Cplant Parallel Application Support: Job Launch, Monitoring and Termination Batch scheduling and debugging Administrator's Guide (1.0)

Lee Ann Fisk

May 11 2001

Contents

Chapter 1

Introduction

This document describes the utilities, daemons, configuration files, and scripts in the Cplant runtime environment that combine to do the following

- determine availability of compute nodes
- launch parallel applications
- set up the environment on the compute node for the application process
- obtain status of running jobs
- debug running parallel applications
- terminate running jobs with a signal
- logically add compute nodes into the virtual machine, remove them or reserve them

It also describes how PBS (Portable Batch System) works on Cplant, including how to install it, configure it, run it, and handle common problems that may occur.

It is intended for administrators who will be setting up a Cplant and maintaining it while it is running. This document is targeted to the Sandia National Laboratories site in Albuquerque. There may be differences at other sites, such as location of configuration files. Other Cplant sites are encouraged to write a site guide to supplement this guide.

It is assumed that the reader is familiar with the Cplant architecture and in particular

- the division of computing resources into a few full service machines (the *service partition*) and many lean compute engines (the *compute partition*)

- the administration of Cplant through a separate hierarchy of workstations from which administrative file systems are mounted and the compute and service nodes are booted and maintained

We frequently refer to *nodes* in this document and *node numbers*. The workstations in the cluster which are playing the role of *service node* or *compute node* are the *nodes*. The service nodes host users loading parallel applications and querying for job status, among other things. The compute nodes each host a single process of a parallel application. The *node numbers* we refer to are indices into a map of Myrinet hardware addresses, are contiguous, and begin at zero. These are the numbers displayed with a label like *Node ID* in the status displays and error messages of **yod** and **pingd** .

Other helpful information may be found at the Cplant web site, `http://www.cs.sandia.gov/cplant` or by email queries to `alaska-help@cs.sandia.gov`.

This document describes revision number 1.0 of the Cplant code.

Chapter 2

Overview

This chapter gives a general overview of Cplant from the standpoint of the application load utilities and daemons. It describes broadly how a system administrator would set up and start the daemons and how a user would start a parallel application.

Cplant is a cluster of workstations dedicated to running message passing parallel applications. It was designed from the early stages of the project to scale to thousands of nodes, and to provide message passing and file IO performance sufficient for the applications that had previously run on the lab's massively parallel supercomputers.

The machines running user parallel codes (the compute nodes) are space shared: only one user process runs on the machine. The application must run to completion on all of it's nodes before another user may claim any of the compute nodes for a new job.

Application message passing is normally handled with calls to an MPI library [7]. The MPI library is layered over Cplant portals [4]. Portals are user level data structures understood by the operating system and were designed to reduce message passing latencies by allowing the user libraries to set up data structures permitting the operating system to copy incoming messages directly into the user's buffers.

Applications are linked with another message passing library called the Cplant server library (also layered over portals), described in [5]. The server library is used by the application before it begins user code to set up the group information required by MPI. It is also used for I/O to the yod proxy process and some other miscellaneous tasks. Use of the server library by the application is transparent to the programmer, but there is no reason why the programmer can't use this library if he or she wishes. Message passing can also be performed with direct calls to the portals library, although we don't know of anyone doing this.

The Cplant daemons and utilities described herein communicate over the Cplant server library. This library allows the various entities involved to send brief control messages, get and put large blocks of data, and to temporarily form groups to do collective communication operations. In subsequent pages of this document, when we refer to *putting* data in another's memory or *getting* data into our memory from another utility or daemon, we are referring to server library operations.

The server library has fault recovery features that enable groups of PCTs to carry on if one of the PCTs crashes during an application load. The library routines report which PCT has malfunctioned, and system adminstrators can view log files to obtain this information. After a failed load, yod will also display to the user which PCT failed in the group operation. The user should report this to system administrators so they can investigate further.

There are two utilities and two daemons involved in node allocation, application load, and application status queries:

- The **bebopd** (a Better Engineered Bag Of PCs daemon) runs in the service partition and serves as a node allocator and a repository of status information about the compute partition and the jobs running in it. The **bebopd** is a single process. It had been our hope to implement this as a distributed server to enhance fault tolerance and improve response time, but time has not permitted this enhancement.

- The **PCT** daemon runs on each compute node. It starts the user process on the node and responds to status queries about the user job. The **PCT** also monitors the health of the compute node.

- The **yod** utility is run on a service node by a user wishing to load a parallel application. The application may consist of one or more different executable files. The application stdout and stderr streams are channeled to **yod** for dispatch.

- The **pingd** utility is executed on a service node and returns status information about parallel jobs running in the compute partition. **pingd** has functions of use to system administrators as well functions for users. **pingd** may be used by administrators to reset busy compute nodes so they can run a new application, to kill the **PCT** on a compute node, to reserve a compute node, or to inform the **bebopd** that the **PCT** on a compute node is dead. **pingd** is also used to modify some elements of the PBS configuration at runtime.

In this chapter we will provide a general overview of the use of these four elements from the standpoint of a user running a job and from the standpoint of an administrator setting up and maintaining a running Cplant. Subsequent chapters will provide details on the interfaces of the daemons and the configuration elements that affect their behavior.

2.1 Initializing and Managing a Virtual Machine

Since the **bebopd** and **PCT** are portals processes, they must not be started on a machine until the system is ready to register portals processes and handle incoming and outgoing portals messages. Currently, since Cplant runs Linux and sends portals traffic over Myrinet, this means that the MCP (Myrinet Control Program) is loaded on the LANai network card and the portals module (p3mod), the Cplant module (cTask), and Myrinet driver module (rtscts) are installed.

The **bebopd** is a single process that runs on a service node and maintains the state of the compute partition. As such, the **bebopd** is synonymous with a *virtual machine*. If two **bebopd** daemons are running on two different service nodes, managing two disjoint sets of compute nodes, then logically the Cplant appears to be two separate clusters. There is nothing physically preventing communication between service and compute nodes of the two machines, but due to the way **bebopd** administers it's partition, it won't happen.

In fact, Cplant can be partitioned into virtual machines, each consisting of a collection of compute nodes and service nodes, each with it's own version of Cplant software and configuration files. The directories and scripts on the administrative network that make this happen are described elsewhere. The point here is that one **bebopd** will be running on each virtual machine. Users logging in to a service node of a given virtual machine will receive compute resources allocated by the **bebopd** running on that virtual machine.

2.1.1 Configuration Files and Log files

There are a few configuration files accessed by the runtime daemons and utilities. The pathname to these files is coded into the daemons and utilities, but may be overridden by environment variables or by entries in a `site` file.

The **yod** application launcher writes a user log file as does the **bebopd**. Also, all utilities and daemons that use the Cplant server library log exceptions and warnings to a log file. The **bebopd** writes a restart file when it exits. The pathnames to these files may also be specified via environment variables or `site` file entries.

The configuration files and log files for PBS are described separately in the PBS chapter.

About the location of configuration and log files

The location of your configuration and log files will be determined by your cluster management decisions. We typically place files required at runtime on the sss1 (the main administrative node) in `/cplant/vm/VM-NAME/nfs-cplant/etc` which we find at `/cplant/etc` on all the service and compute nodes. (`VM-NAME` of course would be replaced with the name of your virtual machine. Then the file system at `/cplant/vm/VM-NAME/nfs-cplant` is mounted at `/cplant` on the all service and compute nodes in virtual machine `VM-NAME`.)

The `/cplant/etc` file system is read only. We need a writable directory for the user log files written by **yod** and **bebopd**, for the restart file written by the **bebopd**, and for the log files written by the PBS daemons.

We have had trouble allowing **yod** processes on different service nodes to write the same NFS mounted log file. This is in spite of the fact that we wrote our own file locking routine in **yod** that should work in the environment in which we were running it. For this reason, we prefer to set up the user log file so that there is a separate one for each service node, either through links to a separate file for each service node or by writing the user log file to a local disk on the service node.

We find installing a local disk in the service node and placing log files and restart files there works best for us.

Site file

The runtime utilities and daemons look for configuration files and log files at certain default locations. These default locations can be overridden by environment variables or by entries in the `site` file. The utilities and daemons will look first for an environment variable. If one is not defined they will look for a `site` file definition. If there is no `site` file definition they will use a default value.

Chapter 11 describes the `site` file entries in detail. The `site` file entry is simply a line of text listing the environment variable name followed by the environment variable value.

Utilities and daemons search for the `site` file at the location specified by the environment variable `SITE_FILE`. If that environment variable is not defined, they look for the `site` file at `/cplant/etc/site` .

Cplant host file

The `cplant-host` file lists the physical node number of the node on which the **bebopd** is running. It must be accessible from every node on which **yod**, **pingd** or a **PCT** is running, since all these processes need to contact the **bebopd** .

The `cplant-host` file also lists the location of **fyod** file servers on systems using **fyod**.

Here is an example:

```
# This file specifies which nodes offers certain services.
# The bebopd format is:
# bebopd node_name1 [node_name2 ... [node_nameN]]
# The fyod format is:
# fyod unit_no node_id

bebopd  31              #bebopd on c-7.SU-3
fyod 010 9
fyod 011 9
fyod 020 10
fyod 021 10
```

At runtime, the default location of the `cplant-host` file may be changed by setting the environment variable `CPLANT_HOST_FILE` to the path name of the `cplant-host` file. Alternatively you can add an entry to the `site` file.

Cplant map file

The Cplant utilities display host names for the compute nodes that indicate what physical part of the machine they are in. For example the cluster *ross* is composed of racks of 32 nodes and the host name displayed for a compute node includes the rack number and the number of the node within the rack.

The utilities derive these hostnames from the `cplant-map` file. The format of the file is a list of host names in physical node number order.

The location of the `cplant-map` file can be specified with the environment variable or `site` file entry `NODE_NAMES_FILE`.

The virtual machine name file

The `vm-name` file is a text file containing the name of the virtual machine. It is located by default at `/cplant/etc/vm-name`. This default may be overruled by defining the environment variable `VM_NAME_FILE` or setting the analygous `site` file entry.

The user log files

When a parallel application completes, **yod** will log the node usage to a log file. This logging by default goes to the file `/tmp/userlog`. The location can be changed by defining the environment variable `LOGFILENAME` or setting the analygous `site` file entry. Logging works best if the user log file is a different file for each service node.

The **bebopd** logs user activity in a file with format identical to the user logs on ASCI Red. The log file is written to the same directory that the **yod** user log file is written. The file name is `run.log.mm-dd-yy`. There is no way to override this file name. It is the same file name as that given the user logs on ASCI Red.

The cplant log file

All utilities and daemons linked with the Cplant server library will log errors and warnings to a log file using the `syslogd` facility. We log to `/var/log/cplant` on our clusters, and often set up the compute and service nodes so that `/var/log/cplant` goes to the `sss0` (or *leader* or *mother*) node. This log file is the place to look if a daemon seems to be experiencing problems.

The file that the Cplant daemons and utilities are logging to is named in `/etc/syslog.conf` as the target for the `local7` facility.

The bebopd restart file

The **bebopd** maintains a list of all compute nodes in the virtual machine and their status. When the **bebopd** exits, it writes a restart file with these PCT records. If the **bebopd** is started later on with the `-r` option (for *restart*, it will read in the restart file and contact all those PCTs to get their current status).

By default the restart file is written to `/etc/local/saved_pct_list`. You may override this default by setting the environment variable `BEBOPD_RESTART_FILE`, or by setting the analygous `site` file entry.

2.1.2 Starting the Daemons

1. As user `root`, start a **bebopd** process on the service node named in the `cplant-host` file. Unless you are debugging, start it with the `-d` (daemon) argument so it will run in the background. The **bebopd** is started at boot time by the `user-env` script on the Sandia Albuquerque Cplants.

2. On every compute node in the virtual machine, start a **PCT** as user `root`. Again, unless you are debugging, start it with the `-d` (daemon) argument so it will run in the background. The **PCT**looks up the location of the **bebopd** in the `cplant-host` file. It sends an update message to the **bebopd** at regular intervals until it receives an acknowledgement. If the **PCT** does not receive this acknowledgement from the **bebopd** within 10 minutes, it gives up and exits. On the Sandia Albuquerque Cplant, the **PCT**s are started by the `user-env` script at virtual machine boot time.

If you now run **pingd** on any service node in the virtual machine, you will see a list of the **PCT**s that the **bebopd** has received update messages from. As far as the **bebopd** is concerned, this is the extent of the virtual machine. Users may now run parallel applications on the listed nodes.

2.1.3 Stopping and Restarting the Daemons

The PCT

When a **PCT** terminates in an orderly fashion it sends a message to the **bebopd** to remove itself from the active list of **PCT**s. A **PCT** will terminate when it has a resource problem (for example, not enough memory for a portal's buffers), when it receives terminating signal, or when instructed by the **bebopd** to exit.

To stop a **PCT** either of the following will work:

- `kill -2` *pct-PID* on the compute node.
- `pingd -kill -l` *pct-NID* on any service node in the the virtual machine. (You must be `root` or in group `bebop` to submit a *kill* command to **pingd**.

In the event that a **PCT** terminates without the knowledge of the **bebopd**, or you just don't want the **bebopd** to allocate the node it is running on, you may use **pingd** to notify the **bebopd** that the **PCT** is gone. If the **PCT** on node 12 has terminated, enter:

```
pingd -gone -l 12
```

You must be user `root` or in group `bebop` to issue a *gone* command to **pingd**.

To restart a **PCT**, simply start it again on the compute node. It will send an update to the **bebopd** and rejoin the virtual machine.

To force a **PCT** to send an update message to the **bebopd**, send the **PCT** a SIGHUP. If the **bebopd** had considered the node to be gone, it will now include it again in the virtual machine.

What if a **PCT** terminates while hosting a member of a parallel application? The member process will continue to execute. It is not impacted by the disappearance of the **PCT**. It is best not to restart a **PCT** on the node until the parallel application finishes. The new **PCT** would not know about the application process and would announce itself to the **bebopd** as ready to run a job.

Note that in this situation there is no **PCT** to catch the termination of the user process, so **yod** will never receive a message that the application is done on that node. It will be necessary for the user to note through other evidence that the application is done and then terminate **yod** with `control-C`. (For instance, output data files are completed, or **pingd** shows all the other processes are done, or simply `rsh compute-node-name ps ax`.) Also, **pingd** will not return any status information for the node on which the **PCT** went away, because **pingd** gets it's information from the **PCT** by way of the **bebopd**.

In this situation (where the PCT is killed when an application is running) it will most likely be necessary to run `pingd -reset` to reset the other PCTs to free status after the application is done.

The bebopd

When a **bebopd** terminates in an orderly fashion, it writes a file listing the active **PCT**s at the time of it's demise. The file it writes to is named by the environment variable `BEBOPD_RESTART_FILE` (or it's `site` file equivalent) and defaults to (`/tmp/saved_pct_list`).

The **bebopd** may be restarted with the `-r` option:

```
bebopd -r {optional-file-name}
```

The **bebopd** will look for `/etc/local/saved_pct_list` (or the file named by the `BEBOPD_RESTART_FILE` variable) and send status requests to all **PCT**s listed there. The **bebopd** will set the next job ID to an ID greater than that of the jobs that were running when the **bebopd** last terminated.

The `-r` option of **bebopd** takes an optional file name argument. **bebopd** will look in this file rather than it's default for a list of **PCT**s to contact.

Again, **bebopd** must be run as user `root`.

2.2 Running a Parallel Application

A user launches a parallel application by running **yod** on a service node. The general command syntax is:

```
yod {yod-options} executable-path-name {application arguments}

OR

yod {yod-options} load-file-name
```

The `yod-options` specify such things as the number of nodes required or a list of the node numbers of the actual nodes requested. (The details can be found in Chapter 5.) If the application consists of more than one command line (*heterogeneous load*), then the executables and their arguments can be listed in a textual load file, and this load file name is then the last argument to **yod** .

The following sequence of events occurs:

1. **yod** creates a node request for the **bebopd** . Depending on the `yod-options` this will be a simple request for some number of compute nodes, or a more complex request involving specific lists of nodes.

2. **yod** sets up portals required for communication with the **bebopd** , with the compute partition **PCT**s , and with the user application processes.

3. **yod** determines the location of the **bebopd** from the `cplant-host` file, sends the **bebopd** a request for compute nodes, and waits for a reply.

4. If the **yod** job is part of a PBS job, the **bebopd** at this point ensures that the PBS job is not requesting more nodes than it has been allocated by the batch scheduling system.

5. If the requested nodes are not available, **yod** informs the user and exits. Otherwise, **yod** receives from the **bebopd** a list of the compute nodes reserved for it and the identifying information for the **PCT**s on those nodes, and a numeric job ID identifying the parallel application. A **PCT** that has been allocated to a **yod** job but has not yet heard from **yod** will be listed as *pending allocation* on the **pingd** display.

6. **yod** sends an initial message to each **PCT** notifying it that it will be hosting a parallel application. **yod** must be snappy because **PCT** reservations expire within 60 seconds if an allocated **PCT** does not hear from **yod** .

7. The **PCT**s form a temporary group for the purpose of hosting the parallel application. After successfully forming a group, they can engage in global communication operations like broadcast and barrier synchronization. The root **PCT** (the one on the node hosting the application rank 0 process) notifies **yod** when this group formation is complete. In

the case of heterogeneous load (more than one command line and possibly more than one executable in the parallel application) the **PCT**s also form subgroups based on which command line in the load file they are hosting. Each of these has a root, the **PCT**hosting the lowest ranked member in the subgroup.

8. Normally, since compute nodes are diskless, the **PCT** stores the executable image in RAM disk . At this point the **PCT**s check to see if they have sufficient room in RAM disk to store the executable. If any **PCT** is unable to reserve local storage for the user's executable file, the **PCT**s will notify **yod** that the executable will need to be copied to a file system from which the PCTs can read it. (This kind of load is much slower that a load where the user's program is sent to the PCT and stored in RAM disk.) At this point **yod** will copy the executable file to the file system specified by the variable `PARALLEL_FILE_SYSTEM`, or to , `/enfs/tmp` if that variable is not defined.

9. **yod** *puts* the argument data in a buffer of the root **PCT**. The root **PCT** broadcasts this data to the group. (In the case of heterogeneous load, **yod** puts the argument data for each executable in a buffer of the subgroup root **PCT** for that executable. These root **PCT**s then broadcast this data to their subgroups.)

10. **yod** *puts* the user's environment variables in a buffer of the root **PCT**. The root **PCT** broadcasts this data.

11. The **PCT**s need to know group ID information for the application owner in order to limit file IO operations. If the owner is a member of more than a few groups, this list of groups is sent up to the root **PCT** for broadcast to the other **PCT**s.

12. If the user wants the **PCT** to run the application under the debugging tool `strace`, **yod** sends the `strace` options up now.

13. If the **PCT**s had been able to reserve space in RAM disk for the executable, **yod** searches for the user executable(s) using the user's PATH environment variable if a fully qualified path name was not provided to **yod**. **yod** reads the executable image into a buffer, and then *puts* this buffer in a buffer of the root **PCT** for broadcast to the other

PCTs. Otherwise, the **PCT**s access the executable from the file system specified by the variable `PARALLEL_FILE_SYSTEM` or , `/enfs/tmp` if that variable is not defined.

14. In the compute partition, each **PCT** sets up the environment for the user process and then forks and execs it. The user process begins executing special initialization code linked with it by the Cplant compilation scripts, but does not procede to user code. This initialization code sets up the process to use portals, and sets up the information required to perform file IO through a remote server.
15. Each **PCT** sends a synchronization message to **yod** at this point.
16. When **yod** has received all these synchronization messages, it knows that the user process has begun on every node, but has not yet entered user code. **yod** sends a message to the root **PCT** to be fanned out to all **PCT**s indicating that the application processes should continue to the entry point of user code.
17. Each **PCT** instructs the user process to procede to user code.
18. The **PCT**s collect a map of the portal process IDs of the new processes and **yod** *gets* this map from the root **PCT**.
19. **yod** serves as a front end to the running application. It processes user application IO calls (opens, reads, writes, etc), and forwards signals sent by the user to **yod**. SIGUSR1 and SIGUSR2 are forwarded to the application processes. Interrupting **yod** will cause **yod** to forward a SIGTERM to the user application processes. If this is ignored by the application, interrupting **yod** again will cause **yod** to forward a SIGKILL.
20. When a user process terminates, the **PCT** sends a completion message to **yod**.
21. If the **yod** user ran **yod** with the `-g` option, and the application process had terminated with a signal, **yod** will request a stack trace of the faulting application process from the **PCT**.
22. When **yod** has received a completion message from every node, it sends an *all done* message to the root **PCT** for broadcast and displays the

completion messages with user process exit code or terminating signal. If stack traces are available **yod** will display them after each node's completion message. **yod** then logs the node usage to a log file and exits.

23. When a **PCT** receives an *all done* message, it notes that it is now available and will report itself as available when queried by the **bebopd**.

Chapter 3

bebopd Node Allocator

3.1 Command line arguments

```
bebopd -D -r {optional-file-name} -alternative
       -S {0|1} -L {0|1}
       -daemon -help
       -PBSsupport  -PBSupdate  -PBSinteractive {num-nodes}
```

-D This debugging option causes the **bebopd** to output information about what it is doing. Repeating the **-D** option on the command line increases the amount of information.

-r {optional-file-name} This option specifies that the **bebopd** is being restarted. The **bebopd** always saves a restart file (`saved_pct_list`) containing a list of active PCTs when it exits. When **bebopd** is started with this option, it reads in this file and contacts the PCTs for their status. If an *optional-file-name* is given (or the `BEBOPD_RESTART_FILE` variable is defined), the bebopd will look there for the PCT list instead of in the `saved_pct_list` file.

-alternative Every portals process has a portal ID. It is this ID that the portals module on a node uses when dispatching received messages to processes. For testing purposes we may want to run another bebopd on the same node. This argument causes the **bebopd** to request an unused portal ID, rather than it's normal fixed ID, from the portals module. The bebopd will display it's alternative portal ID on startup.

-S {0 or 1} The **bebopd** outputs warnings and errors, and, if the `-D` option is used, status information. The `0` switch turns off all output to stderr. The `1` switch turns it on. By default, the **bebopd** *does not* write to stderr.

-L {0 or 1} The **bebopd** outputs warnings and errors, and, if the `-D` option is used, status information. The `0` switch turns off all output to the log file. The `1` switch turns it on. By default, the **bebopd** *does* write to log file.

-daemon By default the **bebopd** runs as a foreground process, which is useful for debugging the daemon. To run it in the background use this option.

-help This option explains the **bebopd** options.

-PBSsupport -PBSupdate PBS (Portal Batch System) on Cplant requires support from the bebopd. The bebopd is running in PBSsupport mode if it is keeping track of the number of live compute nodes in the machine and policing PBS users to ensure they use no more nodes than they were allocated. The bebopd is running in PBSupdate mode if in addition it sends updates to the PBS server whenever the number of live compute nodes changes. These two arguments can be used to turn on PBSsupport or to turn on PBSupdate. Since PBSupdate implies PBSsupport, turning on PBSupdate automatically turns on PBSsupport.

-PBSinteractive {num-nodes} The **bebopd** can reserve `num-nodes` nodes for interactive use. PBS will not be able to schedule these nodes for batch jobs. Note that the **bebopd** does not reserve a specific set of nodes, but simply reduces the number of nodes available to PBS by this amount.

3.2 Typical use

The **bebopd** is typically started from the `user-env` script with options like these:

```
bebopd -d -PBSupdate  -PBSinteractive 32
```

This command starts the **bebopd** as a background process. It runs in PBSupdate mode, meaning it will allocate to PBS jobs only the number of nodes they have been allocated by the PBS scheduler. In addition the **bebopd** will notify the PBS server whenever there is a change in the total number of active compute nodes. The **bebopd** will leave 32 nodes out of that total, and will permit non-PBS jobs to use up to 32 nodes at a time.

3.3 Interfaces

The **bebopd** accepts message from **PCT**s , **yod** processes, and **pingd** processes. It also responds to SIGUSR1, SIGUSR2 and SIGHUP. The SIGUSR responses are left over from the early days when there was no **pingd** interface to **bebopd**.

There is under development now a new interface so that libraries linked with application codes can contact the **bebopd**. These library calls are intended to support MPI-2 dynamic process creation and will return information about the PBS queues and running jobs. These calls will be described in a separate manual.

The **bebopd** reads the `site` file (see chapter 11) for configuration information.

At the present time, the **bebopd** only communicates with clients that send and receive messages over the Portals interface.

3.3.1 SIGUSR1

Upon receiving a SIGUSR1, the bebopd reports the following information. The information will be logged to `/var/log/cplant` normally. The destination can be changed with the `-S` and `-L` options.

```
bebopd: node 22, system pid 396, portal pid 1
bebopd portals: yod requests 10, pingd requests 12, pct updates 11
PBSsupport: 1, PBSupdate: 1, PBSinteractive: 100
node count: free 226  scheduled 700  interactive 92  active 918
location: answer_ping_requests (top)
```

The first line lists the physical node ID, Linux system process ID, and portal process ID of the **bebopd**. The second line lists the numbers of the portals at which the **bebopd** checks for **yod** messages, **pingd** messages and **PCT** messages. Every incoming portals message must specify a portal process ID and a portal number in order for the portals module to identify the destination of the message.

The next line lists whether the **bebopd** is in PBSsupport mode and PBSupdate mode (`1` means *yes*, `0` means *no*), and how many nodes are set aside for non-PBS jobs. These parameters can be changed while the **bebopd** is running by using **pingd** (see chapter 6).

The fourth line provides a count the number of free compute nodes, the number of nodes running PBS jobs, and the number of nodes running non-PBS (interactive) jobs. It appears that in this case, 100 nodes are reserved for non-PBS use (92 of them are in use) and 918 are allocated for PBS use (with 700 of these in use), leaving a total of 226 free nodes.

The fifth line lists the routine that the **bebopd** was in when it received the SIGUSR1. This information can be useful in debugging the **bebopd**. If the location doesn't change in subsequent displays, it may indicate that the **bebopd** is stuck somewhere.

3.3.2 SIGUSR2

In response to SIGUSR2, the **bebopd** will begin logging extra information to the log file (`/var/log/cplant` relating to it's attempts to find free compute nodes. For non-PBS jobs, the **bebopd** just indicates whether all the queried compute nodes responded. For PBS jobs, the bebopd lists certain counts. This may be helpful when debugging a situation where compute nodes are not responding to status requests.

Sending another SIGUSR2 to the **bebopd** will turn off the extra information.

```
allocate_nodes_for_pbs (8), 1 requests, 128 nodes
   got all expected updates
    1018 queries
    minPct 24 maxPct 1047
    226 free 0 pending 0 already in use by PBS job
    98 unneeded pcts released
```

In this example, the **bebopd** reported that it received a **yod** request for 128 nodes for a PBS job. The **bebopd** send 1018 status queries to **PCT**s and got all expected updates back. It queried active compute nodes in the physical node number range of 24 through 1047. (Not all of those nodes were active compute nodes.) Of those, 226 were available to run a job. The **bebopd** sent a list of 128 of them back to **yod** and sent a message to 98 others that they would not be needed. The **bebopd** also lists a count of pending nodes and nodes already in use by this PBS job. *Pending* nodes are compute nodes that have been allocated to a **yod** process but which haven't yet been contacted by **yod** to run a job.

3.3.3 SIGHUP

In response to a SIGHUP, the bebopd will re-read the site file and close and reopen the cplant log file. See chapter 11 for a description of the site file.

3.4 User log file

The **bebopd** logs Cplant parallel applications to a log file in a format identical to that of ASCI Red. There is one file per day and it is named `run.log.`*mm-dd-yy*. Prior to source code version 1.01 the *mm* (month) field ranged from 0 to 11. Then the **bebopd** fixed to express the the *mm* field from the range of 1 to 12. Hopefully this will not cause much confusion.

The **bebopd** write it's user log file to the same directory to which **yod** writes it's log file. This directory is given by the site file value `LOGFILENAME` and defaults to `/tmp`.

3.5 Error and Warnings

The **bebopd** logs errors and warnings to the file `/var/log/cplant` [1] local to the service node it is running on.

The **bebopd** will exit if it encounters an error setting up or using portals for message passing. It will not exit if it receives a message that fails sanity checks, it will log a warning and ignore it.

[1] or whatever you define to be the target of the local7 facility in the syslog.conf file

Chapter 4

PCT Compute Node Daemon

4.1 Command line arguments

```
pct -nid {node-ID} -pid {portal-ID}
    -D -S {0|1} -L {0|1}
    -daemon
```

-nid {node-ID} -pid {portal-ID} The **PCT** normally registers with the **bebopd** managing the virtual machine. (It is identified in the `cplant-host` file.) If an alternative **bebopd** is running for test purposes in addition to the regular **bebopd**, and you wish this **PCT** to register with the alternative **bebopd**, then provide the node number and portal ID of that **bebopd** with these arguments.

-D This option causes the **PCT** to output information about what it is doing. Repeating the `-D` option on the command line increases the amount of information. Normally the information goes to the log file (/var/log/cplant) on the compute node. This can be changed with the `-S` and `-L` options.

-S {0 or 1} The **PCT** outputs warnings and errors, and, if the `-D` option is used, status information. The `0` switch turns off all output from the **PCT** to stderr. The `1` switch turns it on. By default, the **PCT** *does not* write to stderr.

-L {0 or 1} The **PCT** outputs warnings and errors, and, if the `-D` option is used, status information. The `0` switch turns off all output from the

PCT to the log file. The `1` switch turns it on. By default, the **PCT** *does* write to log file.

-daemon The **PCT** by default runs as a foreground process, which is useful for debugging. To run the **PCT** as a background process, use the `-d` option.

4.2 Typical use

The **PCT** is typically started by the `user-env` script like this:

```
pct -d
```

4.3 Interfaces

The **PCT** responds to messages from the **bebopd**, from **yod** processes, from the application process it is hosting, and from user debugging sessions on the service node. It also responds with helpful information to SIGUSR1, and resets certain parameters in response to SIGHUP.

4.3.1 Incoming messages

The main **PCT** loop processes these messages:

- requests from yod
  - to load a new application
  - to interrupt the application process with SIGTERM or SIGKILL
  - to provide a stack trace of a faulted application process
  - to transition back to free because the parallel application is done
- requests from the bebopd
  - for a status report
  - to interrupt application process with SIGTERM (*pingd -interrupt*)
  - to kill application process with SIGKILL (*pingd -reset*)

  – for the **PCT** to exit

- messages from the user application that was started by the **PCT**requesting node ID map and other data necessary to set up the parallel application
- messages from a user's Totalview debugging session or cgdb debugging sesssion to be forwarded to the debugging session on the compute node

4.3.2 Status updates

The **PCT** sends an update to the **bebopd** at these points:

- at **PCT** start up
- at **PCT** exit
- when the **bebopd** requests a status report
- when the **PCT** receives a SIGHUP
- when the **PCT** begins hosting a new parallel application
- when **yod** releases the **PCT** from hosting the parallel application
- when the **PCT** kills the application process due to a reset request

4.3.3 SIGHUP

If the **PCT** receives a SIGHUP, it will re-read the `site` file, and close and reopen the cplant log file. It will also re-read the `cplant-host` file and send an update message to the **bebopd** listed there.

4.3.4 SIGUSR1

If the **PCT** receives a SIGUSR1, it will log certain information (to `/var/log/cplant`) that may be useful in debugging problems. Here is an example:

```
PCT received SIGUSR1 (30)
was in listen_app_process (process start request)
then in wait_for_app (top)
now in bebopd_request_check ()
```

```
PCT status 3
loop counter 805892
attn 0 started 1 done 0
pid 549 ppid 114 child_rc 0 child_term_sig 0
start_time 991104989  end_time 0  bt_size 0
status 0x0000
```

The **PCT** lists the last 3 routines it was in before receiving the signal. In this example the **PCT** was checking for application process messages, then checking for application process termination, then checking for bebopd messages.

It then lists a status code. The code values are:

1 *Free* - the **PCT** is available to run a job.

2 *Pending* - the **PCT** has been allocated to a job but hasn't yet received the initial message from a **yod** process. If it doesn't hear from a **yod** process soon, the **PCT** will transition back to *Free*.

3 *Busy* - the **PCT** is hosting a member of a parallel application.

4 *Down* - the **PCT** has encountered a serious problem and is about to exit.

6 *Trouble* - the **PCT** has encountered a problem that can be fixed by system administrators. It will refuse to load an application process when it is in this state. See `/var/log/cplant` for a description of the problem. (See the description of the `PCT_HEALTH_CHECK` variable to modify the things checked by the **PCT** when determining whether or not there is trouble on the node.)

The loop counter is incremented each time the **PCT** traverses it's main loop. If subsequent displays don't show the counter incrementing, this means the **PCT** is stuck somewhere.

If the **PCT** is busy, the next lines display the state of the application process.

attn - "1" means something just happened with the application (like it terminated, or it requested it's nid map) that the PCT has to handle, but hasn't handled yet.

started 1 if the app process has been forked, 0 if it hasn't

done 1 if the app process has terminated, 0 if it hasn't

pid system PID of the app process

ppid portal PID of the app process

child_rc if done, the exit code of the application process

child_term_sig the terminating signal, if any, of the app process

start time, end time start and end time (in seconds) of app process

bt_size the size in bytes of the stack trace collected if the app process terminated with a signal

status a bit map indicating progress of the application process through the load. The strings describing the bits are listed too. Once the load has completed and the app is running, some of the bits are erased because we don't care anymore how far they got through the load process.

The application status bits of interest are:

0x0002 application process is forked

0x0200 the application process has been sent to user code

0x0400 the application process has terminated

0x0800 the application process encountered an error in startup

0x1000 The **PCT** sent the application process a SIGTERM

0x2000 The **PCT** sent the application process a SIGKILL

These following status bits are set during application startup and cleared after the application procedes to user code. If they are set, then the application may be stuck in it's startup procedure:

0x0001 application load has begun

0x0004 application process has requested it's portal process ID map

0x0008 application process has been sent it's portal process ID map

0x0010 application process has requested it's node ID map

0x0020 application process has been sent it's node ID map

0x0040 application process has requested it's file IO proxy information

0x0080 application process has been sent it's file IO proxy information

0x4000 The application process in startup has sent the PCT it's portal process ID

0x8000 The PCT has sent the application process a list of fyod file servers

0x0100 The startup protocol is done and the application process is ready to procede to user code

Another way to view these status bits is to run *pingd -v*. If the **PCT** is hosting an application, the job status information will be listed.

Chapter 5

yod Parallel Application Launcher

yod is the utility that loads a parallel application onto nodes in the compute partition, serves as an I/O proxy for it's stdout, stderr and stdin streams, and monitors the orderly termination of the application and freeing of the compute nodes.

5.1 Command line arguments

When loading a single executable file and set of command line arguments onto the nodes of the compute partition, list the `executable-path-name` followed by your program arguments on the **yod** command line. Loading more than one executable file or specifying different application command lines is called heterogeneous load. Heterogeneous load is accomplished by specifying the executables and command line arguments in process rank order in a load file. List the `load-file-name` as the argument to **yod**.

```
yod {yod-options} {executable-path-name} {application arguments}

OR

yod {yod-options} {load-file-name}

   General options:
```

```
-batch  -file {file-name}  -help
-interactive  -kill  -list {node-list}
-NOBUF  -quiet  -sz {number-of-nodes}  -vhelp

Debugging options:

-alloc  -attach  -bt
-D  -d {info-type}   -Log
-sleep {where}
-strace {path-name}
-straceoptions {option-list} -stracenodes {rank-list}
-timing

Testing options:

-nid {node-ID} -pid {portal-ID}
-show -xxx
```

5.1.1 General options

Of all these options, most users will only use `-sz` to specify the number of nodes required by the parallel application.

-batch This option informs **yod** that it is not being run interactively, so, for instance, **yod** will not await user responses in certain circumstances.

-file {file-name} When all processes in the parallel application have completed, **yod** displays a one line completion message for each process. This message lists the wall-clock time elapsed from start to finish for the process, and the exit code and terminating signal, if any, for the process. By default the listing goes to stdout, but may be redirected to a file with this option.

-help This option displays a usage message for **yod**.

-interactive This option informs **yod** that it is being run interactively by a living user. This is the default mode. If **yod** is being run by a script, be certain to specify `-batch` on the command line. One difference between interactive mode and batch mode is that if the load fails on one node, interactive mode waits for the user to interrupt **yod** with control-c before cancelling the load on all allocated nodes. Batch mode goes ahead and cancels the load.

-kill When **yod** is run in interactive mode (the default) and a process of a parallel application terminates abnormally, **yod** displays the fact that the process terminated but does not kill the other processes in the job. The user may choose to abort the job by terminating **yod** with control-C.

If the user wishes **yod** to automatically kill the application when one or more processes terminates abnormally, then use the `-kill` option to yod.

-list {node-list} If a *node-list* is provided on the **yod** command line, then the nodes requested will be allocated out of this list. If `-sz` n is specified, then n nodes will be allocated out of the list. If there does not exist n free nodes in the list, **yod** will display an error message. If no `-sz` option is specified, **yod** will assume you want all the nodes in the *node-list*.

A *node-list* is a list of node specifiers separated by commas. A node specifier is a node number or a node range. A node range is specified by two node numbers separated by one or more dots. No white space may be included in the *node-list*. Example: `-l 25..35,112..140,160,165`

-NOBUF **yod** displays it's own messages and also text printed by the parallel application processes while they are running. Normally this combination of buffered (yod's status messages) and unbuffered (application output and yod's error messages) messages appear sensibly on the tty that started yod. But if **yod** was started by an rsh from a remote node, the output appears garbled. The `-NOBUF` option solves this problem by making all **yod** output unbuffered.

-quiet **yod** is quite verbose. It lists many status and error messages as it loads and runs a parallel application. If you wish to have these messages suppressed, run yod with the `-quiet` option.

-sz {nodes} The number of compute nodes required to run the parallel application. One member (process) of the application will run on each node. The default if no node list is specified is `-sz 1`. The default if a node list is specified is the number of nodes in the node list.

-vhelp This option displays a more verbose usage message for **yod**.

5.1.2 Debugging options

These options are useful for debugging parallel applications and for debugging **yod** itself.

-alloc Choosing `-alloc` was useful for compute node debugging before the availability of `cgdb` or `Totalview`. It displays the nodes on which your application has been started and waits for you to press a key before allowing the processes in your parallel application to procede out of system code and into user code. You could at this point log in to a compute node and attach a debugger to your application to catch it before it procedes to *main*. Since users are discouraged from logging into compute nodes, it would be better for you to use `-attach` and cgdb. Also see the `-bt` option of yod.

-attach This option is essentially the same as `-alloc`. It is intended to hold the application processes once they have started executing at an instruction prior to user code (prior to *main*). You can at this point start `cgdb` to attach a debugger to a process. See the `cgdb` man page for more help on debugging compute node processes.

-bt This option will cause **yod** to display a stack trace for user processes that terminate abnormally. **yod** normally displays a completion message for each process in the parallel application, listing the exit code or terminating signal if any. If the completion message indicates that the application process terminated with a signal and you wish to investigate, you may rebuild the application with debugging symbols and re-run it with the `-g` option of **yod**. The PCT will then attach a debugger to the process, collect the stack trace when it faults, and send the stack trace to **yod** for display. This capability may be expanded some day to allow for interactive debugging of node processes from the service node.

-D Turn on debugging of the application load. The steps in the load protocol are displayed as the application load progresses. Application process file IO requests are displayed as **yod** receives them.

-d {info-type} Selective types of information can be displayed by **yod** using this option. Two of the most common requests are `-d io` and `-d iomore` which display I/O requests sent from the application processes to **yod**. The entire list follows.

- **io** Display application IO requests to yod.
- **iomore** Display details of dispatch of IO requests.
- **memory** Display all buffers allocate by yod.
- **load** Display the steps of the load protocol.
- **loadmore** Display the load protocol in great detail.
- **alloc** Display nodes allocated to the job.
- **hetero** Display the heterogeneous load information.
- **pbs** Display the PBS information.
- **environ** Display yod's environment variables.
- **rtl** Display the Fortran run time library messages from compute nodes.
- **progress** Display the progress of the application through load and termination.
- **failure** Display all launch failure information.
- **debug** Display efforts to obtain debugging data for application processes.
- **bebopd** Display yod interactions with bebopd.
- **comm** Display information about portals setup.
- **phase1** Display information only until application starts.
- **phase2** Display information only after application starts.

-Log This option causes the compute node application load protocol steps to be logged to `/var/log/cplant` on the compute node.

-sleep {where} Cplant system debuggers may want to attach a debugger to a Cplant application before it is in user code. This option provides four different points at which the processes can be held for 60 seconds.

1 right after the fork

2 just before the exec

3 right after entering system startup code

4 at the end of system startup code, just before proceeding to *main*

-strace {path-name} Yet another debugging tool. *path-name* should be a directory which is mounted writable on the compute node. This option will cause the **PCT** to run the application process under *strace* which will list all system calls (and their arguments) made by the application process. By default, only the rank 0 process is traced. The strace output goes to a file in directory *path-name*. The file name contains the Cplant job ID and the rank of the process being traced.

-straceoptions {option-list} The **PCT** will invoke *strace* with the options you specify n the quoted string *option-list*. You must use the `-strace` option with this option.

-stracenodes {rank-list} The PCT will invoke *strace* on the processes with the ranks given in the *rank-list*. The format for the *rank-list* is the same as the format for a node list. By default, *strace* is invoked only on the rank 0 process. You must use the `-strace` option with this option.

-timing Interested in how long the different stages of application load are taking? The `-t` option times them and displays the results in seconds. (If our name was **mpirun** instead of **yod** we would display it in minutes!)

5.1.3 Testing options

These may be useful when testing **yod** or running an alternative **bebopd**.

-nid {node-ID} -pid {portal-ID} These arguments will cause **yod** to contact the bebopd on the specified node number and at the specified portal ID rather than the bebopd listed in the `cplant-host` file. This option is only for testing alternative bebopds and should be used only by Cplant developers.

-show Cplant parallel applications are linked with a version string indicating their compatibility with a certain revision of Cplant software. **yod** refuses to load an application if it's version string is wrong. This option shows the version string of the application and the version string **yod** is looking for.

-xxx A secret option that causes **yod** to load an application even if it's version string is incorrect.

5.1.4 Heterogeneous load file

The load file is a text file you create with your favorite text editor. It has two kinds of entries: comments and application members. Comments are lines on which the first text that appears is a pound sign (#). These are ignored by **yod**. The other type of entry lists a member of the parallel application and has this format:

```
{yod-options} {executable-path-name} {application arguments}
```

The only **yod** options accepted in a load file are `-sz` and `-list`.
Example:

```
yod -l 100..200 myLoadFile
```

The contents of `myLoadFile` are listed here:

```
#
# load file to run my computation and parallel vis server
#
-sz 2 -l 500,501 my-vis-code bufsize=2048
-sz 64 my-computational-code
```

In this example, the executable file `my-vis-code` will be loaded on nodes 500 and 501, will be passed the argument `bufsize=2048`, and will be ranks 0 and 1 in the parallel application. The executable file `my-computational-code` will be loaded on 64 free nodes found in the node number list 100 through 200. These processes will have ranks 2 through 65 in the parallel application. MPI users note that the 66 processes described will populate a single MPI_COMM_WORLD on application start up.

If a load file is provided, any size argument given on the **yod** command line is ignored. If there is no node list given in the load file for a member, then the node list given on the **yod** command line will be used. If in addition there is no node list given on the **yod** command line, then the requested nodes will be allocated from anywhere among the general collection of free nodes. If there is no size argument provided in the load file, but a node list is provided, it will be assumed that you want all the nodes in the node list. If there is no size argument provided in the load file and also no node list, it will be assumed that you want one node from anywhere.

5.2 Environment variables

There are a few environment variables that have an impact on application load:

5.2.1 YODRETRYCOUNT

Occasionally a load will fail because a compute node allocated to the parallel application is not working. **yod** will try to obtain a new set of nodes and load again. It will try up to three times. If you want to decrease or increase the number of retries, set the value of the environment variable `YODRETRYCOUNT` to the number of times yod should retry the load.

5.2.2 PATH

If the user does not specify the full path of the executable name, **yod** will search first for the executable in the current working directory. If it is not found, **yod** will use the `PATH` variable in the user's environment to search for the executable.

5.2.3 CPLANT_STDERR and CPLANT_STDOUT

When an application process writes to stdout or stderr, the text is sent to **yod** for display. If there is quite a bit of output, this can have an impact on the performance of the service node, and also on the performance of the parallel application. Users can define these two environment variables to paths on the `enfs` parallel file system and then each node's stdout and stderr stream will be written to a file on that system using parallel file servers.

For example:

```
command>> export CPLANT_STDOUT=/enfs/tmp/lafisk/outputdirectory

command>> export CPLANT_STDERR=/enfs/tmp/lafisk/outputdirectory

command>> yod -sz 4 say_hello

Contacting node allocation daemon...
1 nodes are allocated to your job ID 1814.
Awaiting synchronization of compute nodes before beginning user code.
Application processes begin user code.

 Name           Rank  Node   SPID   Elapsed  Exit  Signal
--------------- ----  ----  -----  --------  ----  ------
 t-17 n-29         0  1047   8749  00:00:01     0   none
 t-17 n-30         1  1047    540  00:00:01     0   none
 t-17 n-31         2  1047   8441  00:00:01     0   none
 t-17 n-32         3  1047    942  00:00:01     0   none

command>> ls -l /enfs/tmp/lafisk/outputdirectory

-rw-r--r--    1 lafisk   200             0 Jun 25 13:40 stderr.job-1813.rank-0
-rw-r--r--    1 lafisk   200             0 Jun 25 13:40 stderr.job-1813.rank-1
-rw-r--r--    1 lafisk   200             0 Jun 25 13:40 stderr.job-1813.rank-2
-rw-r--r--    1 lafisk   200             0 Jun 25 13:40 stderr.job-1813.rank-3
-rw-r--r--    1 lafisk   200             6 Jun 25 13:40 stdout.job-1813.rank-0
-rw-r--r--    1 lafisk   200             6 Jun 25 13:40 stdout.job-1813.rank-1
-rw-r--r--    1 lafisk   200             6 Jun 25 13:40 stdout.job-1813.rank-2
-rw-r--r--    1 lafisk   200             6 Jun 25 13:40 stdout.job-1813.rank-3
```

5.2.4 PBS variables

When **yod** is executed from a PBS job script, there are certain variables defined that are required by the runtime system. If the user does something in their PBS job script like `rsh` to another service node and run `yod` there, they must be sure to set these environment variables in the new shell to the same value they have in the original shell: `PBS_ENVIRONMENT`, `PBS_BATCH`, `PBS_JOBID`, `PBS_NNODES`.

5.3 Typical use

yod has a huge array of options, but in typical use only one or two may be used:

```
yod -sz 64 myParallelCode -ifile /home/me/input.dat
```

The above command requests the `myParallelCode` be run on 64 nodes. The text following the program name contains the program arguments.

```
yod -l 300..363 myParallelCode -ifile /home/me/input.dat
```

The above command requests that `myParallelCode` be run on physical nodes 300 through 363. All 64 nodes must be free to run the job or the load will fail. Note that jobs run under PBS can not choose the specific nodes they run on. Only interactive jobs run without PBS can choose their actual nodes.

```
yod -l 300..1200 -sz 64 myParallelCode -ifile /home/me/input.dat
```

The above command requests that `myParallelCode` be run on any 64 free nodes in the range of physical nodes 300 through 1200. Again, jobs run under PBS can not choose particular nodes or node ranges.

```
yod -sz 64 -f yod.out -bt myParallelCode -ifile /home/me/input.dat
```

The above command requests that `myParallelCode` be run on any 64 free nodes, and that the completion messages go to the file yod.out instead of to stdout. The `-bt` option specifies that the completion messages will contain a stack trace for any node on which the application process terminated with a signal.

5.4 Load errors

There are many different types of errors that may be reported by **yod** when a user attempts to load an application. Some of the errors are detected by **yod**, for example when an executable was compiled with the wrong version of the Cplant libraries. Some of the errors are reported to **yod** by the **PCT**s, such as a compute node failure during the load.

Often when there is a problem in the compute partition, the first symptom of the problem is that an application fails to load. **yod** will attempt to display the compute node or nodes that is failing in the load. You can configure **yod** to send mail to system administrators when a compute node failure is detected. In the `site` file (see chapter 11) define the NOTIFY_LIST to be the list of their email addresses, and be sure to enable mail on the service node.

Some load errors are listed in the user log file written by **yod** when it terminates, and also logged to `/var/log/cplant`. You can check these two files when investigating problems.

5.4.1 Common load errors detected by yod

Some of the more common error messages displayed when **yod** detects a problem are listed here.

```
Unable to find file in current working directory or in your PATH
```

If a fully qualified path name is not given, yod searches first in your current working directory and then along your PATH variable for the program or load file listed on the **yod** command line. This error message results if the file is not found.

```
bad format in load file
```

yod can not parse the load file. If the file named on the **yod** command line is not executable, **yod** assumes it's a load file and that it has the format described in section 5.1.4.

```
Compute nodes requested exceed PBS allocation
```

```
The total number of nodes requested in your load file
exceeds the number of nodes allocated to your job by PBS
```

yod knows how many nodes your PBS job has been allocated. If you request more than that on the **yod** command line or in the load file, **yod** will not load your job.

```
Unable to allocate these nodes to your job.
Allocating them to you would give you more
total nodes than the limit allocated you by PBS.
```

yod contacts the **bebopd** to get compute nodes for the application. The **bebopd** knows how many compute nodes are already in use by the PBS job that started **yod**. If the new nodes would cause the PBS job to have too many compute nodes, **bebopd** will refuse to allocate the nodes and **yod** will display this message.

```
Unable to allocate nodes to your interactive job.  The
compute partition is dedicated completely to PBS jobs
at this point in time.
```

```
Unable to allocate nodes to your job.  The compute partition
is running PBS (scheduled) jobs and has insufficient nodes
reserved for interactive (non-PBS) jobs at this time.
Run "pingd" to learn about nodes reserved for interactive use.
```

If you are running **yod** interactively (not as part of a PBS job), you may see one of these errors. In the first case, there are no compute nodes allocated to non-PBS jobs. In the second case, there are some compute nodes allocated to non-PBS jobs, but there are not enough available to run your job.

```
Unable to allocate the requested nodes.
Insufficient number of free compute nodes.
```

If you are running an interactive (non-PBS) job, this message simply means there are insufficient free nodes to run your job. If your PBS job gets this error, something is out of balance in the runtime system. PBS should not schedule a job unless there are sufficient nodes to run it. The system administrator should at this point investigate the PBS attribute values and queues to determine what is wrong and fix it. See the PBS chapter for more information.

```
Unable to allocate nodes to your PBS batch job.
The node allocator is not supporting PBS scheduling at this
time.
Please contact the system adminstrators.
```

If you see this error, the **bebopd** believes that PBS scheduling is turned off. But if a PBS job requested nodes, PBS must in fact be scheduling jobs. A system administrator must turn on PBS support in the bebopd with the *pingd -PBSupdate on* command, or turn off scheduling in PBS with the `qmgr` command *set server scheduling=false*.

5.4.2 Common load errors reported by the PCTs to yod

Failure of a compute node

When **yod** contacts the **PCT**s it has been allocated, the **PCT**s form a group so they can do scalable collective communication. If one of the **PCT**s fails to execute the protocol, another PCT will report this to **yod**. **yod** normally displays only the first error received from the compute partition. Then it may display a "waiting for" relationship, meaning it displays which PCTs are waiting for which. The PCT at the end of this chain is on a compute node that is malfunctioning, and system administration should log in and try to determine the problem.

```
Failure while hosting application process on node 0 w/ rank 0.
PCT report:  pct group communication failure
Timed out waiting for node 18.
yod error:  load failed, gathering post mortem
Sending RESET messages to compute partition.

Problem report ("->" sending to, "<-" waiting for a message
from):
(job 101) Waiting for node:  0 <- 18 <- 19
```

Above is a sample load error message. **yod** received several messages from **PCT**s indicating that they timed out waiting for another **PCT**. This first message received was from the rank 0 node. **yod** doesn't print the rest of them. However **yod** prints a summary showing that node 0 is waiting for a message from node 18, and node 18 is waiting for a message from node 19. (These are physical node numbers, not process ranks.) It would be good to look at physical node 19 and look for message passing problems. If it repeatedly fails in application loads, remove it from the virtual machine with "pingd -gone".

Failure of application

The application process executes some start up code before entering the user's code. In this code it sets up it's node map and other information needed to function as a parallel application. The application process may encounter an error at this point, in which case one of the following messages will appear. It may be necessary to recompile and relink the code. If this does not fix the problem, then a developer may need to run the application code under a debugger to see if there is a bug in one of the Cplant libraries linked with it.

yod will display one of the following messages, depending on where the application failed.

```
Failure while hosting application process on node...
PCT report:  failure in exec of new process
Unable to run your code on compute node.
Perhaps the executable file is corrupt?
```

```
Failure while hosting application process on node...
PCT report:  application process detected error before user
code
```

```
Failure while hosting application process on node...
PCT report:  application process terminated before user code
App process terminating signal:  SIGSEGV
```

Recall that when the application writes to the stdout stream or does file I/O without using a parallel I/O facility, the operation goes through **yod**. Another error can occur when **yod** encounters an problem processing application I/O. For example if the application process or the compute node it is running on crashed midway through an I/O operation which was being carried out by **yod**, the I/O operation would fail. It may be necessary to kill the parallel application at this point. (That would be the owner's call.) The system administrator should investigate the compute node to determine why the application running on it did not complete the I/O protocol. This is the message displayed in this case:

```
I/O error (node 27 portal ID 4 rank 12):  timeout on message
wait
Interrupt yod to kill the parallel application.
```

5.4.3 Retries upon load failure

When the application load fails because of a problem with one or more compute nodes, **yod** will try again, up to three times. This is because such problems are often intermittent, and retrying has a good chance of success.

To prevent **yod** from retrying the load, you can set the environment variable `YODRETRYCOUNT` to 0 (zero). This is an environment variable that **yod** checks to see how many more times it should retry the load. **yod** decrements the value by one prior to each retry. By default it is set initially to 3.

5.5 User log file

Upon completion of the application, **yod** write a line to a file logging the user's activity. It looks like this:

```
05/30/2001 14:59:31 05/30/2001 14:59:46 00:00:15 0010
lafisk /cplant/test/tshello, Cplant #102 , 0..4,6,16..18,20
```

We show the entry on two lines here for readability, but in the log file it is all on one line.

The fields in order are:

- start date
- start time
- end date
- end time
- duration

- number of nodes
- user ID of job owner
- executable name
- Cplant job ID
- list of nodes on which application ran

If the parallel application were part of a PBS job, the PBS job ID would be listed as well.

When a new virtual machine is set up, you need to determine where the user log file should be written. If the virtual machine has more than one service node, it may be best to have a separate log file for each service node. By default **yod** will try to write log entries to `/tmp/userlog`. You can override this default by creating a `site` file entry named `LOGFILENAME`

Chapter 6

pingd Status Utility

The **pingd** utility contacts the **bebopd** to obtain information about the compute partition, to modify **bebopd** runtime parameters, or to instruct the **bebopd** to remove or reserve compute nodes.

6.1 Command line arguments

```
pingd {pingd-option-list}

   Action options:

      [-fast | -gone | -kill | -reset | -reserve {uid} | -unreserve ]
      -PBSsupport [on | off]  -PBSupdate [on | off]  -PBSinteractive {num-nodes}

   Node specifier options:

      [-job {job-ID} | -pbsjob {job-ID} | -mine | -l {node-list} | -all]

   Display options:

      -help  -xtrahelp  -verbose
      -NoInquire  -summary  -parse

   Testing options:
```

```
-nid {node-ID} -pid {portal-ID}
```

6.1.1 Action options

These options specify the action to be performed by **pingd**.

-fast By default, the bebopd queries all compute nodes for their status before reporting back to **pingd**. For a faster display, `-fast` queries the bebopd for it's most recent update from the compute partition.

-gone It is possible that a **PCT** can terminate without notifying the **bebopd**. Use `-gone` to notify the bebopd that a PCT has disappeared from a node.

-kill System administrators can kill **PCT**s with this option.

-reserve { uid } System adminstrators can reserve a node for a particular user with this option. The argument is either a user name or numeric user ID. The **bebopd** will allow a job running on the node to complete, but will refuse to allocate the node to anyone other than the specified user. To free the node, use the `unreserve` option.

-reset Reset the selected nodes. This option kills the application process, and resets the **PCT** to available status. System administrators can reset any node. Users can only reset nodes running their jobs.

-unreserve Use this option to free a node that has been reserved for a particular user. A job running on the node will not be disturbed.

-PBSsupport [on — off -PBSupdate [on — off] These two options turn on or off PBS support in the running **bebopd**. The bebopd is running in PBSsupport mode if it is keeping track of the number of live compute nodes in the machine and policing PBS users to ensure they use no more nodes than they were allocated. The **bebopd** is running in PBSupdate mode if in addition it sends updates to the PBS server whenever the number of live compute nodes changes. These two arguments can be used to turn on or off PBSsupport and to turn on or off PBSupdate. Since PBSupdate implies PBSsupport, turning on PBSupdate automatically turns on PBSsupport, turning off PBSsupport automatically turns off PBSupdate. If the **bebopd** is already running in PBSupdate mode, the `PBSupdate on` option will cause it to send an update to the PBS server.

When you `reserve` a node you must use care. If PBS is running, the PBS system believes it has use of that node. If PBS has allocated all available nodes to jobs, then a PBS job script may try to run a **yod** job that will require the node you reserved. If the PBS `resources_assigned.size` shows that PBS has not allocated all it's nodes, you can reduce the `resources_available.size` by one. First turn off the `PBSupdate` mode of the **bebopd**. This step will prevent the **bebopd** from correcting the server's `resources_available.size` value.

```
command>> qmgr -c "list server resources_available.size"

Server myri-0.n-4.r-3
        resources_available.size = 917

command>> pingd -PBSupdate off

command>> qmgr -c "set server resources_available.size=916"
```

Later on when you `unreserve` the node, turn the `PBSupdate` mode back on and the server's `resources_available.size` will be updated to the correct value.

```
command>> pingd -PBSupdate on

command>> qmgr -c "list server resources_available.size"

Server myri-0.n-4.r-3
        resources_available.size = 917
```

6.1.2 Node specifier options

These options specify the compute nodes on which the action should be performed.

-all Perform requested action on all nodes in the compute partition. This is the default.

-job {job-ID} Limit the action to compute nodes running Cplant job number *job-ID*.

-list {node-list} Perform the requested action on the specified list of nodes. Node specifiers are delimited by commas. A node specifier is a node number or a node range. A node range is specified by two node numbers separated by one or more dots. No white space may be included in the *node-list*. As a convenience, the node list may be entered without the preceding `-l` option specifier.

-mine Perform function on compute nodes running my jobs only.

-pbsjob {job-ID} Limit the action to compute nodes running PBS job number *job-ID*. A PBS job may start many Cplant jobs.

6.1.3 Display options

These options affect how **pingd** displays results.

-help Display a list of the most common **pingd** options and how to use them.

-NoInquire This option will cause **pingd** to skip it's *are you sure* questions before performing an action. This is useful when running **pingd** from a script.

-parse This option asks **pingd** to display it's output in an alternative format that is easier to parse.

-summary This option asks **pingd** to display only the node count totals at the end of it's display.

-verbose Display extra information about running jobs. You will not see this information about other people's jobs unless you are a very special user.

-xtrahelp Display a list of all **pingd** options and how to use them.

6.1.4 Testing options

System testers may want to use these options.

-nid {node-ID} -pid {portal-ID} These arguments will cause **pingd** to contact the bebopd on the specified node number and at the specified portal ID rather than the bebopd listed in the `cplant-host` file. This option is only for testing alternative bebopds and should probably be used only by Cplant developers.

6.2 Usage Examples

pingd assumes the most restrictive interpretation when nodes are specified. If you provide a list of node numbers, a job ID, and specify `-mine`, **pingd** will perform the operation on the nodes in the list which are running the job specified, if you own it.

To list the current status of all nodes in the compute partition:

```
pingd
```

To list the status most recently reported to the service partition of all nodes in the compute partition (without going out and querying the compute partition):

```
pingd -fast
```

To kill the **PCT** on node 20:

```
pingd -kill -l 20
```

To reset the **PCT**s on nodes 0 through 100 which are running my jobs:

```
pingd -reset -l 0..100 -mine
```

To display much status information about my jobs:

```
pingd -m -v
```

To inform the bebopd that the **PCT**s on nodes 55, 56, 57 and 61 are dead:

```
pingd -gone -l 55..57,61
```

Chapter 7

Debugging parallel applications

There are several options for examining a parallel application's processes at runtime with a debugger or trace utility.

7.1 yod -bt

We say that an application process terminates abnormally if it terminates with a non-zero exit code or if it is terminated with a signal. If a process terminates with a signal, you can use the `-bt` option of **yod** to see a stack trace when the process terminates.

You will detect that an application process has terminated with a signal when **yod** displays a warning to this effect. **yod** only displays the first such termination that occurs. When the application completes, **yod** displays the exit code and terminating signal (if any) for each process (see figure 7.1).

In this example both processes in the two node application experienced a segmentation fault 2 seconds after starting.

If we rerun this test with the `-bt` option we get more information, shown in figure 7.2.

7.2 cgdb - a gdb front end

You can debug a compute node process with gdb (the Gnu debugger). You do this by running `cgdb` on the service node, which forwards your `gdb` commands to the compute node, and displays the `gdb` response to you on the service node.

```
yod -sz 2 /cplant/test/fault
Contacting node allocation daemon...
4 nodes are allocated to your job ID 101.
Awaiting synchronization of compute nodes before beginning user code.
Application processes begin user code.

>>> Terminating signal SIGSEGV on node 1 w/ rank 1<<<
>>> further messages suppressed

 Name            Rank  Node   SPID   Elapsed  Exit  Signal
--------------- ----  ----  -----  --------  ----  ------
 SU-00 n-00        0     0    225  00:00:02     0   11, SIGSEGV
 SU-00 n-01        1     1    225  00:00:02     0   11, SIGSEGV
```

Table 7.1: yod reports an abnormal termination

To use this feature, run **yod** with the `-attach` option. This will display the nodes the application has been allocated and will start a gdb debugging session on each compute node. yod will prompt you before it allows the application processes to procede to user code.

To start a debugging proxy on the service node which can attach to the debugger on a compute node, run `cgdb` with a physical node number from the yod display:

```
cgdb -n 24
```

Now you can enter gdb commands to debug the application process.

TotalView is a commerical graphical parallel debugger available from Etnus.

7.3 Running gdb on the compute node

You can debug the application directly on the compute node by logging in to it and attaching to the process with gdb. We don't recommend this to users, but administrator's may wish to do this from time to time.

If you need to attach to the process at it's very start, there are four points in the application process where you can do this:

- Right after the PCT forks the application process
- After the PCT sets up the application process uid/gid information but before it exec's the application code
- Right after the application code begins execution but before it executes any of it's Cplant startup code.
- After the application process has executed it's Cplant startup code but just before it begins executing the user's code.

Choose the nodes you want to run on by using the `-l` option to **yod**. Then use the `-sleep` option to choose one of the four points at whilch to attach to the code. Then you have 60 seconds in which to log in to the compute node and attach gdb to the process.

7.4 Running the application under strace

The Linux strace utility traces all system calls made by a process. This may be useful in determining why an application process fails. To run an application process under strace, use the `-strace` option to **yod**.

```
yod -bt -sz 2 /cplant/test/fault
Contacting node allocation daemon...
4 nodes are allocated to your job ID 102.
Awaiting synchronization of compute nodes before beginning user code.
Application processes begin user code.

>>> Terminating signal SIGSEGV on node 1 w/ rank 1<<<
>>> further messages suppressed


 Name           Rank  Node   SPID   Elapsed  Exit  Signal
--------------- ----  ----  -----  --------  ----  ------
 SU-00 n-00        0     0    228  00:00:07     0   11, SIGSEGV

GNU gdb 4.17
Copyright 1998 Free Software Foundation, Inc.
Program received signal SIGSEGV, Segmentation fault.
0x120001c1c in segv (x=0x4) at ../fault.c:29
#0  0x120001c1c in segv (x=0x4) at ../fault.c:29
#1  0x120001b8c in main () at ../fault.c:16
Program terminated with signal SIGSEGV, Segmentation fault.
The program no longer exists.
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
 SU-00 n-01        1     1    228  00:00:08     0   11, SIGSEGV

GNU gdb 4.17
Copyright 1998 Free Software Foundation, Inc.
Program received signal SIGSEGV, Segmentation fault.
0x120001c1c in segv (x=0x4) at ../fault.c:29
#0  0x120001c1c in segv (x=0x4) at ../fault.c:29
#1  0x120001b8c in main () at ../fault.c:16
Program terminated with signal SIGSEGV, Segmentation fault.
The program no longer exists.
```

Table 7.2: yod reports an abnormal termination with stack trace information

Chapter 8

PBS for Cplant

PBS (Portable Batch System, [2]) is an open source system of daemons and utilities that may be used to schedule jobs on a parallel machine or distributed computing resource. We have made several modifications to PBS and some of these are described at [1]. We have also made changes to the Cplant runtime components to integrate them into PBS.

To run Cplant jobs under PBS, users generally write a script that runs **yod** to launch their application. They submit the script to PBS, and PBS schedules it to run at a later time. When they submit the script they also request a certain number of nodes and the length of time they wish to use the nodes. Their job script can run **yod** many times, in succession or in parallel. But at no time may their job script use more total nodes than they requested at submission time. Complete instructions for using PBS are given at [1].

This chapter describes the PBS components, and the enhancements to Cplant that support PBS. It also describes the PBS configuration and log files that are of use in configuring and trouble shooting PBS. Some common runtime problems are described as well.

The details of running PBS are described in the PBS Administrator's Guide and in the PBS External Reference Specification, both available at the PBS web site [2]. (The Administrator's Guide can be downloaded directly. The ERS must be built from the PBS source tree available at [2].)

8.1 PBS components

The software components of a PBS system are:

the PBS server The PBS server (`pbs_server`) is the heart of the PBS system. A single server runs on one of the Cplant service nodes in a virtual machine. The server's behavior is governed by *attributes*, which may be set by authorized users with the `qmgr` program. Users submit requests to the server to add jobs to a queue, delete jobs, alter jobs, and so on.

the PBS scheduler The PBS scheduler (`pbs_sched`) is a single daemon process that evaluates jobs in the queues and selects those that will run next. PBS sites that wish to alter their scheduling policy may do so by rewriting the scheduler.

the PBS MOM processes The PBS MOM (machine oriented miniserver) process (`pbs_mom`) is the process that starts the user's job script and ensures that it completes within it's allotted time. We run one PBS MOM on each service node that we wish to run job scripts on.

PBS client programs To submit a job to a queue, users run `qsub`. To check the status of jobs in a queue, users run `qstat`. To change the operating parameters (called *attributes*) of the PBS server, administrators run `qmgr`. These are all client programs that contact the PBS server for help. A complete list of these may be found at [1].

The life-cycle of a PBS job through these components looks like this:

- A user runs `qsub` to submit a job script to PBS. The script contains commands to run **yod** to start Cplant parallel applications. The user requests a number of nodes (the *size* request) for a duration of time (the *walltime* request), either on the `qsub` command line or as directives within the job script file.
- The PBS server adds the job to a queue.
- The PBS server sends a message to the PBS scheduler asking it to evaluate the queue. It does this everytime a job is submitted, every time a job completes, and whenever a certain interval of time elapses

(defined as the `scheduler_iteration` attribute of the server). It will also initiate a scheduling cycle whenever an administrator enters the `qmgr` command *set server scheduling=true*.

- The scheduler requests all the queue information from the server, and requests the server's attribute values as well. The server's attributes provide information such as how many compute nodes are available to PBS, how many have been allocated, the maximum number of jobs any user can have running at one time, and so on.

- If possible, the scheduler chooses jobs to run. It also chooses the service node on which each job should run. It does this by contacting the MOM on each node and requesting the system load of the service node. The scheduler assigns the PBS job to the least loaded service node.

- The scheduler sends the server a message telling it which jobs to run and what service nodes to run them on.

- For each job the scheduler wishes to place into execution, the server contacts the MOM on the service node and tells it to run the job.

- The MOM process starts the PBS job. It redirects the job's stdout and stderr streams to files in /tmp/pbs/working/spool.

- The MOM monitors the job for termination. When it terminates, the MOM copies the stdout and stderr files to the directory from which the user submitted the job. Then the MOM sends an obituary to the server.

- If the PBS job does not terminate within it's allotted time, the MOM sends a SIGTERM to it's parallel applications. After awhile it sends the parallel applications a SIGKILL and then kills the job script. Then it copies over the stdout and stderr streams, and sends an obituary to the server.

8.2 Cplant runtime components

The **bebopd** node allocator has been enhanced to support PBS operation on Cplant. In particular:

- The **bebopd** node allocator daemon (when run in PBSsupport mode) ensures that **yod** requests from a PBS job do not ask for too many nodes. It determines how many nodes a PBS job is already using before granting more nodes to the **yod** request.

- The **bebopd** knows how many live, working compute nodes there are in the virtual machine. The PBS server has an attribute called `resources_available.size` which it hopes is this value. The **bebopd** (when run in `PBSupdate` mode) sets this value in the server and keeps it up to date when it changes.

- The **bebopd** can be configured to reserve some number of nodes for non-PBS jobs. See the `PBSinteractive` options of **bebopd** and **pingd**. Users can acquire these nodes for their application simply by running **yod** on a service node.

Note that the **bebopd** does not reserve specific nodes for non-PBS jobs. It just ensures that the non-PBS jobs are not using more than the specified number of nodes. Also, PBS does not allocate specific nodes to PBS jobs, it allocates them some number of nodes. The **bebopd** selects the particular nodes that will be allocated to **yod** jobs started by a PBS job.

pingd has been enhanced as well:

- The **pingd** utility has been enhanced to list the PBS job ID (if any) of the Cplant jobs it displays. Each PBS job may start more than one Cplant job.

- **pingd** may also be used to alter the bebopd's support of PBS while the **bebopd** is running. See the `-PBSsupport`, `-PBSupdate`, and `-PBSinteractive` options of **pingd**.

- **pingd** also allows you to specify a PBS job ID on it's command line. It will list every Cplant job that is part of that PBS job.

- The summary information provided by **pingd** mentions how many nodes are running PBS jobs, how many are running interactive jobs, and how many are reserved for interactive jobs.

8.3 PBS runtime configuration files and logging

A typical Cplant PBS set-up will have the PBS server and scheduler running on one service node, and one PBS MOM on each of several other service nodes. The **bebopd** node allocator will be on still another service node.

A small Cplant may have just one service node and can run server, scheduler, MOM and **bebopd** all on the same service node.

In any case, the PBS components (both daemons and client programs) and a **bebopd** running in `PBSupdate` mode require a runtime directory of PBS configuration files. The runtime configuration files are identical on each node. In addition, the PBS deamons log information into this directory, so on their nodes the runtime directory must be private, writable storage that persists after reboots.

When you build the PBS codes and install them they will create a template of this directory on your build machine. You must then edit certain configuration files to have the specific hostnames and pathnames for your cluster. Complete instructions are in the README file in the Cplant source code repository [3]. We suggest copying the template directory to a location readable by all service nodes. Then at startup time if the runtime directory is not set up on a node, a startup script (/cplant/etc/pbs-env) can create it from the template.

You should know where this runtime directory is on the service node. When you encounter a problem with PBS, the log files in this directory may help you to diagnose and solve the problem.

We typically place the runtime directory at `/tmp/pbs/working` on the service node. This is on a local disk, if we have one, or is a link to a separate directory for each service node on a file server if we have no local disk.

8.3.1 Log files useful for troubleshooting

The files listed here are all text files that may be useful when troubleshooting problems:

- Written by the PBS server
  - `/tmp/pbs/working/server_priv/accounting` Each PBS job is logged here when it is queued, when it starts and when it exits.

  - `/tmp/pbs/working/server_logs/yyyymmdd` The server logs every status request it receives, job submissions, interactions with the scheduler, interactions with MOMs, and so on. It also logs problems communicating with other service nodes.

- Written by a PBS MOM
  - `/tmp/pbs/working/mom_logs/yyyymmdd` The MOM logs every PBS job that it starts, every job that it kills for being over it's walltime limit, and every normal job termination.
  - `.e and .o files` When a PBS job completes, the MOM copies the script's stdout and stderr stream to files in the submission directory. While the job is still running, these files may be found in `/tmp/pbs/working/spool`.

- Written by the PBS scheduler
  - `/tmp/pbs/working/sched_logs/yyyymmdd` The scheduler lists the jobs it is scheduling to run. We enhanced the scheduler also to list jobs it chooses not to run and the reason it does not run them. The scheduler also lists service nodes it has trouble communicating with.

- Written by yod
  - `/tmp/userlog` A single line is written to this file by **yod** when a load terminates with an error or when an application completes. It lists the nodes the application ran on, it's start and end time, it's Cplant job ID and PBS job ID, and any abnormal termination of the application processes.
  - `/var/log/cplant` Warnings and exceptions that occur during application load are logged by **yod** to this file. You must have `syslogd` running on the service node to enable logging.

8.3.2 Configuration files

The entries in the configuration files list nodes that PBS daemons will accept messages from, determine how the MOM copies the `.e` and `.o` files to the user's submission directory, and so on. Certain problems that occur at run-time may be due to incorrect or missing entries. Sample files and a detailed README file are in the source code repository [3]. The files you would most commonly need to check are these:

- `/tmp/pbs/working/mom_priv/config`
  - The `$clienthost` directives in this file should list every node on which a PBS server, scheduler or MOM is running. These are the nodes that the MOM will accept messages from.
  - The `$usecp` directives should contain the pathname prefix for all user directories that are mounted on the service nodes. If a directory is not listed here, the MOM will try to use PBS's own version of `rcp` (rather than `cp`) to copy the user's `.e` and `.o` files back to their submission directory. The `pbs_rcp` command may fail if certain permissions are not set correctly, and also may take longer than `cp`.

- `/tmp/pbs/working/sched_priv/usage`

  This is the file in which the scheduler keeps track of the individual users' usage of the machine. It uses these figures in deciding whose job to schedule next. On a new system, the `usage` file must be created (0 length, permission 644) or the scheduler will report an error.

- `/tmp/pbs/working/server_priv/nodes`

  This file should list every node on which a PBS MOM will run, followed by the modifier `:ts` indicating that it is *timeshared*, i.e. more than one PBS job may run on the node at any given time. Warning: *a blank line at the end of this file will cause the PBS server to exit with an error.*

- `/tmp/pbs/working/PBSserver`

  This file names the node on which the PBS server is running.

8.4 Starting and stopping PBS

We have supplied a script (`pbs-env`) that can be used to stop, start and restart the PBS daemons. It is normally found at `/cplant/etc/pbs-env` on the service node. See the comments at the top of the script for information on how to use the script.

Here are some notes about stopping, starting and restarting:

8.4.1 pbs_server

The first time you *start* the PBS server on a node, you start it with the option `-t create`. This tells the server there is no PBS server attribute database yet. Once PBS is started you use `qmgr` to set the PBS attributes. There is a sample file `qmgr-setup` in the source code repository [3]. It shows the sort of attributes that must be set before you start using PBS. For example, you must enter the name of all users allowed to change the PBS attributes, and you must enter the hostnames of all machines allows to send requests to the server. The PBS Adminstrator's Guide explains the use of server attributes, and the PBS External Reference Specification lists all attributes.

The PBS server creates the `serverdb` file in the `server_priv` directory of the runtime directory. This file contains the server attributes.

Subsequently when you start the server, you start it with no options. The server will look for the `serverdb` file and read it's attributes from there.

If you wish to *restart* the PBS server, you must be cautious. To restart the server in an orderly way, first stop it using the `qterm` command:

```
qterm -t quick
```

The `qterm` command shuts down the server. But by default it kills all running PBS jobs first. You must use the `-quick` option if you wish PBS to leave running jobs alone.

If `qterm` fails, you can use `kill -9` to kill the server without harming running jobs, but it is much prefered to use `qterm`.

Once you are ready to restart the server, just start `pbs_server` with no options.

8.4.2 pbs_mom

The PBS MOM process is the parent process of the PBS jobs on it's service node. You can kill the `pbs_mom` and the PBS jobs will continue executing. But when you restart the MOM you must start it with an argument that tells it to monitor the running PBS jobs. Otherwise it will kill them.

To stop the `pbs_mom`:

```
kill -2 {pbs_mom process ID}
```

To restart the MOM so it will monitor the running PBS jobs:

```
pbs_mom -p
```

8.4.3 pbs_sched

You can kill and restart the PBS scheduler at any time. You should kill it with a signal it can catch, like `-2`. There are no special arguments needed to safely restart the scheduler.

8.5 Common runtime problems and solutions

Because it is a network of communicating daemons with dependency on the Cplant runtime system, the PBS system can find itself out of balance. A node may die while the MOM is communicating with the server, or an administrator may inadvertantly change the `resources_available.size` attribute of the server while using `qmgr`. An NFS file system that the MOM needs to copy user files to may be unavailable, hanging the MOM daemon. Any number of mishaps can cause a problem.

When trouble shooting a problem, you should look in the log files of the server, MOMs and scheduler. Each writes one log file per day, and they are located in `/tmp/pbs/working/server_logs`, `/tmp/pbs/working/mom_logs`, and `/tmp/pbs/working/sched_logs` respectively.

If you are tracing a problem with a certain job, look first in the server's accounting file to determine which service node it ran on. The accounting files (one per day) may be found in `/tmp/pbs/working/server_priv/accounting`. Look for the `exec_host` field in the "S" or "E" record for the job. (These are records written by the server when the job is scheduled to run and when it exits.) Then go to that node and look for entries for the job in the MOMs log file.

This section describes some of the common problems and how to solve them.

8.5.1 Clients can't reach server

You may notice that the **bebopd** does not seem to be updating the `resources_available.size` attribute of the server. Perhaps this is because the `qmgr` client it is invoking can't reach the server. (Incidentally, the **bebopd** should log this problem in `/var/log/cplant`). Or a user may tell you that `qsub` is failing to contact the server. These are examples of PBS clients that can not contact the PBS server.

Verify that the server is running and you can ping it's node on the network being used by PBS. Check the server's log file to see if it received and then rejected the request, or never received it at all.

If the server never received the request, check the `/tmp/pbs/working/PBSserver` file on the client node. This is the file that clients (and the scheduler and MOMs) look in to find the hostname of the server. Verify that the hostname in this file is the hostname on which the server is running. You may need to modify the name (add or delete a domain name, for example) to get it to work.

The next thing to check is that the `acl_hosts` attribute of the PBS server includes the client node. The PBS server will only accept messages for hosts in this list. If the hostname does appear on the list, make sure that it exactly matches the hostname the server got from the incoming connection. (The server writes to it's log file the name of the incoming host of the request that it rejected.) You may need to add or delete a domain name, for example, from the hostname in the acl list.

In the case of the **bebopd** invoking `qmgr`, verify that `root` on the bebopd node is listed as a `manager` in the PBS server attributes. Only managers can set resource information.

8.5.2 Node counts out of balance

Count of active compute nodes

The PBS server believes it's `resources_available.size` attribute is the total number of compute nodes available for use by PBS jobs. This value is set and updated by the **bebopd**. If it is incorrect, PBS may schedule too many jobs to run, or may not schedule jobs when there are nodes available. To verify the count is correct run both `qmgr` and `pingd` and compare the numbers as follows:

```
command>> qmgr -c "l s resources_available.size"

Server myri-0.n-4.r-3
        resources_available.size = 917

command>> pingd -s
Awaiting status from bebopd...
Awaiting pct list from bebopd

Total: 1018
Total busy: 907
Total free: 110
Total nodes unavailable: 1

Compute nodes are being scheduled by PBS, but 100
nodes are currently reserved for non-PBS interactive use.

Nodes currently hosting PBS jobs:         865
Nodes currently hosting interactive jobs: 43

Free nodes remaining for interactive jobs: 57
```

The `qmgr` request shows that PBS thinks there are 917 compute nodes at it's disposal. The `pingd` command shows that there are 1017 healthy nodes (*Total* minus *unavailable*) , but 100 are reserved for non-PBS use. So the PBS total is correct.

If the total is incorrect, you can force the **bebopd** to update the PBS server again by entering this command:

```
pingd -PBSupdate on
```

Normally the **bebopd** only updates the server when the total changes. So, for example, if that unavailable node came back online, the **bebopd** would invoke `qmgr` to update the server's total.

Count of assigned nodes

The PBS server also maintains a count of the number of nodes it has assigned to jobs. This is the value of the server's `resources_assigned.size` attribute. There are two ways this can get out of balance.

First, if you run the PBS command `qstat -a` it lists a total of assigned nodes at the end of the display. This should match the server's `resources_assigned.size` value.

```
command>> qstat -a

myri-0.n-4.r-3:
                                                    Time In Req'd  Req'd   Elap
Job ID Username Queue    Jobname    SessID  Queue  Nodes  Time  S Time
------ -------- -------- ---------- ------ ------- ------ ----- - -----
64     jrobbin  dedicate fine_rb3       --  117:32    700 100:0 Q   --
137    gsgrest  default  rrr10         641  072:35      8 96:00 R 72:35
139    gsgrest  default  runbead2      736  072:31    150 100:0 R 72:31
140    gsgrest  default  runpe-400     681  072:29     64 100:0 R 72:29
176    mechand  default  C6.33sh1     1985  064:17    128 40:00 R 64:17
200    swsides  default  ea          30304  007:23     16 40:00 R 07:13

Total compute nodes allocated: 366


command>>  qmgr -c "l s resources_assigned.size"
Server myri-0.n-4.r-3
        resources_assigned.size = 366
```

Here, the total number of node assigned to running job's is equal to the PBS server's `resources_assigned.size`.

If they are not equal, you should restart the PBS server. This usually fixes the imbalance.

Secondly, the PBS server's count of assigned nodes should never be less than the number of nodes actually in use by PBS jobs. The PBS jobs may be using fewer nodes than they have been assigned, but they should never be using more. You can determine how many nodes are actually in use with `pingd`.

```
command>> pingd -s
Awaiting status from bebopd...
Awaiting pct list from bebopd
```

```
Total: 1018
Total busy: 411
Total free: 605
Total not responding to ping (try again): 2

Compute nodes are being scheduled by PBS, but 100
nodes are currently reserved for non-PBS interactive use.

Nodes currently hosting PBS jobs:         366
Nodes currently hosting interactive jobs: 45

Free nodes remaining for interactive jobs: 55
```

`pingd` shows the 366 nodes are in use by PBS jobs. This is correct. The *Nodes currently hosting PBS jobs* must by no greater than the PBS server's `resources_assigned.size`.

If there are more nodes in actual use than the PBS server has assigned, then there is a PBS job running that the PBS server believes has exited. (This may happen when the network fails when the MOM is trying to kill the job. It notifies the PBS server that the job has been killed when in fact it has not.) To determine which job or jobs this is, run `pingd` and match the PBS job IDs displayed there with the running jobs listed in `qstat`. You should find a job or jobs listed by `pingd` that is not in `qstat`. Remove the job with `pingd`.

There is a useful script called `check_for_invalid_pbs_jobs` in the Cplant source tree under `support/cplant/sbin`.

8.5.3 Can't delete a job that has completed

Occasionally a job is listed as running by `qstat` but it has actually completed, and attempts to delete the job from the PBS system with the command `qdel` fail. This can happen when the MOM catches the termination of the job, but the network fails when it sends the obituary to the server. When you run `qdel` to delete it, the PBS server sends a command to the MOM to delete the job, but the MOM refuses because it believes it has already done that. So the PBS server refuses to delete the job from it's records.

This situation can also arise when the MOM on the node that had been hosting the job is no longer running, or when the MOM's node is unreachable. PBS can't delete the job because it can't reach the MOM.

To fix this situation remove all files and directories related to the job that were maintained by the PBS server and the MOM daemon. It is polite to send the stdout and stderr streams to the user if you find them. Then restart the PBS server.

Suppose for example that PBS job number 64 has completed, but `qstat` shows it is still running. Then follow these steps to remove the job:

- Search the MOM node that ran the job for undelivered stdout and stderr streams and mail these to the job owner, then delete them. (These would be in either the `undelivered` or the `spool` directory.)
- Search the MOM node for directories set up for the job and delete them.
- Delete the files representing the job in the PBS server's runtime directory.
- Restart the server.

On the MOM's node (the service node on which the job ran):

```
command>> cd /tmp/pbs/working/undelivered
                     OR
          cd /tmp/pbs/working/spool

command>> ls
275.myri-.ER  323.myri-.OU  503.myri-.OU    64.myri-15.ER
275.myri-.OU  547.myri-.ER  1180.myri-1.ER  64.myri-15.OU
323.myri-.ER  547.myri-.OU  1180.myri-1.OU

command>> mail JoeUser -s "your pbs output" < 64.myri-15.ER
command>> mail JoeUser -s "your pbs output" < 64.myri-15.OU
command>> rm 64.*

command>> cd /tmp/pbs/working/mom_priv/jobs
command>> ls
```

```
820.myri-.JB   881.myri-.SC*  64.myri-.TK/
820.myri-.SC*  881.myri-.TK/  820.myri-.TK/
881.myri-.JB   64.myri-.SC*   64.myri-.JB

command>> rm -rf 64.*
```

On the server's node:

```
command>> cd /tmp/pbs/working/server_priv/jobs

command>> ls -l
total 92
-rw-------    1 root     root         4577 Jun  1 09:18 820.myri-0..JB
-rw-------    1 root     root          284 Jun  1 09:18 820.myri-0..SC
-rw-------    1 root     root         4598 Jun  1 09:22 881.myri-0..JB
-rw-------    1 root     root          182 Jun  1 09:22 881.myri-0..SC
-rw-------    1 root     root         3911 May 30 12:21 64.myri-0.n.JB
-rw-------    1 root     root          462 May 30 12:21 64.myri-0.n.SC

command>> rm 64.*

command> /cplant/etc/pbs-env restart server

-->> Executing /cplant/etc/pbs-env  restart server
restarting pbs_server, please wait
-->> Done with /cplant/etc/pbs-env restart
```

Note that the fact that a PBS job is not using any compute nodes does not mean it has completed. The PBS job is a job script, and the script may be doing other work at the moment like copying files. You can tell a job has *not* completed if there is a process running on the service node it ran on that has the PBS job ID in it's name. The job *has* completed if you find entries in the MOM's log file that indicate that the job terminated.

This is what job termination looks like in the MOM's log file:

```
Job;177.myri-0.n-4.r-3;task 1 terminated
Job;177.myri-0.n-4.r-3;Terminated
```

```
Job;177.myri-0.n-4.r-3;/cplant/sbin/pingd -reset -pbs 177
Job;177.myri-0.n-4.r-3;kill_job
Job;177.myri-0.n-4.r-3;Obit sent
```

The log entries show that the MOM caught the termination of PBS job number 177, ran `pingd -reset` to ensure that no parallel applications were left running by the job script, and sent an obituary to the PBS server.

8.5.4 Nodes are free but scheduler won't run any jobs

Another mysterious problem that occurs at times is that PBS jobs waiting for nodes are not being scheduled to run, even though there are sufficient nodes to run the jobs.

First verify that the system is not being drained so that a starving job can run. `qstat -s` will display a message under a job if it is being held to free up nodes for a starving job. (A starving job is a job that has been waiting for more than 24 hours. PBS will refuse to run any jobs until it has sufficient nodes to run the starving job.)

Next you should force a scheduling cycle with a `qmgr` command:

```
command>> qmgr
Max open servers: 4
Qmgr: set server scheduling=true
Qmgr: quit
```

If `qstat` reveals that the jobs are still not being scheduled to run, then look at the scheduler's log file (in `/tmp/pbs/working/sched_logs`) for the scheduler's explanation of why it is not running the job.

It may also help to check the server's totals, since the scheduler will not try to run jobs unless the `resources_available.size` less the `resources_assigned.size` is large enough to run at least one of the jobs.

```
command>> qmgr -c "list server resources_available.size"

Server myri-0.n-4.r-3
        resources_available.size = 918

command>> qmgr -c "list server resources_assigned.size"
```

```
Server myri-0.n-4.r-3
        resources_assigned.size = 603
```

If it's `resources_available.size` is too low, force an update with `pingd -PBSupdate on`. If it's `resources_assigned.size` is too high, restart the server so it will read it's database and recalculate this value.

If the scheduler is in fact scheduling jobs to run, look in the server's log file. The server may indicate that the MOMs are refusing to run the jobs. In this case, try restarting the MOMs, and then restarting the server. If it is only one MOM that is refusing to run jobs, then take that MOM offline with a `qmgr` command.

```
command>> qmgr
Max open servers: 4
Qmgr: set node myri-0.n-2.r-3 state=offline
Qmgr: quit

command>> pbsnodes -a

myri-0.n-0.r-3
     state = free
     np = 1
     ntype = time-shared

myri-0.n-2.r-3
     state = offline
     np = 1
     ntype = time-shared
```

8.5.5 User doesn't get their output files

While a PBS job is active, it's stdout and stderr streams are saved in a file on the MOM's node. These two files are in `/tmp/pbs/working/spool`. When the job completes, the MOM copies the files into the directory from which the PBS job was submitted.

If the directory pathname prefix is listed in a `usecp` command in the MOM's `config` file, the MOM will use `cp` to move the files over. If the pathname is not listed in a `usecp` command, the MOM will use a PBS remote copy command called `pbs_rcp`. It is preferable to use `cp`, since there are many reasons why a *pbs_rcp* command may fail. For this reason, all file systems that host users directories and that are mounted on all service nodes should be listed in the MOM's `config` file. (The `config` file is located in `/tmp/pbs/working/mom_priv` and is described in the our PBS README file.)

If for some reason the MOM can not copy over the output files, it copies them instead to the directory `/tmp/pbs/working/undelivered`. You can recover these and send them to the user.

8.5.6 User doesn't know why their job exited early

Occasionally a user will report that their job exited before completing and they don't know why. There are several possible reasons for this:

- Their job ran out of time. Look in the server's accounting file to see which service node their job ran on. Then look in the MOM's log file on that node. If the MOM killed the job because it ran over it's walltime allocation, you will see a message like `walltime used (24:30:52) exceeds limit (24:00:00)`. There is also a message in their `.e` file telling them that PBS killed their job.

- One of their processes exited with an error or a signal. They should be able to determine this by looking at their `.o` and `.e` files, since **yod** will report such an error. Alternatively, look in the server's accounting file to determine what node their job ran on, then look in the `userlog` file written by **yod** for an entry about their job. The entry will indicate if one of their processes terminated with an error code or a signal.

- The MOM refused to run the job that was handed to it by the server. Again, look in the server's accounting file to determine which node that job ran on. Then look in the MOM's log file on that node. If the MOM refused to run the job, it will log that fact to the file, along with a reason. This happens rarely, and if it does the best course of action is to restart the server and all the MOMs.

8.6 Building and installing PBS on Cplant

The README file in the source code repository [3] explains how to build and install PBS on a new machine. It is copied here for convenience.

```
Introduction:
-------------
This directory contains

  o cplantPatch       - a patch file of Cplant changes for PBS
  o obj - a directory containing two scripts to configure a directory
  for building and installing PBS
  o configs.onequeue - a directory containing runtime configuration files for
  PBS and a file of commands for setting the initial parameters of the
  running PBS server.

The patches were created for version 2.2, patch level 8 of PBS.  The PBS
source is available for free at www.openpbs.com.

there is a good chance they will work on a later version of PBS.

Building PBS for Cplant:
------------------------
1. Obtain the Open PBS source from www.openpbs.com.  Tar up that source and
   the directory containing this README file and copy it to the
   file system where you are going to build and install PBS.  (We'll refer
   to the directory into which you install these files as PBSTOP.)
   The binaries will be executed from the Cplant service nodes, so
   the file system should be one that is mounted on the service nodes
   or one that gets distributed to the sss0s.  We install our PBS
   executables in /cplant/nfs-root-alpha/sbin and /cplant/nfs-root-alpha/bin
   on the sss1.  We build in /usr/local/pbs on the sss1.  But you could
   also build and install on a file server that is mounted by the service
   nodes.

   Note that the stat structure changed sometime between glibc 2.0.7 and
   glibc 2.1.2, so a PBS server, scheduler or MOM built on one will report
   errors if run on the other.  (Calls to chk_file_sec() will fail.)
   Beware of this if you have different Linux distributions running on
```

```
   service nodes and admin or file server nodes.

2. Uncompress and untar the PBS distribution.

3. Move the directory created by this tar file to a directory called
   "patched".  (Or don't, but we'll call it "patched" from now on.)
   If you believe you may want to modify PBS and subsequently build
   a new Cplant patch file, then copy the entire "patched" directory
   to a directory named "origsrc".  Later on you can create a new
   patch file by diff'ing origsrc and patched.

   The "patched" directory contains a copy of the PBS adminstrator's
   guide.  You may also want to obtain the External Reference Specification
   for more detailed information about PBS.  This can be built from this
   directory, or obtained from http://mrj.pbs.com.

4. Move the cplantPatch to PBSTOP/patched, and apply it in that directory
   with this command:

   "patch -N -p1 -l < cplantPatch"

   If you want to know what all these patches do, there's a description
   toward the end of this README.

5. PBSTOP/patched contains all the source files required to build PBS.
   PBSTOP/obj is the target directory that will contain the object files.
   In PBSTOP/obj edit the scripts configBuild and configInstall to have
   the correct pathnames for your build and installation of PBS
   executables and runtime files.  See the comments in the two
   files for the meaning of the values in the file.

   The pathnames in configBuild (except for SRCDIR) are compiled into the
   code, and so should represent the path name at execution time (from the
   service nodes).  The pathnames in configInstall are used when copying
   the executables to their home, and so should represent the pathname on
   the build machine.

   Also ensure that the directory defined with the "prefix" argument
   exists and is owned by root.
```

```
8. In PBSTOP/obj execute the configBuild script.  PBSTOP/obj is now set
   up to be the target directory.  Type "make" to build the code.

9. In PBSTOP/obj execute the configInstall script.  PBSTOP/obj is now set
   up to copy the compiled PBS codes to the proper location and to build
   a template of the runtime working directory.  Type "make install".

10. The PBS code is built and installed.  Now it is necessary to set up
    the runtime configuration files, and start PBS.

Setting up runtime configuration files:
--------------------------------------

The PBS server, scheduler, MOMs and clients access a runtime directory
for configuration information and logging.  The install process created
a template for this directory.  (It's at the pathname specified by
"set-server-home" in configInstall.)  The configuration files in this
template must be updated with Cplant hostnames and scheduling policy
configuration parameters, and copied into place on the service nodes.
Cplant specific configuration files are in PBSTOP/configs.onequeue.
(This is for the single queue policy.)

I should write a script to do this.  Until then, follow these steps.

All hostnames in the working directory files should be the hostnames to
be used by PBS codes on service nodes when they establish socket connections.
These hostnames will be used by the PBS clients and daemons to establish
connections with each other and to authenticate incoming requests.

1. Copy the MOMs' config file from PBSTOP/config.onequeue/mom_priv/config
to the working directory template.  Set these values in the file:

$clienthost - Host names of nodes on which PBS server, scheduler and other
MOMs are running.

$usecp - Directs the MOM to use "cp" instead of "rcp" when copying files
back to users directories.  List file systems that are shared by
the service nodes on which the MOMs are running.

2. Copy the scheduler's configuration files from
```

PBSTOP/config.onequeue/sched_priv to the sched_priv directory of the working directory template. Edit them as follows:

sched_priv/resource_group - This file should list every PBS user, followed by a unique ID, followed by a parent group, followed by a priority. You can use the user ID from the passwd file for the unique ID, use "root" for everybody's parent group, and 10 for everybody's priority. If all users have the same priority, you may leave the resource_group file empty.

sched_priv/dedicated_time - see the file for how to set up dedicated time periods. It you don't have dedicated time you still need the file.

sched_priv/holidays - lists the prime and non prime time periods, and all holidays for the year. You need the prime and non prime periods even if you are running the "one queue" scheduler, because the scheduler sorts the jobs in the queue differently during prime and non prime time.

sched_priv/sched_config - lists parameters of the scheduling policy. You may be able to copy this straight into the working directory without any changes.

sched_priv/usage - you need a zero length usage file with permission 644.

3. Copy the server's "nodes" file from PBSTOP/config.onequeue/server_priv to the server_priv directory of the working directory template. You need one line for each node on which a PBS MOM daemon will run. Each line should be appended with ":ts" to tell the server it is timeshared. Do not put any blank lines or comments in this file.

4. The PBSserver file should contain the hostname of the node on which the PBS server is running.

5. Decide where you want the runtime configuration files to go. Actually you already decided because it's the set-server-home value in configBuild. It is recommended that each service node have a local working directory. If you can't do this, be certain you built with the CPLANT_REMOTE_WORKING_DIR switch.

6. Now copy the working directory to each of the service nodes on which PBS clients, MOMs, the scheduler or the server will run. This

```
would be to the location named by "set-server-home" in configBuild.  Use
the -p option in your copy to ensure the permissions are maintained.

Running the PBS server for the first time:
------------------------------------------

The first time you run the PBS server, you must run it with the
argument "-t create", which creates a new database for the server
in the working directory.  Then you need to run "qmgr" and set
several operating parameters.  These parameters are listed in the
file PBSTOP/configs.onequeue/qmgr-setup.  See the PBS External
Reference Specification for an explanation of the values in
the file.

It is important to set the default queue's resources_max.size
attribute.  This attribute defines limit on the size of jobs in
this queue.  If users attempt to submit a job that requests more
nodes than this size, "qsub" will display an error and suggest
the possibility that their job belongs in a dedicated time queue.
(Use qmgr command "set queue default resources_max.size = 75".)
You may want to change this value from time to time as the size
of your compute partition changes.

You must be root on the node the server is running on to enter
the first qmgr commands.  There are qmgr commands that set up other
users as PBS managers, and once those are entered, these other users
and users on other hosts may enter qmgr commands.

The PBS server verifies the incoming qmgr configuration requests
by checking the hostname of the source of the command.  If the primary
name on the interface is not the name in the PBSserver file, you will get
get an "Unauthorized Request" error when you attempt to configure
the server with qmgr.  By looking at the server's log file in the
server_logs directory, you can see what hostname the server resolved
the incoming request to.

Making a patch file:
-------------------

If you change the PBS source you need to make a new patch file.  This script
```

```
would create the patch file when the 2.2 PBS distribution is in "origsrc"
and the Cplant version is in "patched".  (See step 3 above in "Building PBS".)

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
#!/bin/bash
#
export TZ=UTC0
export LC_ALL=C
echo "diff -Naurw origsrc patched > cplantPatch"
diff -Naurw origsrc patched > cplantPatch
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

The patches
------------

The Cplant patches to PBS 2.2, patch level 8, accomplish the following:

1. We added a "size" resource to PBS.  It's analygous to the yod
   "-size" argument.  It represents the number of compute nodes
   requested by the job.  PBS does not run on our compute nodes.
   They are an abstract resource managed by PBS with help from
   the Cplant runtime system.  The MOM puts the "size" in the
   PBS job's environment so yod can pass it along to the cplant
   node allocator.  (The "nodes" resource in PBS represents the
   service nodes on which the job scripts run.)

   (server/resc_def_all.c, qstat.c, resmom/start_exec.c, fifo/check.c,
   fifo/fairshare.c, fifo/globals.c, fifo/server_info.c, fifo/sort.c
   fifo/sort.h)

2. Changes to qstat display the "size" and "walltime" requests
   of a job and the total compute nodes in use by jobs.  We also
   display the amount of time the job has been in the queue.
   (qstat.c)


3. MOM, scheduler and clients use our gethostbypeer() instead
   of gethostname() to determine their host name on the network
   connecting them to the other service nodes.  (With multiple
   network interfaces, gethostname() may return the wrong name.)
```

```
   gethostbypeer() uses "ping -R pbs-server-host-name" to find my
   host name on this network.)

   (qsub.c, net_connect.h, prepare_path.c, Libnet/Makefile.in,
   Libnet/gethostbypeer.c, mom_main.c, pbs_sched.c)

4. PBS server gets it's hostname from the file in the runtime working
   directory that lists the hostname of the server, instead of from
   gethostname().  (pbsd_main.c)

5. We skip the check of the the full path name starting at root
   when verifying that runtime files and directories have the
   correct permissions.  (Liblog/chk_file_sec.c)

6. Log file entries contain the physical node number of the reporter.
   (Liblog/pbs_log.c, pbsd_main.c, mom_main.c, pbs_sched.c)

7. If the compile time flag CPLANT_SCHEDULER_PING_NODES is set, the
   scheduler checks to see if node is alive before trying to talk_with_mom.
   (If node crashed after server last queried it, scheduler will hang
   trying to talk to it.)  This time-consuming check is far less likely
   to be needed if CPLANT_NONBLOCKING_CONNECTIONS is on.

   (Libnet/node_is_alive.c, Libnet/Makefile.in, node_info.c)

8. MOM uses "$cplant_grace" directive in it's config file to determine
   how long to wait before killing overlimit jobs.  If directive value
   is negative, overlimit jobs are not killed.  MOM only kills overlimit
   jobs if there are jobs in the queues waiting for nodes.  (Scheduler
   notifies MOMs after each scheduling cycle if there are jobs waiting
   or if there are no jobs waiting.)

   MOMs kill all parallel apps started by a PBS job when the PBS job
   terminates.  It simply invokes the Cplant utility "pingd".

   MOM adds the "size" resource request to PBS job's environment as PBS_NNODES.

   (catch_child.c, linux/mom_mach.c, mom_main.c)

8. PBS codes only try to lock files in runtime working directory if
```

CPLANT_REMOTE_WORKING_DIR is undefined. (mom_main.c, pbs_sched.c, pbsd_main.c)

9. If CPLANT_PRIME_NONPRIME_POLICY is defined, the scheduler only runs jobs from the "prime" queue during prime time, refusing to start jobs that will run past the end of prime time. No user is permitted to have running jobs that requested more than N/2 node hours total, where N is the number of compute nodes in the machine. During nonprime time, the scheduler runs jobs from both the prime and nonprime queue, with higher priority given to jobs in the nonprime queue. (Queues need to be setup with priorities this way.) Nonprime jobs will not be started if they would extend into prime time. If this value is not defined, the scheduler ignores prime/nonprime boundaries when scheduling jobs whose walltime exceeds the boundary, and the scheduler runs jobs from all queues. (So use sched_config values and queue settings to get desired policy.)

   If CPLANT_RUN_ON_SUBMITTING_NODE is defined, the scheduler will try to run the job on the node from which the job was submitted, otherwise it will run it on the least loaded service node.

   We added two sorts to sort in increasing and decreasing order by the job's "size" resource request.

   The scheduler sends a message to all MOMs at the end of each scheduling cycle to inform them of whether or not there are jobs in the queues that are waiting to run but can't because of a lack of compute nodes. We hacked the Libnet/rm.c facility to do this.

   (check.c, config.h, constant.h, data_types.h, fairshare.c, globals.c, fifo.c, globals.h, job_info.c, node_info.c, node_info.h, prime.c, queue_info.c, server_info.c, sort.h, sort.c, node_manager.c, Libnet/rm.c)

10. If the scheduler receives a SIGHUP, it will re-read all of it's configuration files. (It will do all the actions it does on receiving a SCH_CONFIGURE command.) (pbs_sched.c)

11. If qsub rejects a request because of a resource request that is not valid for the queue, information is displayed about how to find the resource limits on the default and dedicated time queues. (qsub.c)

```
12. If CPLANT_NONBLOCKING_CONNECTIONS is defined at compile time, the
   server will open non-blocking sockets to talk to moms and scheduler and
   will time out and log a warning if the it can't reach the process.  (The
   mom's use this code also to talk to other moms.)  Server will check
   a node's status before attempting connection. (pbs_nodes.h,
   Libnet/net_client.c, server/node_func.c, server/run_sched.c,
   server/svr_connect.c)

13. When mom is started with the "-p" option, it will add the jobs
   already running on the node when it started to it's list of jobs
   to poll.  (resmom/catch_child.c)

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

The compile switches
--------------------

CPLANT_SERVICE_NODE - general cplant patches for "size" resource,
   use of gethostbypeer, scheduler interface to MOMs for notification
   of waiting jobs, MOMs killing of jobs that go over walltime
   request if there are jobs waiting in the queue, MOMs killing of
   parallel applications when PBS job completes


CPLANT_PRIME_NONPRIME_POLICY - Define this and rebuild the scheduler
   to turn on Cplant's prime/non-prime policy.  During prime time,
   the scheduler will only run jobs from the "prime" queue.  It
   will not allow any user to have running jobs whose node-hour
   total exceed N/2, where N is the number of nodes available in
   the machine.  It will not start jobs from the "prime" queue
   whose walltime request would extend past the prime time period.
   During non-prime time, the scheduler will run jobs from the
   prime and non-prime queues, with higher priority going to jobs
   in the non-prime queue.  Jobs in the non-prime queue will not
   be started if their walltime request would run past the end
   of non-prime time.

   If this is not turned on, all queues are evaluated during prime
   and nonprime time.  The settings in the scheduler sched_config
```

```
   file determine the priority given to the jobs in the queues
   during prime and nonprime time.  We don't care about prime/nonprime
   time boundaries.

CPLANT_RUN_ON_SUBMITTING_HOST - If this is defined, the scheduler
   will schedule each job to run on the service node from which
   it was submitted.  If the MOM is not running on that node, it
   will choose another node.  If this is not defined, the scheduler
   load balances PBS jobs across the service nodes.

CPLANT_REMOTE_WORKING_DIR - define this if the working directory
  is NFS mounted from a remote server.  If it's defined we won't
  try to lock files with fcntl(), which often fails.  It is much
  preferable to place the working directory locally.

CPLANT_NONBLOCKING_CONNECTIONS lib/Libnet/net_client.c will open
   non-blocking sockets, so server and mom don't hang when connecting
   to a mom that's exited or a mom on a node that's crashed.  Also,
   server will check state of a node (in server's data structure)
   before attempting to connect.  Server will log a big warning if
   a node is unreachable.
```

8.7 Maintaining the runtime directories

It would be good to clean up the runtime directories occasionally. In particular:

Archive these log files:

- `/tmp/pbs/working/server_logs/*` on the server node
- `/tmp/pbs/working/server_priv/accounting/*` on the server node
- `/tmp/pbs/working/sched_logs/*` on the scheduler node
- `/tmp/pbs/working/mom_logs/*` on every MOM node

Remove old junk from:

- `/tmp/pbs/working/spool/*` on every MOM node

- `/tmp/pbs/working/undelivered/*` on every MOM node
- `/tmp/pbs/working/mom_priv/jobs/*` on every MOM node
- `/tmp/pbs/working/server_priv/jobs/*` on the server node

Chapter 9

Setting up a virtual machine

This chapter provides an example of setting up a virtual machine from the standpoint of runtime utilities and daemons. See the installation document [6] for a broader description of setting up a VM.

9.1 The VM software tree

Assume we are setting up a virtual machine named `vmnew` on a cluster and we wish it to include 4 service nodes and 60 compute nodes.

On the main administrative node (also called the *sss1*) we maintain a directory of the Cplant system software. We would typically place the software tree at `/cplant/vm/vmnew`. From here it is either directly mounted onto service and compute nodes, or it is distributed by way of a script to the *leader* nodes from which the service and compute nodes can mount it.

You will see these directories in `/cplant/vm/vmnew`:

```
command>> ls /cplant/vm/vmnew

kernels  nfs-cplant  nfs-root-alpha
```

The `kernels` directory may contain a special Linux kernel for the compute and/or service nodes. If not, there is a default kernel that will be booted on those nodes.

The `nfs-root-alpha` directory may contain files that override the files in the root file system on the service and compute nodes. Normally the root file system is standard for all VMs. But if your VM requires a different

`/etc/passwd` file for example, they you should create an `/etc` directory in `nfs-root-alpha` and place your special `passwd` file in it.

The contents of `nfs-cplant` looks like this:

```
command>> ls /cplant/vm/vmnew/nfs-cplant


bin  etc  etc.in  hpti  man  mcps  misc  modules  routes  sbin  test
```

The directories under `/cplant/vm/vmnew/nfs-cplant` will appear under `/cplant` on the service and compute nodes.

You will want to look at these files in `/etc` and modify them if necessary to to set up the VM as you wish:

9.1.1 user-env

We use this script to start and stop **PCT**s, to start and stop the **bebopd**, or to ensure the user log file is set up on a node that **yod** will run on. It is invoked with two options:

```
user-env start [ bebopd | bebopd+pbs | pct | yod-support]
user-env stop  [ bebopd | pct ]
```

The `vm-config` file determines how the `user-env` script will be invoked at start-up time.

`user-env start bebopd` starts the bebopd in the background without any PBS support.

`user-env start bebopd+pbs` starts the bebopd in `PBSupdate` mode and allocates some nodes to non-PBS jobs. Edit this if you want to change the number of nodes allocated to non-PBS jobs when the **bebopd** starts.

`user-env start pct` mounts the RAM disk required by the **PCT** and then starts the **PCT** in the background.

`user-env start yod-support` ensures that the user log file used by **yod** is set up on the node. There may be code in the script to create a log file and then make a link to it from `/tmp/userlog`. (`/tmp/userlog` is the default location to which **yod** will write the log file.) The best situation is to have `/tmp` be on a local disk and to have the **yod** processes on each service node write the userlog file there. You should edit this section of the file to reflect where you want the userlog file to be. You may not need to invoke it at all, depending on your setup.

The `user-env` script can be invoked with the `stop` argument to kill the **PCT** or to kill the **bebopd**.

9.1.2 pbs-env

The `pbs-env` script can start the PBS deamons, restart them and stop them. It also sets up a **bebopd** node so the **bebopd** can use `qmgr` to update the PBS server.

```
pbs-env start [mom|server|scheduler|bebopd-support-only]
pbs-env stop  [mom|server|scheduler]
pbs-env restart [mom|server|scheduler]
```

There is only one situation where the `pbs-env` script should not be used to start a PBS daemon. The first time the PBS server is started with a new runtime directory, it should be started with this command:

```
pbs_server -t create
```

This informs the server that the database created when you enter the initial `qmgr` commands to configure the server is not yet set up. If you use the `pbs-env` script to start the server, no harm is done, but the server will exit with an error.

The `pbs-env` script assumes there is a template of the PBS runtime directory somewhere. (See the definition of `TEMPLATE` in the script.) When you run `pbs-env start` *service* on a service node, and there is no working directory, the script will create a working directory from this template. So you may wish to edit the script to have the correct path of the working directory template you created when you built and installed the PBS software.

The `bebopd-support-only` option should more accurately be called `pbs-client-support`. On any node where PBS clients like `qmgr` or `qsub` will be invoked, the PBS runtime directory must be set up. `pbs-env start bebopd-support-only` ensures there is a PBS runtime directory on the node.

9.1.3 vm-config and types files

The `vm-config` file lists the scripts that should be run to start up each node. For nodes that are not listed in `vm-config`, a default start up routine is provided in the `types` file.

Suppose we decide to place our runtime daemons on the service nodes according to following table, and that **yod** processes will run only on the two nodes on which the MOMs are running.

if-0.n-6.r-3	bebopd
if-0.n-4.r-3	PBS server and scheduler
if-0.n-2.r-3	PBS MOM
if-0.n-0.r-3	PBS MOM

The `vm-config` file lines for these service nodes would look like this:

```
#
# Bebopd
if-0.n-6.r-3:ptl-config=  ... user-env=bebopd+pbs pbs-env=bebopd-support-only

# PBS server/scheduler
if-0.n-4.r-3:ptl-config=  ... pbs-env=scheduler pbs-env=server

# PBS MOM nodes
if-0.n-0.r-3:ptl-config=  ... pbs-env=mom user-env=yod-support
if-0.n-2.r-3:ptl-config=  ... pbs-env=mom user-env=yod-support
```

The `config` script invokes the scripts listed in `vm-config` with the argument `start` and followed by the arguments named in the `vm-config` file.

The ... indicates several other scripts were left out that would normally be run. For example, any scripts that mount file systems should be executed before `user-env` and `pbs-env`. See the `vm-config` file on a running Cplant VM, or in the source code repository in `/top/scripts/etc.in`, for an example.

The `ptl-config` script starts the Cplant message passing modules. These must be running before any portals processes are started.

9.1.4 `site file`

The site file defines path names and other configuration information used by the runtime daemons and utilities. It's fields are described in chapter 11. For example, if you want to override the default location of the userlog file, or if you want to provide a list of administrators who should receive mail when there are problems on the machine, you need a site file entry.

9.1.5 `cplant-host file`

The `cplant-host` file names the service node on which the **bebopd** is running. When **yod**, **pingd**, or the **PCT** need to contact the **bebopd**, they look in this file to determine which machine they should send a message to. Edit this file to contain the physical node number of the service node on which the **bebopd** is running.

9.1.6 `vmname file`

This file should contain the name of your VM.

9.2 Start it up

Now you should be able to start the virtual machine. Depending on what cluster management software you are using, you may need to run a script that updates the `leader` nodes with the VM software tree. Then you must reboot the service and compute nodes in the VM. (Again, there is a script to do this, depending on what cluster management software you use.)

After a minute or so, log in to a service node and run **pingd**. You should see a display listing the compute nodes in the VM. If they are not all listed, it is possible they have not finished coming up yet.

Chapter 10

Solutions to common runtime problems

This section is in progress.

10.1 moving the bebopd to a different node

10.2 executable check sum error

10.3 users notice there are free nodes (PBS jobs aren't using all their nodes)

10.4 stale PCTs

10.5 load failure scenarios

10.6 yod can't write userlog file because of old LOCK files

Chapter 11

Site definition file format

Some of the values governing the behavior of the load utilities and daemons will most likely vary from site to site. For example, values naming pathnames to files may differ. These values may be specified in a site file, or specified with environment variables. An environment variable definition overrides a definition in the site file.

For example, when the **PCT** tries to determine whether the user executable will fit in RAM disk, it needs to know the path to the RAM disk. If the environment variable `PCT_SCRATCH` is set, the **PCT** uses this value as the path name. If this environment variable is not set, the **PCT** looks in the site file for the string `PCT_SCRATCH` and takes the string following it to be the path name. If there is no site file, or no `PCT_SCRATCH` entry in the site file, then the **PCT** uses the default of `/tmp/pct-scratch` which is compiled into the code.

The site file location is named in the source file `top/include/config/cplant.h` as `/cplant/etc/site`. This pathname may be overridden by defining the environment variable `SITE_FILE` to be the pathname of the site file.

The format of the site file is a list of definitions. A definition is an environment variable name, followed by the value, delimited by white space. Other text can appear before or after this definition. The load utilites seek the environment variable name string in the file, and then take the following string to be it's definition, so don't name the variable elsewhere in text.

Most variables can have the special value `none` meaning in general that that resource is not provided on your machine.

The following definitions can be made in the site file.

Site file definitions	
Name Default if Not Set	Description
PCT_SCRATCH `/tmp/pct-scratch`	To start a user process on a diskless compute node, the **PCT** must first copy the executable to RAM disk. This environment variable defines the path name of the directory wherein user executable file will be written.
PCT_OVERFLOW `none`	If there is local disk storage on the compute node, or you have some alternative to RAM disk, recompile the PCT with HAVE_WRITABLE_DISK defined and set PCT_OVERFLOW to the path name on the disk to which applications can be written. The PCT will attempt to write the executable file to this location if there is insufficient space in RAM disk. Set to *none* if there is no such file system.
LOGFILENAME `/tmp/userlog`	**yod** logs a Cplant job to a log file when the parallel application completes. The **bebopd** logs Cplant jobs to another log file with a format identical to that of ASCI Red. The name of the **yod** log file is defined here. The **bebopd** log file is written to the same directory, but has the name *run.log.mm-dd-yy*. Set LOGFILENAME to `none` if you want to turn off logging.
VM_NAME_FILE `/cplant/etc/vmname`	Location of a text file containing the name of the virtual machine. If you don't use virtual machine names, set this to "none".
NOTIFYLIST `none`	**yod** can send mail to administrators when a load fails detailing the physical node numbers of the compute nodes that are failing. The NOTIFYLIST is the list of recipients of this mail. Set to `none` if you don't want yod to send mail upon load failures.

Site file definitions	
Name Default if Not Set	Description
BEBOPD_RESTART_FILE `/tmp/saved_pct_list`	Upon exit the **bebopd** writes a restart file listing the PCTs in the virtual machine. This is the pathname to which the restart file is written. Typically, `/etc/local` is a link to global writable storage that is private to the virtual machine.
NODE_NAMES_FILE `/cplant/etc/cplant-map`	Location of a text file containing a list of compute node names in physical node number order. The names are displayed by yod and pingd.
CPLANT_HOST_FILE `/cplant/etc/cplant-host`	Location of a text file naming the node on which the **bebopd** is running.
PARALLEL_FILE_SYSTEM `/enfs/tmp`	Path name to a global parallel file system. When executables fail to fit in RAM disk, **yod** will copy them here and the **PCT**s will exec them from here. Set to `none` if you have no such file system.
PBS_PREFIX `/`	The pathname where the PBS `bin`, `sbin`, and `man` directories may be found.
PCT_HEALTH_CHECK `zombies:memory:scratch`	The PCT can check the health of the compute node every time it receives a status request from the bebopd. (It omits the check however if it is running an application process.) It can check for zombie processes, low free memory, and insufficient RAM disk space. Set this value to the parameters you would like the PCT to check, or set it to `none` if you don't want the PCT to check these. The PCT will re-read the `site` file upon receiving a SIGHUP, so you can change the health check parameters while the PCT is running.

Bibliography

[1] http://www.cs.sandia.gov/cplant/doc/pbs/pbs.html.

[2] http://www.openpbs.org.

[3] top/service/pbs4cplant/build-pbs-2.2.

[4] R. B. Brightwell, T. B. Hudson, A. B. Maccabe, and R. E. Reisen. The portals 3.0 message passing interface. Technical Report SAND99-2959, Sandia National Laboratories, December 1999.

[5] Lee Ann Fisk. Cplant server library, reference guide. http://www.cs.sandia.gov/cplant/doc/libsrvr/libsrvr.html, February 2001.

[6] James Otto. Building, installing, and running cplant software. http://www.cs.sandia.gov/cplant/doc/build/build.html.

[7] Mark Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI The Complete Reference: Volume 1, The MPI Core, second edition*. MIT Press, 1998.

Index