# Table of Contents

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
</tr>
<tr>
<td>Feedback</td>
</tr>
<tr>
<td><strong>1. Scyld Beowulf Overview</strong></td>
</tr>
<tr>
<td>What is a Beowulf Cluster?</td>
</tr>
<tr>
<td>A Brief History of the Beowulf</td>
</tr>
<tr>
<td>First-Generation Beowulf Clusters</td>
</tr>
<tr>
<td>Scyld Beowulf</td>
</tr>
<tr>
<td>Top-level Features of Scyld Beowulf</td>
</tr>
<tr>
<td>Scyld Beowulf Technical Summary</td>
</tr>
<tr>
<td>Beowulf Process Space Migration Technology</td>
</tr>
<tr>
<td>Compute Node Boot Procedure</td>
</tr>
<tr>
<td>Compute Node Categories</td>
</tr>
<tr>
<td>Compute Node States</td>
</tr>
<tr>
<td>Major Software Components</td>
</tr>
<tr>
<td>Typical Applications of Scyld Beowulf</td>
</tr>
<tr>
<td><strong>2. Interacting with the System</strong></td>
</tr>
<tr>
<td>Verifying that the Cluster is Up and Running</td>
</tr>
<tr>
<td>Issuing Commands</td>
</tr>
<tr>
<td>head node</td>
</tr>
<tr>
<td>Compute Node</td>
</tr>
<tr>
<td>bpsh and Shell Interaction</td>
</tr>
<tr>
<td>Copying Data to Compute Nodes</td>
</tr>
<tr>
<td>Monitoring and Signaling Processes</td>
</tr>
<tr>
<td>top, ps, and kill</td>
</tr>
<tr>
<td>Monitoring Node Status</td>
</tr>
<tr>
<td>Beostatus GUI Tool</td>
</tr>
<tr>
<td>bpstat Command Line Tool</td>
</tr>
<tr>
<td>beostat Command Line Tool</td>
</tr>
<tr>
<td><strong>3. Running Programs</strong></td>
</tr>
<tr>
<td>Introduction</td>
</tr>
<tr>
<td>Program Execution Description on a Stand-alone Computer</td>
</tr>
<tr>
<td>What’s Different About Program Execution on a Scyld Beowulf Cluster?</td>
</tr>
<tr>
<td>Traditional Beowulf Cluster - Description</td>
</tr>
<tr>
<td>Scyld Beowulf Cluster - Description</td>
</tr>
<tr>
<td>Executing Programs That Aren’t Parallelized</td>
</tr>
<tr>
<td>Starting and Migrating Programs to Compute Nodes (bpsh)</td>
</tr>
<tr>
<td>Copying information to the Compute Nodes for Your Program (bpct)</td>
</tr>
<tr>
<td>An Introduction to Parallel Programming APIs</td>
</tr>
<tr>
<td>MPI - A Brief Description</td>
</tr>
<tr>
<td>PVM - A Brief Description</td>
</tr>
<tr>
<td>Others</td>
</tr>
<tr>
<td>Mapping Jobs - How do Jobs get &quot;Mapped&quot; to the Compute Nodes?</td>
</tr>
<tr>
<td>Running Serial Programs in Parallel (mpirun and beorun)</td>
</tr>
<tr>
<td>mpirun</td>
</tr>
<tr>
<td>beorun</td>
</tr>
<tr>
<td>Running MPI-Aware Programs</td>
</tr>
<tr>
<td>Direct Execution</td>
</tr>
</tbody>
</table>
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.

Also not covered is using the Linux operating system, on which the Scyld software is based. Some of the basics are presented here, but if you’ve not used Linux or Unix before a book or online resource will be helpful. A good source of information are books by O’Reilly and Associates.


This manual will provide you the information about 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 Series 30, published Feb 2006.

Notes

Chapter 1. Scyld Beowulf Overview

The Scyld Beowulf Cluster Operating System is a Linux-based high-performance computing system. 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 presents a high-level overview of the Scyld Beowulf Cluster Operating System, a history, and specifics of a Scyld Beowulf; essentially what a Beowulf cluster is, how it can be used, and 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 presented, with further details provided throughout the Scyld Beowulf Series 30.

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 off-the-shelf computers (COTS), each of which is 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 head node, is usually attached to both the private and public networks, and is the cluster’s administration console. The remaining nodes are commonly referred to as compute nodes or slave nodes. The head 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 head node. Typically these compute nodes require neither keyboards nor monitors; they are accessed solely through the head node. From the viewpoint of the cluster’s head node the compute nodes are simply additional processor and memory resources.

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 day, allowing users to execute high-performance computing applications, the original software implementations were not without their problems. The Scyld Beowulf Cluster Operating System addresses - and solves - many of these problems.

A Brief History of the Beowulf

Cluster computer architectures have a long history. The early network of workstations (NOW) architecture used a group of standalone processors connected through a typical office network, their idle cycles harnessed by a small piece of special software, as shown below.
Chapter 1. Scyld Beowulf Overview

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 shown below.

For a cluster to be properly termed a *Beowulf*, it must adhere to the *Beowulf philosophy*, which requires

1. scalable performance
2. the use of commodity off-the-shelf (COTS) hardware
3. the use of an open-source operating system, 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 distributions included all of the software needed for a networked workstation: the kernel, various 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 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 head 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 log into one of the compute nodes nor worry about which compute node is which. To the user, the head node is the computer, and the compute nodes appear merely as attached processors capable of providing computing resources. With a Scyld Beowulf the cluster appears to the user as a single computer. Specifically,

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

The Scyld Beowulf architecture simplifies cluster setup and node integration, requires minimal system administration (and provides tools for easy administration where necessary), increases cluster reliability through 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:
Chapter 1. Scyld Beowulf Overview

Easy Installation

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

Install Once, Execute Everywhere

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

Single System Image

Scyld Beowulf makes a cluster appear as a multi-processor parallel computer. The head 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 of the Scyld Beowulf Administrator’s Guide.

Execution Time Process Migration

Scyld Beowulf stores applications on the head node. At execution time processes are migrated from the head node to the compute nodes. This approach both virtually eliminates the risk of version skew and eliminates the requirement of hard disks for the compute nodes. See the section on the BProc: Beowulf Distributed Process Space in the System Design Description chapter of the Scyld Beowulf Administrator’s Guide.

Seamless Cluster Scalability

Scyld Beowulf seamlessly supports the dynamic addition and deletion of compute nodes without modification to existing source code or configuration files. See the BeoSetup chapter of 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, job execution, and monitoring of the cluster. See the Administrator’s Guide for more information.

Batch Queuing Tools

Scyld Beowulf includes a robust batch queuing and job-scheduling system. BBQ, the Scyld Beowulf Batch Queuing system, includes CLI, 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 view 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 both 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 libraries and utilities specifically improved for high-performance computing applications. 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 compute nodes to be visible and managed on the head node. All processes appear in the head node’s process table, from which they are migrated to the appropriate compute node by BProc. Both process parent-child relationships and Unix job-control information are maintained with migrated jobs. The stdout and stderr streams are redirected back to the head node across 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 of the Scyld Beowulf Administrator’s Guide.

Compute Node Boot Procedure

Scyld Beowulf compute nodes boot in two stages.

First is a local stage 1 boot image. This contains a minimal Linux kernel with just enough functionality to configure a reliable TCP/IP socket connection between the compute node and the head node.

Then the compute node attempts to communicate with the head node to obtain its required runtime files and to complete its initialization procedure. After the head 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 of the Scyld Beowulf Administrator’s Guide.

Compute Node Categories

Each compute node in the cluster is classified into one of three categories by the head node: unknown, ignored, or configured.

Unknown

A node is one not formally recognized by the cluster as being either a configured or 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 which, for one reason or another, you’d like the head node to ignore. These are not considered part of the cluster, nor will they receive a response from the head node during their boot process.

Configured

Nodes are those listed in the cluster configuration file using the node tag. These are formally part of the cluster, recognized as such by the head node, and used as computational resources by your Beowulf cluster. For more information on node categories, see the System Design Description chapter of the Scyld Beowulf Administrator’s Guide.
Chapter 1. Scyld Beowulf Overview

**Compute Node States**

For each of the configured nodes in the cluster, BProc maintains the current condition of the node. This information, known as the node's state, is always one of the following: 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 stage 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 of the Administrator’s Guide.

**Major Software Components**


**bproc**

The Beowulf process migration technology; an integral part of Scyld Beowulf.
beosetup
   A GUI for configuring the cluster.

beostatus
   A GUI 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 runs on the head 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 CLI 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.
Chapter 1. Scyld Beowulf Overview

beoqstat

The GUI BBQ tool; a GUI for viewing and deleting batch jobs.

Typical Applications of Scyld Beowulf

Beowulf clustering provides a facile solution for anyone executing jobs that involve either a large number of computations or large amounts of data (or both), such as image-rendering. Some of the special effects used in the movie *Titanic* were done using clusters. Beowulf clustering is ideal for both large, monolithic, parallel jobs and for 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, and computational drug development.

These types of jobs can be performed many times faster on a Scyld Beowulf cluster than on a single computer. Increased speed depends on the application code, the number of nodes in the cluster, and the type of equipment used in the cluster. All of these can be easily tailored and optimized to suit the needs of your applications.

The increased computational resources needs of modern applications are frequently being met by Beowulf clusters; some of these problem domains include:

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

The first priority when interacting with a cluster is ascertaining the availability of nodes. Unlike traditional Beowulf clusters, Scyld Beowulf provides rich reporting 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 is an X window. This happens when you log in and can be done thereafter by typing beostatus at the command prompt.

In order to use a Scyld Beowulf cluster for computation you must have at least one node available; "up". As shown in the screenshot above, all of the nodes that are up have a green checkmark on their row. If there are less nodes up than you think there should be, or some nodes report an error, ask your systems administrator to fix the problem.

Another reporting utility 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 search for the node’s state being up, the equivalent of the green checkmark.

Issuing Commands

head node

When you log into the cluster, you are actually logging into the head node. As such, any commands that you type on the command line will execute on the head node. The only exception is when you use 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 on which to run the command and the command. Bpsh allows you to run a command on more than one node without having to type the command once for each node, but it doesn’t provide an interactive shell on the remote node like rsh does.

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

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

and the command output would appear on the head node terminal where you issued the command. A range of nodes can also be specified for bpsh. To run the same command on nodes 3, 4, and 5 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.

This output may be hard to read if you have a large cluster.

The previous command, run on a hypothetical 64-node cluster on which half of the machines have the file being examined, would result in 32 lines of “/tmp/output” and another 32 of “ls: /tmp/output: no such file or directory”. The lack of node identifiers, plus the returned results sorted by the response speed of the compute nodes, makes it impossible to ascertain the existence of the target file on a particular node.

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 node 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 generates a divider between nodes and causes the functionality of -s to be used even if -s isn’t specified.

The command:

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

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

```
0: ls: /tmp/output: No such file or directory
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’s 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**

Special shell features, such as piping and input and output redirection, are available to advanced users. (For the examples below we’ll use the following conventions: the command running will be cmda. If it is piped to anything, it will be piped to cmdb. If an input file is used, it will be /tmp/input. If an output file is used, it will be /tmp/output. The node used will always be node 0.)

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

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

A bit more complex is to run the command on the compute node and do something with its input (or output) on that same compute node. There are two ways of accomplishing this. The first solution 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. Then you would type:

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

The second solution doesn’t require any of the programs to be on the compute node. It uses a lot of network bandwidth as it takes the output, sends it to the head node, then right back to the compute node. The appropriate commands are:

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

Lastly, to run a command on the head node and do something with its input or output on the compute nodes, one would type:

```sh
cmda | bpsh 0 cmdb
cmda | bpsh 0 dd of=/tmp/output
bpsh 0 cat /tmp/input | cmda
```
Chapter 2. Interacting with the System

Copying Data to Compute Nodes

There are a few ways to get data from the head node to the compute nodes. The easiest way is NFS. All files in /home, including the files in your home directory, are shared to all compute nodes via NFS (by default). Opening an NFS-shared file on a compute node will, in fact, open the file on the head node; no copying takes place.

To copy a file, rather than changing on the original across the network, you can use bpcp. This works much like the standard Unix file-copying command cp: 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, may be prepended with a node number and colon to specify that the file in question is on that node. If you wanted to copy the file /tmp/foo to the same location on node 1, you would:

bpcp /tmp/foo 1:/tmp/foo

The third method for transferring data is to do it programatically. This is a bit more complex and will only be described here conceptually. If you are using an MPI job, you can have your rank 0 process on the head 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 head 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 head 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 running on the compute nodes from the head node using standard Unix tools such as top, ps, and kill.

Scyld Beowulf adds a tool called bpstat to determine upon which node a process is running.

bpstat -p lists all of the processes that are currently being sent to compute nodes and reports their pid and the node upon which they’re running.

[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 the process ID. The Node column shows upon which node the process is running.

Using an uppercase ’P’ instead of a lowercase ’p’ (as above) tells bpstat to take the output of ps, a sample of which is shown here,

[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
Monitoring Node Status

Scyld Beowulf includes both graphical and text-based tools for monitoring the state and performance information for each node on the cluster.

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 monitors the reported information. The -u flag modifies the update interval from the default 4 seconds. Each update places additional load on the master and compute nodes and the interconnection network. Too-frequent updates can degrade the overall system performance.
Chapter 2. Interacting with the System

In environments where the X Window System isn’t either undesirable or impractical, such as accessing your Beowulf head node through a slow remote network connection, the `beostatus` command can still be used. The `-c` flag causes beostatus to use a curses text output appropriate to slow network connections. The data displayed in the window will be updated just as they would with the graphical display.

![Figure 2-1. Screenshot of Beostatus in Gnome/X Mode](image)

![Figure 2-2. Beostatus in Curses Mode](image)
bpstat Command Line Tool

The `bpstat` command shows a snapshot of the cluster but doesn’t continually update. When run without any arguments it prints a list of the nodes and information including the node’s status. (A node has to be in the ‘up’ state before it can be used.)

```
[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

The `beostat` command will show the raw data for the compute nodes’ status. The Reference Guide details the use of, and options for, this command.
Chapter 3. Running Programs

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

First we’ll manually run a simple non-cluster-aware program on a single computational node. Then we’ll map multi-process jobs on to the cluster computational nodes. Finally, we’ll run non-cluster-aware program across multiple nodes, an MPI program, a PVM program, and other types of cluster-aware programs.

Introduction

Let’s contrast executing a program on a stand-alone computer versus doing the same on a Scyld Beowulf cluster.

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, you type `ls` followed by `<return>` to generate a list of the files in the current working directory.

Typing the return key causes the command shell - a program which listens to and interprets commands typed in the terminal window - to start the `ls` program (stored at `/bin/ls`). The output is captured and directed to the standard output stream which also appears in the same window as you typed the command.

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

A Scyld Beowulf cluster isn’t simply a group of networked standalone computers. Only the head node resembles the computing system with which you are familiar. The compute nodes have only the minimal operating system necessary to support an application initiated from a head node.

Running `ls` on the head node causes the same series of actions as described (above) for a standalone computer.

Running `ls` on a compute node causes a very different series of actions. Remember that on a Scyld Beowulf there are no resident applications on the compute nodes of the cluster; they reside only on the head node. Typing `bpsh 1 ls` causes node 1 to receive the `ls` process via Scyld’s `BProc` software. The output appears in the terminal window on the head node, where you typed the command.

A full description of `bproc` and `bpsh` and how they function are covered here 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.
- Copy the program binary to each node, either manually, with a script, or through a network file system.
Chapter 3. Running Programs

- Ensure that each node has available identical copies of all the dependencies; the 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 head node and are migrated out to the compute nodes using BProc. By using a cluster architecture where jobs may only be initiated from the head 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). 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 the following pages.

Executing Programs That Aren’t Parallelized

Starting and Migrating Programs to Compute Nodes (bpsh)

There are no executable programs - binaries - on the filesystem of the computational nodes. Thus, by default, there’s no getty, login, or any shells on the compute nodes. How does one run a program on a compute node?

Just as one has remote shell (rsh) and secure shell (ssh) commands on networked standalone computers, each of which has its own collection of binaries, on a Scyld Beowulf one has the bpsh command.

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

The example shows the standard ls command running on node 2. From the output one can see that there’s no /bin directory. How did ls execute? It started on the head node; at the main() routine it’s memory-mapped to node 2, where it continues executing. Output is forwarded to the head node.

Note please that this isn’t a special version of ls, but a special way of handling execution. This process works with any program. All three standard I/O streams - STDIN, STDOUT, and STDERR - will be forwarded to the head node. Some programs stop working if they’re run in the background, as they need to read standard input. When you run a program in the background on a compute node use the -n flag, which will close standard input at invocation.

Because shell scripts expect executables to be present, and because compute nodes don’t meet this requirement, shell scripts should be modified to include bpsh commands required to affect the compute nodes and run on the head node.
Chapter 3. Running Programs

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

Just as traditional Unix has copy (cp), remote copy (rcp), and secure copy (scp) to move files to and from networked machines, Scyld Beowulf has the `bpcp` command.

Having the compute nodes read large data files served via NFS from the head node will result in major network congestion, or even a overload and shutdown of the NFS server. Thus the default sharing of the home directories via NFS, while useful for sharing small files, is not a good solution for large data files. `bpcp` is a better solution.

The following example shows a data file named `f001.dat` being copied from the current directory to the `/tmp` directory on node 6.

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

The default directory on the compute node is the current directory on the head node. The current directory on the compute node may not exist or already be NFS-mounted from the head node.

This example shows the creation of a temporary directory and then the transfer of a data file to node 2.

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

The following example copies a data file from node 2 to node 3 directly, without the data being stored on the head node.

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

**An Introduction to Parallel Programming APIs**

What does it mean to run in parallel?

Programmers are generally familiar with serial, or sequential, programs. Simple programs, like "Hello World" and the basic suite of searching and sorting programs, are typical of sequential programs. These have a beginning, an execution sequence, and an end. At any time while during the run of the program 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. At any time while a thread is running, there is a single point of execution. A thread differs in that it isn’t 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 of one computer, or in the case of a cluster, different computers). A few things are required to make parallelism work and be useful: the program has to migrate to another computer(s) and get started and, at some point, the data upon which the program is working has to be exchanged between the processes.

The simplest case is when the same single-process program is run with different input parameters on all the nodes and the results are gathered at the end of the run. Using a cluster to get faster results of the same non-parallel program with different input is called *parametric* execution.

Imagine a much more complicated example, say, 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*, an MPI.
To solve these two problems - 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, such as the well-named MPI (which we recommend).

**MPI - A Brief Description**

The MPI API is currently the post popular choice for writing parallel programs. The MPI standard leaves implementation details to the system vendors (like Scyld). This is useful because they can make appropriate implementation choice 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 project managed by Argonne National Laboratory and Mississippi State University; visit the MPICH web site[^1] for more information.

A program which uses MPI 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 goodly number of conditionals are evaluated to determine what each process’ actions. Messages may be sent and received between processes.

The advantages of MPI are that the programmer:

- doesn’t have to worry about how the program gets started on all the machines
- has a simplified interface for interprocess messages
- doesn’t have to worry about mapping processes to nodes
- abstracts the network details resulting in more portable hardware-agnostic software

**PVM - A Brief Description**

Parallel Virtual Machine (PVM) was an earlier parallel programming interface; 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. It fails to abstract implementation details as well as MPI does. It’s deprecated, but still in use by legacy code. We generally advise against writing new programs in PVM but 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. In terms of starting a remote program, there are programs written:

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

These solutions come with their own problems, particularly in the implementation details. 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?

Scyld Beowulf, which doesn’t have binaries installed on the cluster nodes, may not lend itself to these techniques; your mileage may vary.
We recommend you write your parallel code in MPI. That having been said, we can say that Scyld has some experience with getting `rexe()` calls to work, and that one can simply substitute calls to `rsh` with the more cluster-friendly `bpsh`.

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

Running programs specifically designed to execute in parallel across a cluster requires at least the knowledge of the number of nodes to be used.

Scyld Beowulf uses the `NP` environment variable:

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

The example above shows an MPI-saavy program, `a.out`, located in the current directory, run with four processes. (Our examples show the syntax of the Bourne shell (/bin/sh or /bin/bash). Each kind of shell has its own syntax for setting environment variables.) Unspecified is where the processes will execute; the job of the mapper. The mapper includes both mapping and batching (also known as queuing).

Mapping is the act of deciding upon which node each process will execute. While it seems awfully simple, as various requirements are added, it can get complex. The mapper scans available resources at the time of job submission to decide which processors to use.

Batching is the queuing of jobs until the mapped resources become available. Scyld Beowulf includes a mapping API (documented in the Programmer’s Guide) and a details of writing your own mapper.

The mapper’s default behavior is controlled by 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. Similar to 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 head node. (For debugging purposes.)
- **NO_LOCAL** - Don’t run any processes on the head node.
- **EXCLUDE** - A colon-delimited list of nodes to be avoided during 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.

The `beomap` program shows the current mapping (for the current user in the current environment with the current resources at the current time). Generally used in shell scripts; useful also to learn about these environment variables.

Some more examples:

```bash
> 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, can run tasks on the head 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 processors, 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 **mpprun** is:

```
mpmprun [options] app arg1 arg2 ...
```

where *app* is the application program you wish to run; it need not be a parallel program. The *arg* arguments are the values passed to each copy of the program being run.

See the Reference Guide for a complete list of options for **mpprun**. Options exist to control the number of processors upon which to start copies of the program, 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 head node, or to prevent any jobs from running on the head node. The most interesting of the options is the **--map** option. The map option takes a colon-delimited list of nodes upon which copies of a program should be run. This argument, if specified, overrides the mapper’s selection of resources that it would otherwise use.

This example shows 16 tasks of program *app*: Some examples of using **mpprun** are:

```
mpmprun -np 16 app infile outfile
```

This example does the same on any available nodes except nodes 2 and 3:

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

This example 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:

```
mpmprun --map 4:2:1:5 app infile outfile
```

**beorun**

**Beorun**, designed to run applications rather than utilities, runs them concurrently on the selected nodes. The basic syntax is:

```
beorun [options] app arg1 arg2 ...
```

where *app* is the application program you wish to run; it need not be a parallel program. The *arg* arguments are the values passed to each copy of the program being run.

See the Reference Guide for a complete list of options for **beorun**. Options exist to control the number of processors upon which to start copies of the program, 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 head node, or to prevent any jobs from running on the head node. The most interesting
of the options is the --map option. The map option takes a colon-delimited list of nodes upon which copies of a program should be run. This argument, if specified, overrides the mapper’s selection of resources that it would otherwise use.

This example runs 16 tasks of program app.

beorun -np 16 app infile outfile

This example does the same, on any available nodes except nodes 2 and 3.

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

This example 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 --map 4:2:1:5 app infile outfile

Running MPI-Aware Programs

MPI-aware programs are those written to the MPI specification and linked with the Scyld MPICH library.

Direct Execution

The following examples show a hypothetical MPI-aware program named my-mpi-prog. (Details about how to build MPI programs can be found in the Programmer’s Guide.)

This example shows a cluster execution of my-mpi-prog run with 4 processes:

NP=4 ./my-mpi-prog

An alternative syntax is:

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

Note please that the user specified neither the nodes to be used nor a mechanism for migrating the program to the nodes. The mapper does these tasks; jobs are run on the nodes with the lowest CPU utilization.

The following example specifically states the number of processes to be used (NP=6), the stricture against using the head node as a computing resource (NO_LOCAL=1), and the exclusion of some cluster nodes (EXCLUDE=2:4:5).

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

mpirun

Almost all implementations of MPI have a mpirun program (which shares the syntax of mpprun but which boasts of additional features for MPI-aware programs.)
Chapter 3. Running Programs

Using mpirun

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

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

is equivalent to running, in the Bourne shell, the commands:

```
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. 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 source code to invoke `mpirun` with the `-np 16` option from within a program is shown below.

**Example 3-1. MPI Programming Example**

```c
/* Standard MPI include file */
#include <mpi.h>

main(int argc, char **argv) {
    setenv("NP","16",1); // set up mpirun env vars
    MPI_Init(&argc,&argv);
    MPI_Finalize();
}
```

Running PVM-Aware Programs

*Parallel Virtual Machine (PVM)* is an *application programming interface* (API) for writing parallel applications, enabling a collection of heterogeneous computers to be used as a coherent and flexible concurrent computational resource. It has been specifically tailored to take advantage of the technologies used in a Scyld Beowulf. 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 cluster configuration for PVM is beyond the scope of this document. 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 head node using the PVM console, `pvm`. Adding compute nodes to the virtual machine is done by issuing a `add .#` command, where `'#` is replaced by node’s assigned number in the cluster (as listed by `beosetup` or `bpstat`).

Alternately, you can start the PVM console with a hostfile filename on the command line. The hostfile 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; a separate line for each node being used is shown. Once your virtual machine has been configured, run your PVM applications as 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` or `rcp` can instead call `bpsh` and `bpcp`, respectively. `beomap` may be used by any script to load balance programs that are to be dispatched to the compute nodes. Application porting details 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 (based on the standard `at` package; enhanced for the Scyld cluster environment). It is installed as part of the default installation procedure.

`bbq` releases submitted jobs as the requested resources become available to keep the cluster at a predetermined level of utilization, the *load average*. (Resources are based upon the same variables examined by `mpirun` and `mpprun`.) Users can schedule programs to run at specific times or when resources become available.

`bbq` consists of three programs, the daemon `atd`, a job-submission tool `at`, and a queue viewer `bbq`. Standard `at` queues are available, but only the `b` queue has knowledge of the cluster resources (via Scyld software enhancements). The `batch` command has been aliased to `at -q b`, using the `b` queue for job submission, as a convenience.

User job output - both STDOUT and STDERR - will be captured as text and sent via email to the user at `username@localhost`. To forward the mail to your regular email account create a `.forward` text file in your home directory on the head node containing the fully-qualified destination email address. (You can have multiple entries in this file; mail will be sent to each destination).

**Submitting Jobs**

Users may submit jobs to the `batch` program, either by file or interactively. Both use the same command syntax.

Time may be specified in user-friendly terms like `now, noon, midnight, today`, with simple arithmetic expressions such as `now + 3hours, 4pm + 2days`, and with specific timestamps such as `batch 06:32 2001-06-04` and `batch 6:32am jun 4 2001`. These can be combined, as in `batch noon sun, batch tomorrow`.

When only the time of days is specified and the time is in the future, today’s date is used. When the time is in the past, tomorrow is assumed. When only the date is specified the current time is used. 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.

This example of batch file usage shows `jobfile` being submitted for execution at noon tomorrow.

```
[user1@cluster]$ batch noon tomorrow -f jobfile
```

An alternate syntax used input redirection:

```
[user1@cluster]$ batch noon tomorrow < jobfile
```
An interactive session is started with a `batch` `timestamp` command, where `timestamp` is as described above. At the `at>` prompt commands may be typed, separated by a carriage return, the session terminated the sequence `^d` (Control-D, EOF) character. `batch` will then parse any time and date parameters, display the time to execute the sequence, and exit.

We recommend that the first line be a comment identifying the job, regardless of method used to submit jobs. Comments start with the `#` character. `bbq` displays the first 38 characters of user input; the comment and characters following may assist in identifying the job later.

Users must be aware of their computing environment, and take pains not to mix different resource requests in one job. For example:

```
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. For clarity, users should have no 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 execution time and queue name are encoded in the file name. When execution time arrives, as calculated by `at`, `bbq` uses the process request mapped against the available processor resources to determine whether or not to run the job. Jobs will wait 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 per cent CPU utilization by default.

To determine the processor request, `bbq` calls the `beomap` function (described previously) with all of the MPI parameters and the user’s environment; a list of nodes to use is returned. Jobs are released in a first come, first served basis; there are currently no ways to assign priorities. If one job requests all of the cluster’s processors, that job will wait 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.

Notice please 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 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 dual-processor nodes the same example will have `bbq` map 1 process per processor.

Currently `mpprun` does not support processor requests. The environment variable equivalents are:

```
-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

The job queue may be viewed as text (bbq lists the pending and running batch jobs) or via a GUI (beoqstat, which can sort and delete jobs). bbq will generate HTML to STDOUT when the GATEWAY_INTERFACE environment variable is set (generally only when called via an Apache web server as a CGI script.) Details are available in the Reference Guide.

![Figure 3-1. Beoqstat - The GUI BBQ Monitor](image)

Anyone may view the queue; only the job owner or root may remove a job (use atrm 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 point-of-entry through which the administrator can easily gather job statistics (including user name, processor resource requests, and start/end times). To do this 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 daemon to start all jobs with the daemon’s group ID, overriding the user’s normal group, allowing the job to access the cluster compute nodes.

PBS

PBS, originally Portable Batch Scheduler, now POSIX Batch Scheduler, defines jobs to be a shell script. As Scyld Beowulf can, by default, run shell scripts, PBS and other schedulers - such as NQS - will work on the head node but cannot schedule jobs on the compute nodes unless the master configuration is modified.

PBS-Pro takes advantage of Scyld’s BProc technology and therefore works on a Scyld Beowulf cluster. Various Scyld users have used traditional schedulers - like PBS - in a cluster of clusters. Jobs are dispatched to the head node of each sub-cluster, which can then map the jobs to its sub-cluster. This strategem may not be necessary in smaller facilities.

File Systems

File System Options

Data files used by the applications processed on the cluster may be stored in a variety of locations:

- on the local disk of each node
Chapter 3. Running Programs

- on the head node’s disk, shared with the nodes through a network filesystem
- on disks on multiple nodes, shared with all nodes through the use of a parallel filesystem

The simplest approach is to store all files on the head node, as with the standard Network File System. Any files in your /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 head 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 head node. 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, which 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.

PVFS

Parallel Virtual File System allows applications, both serial and parallel, to store and retrieve data which has been distributed across a set of I/O servers. This is done through traditional file I/O semantics; you can open, close, read, write, and seek in PVFS files just as you can in locally-stored files.

The primary goal of PVFS is to provide a high performance “global scratch space” for Beowulf clusters running parallel applications. PVFS will “stripe” files across the disks of the nodes in your cluster, resulting in file access faster than that of a single disk.

Within your cluster any given node may take on one or more of the following roles:

- metadata server
- I/O server
- client

The metadata server, one per PVFS filesystem, maintains information on files and directories stored (including 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.
Chapter 3. Running Programs

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. On our example system we will configure the head node as the metadata server and the other eight nodes as both clients and I/O servers. This will allow us to run parallel jobs accessing PVFS files from any node, striping these files across all the cluster compute nodes.

A PVFS filesystem appears to a user much as any other filesystem. (Once the system administrator mounts the PVFS filesystem on your local directory tree 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**

PVFS will provide a default striping of your data across the I/O servers for your file when you use a standard Unix command, like `cp`, to copy files into a PVFS directory. The `u2p` command, supplied with PVFS, is used to copy an existing Unix file to a PVFS filesystem while specifying physical distribution parameters. These are:

- base - the index of the starting I/O node, with 0 being the first file system node
- pcount - partition count (a bit of a misnomer), the number of I/O servers on which data will be stored
- ssize - strip size, the size of the contiguous chunks stored on I/O servers
Chapter 3. Running Programs

Figure 3-3. Striping Example

The example shows a PVFS filesystem with a base node of 0 and a pcount of 4. 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

`pvstat` will show the physical distribution parameters for a PVFS file. For example, for a file named `foo` in the PVFS filesystem mounted at `/pvfs`, you would:

```
[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 determines the state of a given I/O server:

```
[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, reported as responding. As there’s no I/O server on the head node it is reported as down. Likewise the `mgr-ping` utility is 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 response shows a metadata server responding on the head node, but not on node 1.
These two utilities also set their exit values appropriately (for use with shell scripts; 0 for success (responding), 1 on failure (down)).

When no additional command-line parameters are passed the programs will automatically check for servers on localhost, the local machine, at the default port, 7000 for I/O server and 3000 for metadata server. A “-p” may be used to alter the port.

**Sample Programs Included in the Distribution**

**Linpack**

The Linpack benchmark suite, used to evaluate computer performance (see the Top 500 page), stresses a cluster by solving a random dense linear system, maximizing your CPU and network usage. Administrators use Linpack to evaluate the cluster fitness.

The hpl includes Linpack.

Start Linpack with the Scyld Beowulf-provide shell script, linpack, which starts xhpl after creating a configuration/input file. (The default settings are too general to result in good performance on clusters larger than a few nodes; consult the file /usr/share/doc/hpl-1.0/TUNING for tuning tips appropriate to your cluster. A first step is increasing the problem size, set around line 15 to a default value of 3000. Too high a value will cause failure by memory starvation.) If it doesn’t run to completion, or takes too long, investigate network problems (a bad switch, incorrect switch configuration).

![Figure 3-4. Testing your cluster with linpack](image)

---

*Figure 3-4. Testing your cluster with linpack*
Chapter 3. Running Programs

**MPI-Mandel**

`mpi-mandel` is a graphical, interactive demonstration of the Mandelbrot set. It is a classic MPI program, and therefore gives us a chance to use our mapping environment variables:

```
prompt# ALL_CPUS=1 /usr/bin/mpi-mandel
```

This example invocation will run on all available CPUs.

![Figure 3-5. Demonstrating your cluster with MPI-Mandel](image)

In the screen capture above notice the slave (compute node) count in the bottom-left corner. Running on x86 CPUs with a performance counter support-enabled kernel, the number of integer and floating point calculations is given in the status bar, at the bottom of the window.

To have `mpi-mandel` run as a free-flowing demonstration, you can load a favorites file:

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

It will refresh more quickly if you turn off Favorites > delay. (Consider restoring the delay if you are running a very non-homogeneous cluster, or if your video card isn’t fast enough. Both conditions may result in a partial screen refresh.)
Notes

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 program’s 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).
Appendix A. Glossary of Parallel Computing Terms

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
Appendix A. Glossary of Parallel Computing Terms

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