 80% CPU utilization by default.

To determine the processor request, **bbq** calls the **beomap** function described in the above sections, with all of the MPI parameters and the user's environment. **beomap** then returns a list of nodes to use. Jobs are released in a first come first serve basis, there are currently no ways to assign priorities. If one job requests all of the cluster's processors, that job will pend in the queue until the load of all the cluster processors falls below the threshold. No other jobs will be released while that job is pending.

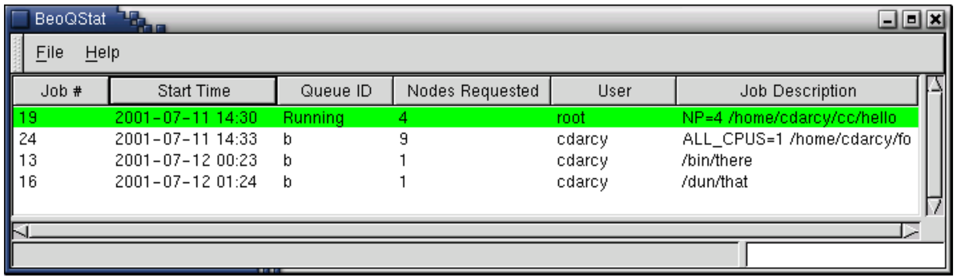
Note, that the terms process and processor have been used interchangeably, even though this is technically incorrect. Normally, **bbq** will use one processor for each primary process the user requests, as this maximizes the job throughput. Only when **bbq** is constrained otherwise, will it fall back to multiple processes per processor. These constraints are reached when the user selects a limited processor list or the cluster administrator has limited this user's access to the full cluster. For example, the user sets NP=32, but only has permission to run on 16 uniprocessor nodes. **bbq** will map 2 processes per processor. With the same example but with dual processor nodes, **bbq** will map 1 process per processor.

Currently, **mpprun** arguments for processor requests are not supported. It is expected that **mpprun** will be fully supported in the next release. You may use the environment variable equivalent for the mpprun argument until support is added.

```
 -all-local   ALL_LOCAL=1
 -all-cpus    ALL_CPUS=1
 -allcpus     ALL_CPUS=1
 -np x        NP=x
 -nolocal     NO_LOCAL=1
 -map         BEOWULF_JOB_MAP=x1:x2...
 -exclude     EXCLUDE=x1:x2...
```

Queue Management

Users have a few options when viewing the queue. The command **bbq** lists the pending and running batch jobs to standard out. There is also a GUI display, **beoqstat**, that constantly monitors the queue and updates its display every 5 seconds. This GUI also allows jobs to be sorted and deleted. In addition, whenever **bbq** detects the environment variable *GATEWAY_INTERFACE*, **bbq** will generate HTML to standard out. This environment variable will only be set when **bbq** is called as a CGI script via an APACHE web server. Details on these options and the available queue sorting options are available in the *Reference Guide*.



Job #	Start Time	Queue ID	Nodes Requested	User	Job Description
19	2001-07-11 14:30	Running	4	root	NP=4 /home/cdarcy/cc/hello
24	2001-07-11 14:33	b	9	cdarcy	ALL_CPUS=1 /home/cdarcy/fo
13	2001-07-12 00:23	b	1	cdarcy	/bin/there
16	2001-07-12 01:24	b	1	cdarcy	/dun/that

Figure 3-1. Beoqstat - The GUI BBQ Monitor

Although any user can view all the jobs, only the job owner or root can remove a job from the queue. To do this, simply type **atrm** with the job number. Both pending and running jobs may be removed, but the reader should note that **bbq** does not have knowledge of all the individual processes that an application has started. To effectively remove a running job the user should delete the job from the queue, and then kill each of the associated processes. To assist in maintaining high cluster loading and accounting, the cluster administrator may require all jobs to use the job queuing system. This creates a single job entry point where the administrator could gather job statistics, including user name, processor resource requests, and start/end times via a simple script. To setup, the administrator sets the group on each compute node to **daemon** and restarts the **atd** daemon with **/usr/bin/atd -r**. The **-r** options tells the **atd** daemon to start all jobs with a group id of daemon, overriding the user's normal group allowing the job to access the cluster compute nodes.

PBS

PBS originally stood for Portable Batch Scheduler. The POSIX committee adopted it as a standard thus it also stands for POSIX Batch Scheduler. The fundamental problem with PBS is that its concept of a job is defined to be a shell script. As we said earlier, in the default configuration Scyld nodes do run shell scripts on the compute nodes. So PBS and other schedulers such as NQS will work fine on the master node, but can not schedule jobs on the compute nodes unless the master configuration is modified. PBS-Pro provides a version of PBS that works on a Scyld Beowulf cluster and takes advantage of Scyld's BProc technology. Various Scyld users have used traditional schedulers like PBS in a cluster of clusters environment by dispatching jobs to the master node of each sub-cluster. The system specific scheduler of each master node can then map the jobs to its nodes. However, it is not needed if you have just one or a few clusters at your facility.

File Systems

File System Options

As you run programs on your cluster, you will often have to deal with where to put the files that those programs use or create. Because your cluster nodes each have separate disk drives, there are a number of options for where and how you store your files. The possibilities are:

- Store files on the local disk of each node
- Store files on the master node's disk, and share these files with the nodes through a network filesystem
- Store files on disks on multiple nodes and share them with all nodes through the use of a parallel filesystem

Each of these approaches has advantages and disadvantages. The simplest approach is to store all files on the master node. Scyld supports this approach through the use of the standard network filesystem NFS. By default, any files in the `/home` directory are shared via NFS with all the nodes in your cluster. This makes management of the files very simple, but in larger clusters the performance of NFS on the master node can become a bottleneck for I/O intensive applications.

Storing files on the local disk of each node removes the performance problem, but makes it difficult to share data between tasks on different nodes. Input files for programs must be distributed manually to each of the nodes, and output files from the nodes must be manually collected back on the master. However, this mode of operation can still be useful for temporary files created by a process and then later reused on that same node.

An alternate solution is to use a parallel filesystem. A parallel filesystem provides an interface much like a network filesystem, but distributes files across disks on more than one node. Scyld provides a version of PVFS, the Parallel Virtual Filesystem, which is described in more detail in the next section.

PVFS

The Parallel Virtual File System (PVFS) is a parallel file system. It allows applications, both serial and parallel, to store and retrieve data which is distributed across a set of I/O servers. This is done through traditional file I/O semantics, which is to say that you can open, close, read, write, and seek in PVFS files just as in files stored in your home directory.

The primary goal of PVFS is to provide a high performance “global scratch space” for Beowulf clusters running parallel applications. If PVFS is installed and configured on your cluster, it will “stripe” files across the disks of the nodes in your cluster, allowing you to read and write files faster than you could to a single disk.

Within your cluster, there are three different roles that a given node might play:

- metadata server
- I/O server
- client

Any node can fill one or more of these roles.

The metadata server, of which there is one per PVFS file system, maintains information on files and directories stored in a PVFS file system. This includes permissions, owners, and the locations of data. Clients contact the metadata server when they want to create, remove, open, or close files. They also read directories from the metadata server.

I/O servers, of which there may be many, store PVFS file data. Each one does so by creating files on local file systems mounted on the machine, such as an existing ext2fs partition. Clients contact these servers in order to store and retrieve PVFS file data.

Clients are the users of the PVFS system. Applications accessing PVFS files and directories run on client machines. There are PVFS system components which perform operations on behalf of these clients.

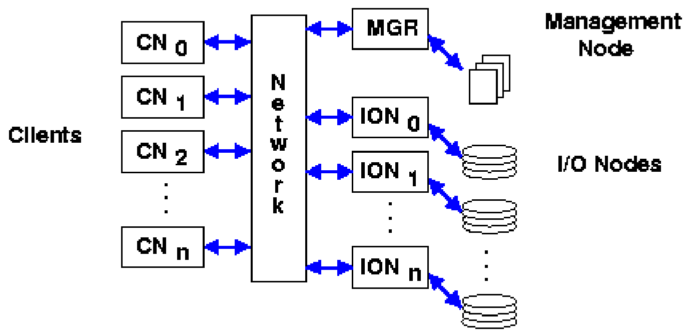


Figure 3-2. PVFS System Diagram

The figure above shows the PVFS system view, including a metadata server (mgr), a number of I/O servers (IONi) each with a local disk, and a set of clients. For our example system, we will set up the master node as the metadata server, the eight other nodes as both I/O servers and as clients (CNi). This would allow us to run parallel jobs accessing PVFS files from any node and striping these files across all the nodes except the head node.

To a user, a PVFS filesystem works much the same as any other filesystem. If a PVFS filesystem exists on your cluster, the system administrator will mount this filesystem somewhere in your directory tree, say /pvfs. Once mounted, the PVFS directory functions the same as any other directory, and normal UNIX utilities will work on it. You can cd into the directory, list the files in the directory with ls, and copy, move or delete files with cp, mv, or rm.

Copying Files to PVFS

If you use a standard UNIX command like cp to copy files into a PVFS directory, PVFS will provide a default striping of your data across the I/O servers for your file. However, in many instances, the user may wish to have control of the physical distribution of the file. The u2p command supplied with PVFS can be used to copy an existing UNIX file to a PVFS file system while specifying physical distribution parameters.

The current physical distribution mechanism used by PVFS is a simple striping scheme. The distribution of data is described with three parameters:

- base - the index of the starting I/O node, with 0 being the first node in the file system
- pcount - the number of I/O servers on which data will be stored (partitions, a bit of a misnomer)
- ssize - strip size, the size of the contiguous chunks stored on I/O servers

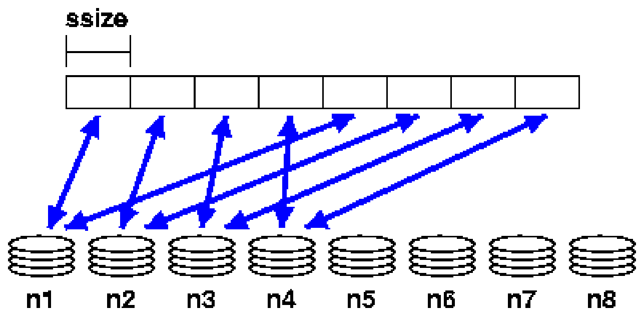


Figure 3-3. Striping Example

In the figure above we show an example where the base node is 0 and the pcount is 4 for a file stored on our example PVFS file system. As you can see, only four of the I/O servers will hold data for this file due to the striping parameters.

The syntax for u2p is:

u2p -s <stripe size> -b <base> -n <# of nodes> <srcfile> <destfile>

This function is most useful in converting pre-existing data files to PVFS so that they can be used in parallel programs.

Examining File Distributions

The pvstat utility will print out the physical distribution parameters for a PVFS file. For example, to examine a file named foo in our PVFS file system, we see:

[root@head /root]# /usr/local/bin/pvstat /pvfs/foo /pvfs/foo: base = 0, pcount = 8, ssize = 65536

which tells us that our file foo has a stripe size of 64k and is currently striped among 8 I/O servers beginning at server 0.

Checking on Server Status

The iod-ping utility can be used to determine if a given I/O server is up and running:

[root@head /root]# /usr/local/bin/iod-ping -h 1 -p 7000 1:7000 is responding. [root@head /root]# /usr/local/bin/iod-ping -h head -p 7000 head:7000 is down.

In this case, we have started the I/O server on node 1, so it is up and running. We are not running an I/O server on the head, so it is reported as down. Likewise the mgr-ping utility can be used to check the status of metadata servers:

[root@head /root]# /usr/local/bin/mgr-ping -h head -p 3000 head:3000 is responding. [root@head /root]# /usr/local/bin/mgr-ping -h 1 -p 3000 1:3000 is down.

The mgr is up and running on the head, but not running one on node 1.

These two utilities also set their exit values appropriately for use with scripts; in other words, they set their exit value to 0 on success (responding) and 1 on failure (down). Furthermore, specifying no additional parameters will cause the program to automatically check for a server on localhost at the default port for the server type (7000 for I/O server, 3000 for metadata server). Not specifying a “-p” option will use the default port.

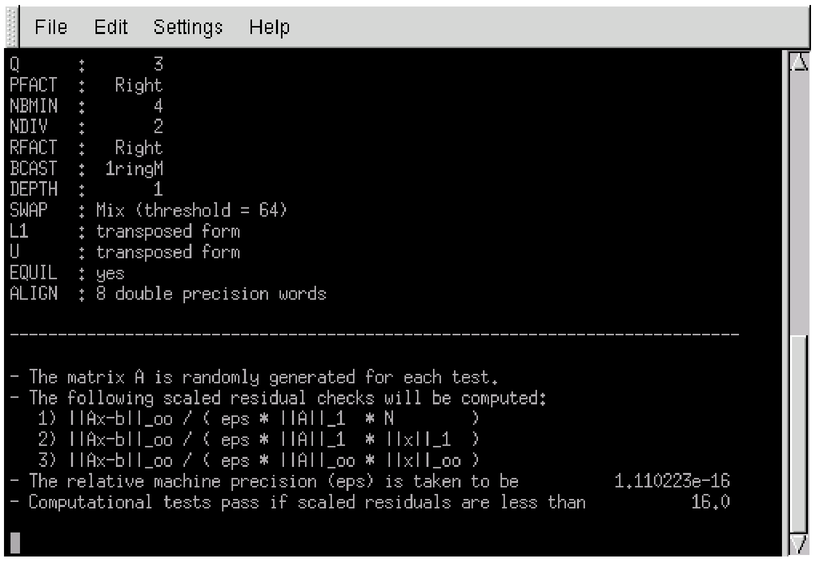
Sample Programs Included in the Distribution

Linpack

The Top 500 page[2] lists the fastest known computers in the world. The benchmark they use is named Linpack. The Linpack benchmark involves solving a random dense linear system. A version of this benchmark is available via a package called **hpl**.

Scyld Beowulf provides a small shell script called **linpack** that will automatically create a configuration/input file for **xhpl** and start it up. It is located in `/usr/bin`. This script is intended to work for everyone, thus the default dimensions are small and will not result in good performance on clusters larger than a few nodes. Feel free to read the script and customize performance for your cluster. The file `/usr/share/doc/hpl-1.0/TUNING` gives some ideas on how to optimize the input file for your cluster. The easy change to try is to increase the problem size that's currently set to `3000` at around line 15, but it will fail if you set it too high and the nodes run out of memory.

The really useful thing about **linpack** is that it will stress your cluster by maximizing your CPU and network usage. If it doesn't run to completion correctly, or takes too long to run, you probably have a network problem such as a bad switch or incorrect configuration.



```
Q      :       3
PFACT  :   Right
NBMIN  :       4
NDIV   :       2
RFACT  :   Right
BCAST  :  1ringM
DEPTH  :       1
SWAP   : Mix (threshold = 64)
L1     : transposed form
U      : transposed form
EQUIL  : yes
ALIGN  : 8 double precision words

----------------------------------------------------------------------------

- The matrix A is randomly generated for each test.
- The following scaled residual checks will be computed:
   1) ||Ax-b||_oo / ( eps * ||A||_1  * N        )
   2) ||Ax-b||_oo / ( eps * ||A||_1  * ||x||_1  )
   3) ||Ax-b||_oo / ( eps * ||A||_oo * ||x||_oo )
- The relative machine precision (eps) is taken to be          1.110223e-16
- Computational tests pass if scaled residuals are less than          16.0
```

Figure 3-4. Testing your cluster with linpack

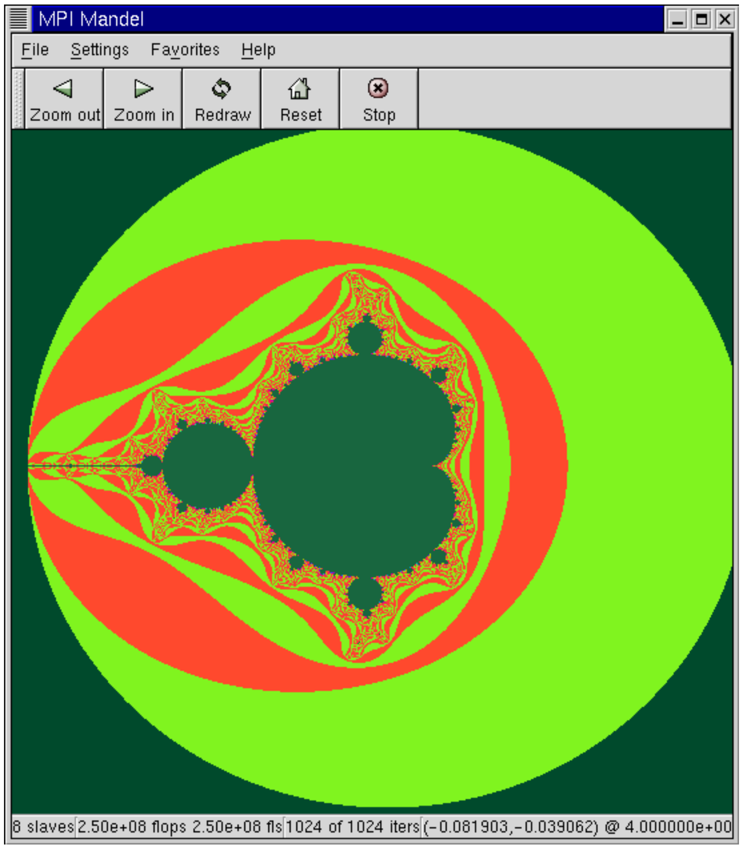
MPI-Mandel

mpi-mandel is a graphical interactive demonstration of the Mandelbrot set. Also, since **mpi-mandel** is a classic MPI program it gives us a chance to practice our mapping environment variables.

Here's a sample invocation:

```
prompt#  ALL_CPUS=1 mpi-mandel
```

This will run the program on all available CPUs. Notice the slave (compute node) count in the bottom left corner. Also, this program demonstrates the performance counter library. If you are running on x86 CPUs and a kernel with performance counter support, you'll see that the number of integer and floating point calculations is given on the bottom of the window. The **mpi-mandel** program is located in /usr/bin.



```
8 slaves 2.50e+08 flops 2.50e+08 fls 1024 of 1024 iters (-0.081903,-0.039062) @ 4.000000e+00
```

Figure 3-5. Demonstrating your cluster with MPI-Mandel

If you would like to have **mpi-mandel** run as a free flowing demo you can load a favorites file.

```
prompt# NP=4 mpi-mandel --demo /usr/share/doc/mpi-mandel-1.0.20a/mandel.fav
```

If you shut off the "delay" under the favorites menu you'll find that it will refresh much faster. However if your video card is not fast enough or you are running a very non-homogeneous cluster, you'll see only part of the screen refresh. In that case you'll probably want to leave the delay in.

Notes

1. http://www-unix.mcs.anl.gov/mpi/mpich/
2. http://www.top500.org/

Appendix A. Glossary of Parallel Computing Terms

Bandwidth

A measure of the total amount of information delivered by a network. This metric is typically expressed in Millions of bits per Second (Mbps) for data rate on the physical communication media or Megabytes per Second (MBps) for the performance seen by the application.

Backplane Bandwidth

The total amount of data that a switch can move through it in a given time. Typically much higher than the bandwidth delivered to a single node.

Bisection Bandwidth

The amount of data that can be delivered from one half of a network to the other half in a given time, through the least favorable halving of the network fabric.

Boot image

The filesystem and kernel seen by the machine at boot time; contains enough drivers and information to get the system up and running on the network

Cluster

A collection of nodes, usually dedicated to a single purpose

Compute node

Synonymous with slave node

Data Parallel

A style of programming in which multiple copies of a single program run on each node, performing the same instructions while operating on different data

Efficiency

The ratio of a programs actual speedup to its theoretical maximum.

FLOPS

Floating-point operations per second. A key measure of performance for many scientific and numerical applications

Grain size
Granularity

A measure of the amount of computation a node can perform in a given problem between communications with other nodes. Typically defined as "coarse" (large amount of computation) or "fine" (small amount of computation). Granularity is a key in determining the performance of a particular problem on a particular cluster

High Availability

Refers to level of reliability. Usually implies some level of fault tolerance (ability to operate in the presence of a hardware failure)

Hub

A device for connecting the NICs in an interconnection network. Only one pair of ports can be active at any time (a bus). Modern interconnections utilize switches, not hubs.

Isoefficiency

The ability of a problem to maintain a constant efficiency if the size of the problems scales with the size of the machine

Jobs

In traditional computing, a job is a single task. A parallel job can be a collection of tasks, all working on the same problem but running on different nodes

Kernel

The core of the operating system, the kernel is responsible for processing all system calls and managing the system's physical resources

LAM

The Local Area Multicomputer, a communication library available with MPI or PVM interfaces

Latency

The length of time from when a bit is sent across the network until the same bit is received. Can be measured for just the network hardware (wire latency) or application-application (includes software overhead)

Local area network (LAN)

An interconnection scheme designed for short physical distances and high bandwidth. Usually self-contained behind a single router

MAC address

On an Ethernet NIC, the hardware address of the card. MAC addresses are unique to the specific NIC, and are useful for identifying specific nodes

Master node

Node responsible for interacting with users; connected to both the public network and interconnection network; controls the slave nodes

Message Passing

Exchanging information between processes, frequently on separate nodes

Middleware

A layer of software between the user's application and the operating system

MPI

The Message Passing Interface, the standard for producing message passing libraries

MPICH

A commonly used MPI implementation, built on the chameleon communications layer

Network Interface Card (NIC)

The device through which a node connects to the interconnection network. The performance of the NIC and the network it attaches to limit the amount of communication which can be done by a parallel program

Node

Single computer system (motherboard, one or more processors, memory, possibly disk, network interface)

Parallel Programming

The art of writing programs which are capable of being executed on many processors simultaneously.

Process

An instance of a running program

Process Migration

Moving a process from one computer to another after the process begins execution

PVM

The Parallel Virtual Machine, a common message passing library that predates MPI

Scalability

The ability of a problem to maintain efficiency as the number of processors in the parallel machine increases

Single System Image

All nodes in the system see identical system files. Same kernel, libraries header files, etc, guaranteeing that a program which will run on one node will run on all nodes

Slave node

Nodes attached to master through interconnection network; used as dedicated attached processors. With Scyld, users should never need to directly login to slave nodes

Socket

A low-level construct for creating a connection between processes on remote system

Speedup

A measure of the improvement in the execution time of a program on a parallel computer vs. time on a serial computer

Switch

A device for connecting the NICs in an interconnection network. All pairs of ports can communicate simultaneously.

Version skew

The problem of having more than one version of software or files (kernel, tools, shared libraries, header files) on different nodes