CCMB Cluster Usage Guide
Jonathan Poisson
08/23/10

Table of Contents

The Raw Basics....2
Logging in to the cluster....2
Copying files to and from the cluster....2
Copying to the cluster -....3
Copying from the cluster -....4
Using the Cluster – Basic....4
Understanding the Cluster -....5
Understanding Storage -....5
Understanding the Head Node -....6
Understanding the Compute Nodes -....7
What's in a Name -....7
Unix Commands....8
man....8
apropos....8
df....9
du....9
ls....10
cd and pwd....10
cp....11
rm....11
screen....12
Job submission....12
A Note On Language....13
Queue commands – Basic....13
Qstat Basics....13
Qsub Basics....15
Job Submission – Doing something useful....17
Getting started – BLAST....17
Example 1....17
Example 2....18
Example 3....19
Job submission – Requesting resources....20
Getting Started – Discovering Queues....20
Requesting Resources - Reference....22
Requesting Queues....23
Requesting Walltime....24
Requesting Multiple Nodes and cpus....25
Requesting Multiple Nodes and CPUs - Advanced....26
Explicitly Assigning Slices to Separate Nodes....26
Explicitly Restricting a Node to a Single Job 27

The Raw Basics

This section contains very simple generalities on how to perform basic everyday tasks to interface with the cluster system. Such as accessing the system, copying files to the system and back, and general system navigation. Also covered are several “best practice” recommendations, and general policies for using a cluster system.

This portion of the documentation assumes little knowledge of Linux/Unix or clusters in general. However, neither are these topics fully explored and explained here. If this is your first time with any Linux/Unix system, it is recommended that you seek additional tutorials and documentation to get up to speed. This section though informative enough to start you working with a cluster system, does not contain the necessary details or fundamentals to effectively work in a Linux/Unix environment.

Logging in to the cluster

The preferred method of logging into a cluster is using `ssh` (secure shell). If using a Windows workstation you will have to use a third party ssh client such as `putty`.

Here is an example which uses ssh to connect to the cluster “axiom” from the workstation “legio.”

Example:

```
[jdpoisso@legio ~]$ ssh axiom.ccmb.med.umich.edu
 jdpoisso@axiom.ccmb.med.umich.edu's password:
Last login: Thu May 27 12:53:15 2010 from www.bioinformatics.med.umich.edu
[jdpoisso@axiom ~]$
```

Important: Axiom may not be the name of your cluster, please ask your advisor or technical contact which cluster you should be using.

Note that the when connecting to the cluster it may be necessary to use it's full name, as shown above. The full name of the cluster “axiom” is “axiom.ccmb.med.umich.edu.” Please see your technical staff or supervisor to find the full name of your cluster system.

Also notice, when logging into a cluster system, you are asked for a password. Different clusters have different password policies, so some may accept your campus uniqname and password, others may have a different set of usernames and passwords. Again, see your technical staff or supervisor for details regarding your login and password.

The detailed usage of Linux, Windows, or alternative ssh clients is beyond the scope of this document, please reference appropriate documentation for this software.

Copying files to and from the cluster

The preferred method of copying files to a cluster is using `scp` (secure copy). Using a Linux

workstation you may use this command to copy files to and from the cluster system. If using a Windows based system, there are third party utilities, such as `WinSCP,` that you may use to copy file. The usage of this utility, however, is not covered in this document.

Copying to the cluster -

Here is an example of copying the file `foo` from workstation `legio` to the cluster `axiom`.

Example:

```
[jdpoisso@legio ~]$ scp foo axiom.ccmb.med.umich.edu:~
jdpoisso@axiom.ccmb.med.umich.edu's password:
foo                                           100%   13KB  13.3KB/s   00:00
[jdpoisso@legio ~]$
```

The file `foo` has just been copied to from workstation `legio` to the home directory of `jdpoisso` (represented by the ~) on `axiom`. Notice as with `ssh` a password is requested, and then the progress of your file copy is shown on subsequent lines before returning you to your command prompt.

This command however assumes you are copying to your home directory on the axiom cluster, what if you wanted to copy the file somewhere different? Say maybe to `/tmp` ? In this case you can change the part of the command after the colon as in this example.

Example:

```
[jdpoisso@legio ~]$ scp foo axiom.ccmb.med.umich.edu:/tmp
jdpoisso@axiom.ccmb.med.umich.edu's password:
foo                                           100%   13KB  13.3KB/s   00:00
[jdpoisso@legio ~]$
```

So far the examples have only demonstrated the copying of a single file. What if, instead of a single file, you want to copy a whole directory or folder? This is accomplished with an argument to the scp command. For basic purposes, an argument may be defined as any additional data appended to a command, where each argument is separated by white space. So the command:

```
scp foo axiom.ccmb.med.umich.edu:/tmp
```

Has two arguments to the scp command "`foo`" and "`axiom.ccmb.med.umich.edu:/tmp`". Most commands in Linux/Unix accept arguments, and tend to expect them in a certain order. The `scp` command expects the last argument on the command line to be the destination of the file copy, while the preceding argument is generally considered to be the source file. You may also give the `scp` command an optional argument of the form "`-r`" which if present will copy any and all directories found at the source.

Example:

```
[jdpoisso@legio ~]$ scp -r data axiom.ccmb.med.umich.edu:~
jdpoisso@axiom.ccmb.med.umich.edu's password:
interface.c                                   100% 2730     2.7KB/s   00:00
architechture.c                               100% 1214     1.2KB/s   00:00
DSCF1692-1.jpg                                100%   61KB  61.1KB/s   00:00
architechture.h                               100% 7895     7.7KB/s   00:00
main.c                                        100%  420     0.4KB/s   00:00
alcove.c                                      100%  971     1.0KB/s   00:00
```

```
interface.h                                   100%  182     0.2KB/s   00:00
alcove.h                                      100% 6781     6.6KB/s   00:00
[jdpoisso@legio ~]$
```

Notice now that there are multiple files copied, this is because the source, `data` , rather than being a single file, was a directory. The `-r` after the `scp` was an argument that directed the `scp` command to copy from the source, all directories and all contents in those directories, to the destination location.

Copying from the cluster -

Assuming you've read the previous section on how to copy files to the cluster, the reverse, copying files back from the cluster, is easy considering the information given.

Here is an example copying the file `foo` from the home directory of `axiom` to the home directory on `legio`.

Example:

```
[jdpoisso@legio ~]$ scp axiom.ccmb.med.umich.edu:~/foo ~
jdpoisso@axiom.ccmb.med.umich.edu's password:
foo                                           100%   13KB  13.3KB/s   00:00
[jdpoisso@legio ~]$
```

Notice that this `scp` command is little more than a reversal of two arguments. This is because as mentioned before, the last argument to an `scp` command is the destination, while the one that precedes that argument is the source. Since we want our destination to be the home directory on `legio`, we simply put "`~`" in as the destination. In Linux/Unix, "`~`" is a shorthand representation for your home directory, and since the `scp` command is being run on `legio`, without any directives to the contrary, it will copy the file to the home directory on `legio`. Also notice that using `scp`, files on the cluster may be used as a source. The command will ask for your password and log into the cluster to find the file at the location specified and copy it to the destination if found. You may even still use the "`-r`" argument if the source on the cluster is a directory!

As you should be able to deduce, the `scp` command is sensitive to location. Meaning the computer you are currently logged into. The command will automatically assume, unless there is instruction otherwise, that the source and destination is on the computer you are currently logged into. In the preceding examples, we specified a computer other than the one we are currently logged into by typing it's full name followed (without spaces) by a colon. We then write out the source location as we would normally. So "`~/foo`" means in the home directory, the file `foo`. We could instead write "`/tmp/foo`" to mean in the `tmp` directory, the file `foo`. Using this format you can copy to and from any location (which you have access to) on the cluster system.

Using the Cluster – Basic

So far this document has focused on clearly defined instructions on how to log into the clusters, and a means on which to get your data to and from the cluster system. These instructions, however, do not cover all circumstances, and depending on various factors may not work in once place the way they work in another. Neither has it been explained exactly *where* you're logging into, and *where* you are

copying files to and from. Understanding some of the underlying structure and details of the system will help you understand the limitations of the system and why things may not always work the same different places.

Also in this section, if you're short of Linux/Unix knowledge, there is a set of examples for basic commands to help you navigate the system. These examples are not complete, and it is recommended that you pursue other means to improve your knowledge of these systems.

Understanding the Cluster -

A classic cluster is essentially a number of computers grouped together in a manner that allows them to share infrastructure, such as disk space, and work together by sharing program data while those programs are running. However, this simple definition, though accurate, does not really capture the full capability of a modern cluster system, as it excludes a very important concept. This concept, which has been developed to essentially become the core of clustering in general, is the scheduling system. The functional purpose of the scheduling system is to eliminate the need to know what individual computers are doing.

When presented with multiple computers, you do not know what they are doing without individually checking them. Anything could be running on them, by anybody who has access to them. If you want to run a program, you would have to check each computer to see which, if any, have enough available resources, disk space, processors, memory, to run your program. Not only is it inconvenient to manually check each computer, but if none of them have any available resources, then you will be forced to check again (manually) at a later time.

A scheduling system removes this need, by aggregating data, and monitoring it's system, a scheduler will keep an accurate and up to date picture of what resources are available and where. Even beyond tracking resources, a scheduler will allow you to submit instructions for running your program, and then run your program on your behalf once the necessary resources are available.

Understanding Storage -

Knowledge of your storage system is an often overlooked, but critical aspect of any cluster system. For most users, it may be enough to see that there is space, and therefore they use it accordingly. Unfortunately this is a mistake that often has a radical impact on the performance of the cluster system, causing it to slow or fail, and by extension slowing and failing everything that runs on that cluster. ***The key thing to realize is that there are different kinds of storage, and different storages are optimized and fit for particular purposes.*** When it is time for you to use a cluster system, it is important to understand the demands of your programs with regards to storage. Consider whether you have needs for fast storage, local storage, shared storage, long term storage, short term storage, or large amounts of storage.

Not all cluster systems have all types of storage available to them. It's not unusual for a cluster to have only a few kinds of storage, or be designed primarily around a single type of storage with other types being merely ancillary. For example, the `umms-amino` cluster is optimized for fast file access and reads, and is built around a fast storage system. While the `axiom` general cluster (at the time of this writing) has a solution that is engineered for data volume rather than performance. Most cluster systems have *local scratch* (storage located on each compute node, i.e. not shared (in most local systems this storage is located at `/tmp`)) available, however some may not and rely entirely on a shared storage system.

As any shared space can be saturated by the number of programs running on a cluster system, any need to access or write files to a shared space (like your home directory) can be affected by other jobs accessing that same general location. Ideally a cluster system may have hundreds, if not thousands, of jobs running on it at any given time. If you place a thousand cars at a traffic intersection, no matter how well the intersection is designed or how many lanes there are, it will still take some time for all those cars to clear the intersection.

Using storage on a cluster system is often a transparent process. The details of what the storage is and where it is, and its available features is often not readily apparent. As you change directories on any individual node in the cluster system, you may move seamlessly between different storage systems, each with different characteristics. For example, home directories (more on these later), are often on a shared storage system, but this may not be made apparent to you without querying the system information and or asking a system administrator. In most cases you will be directed to a preferred location (such as `/tmp`) from which to run your program, and it will be up to you to take advantage of this location when you submit jobs to the scheduling system.

Note: When you log into a cluster system you typically start out in your *home directory.* The home directory is usually a shared location which you can use to setup your personal configuration and preferences, as well as test and build programs to run on the cluster. The previously mentioned guidelines for a preferred run location apply mostly to *program data*, and not the programs themselves. If programs (jobs) that you plan on submitting to the cluster read specific data files (databases or the like) or write out large data files (building and assembling a large data structure or time step analysis), it is best for this data be copied or staged in preferred locations according to any guidelines given to you (i.e. like `/tmp`).

Understanding the Head Node -

Typically when you access a cluster system you are accessing a *head node,* or gateway node. A head node is setup to be the launching point for jobs running on the cluster. When you are told or asked to login or access a cluster system, invariably you are being directed to log into the head node.

A head node is often nothing more than a simply configured system that is configured to act as a middle point between the actual cluster and the outside network. When you are on a head node, you are not actually running or working on the cluster proper. However, all the tools and necessary programs are made available to test and then submit your programs to the cluster, as well as view and manage your data. The purpose for this arrangement is to keep your cluster system separate and distinct from other systems and “appear” as a single system rather than a aggregation of many individual systems. This has the effect of simplifying access, usage, and efficiency by having all interactions filtered and managed through a few designated points. In theory, you should never have any need to access any of the individual compute nodes of a cluster system, instead relying on the head node and the tools it provides to submit jobs to the cluster, monitor them, and view and retrieve their results.

Note: From a best practices standpoint although the head nodes are often setup to provide a point of access and testing of the programs you want to run on a cluster system, ideally you do not want to run computational programs on the head node itself. Meaning specifically, any programs you want to run on the cluster should not be run on the head node, instead they should be run and tested on the cluster itself using the scheduling system tools on the head node. You should restrict your usage of the head node to programs that let you build and prepare your cluster programs and manage and view your

data. In many cluster systems, there are often resources (cluster nodes) explicitly reserved for testing purposes.

Understanding the Compute Nodes -

Although in theory you may have no need to access any of the individual compute nodes, in practice there are many potential reasons you may need to access any given compute node. You may need to monitor the actual data being generated by a program, which may not be readily apparent from the head node, especially if your program is using *local scratch*. Also, it is generally accepted that computers are fallible things, and in the event of any errors, the scheduling system or your script may leave data behind on the head node that you need to retrieve.

In cluster systems access to the compute nodes is restricted to the head node or gateway nodes and other compute nodes. Therefore, access to the compute nodes usually starts from the cluster head node. So accessing any particular compute node is as simple as using **ssh** to connect to that compute node.

Example:

```
[jdpoisso@axiom ~]$ ssh compute-0-4
[jdpoisso@compute-0-4 ~]$
```

Note: You may not be asked for a password when using ssh from the head node to a compute node. In cluster configurations the compute nodes are configured to "trust" the head node. If you are asked for a password it may represent a problem with the system or your account.

Note: The names of the individual compute nodes vary from cluster to cluster, but most clusters are based on a particular schema, such as greek letters (alpha, beta, gamma, delta, etc...) or numbers (one, two, three, four, etc...). In the axiom and amino clusters, the node names have a reference to their physical locations. So compute-0-4 would be the fourth system in rack zero. Some clusters may also have shorthand names, compute-0-4 could also be called c0-4 on the amino cluster.

What's in a Name -

Most computer systems have names: **axiom**, **amino**, **legio**, or **compute-0-4**. When you want to interact with these particular systems you contact them by name. You may notice though that not all names work at all places. Like **legio** may know **axiom** and **amino** by name, but **axiom** and **amino** do not know **legio** and are unable to establish a connection back to that workstation.

It is important to realize that not all systems know the names for other systems (or may know them by different names). Conceptually, you should think of each system as having the computer equivalent of a personal address book, in it are the names and numbers of all the systems that it needs to talk to. Once it calls a system (using **ssh** or **scp**), it's like any phone conversation and information can be passed back and forth on that phone line until the connection is broken. Unfortunately, not all systems have everyone in their address books. Just like in real life, if an address book is too large, it can take a long time (relatively) to find what the system is looking for. This leads to situations where **legio** might have **axiom's** number, but **axiom** may not have **legio**'s number.

Practically, this concept is important to understand when you want to copy data from the cluster to another location: a workstation, a backup system, or another cluster. One system may not know the name of another system on a first name basis. So when you try "`scp data.file legio:~`" on

axiom, **axiom** will return an error saying it can't find **legio**. In these cases, most often you need to look at a phone book (in computer terms - DNS) and to find someone in the phone book you need their full name. As briefly shown above, the full name of "**axiom**" is "**axiom.ccmb.med.umich.edu**". So if a system doesn't know **axiom** by first name, it may have more luck with its full name. It's also important to remember, just like in real life, most computer phone books (DNS) are for local areas. So while it may have all the entries for everything in its area, and be able to look up most internet names, there may not be a listing for a system located in a another area. In real life a phone book for Washtenaw county doesn't have a listing of all the telephone numbers of people in Wayne county, even though the two areas are neighbors.

Unix Commands

Going over the breadth of Unix/Linux commands could cover volumes. The details of most of these commands will be left to other sources of documentation. What follows are a brief explanation and a few example of some crucial Unix/Linux commands you may use to navigate a cluster system. Most technical details will be omitted from this section as will the why and how things work the way they do.

man -

The **man** command represents "manual." Namely you can use this command to lookup and discover details of nearly any (standard) command on a system. Not all commands work with **man**, however, the lack is mostly restricted to custom programs and software packages

Once invoked your terminal screen will change to show the details of the command you requested. You can use the arrow keys to scroll up or down, and press the "q" key to quit.

Examples:

```
[jdpoisso@axiom ~]$ man apropos
<-- apropos man page displayed -->
[jdpoisso@axiom ~]$
[jdpoisso@axiom ~]$ man ls
<-- ls man page displayed -->
[jdpoisso@axiom ~]$
```

apropos -

The **apropos** command is a sort of companion to the **man** command. It is used to search the system's database of commands to find anything "appropriate" to the words or phrases you search for.

Apropos returns a list of commands along with a brief description of those commands. The search is primarly focused on the command descriptions, so this is a useful tool to find something when you aren't entirely sure what the name of the command is. However, the searched database is large and contains programming information as well as system commands, so not all results will be relevant. You may want to enclose your phrase in quotes ("") or the command will search for each word separately.

Examples:

```
[jdpoisso@axiom ~]$ apropos "disk space"
df                   (1)  - report file system disk space usage
```

```
df                    (1p)  - report free disk space
[jdpoisso@axiom ~]$

[jdpoisso@axiom ~]$ apropos magnify
xmag                  (1x)  - magnify parts of the screen
[jdpoisso@axiom ~]$
```

df -

The **df** command is used to display the disk file usage. It shows a summary of the amount of space used and available on the system.

When run with no arguments it reports information about every “filesystem”

Example:

```
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/sda4            120576100  73252260  41198892  65% /
tmpfs                  4092144       656   4091488   1% /dev/shm
/dev/sda2               198337     69000    119097  37% /boot
```

However, if you want to break down to only the relevant information, like how much space is available in my home directory or in /tmp, you can use those directories as an argument.

Example:

```
[jdpoisso@umms-amino ~]$ df ~
Filesystem           1K-blocks      Used Available Use% Mounted on
10.2.1.18:/export/umms-amino/home/jdpoisso
                     11134798720 9499291008 1635507712  86% /home/jdpoisso
[jdpoisso@umms-amino ~]$ df /tmp
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/cciss/c0d0p2     39674224   5665648  31960692  16% /
[jdpoisso@umms-amino ~]$
```

If you find the numbers a bit confusing, try using the “-h” argument. **K** stands for kilobytes, **M** for megabytes, **G** for gigabytes, and **T** for terabytes.

Example:

```
[jdpoisso@umms-amino ~]$ df -h ~
Filesystem            Size  Used Avail Use% Mounted on
10.2.1.18:/export/umms-amino/home/jdpoisso
                       11T  8.9T  1.6T  86% /home/jdpoisso
```

du -

The **du** command is used to display the actual disk usage of a file or directory (and all its files) as well as the usage of every subdirectory in that directory.

By default the numbers are in kilobytes.

Example:

```
[jdpoisso@legio ~]$ du data
132   data/image
208   data
[jdpoisso@legio ~]$ du summary.pdf
```

```
232   summary.pdf
[jdpoisso@legio ~]$
```

You can use the “-h” argument the same as with **df**.

Example:

```
[jdpoisso@legio ~]$ du jdpoisso.strace
3224  jdpoisso.strace
[jdpoisso@legio ~]$ du -h jdpoisso.strace
3.2M  jdpoisso.strace
```

ls -

The **ls** command stands for “list.” You can use this command to view a list of all the files in a directory.

Example:

```
[jdpoisso@legio data]$ ls
architechture.c  architechture.h  image  interface.c  interface.h  main.c
```

A common usage is to use “**ls -la”** to list additional details, including permissions, file ownership and file size.

Example:

```
[jdpoisso@legio data]$ ls -la
total 60
drwxrwxr-x.  3 jdpoisso jdpoisso 4096 2010-07-02 11:56 .
drwx------. 65 jdpoisso jdpoisso 4096 2010-07-02 11:56 ..
-rw-rw-r--.  1 jdpoisso jdpoisso 1214 2010-05-27 13:39 architechture.c
-rw-rw-r--.  1 jdpoisso jdpoisso 7895 2010-05-27 13:39 architechture.h
drwxrwxr-x.  2 jdpoisso jdpoisso 4096 2010-05-27 14:12 image
-rw-rw-r--.  1 jdpoisso jdpoisso 2730 2010-05-27 13:39 interface.c
-rw-rw-r--.  1 jdpoisso jdpoisso  182 2010-05-27 13:39 interface.h
-rw-rw-r--.  1 jdpoisso jdpoisso  420 2010-05-27 13:39 main.c
```

cd and **pwd-**

The **cd** command stands for “change directory.” You use it to move between directories on a system. If the directory has a “/” in front of it, the directory change will occur “absolutely” without a “/” the directory change will occur relatively.

The **pwd** shows your “present working directory.” If you're not sure where you are on a system, use pwd.

Example:

```
[jdpoisso@legio data]$ cd /tmp
[jdpoisso@legio tmp]$ pwd
/tmp
[jdpoisso@legio tmp]$ cd ~
[jdpoisso@legio ~]$ pwd
/home/jdpoisso
[jdpoisso@legio ~]$ cd data
[jdpoisso@legio data]$ pwd
```

```
/home/jdpoisso/data
[jdpoisso@legio data]$
```

cp-

The **cp** command stands for “copy.” Use this to copy from one location to another location. The first arguments are the source, while the last argument is the destination. If the destination is a file, the sources will be copied over that file. If the file doesn't exist yet, it is created and the sources copied over to that file. If the destination is a directory, the sources will be copied to that directory, potentially copying over files already there.

Example:

```
[jdpoisso@legio data]$ ls
architechture.c  architechture.h  image  interface.c  interface.h  main.c
[jdpoisso@legio data]$ cp main.c old_main.c
[jdpoisso@legio data]$ ls
architechture.c  image        interface.h  old_main.c
architechture.h  interface.c  main.c
[jdpoisso@legio data]$ ls image
DSCF1692-1.jpg
[jdpoisso@legio data]$ cp old_main.c image
[jdpoisso@legio data]$ ls image
DSCF1692-1.jpg  old_main.c
[jdpoisso@legio data]$
```

If the source is a directory you need to use the “-r” argument. The source directory (and all the files and directories it contains) will be copied to the destination. The “image” in the example below is a directory.

Example:

```
[jdpoisso@legio data]$ ls
architechture.c  image        interface.h  old_main.c
architechture.h  interface.c  main.c
[jdpoisso@legio data]$ cp -r image new_image
[jdpoisso@legio data]$ ls
architechture.c  image        interface.h  new_image
architechture.h  interface.c  main.c       old_main.c
[jdpoisso@legio data]$ ls new_image
DSCF1692-1.jpg  old_main.c
[jdpoisso@legio data]$
```

rm -

The **rm** stands for “remove.” You may use this command to delete files. As with copying, if you are removing a directory, you need to use the “-r” argument.

Example:

```
[jdpoisso@legio data]$ ls
architechture.c  image        interface.h  new_image
architechture.h  interface.c  main.c       old_main.c
[jdpoisso@legio data]$ rm old_main.c
[jdpoisso@legio data]$ rm image/old_main.c
[jdpoisso@legio data]$ rm -r new_image
[jdpoisso@legio data]$ ls
architechture.c  architechture.h  image  interface.c  interface.h  main.c
```

```
[jdpoisso@legio data]$
```

screen -

The **screen** command is a very powerful and useful command. As you work with a cluster system you may notice that if you lose your connection, or close the terminal you are working on, whatever you were doing on the cluster is terminated (this is not true of jobs submitted to the scheduling system, they are held separately and are not terminated if you log out of the cluster). So if you were copying files and waiting for the process to finish and accidentely close your terminal, the file copy is terminated and you will have to restart it. This is why the **screen** command is so useful. After the "screen terminal" has been activated, anything in that terminal stays running even if you log out.

Simple usage of the **screen** command is easy, however, it may not be readily apparent that it's active.

Example:

```
[jdpoisso@legio ~]$ screen
<- Screen clears ->
[jdpoisso@legio ~]$
```

The screen clears itself and it appears as if nothing happened. However, you are now in the "screen terminal" which is functionally equivalent to any other terminal, but in case the window is closed, or you want to detach this terminal will instead remain active.

You may go ahead and use this terminal normally. If you're waiting for something to complete you can *detach*, which leaves the screen terminal active but removes it from view and protects it in case of a connection loss. To detach you must press "**<Ctrl-A>-D**" (this is a key sequence, press control(ctrl) then **A** while holding control(ctrl) then release those keys and press **D**). If done properly you will see this.

Example:

```
[detached]
[jdpoisso@legio ~]$
```

To *reattach* or to bring up the screen terminal you need to invoke "**screen -r**" and this will show your screen terminal exactly where it left off.

Note: The screen terminal may be exited like any other terminal, by using the "**exit**" command. This destroys the screen terminal and exits the screen program.

Note: screen may not be installed by default on your cluster system. Ask your system administrator to install it for you.

Job submission

As previously explained, the scheduling system is in many ways the most important feature of a modern cluster system. Learning to use this system is key to making the most efficient use of a cluster system as well as sharing it with others. In this section, a detailed explanation of job submission using one of the more common job scheduler packages, TORQUE/MOAB, will be given. This document will

start from a basic understanding of the available commands to a sample rundown of submitting simple jobs to the job queue.

Once basics are explained a detailed reference of all available tools included in the scheduling system is provided, followed by advanced scripting, which will explain how to write scripts for distributed programs that use MPI.

A Note On Language

When using a scheduling system there is a unique vocabulary used to describe various actions and information presented by the system. For example, a program being run, or preparing to be run by the scheduling system is called a “job.” Jobs are “submitted” to the system and are then organized into “queues” which define the rules for running a job and where, using what resources. New terms will be explained as they come up, but be aware that there is a preferred method of describing features and actions performed by the system.

Queue commands – Basic

The two most valuable commands when using TORQUE, are `qsub` and `qstat`. The `qsub` command is used to “submit” jobs to the “queue.” While the `qstat` command is used to retrieve the status of the “queue.” You can effectively use a cluster system with nothing but these two commands, as they effectively comprise the most of the useful scheduler commands allowing you to run your jobs and check their status.

`Qstat Basics -`

As previously stated, the qstat command will display the status of the queue(s), specifically, it's most common usage is to retrieve information regarding the jobs running on the scheduling system. Depending on the cluster and the configuration, this command may return a list of jobs submitted *only* by yourself, or a list of all jobs submitted, as seen in these examples. `Axiom` is configured to report only jobs submitted by the user, while `umms-amino` reports on all jobs currently submitted to the system.

Note: Once a job is complete, it will no longer be reported in qstat, completed jobs will be discussed in another section.

Example:

```
[jdpoisso@axiom ~]$ qstat
Job id                    Name             User            Time Use S Queue
------------------------- ---------------- --------------- -------- - -----
1027.axiom                15-34            jdpoisso               0 Q first
1028.axiom                 16-34            jdpoisso               0 Q first
1029.axiom                 17-34            jdpoisso               0 Q first
1030.axiom                 18-34            jdpoisso               0 Q first
1031.axiom                19-34            jdpoisso               0 Q first
1032.axiom                 20-34            jdpoisso               0 Q first
1033.axiom                 21-34            jdpoisso               0 Q first
1034.axiom                 22-34            jdpoisso               0 Q first
[jdpoisso@axiom ~]$
```

Example:

```
[jdpoisso@umms-amino ~]$ qstat
Job id                    Name             User            Time Use S Queue
------------------------- ---------------- --------------- -------- - -----
1036842.umms-amino        T10372_2_AMF_Z   yzhang          00:05:55 R casp
1036843.umms-amino        T10372_2_A_closc yzhang          00:05:53 R casp
1036850.umms-amino        d1l5ja2          jinrui          00:03:28 R default
1036852.umms-amino        S46334_2F_1_run  zhanglabs       00:03:01 R urgent
1036853.umms-amino        S46334_3F_1_run  zhanglabs       00:02:51 R urgent
1036854.umms-amino        d1l5ja3          jinrui          00:02:43 R default

[jdpoisso@umms-amino ~]$
```

The `qstat` command (when run with no arguments) will return information broken down into six columns. The first column is the `Job ID`, this is the unique job number of the job and the scheduling server that job number was submitted to. Each job submitted is given a unique number that is used by the scheduler to reference the job. This job number is required for many other **TORQUE** commands, and can even be given to the `qstat` command to gain more information about a particular job.

The second column, `Name`, is the job's defined name. Defining names will be covered when discussing the `qsub` command. For now it is enough to know that each job may optionally be given a non-unique name to describe the job in the queue. This is useful as it allows you to describe the job with a word, or phrase, or input, that means more to a human reader than a job number. Also, for future reference, job outputs returned by the scheduler are written to files using this defined name.

The third column, `User`, lists the username which submitted the job. This allows you to know who submitted the job to the queue, and search for jobs submitted by collaborators, or by yourself. The system does allow you to submit on behalf of another user, so all jobs in this field are accurate as to who is used the qsub command.

The fourth column, `Time Use`, lists how long your job has been running for. Until the job actually starts running, the number is set to 0. Depending on how the job is run, and the arguments being used this can be displayed either in `CPU time` (default), or `walltime`. CPU time is the aggregate amount of time each CPU in the system has devoted to running your job, so if one of your job used two CPUs for thirty seconds (00:00:30) each, your `CPU time` will be one minute (00:01:00). `Walltime`, on the other hand is defined by how long the job took to run according to the clock on the wall, i.e. a normal clock. So that same job that used two CPUs for thirty seconds each, if using the CPUs at the exact same time, will finish in thirty seconds, and the `walltime` will be thirty seconds (00:00:30).

Note: `Walltime` is also considered a **resource**, as briefly mentioned before, the system will run jobs based on availability of resources. If you expect your job to run for a long period, say twenty hours, you must request a `walltime` resource for twenty (20:00:00) hours, meaning the cluster will schedule your job when it most optimally can allocate twenty hours according to it's queue and configuration. If a `walltime` resource is not specified when you submit your job, the default `walltime` allotment will be used, the exact amount of which varies by system and by queue. Exceeding the `walltime` allotment may cause the scheduler to forcibly terminate your job, whether it is complete or not. `Walltime` will be covered further in sections about advanced job submission.

The fifth column, `S`, represents that status of the job. This is a one letter code for what your job's current state is. The possible job states are:

Q – Queued, waiting to be run
R - Running, job has been assigned resources and is running
E - Exiting, job is complete and is cleaning up, and copying files
H - Held, the job has been held, either by the submitter or an administrator

The sixth column, `Queue`, lists the current job queue that the job is submitted to. As each queue has varying resources and rules, you are able to use this information to help deduce what resources the job is using, or what resources the job is waiting to become available for.

`Qsub Basics -`

The `qsub` command is used to submit jobs to the queue. A job, as previously mentioned, is a program or task that may be assigned to run on a cluster system. The `qsub` command is itself simple, however, using it to actually run your desired program may be a bit tricky. This is because `qsub`, when used as designed, will only run **scripts**. A script is a text file containing a series of instructions or commands that are carried out in sequence when run on a computer. Scripts can vary a great deal in complexity, a simple script may just change your directory and run a program, while a complex script, may change your directory, run multiple programs, sort the output, compress it, and then copy it to a specific location.

Note: For advanced users, although you can technically submit any type of script to qsub, as a best practice you should only submit shell scripts (i.e. bash), even if this script only invokes another script and scripting language. This is because the data the scheduling system provides to a job during runtime is set as environment variables to the shell. Also, in general, to simplify debugging, it is prudent to keep the actual job submission scripting, separate from the program or task you want to run.

Let's say I have a program that calculates an arbitrary fibonacci number, and I want to run this program on the cluster. I have written a script file “script.sh” which runs this program. To submit this job I can use the following method.

Example:

```
[jdpoisso@axiom ~]$ cd qsub
[jdpoisso@axiom qsub]$ ls
fib  script.sh
[jdpoisso@axiom qsub]$ qsub script.sh
1056.axiom.localdomain
[jdpoisso@axiom qsub]$ qstat
Job id                    Name             User            Time Use S Queue
------------------------- ---------------- --------------- -------- - -----
1056.axiom                script.sh        jdpoisso        00:00:38 R first
[jdpoisso@axiom qsub]$
```

Here the qsub command was given the script as an argument. This tells the `qsub` command to contact to the scheduling system and submit the script “`script.sh`” to the cluster, if the submission is accepted, you are given a number, in this case 1055, which we can use to monitor the job. Once resources are available to run the program, the scheduling system will send the script to one of the cluster nodes and follow its instructions to run the program. As shown in the example by the `qstat` command, the script started running soon after submission. The job will continue to run until it completes, at which point it will copy back the results and will disappear from job listing provided by

qstat.

Example:

```
[jdpoisso@axiom qsub]$ qstat
[jdpoisso@axiom qsub]$ ls
fib  script.sh  script.sh.o1056
[jdpoisso@axiom qsub]$
```

By default the only results copied back is the *standard output* and *standard error* (`script.sh.o1056`). These are Unix/Linux terms for anything that would be printed by the program being run. Many programs write their results to a separate file, rather than printing it. On most cluster configurations, the copying of these files (if necessary) are the responsibility of the user. The instructions to do so may be done at the end of your submitted script. More will be said about this in the sections on scripting.

Note: You may or may not be notified by email when your job completes, depending on the cluster configuration. If you wish to receive an email when your job completes (or fails) instructions for doing so may be found in Scripting Details section under PBS directives.

Example Script -

The script used in the previous qsub example, and its output are listed below. To learn about scripting, see the next section and the section on Scripting Details or refer to other documentation on scripting in Unix/Linux.

`script.sh :`

```
#!/bin/bash
#PBS -j oe

echo "Running on: "
cat ${PBS_NODEFILE}

echo
echo "Program Output begins: "

cd ${PBS_O_WORKDIR}

./fib 46
```

`script.sh.o1056 :`

```
Running on:
compute-0-19

Program Output begins:
1,       1,       2,       3,       5,
8,       13,      21,      34,      55,
89,      144,     233,     377,     610,
987,     1597,    2584,    4181,    6765,
10946,   17711,   28657,   46368,   75025,
121393, 196418, 317811, 514229, 832040,
1346269,         2178309,         3524578,         5702887,         9227465,
```

```
14930352,       24157817,       39088169,       63245986,       102334155,
165580141,      267914296,      433494437,      701408733,      1134903170,
1836311903,
```

Job Submission – Doing something useful

So far this documentation has covered the basics of using a few TORQUE commands, and accessing and using a Unix/Linux system. This information covers the rawest of fundamentals of using the cluster system for your programs. At this point you are ready to start experimenting with the cluster and using it to facilitate your research. This section will go into further details on scripting and setting up jobs and elaborating on best practices and suggestions on how to script them. However, this section will only cover *serial* jobs, jobs using only a single processor. Running distributed or threaded jobs will be discussed in the section on advanced job submission.

This purpose of this section is to teach technique and method, rather than particular details of how and why things work. While these details may be mentioned, they may also be only briefly explained. If there are questions on any of the particular commands, variables, or arguments used in this section please see the appropriate sections of this guide, or feel free to examine an external documentation source.

Getting started – BLAST

BLAST (Basic Local Alignment Search Tool) is a tool used in bioinformatics to find regions of local similarity between sequences. **BLAST** is a software package that contains several different tools that search existing databases given an input of nucleotides or a protein. The exact details of **BLAST** and how to run and use the software are beyond the scope of this documentation. The examples that follow are to illustrate how to use the program with a cluster scheduling system and a variety of scripting techniques to streamline program operation.

Note: The example inputs in this section are taken from the **tcoffee** package, which has many different kinds of inputs in a variety of formats.

Note: The scripts in this section are primarily for example purposes, they are not a “best method” to run a particular program in all cases, instead they are meant to showcase different scripting and job setup techniques. In general, the best submission scripts are those where the process is standardized and organized to a degree that you seldom (if ever) have to change the script. You are encouraged to write scripts that suit your particular style and preferences.

Here we have already set up a blast_test directory with an input file ready to go.

Example:

```
[jdpoisso@umms-amino ~]$ cd blast_test/
[jdpoisso@umms-amino blast_test]$ ls
sv.fasta
[jdpoisso@umms-amino blast_test]$
```

Let's say you wanted to run this without using the scheduling system. Your command might be something like this:

```
/opt/bio/ncbi/bin/blastall -p blastp -i sv.fasta -d /library/yzhang/nr/nr
```

Note: Depending on how paths are set up on your cluster system or by your personal settings you may not need to use the full path "`/opt/bio/ncbi/bin/blastall`" Also, depending on your system you may need to be aware of having multiple versions of a software, and only one may be the default at a given time. Be aware of these factors when writing your script, and be sure to run the correct version if your input is version sensitive.

To submit that exact command to the scheduling system means writing it into a script. Here is a script that will run that exact command through the scheduling system:

`blast.sh` :

```
 #!/bin/bash
cd ${PBS_O_WORKDIR}
/opt/bio/ncbi/bin/blastall -p blastp -i sv.fasta -d /library/yzhang/nr/nr
```

As you can see, just to run a command there is only a small amount of setup required. The first line identifies the file as a script and specifies what shell (for our purposes - the language of the script) to use. The second line changes the directory using the `${PBS_O_WORKDIR}` environment variable. As previously mentioned, the scheduling system may set environment variables for a job. The `${PBS_O_WORKDIR}` variable is set to the directory from which the job is submitted, our *submission directory*. So if we submit the job in our `blast_test` directory, then the variable is set to be "`blast_test`" (actually it's the absolute path - `/home/jdpoisso/blast_test`). This is necessary because (by default) when the scheduling system starts your job, the directory is set to be your home directory, which may have any input (or worse, different input) causing your job to try to run, failing to find your input and ending.

Example :

```
[jdpoisso@umms-amino blast_test]$ qsub blast.sh
1231700.umms-amino.ccmb.med.umich.edu
[jdpoisso@umms-amino blast_test]$ qstat 1231700
Job id                    Name             User            Time Use S Queue
------------------------- ---------------- --------------- -------- - -----
1231700.umms-amino        blast.sh         jdpoisso        00:03:20 R default
<----job finishes---->
[jdpoisso@umms-amino blast_test]$ ls
blast.sh  blast.sh.e1231700  blast.sh.o1231700  sv.fasta
[jdpoisso@umms-amino blast_test]$
```

Having run that script, the job is allowed to run, and to finish, and the output is placed in submission directory. For those familiar with **`BLAST`**, you may know that when the specified command is run without the scheduling system, all the results are printed to the screen, and not saved to a file. However, all the results normally printed to the screen and instead in `blast.sh.o1231700`, as mentioned in a previous section, the scheduling system captures the standard output and saves it into a file, which is then posted back to your submission directory as a result.

What if your job, instead of writing its results to the screen, writes it out to a file? Or multiple files? In this case, there may be nothing in that `blast.sh.o1231700` file. Instead your data may be in the files you specified, or specified by the program.

`blast.sh` :

```
#!/bin/bash
 cd ${PBS_O_WORKDIR}
 /opt/bio/ncbi/bin/blastall -p blastp -i sv.fasta -d /library/yzhang/nr/nr -o
blast.out -O blast.seq
```

Example :

```
[jdpoisso@umms-amino blast_test]$ qsub blast.sh
1231986.umms-amino.ccmb.med.umich.edu
[jdpoisso@umms-amino blast_test]$ qstat 1231986
Job id                    Name             User            Time Use S Queue
------------------------- ---------------- --------------- -------- - -----
1231986.umms-amino        blast.sh         jdpoisso        00:00:36 R default
[jdpoisso@umms-amino blast_test]$ ls
blast.out  blast.seq  blast.sh  blast.sh.e1231986  blast.sh.o1231986  sv.fasta
[jdpoisso@umms-amino blast_test]$
```

The command in the script has been changed to produce two output files, instead of anything that would be printed to the screen. Both these files have been written to our submission directory, as our script still explicitly changes to our submission directory. In this example, the files are small and manageable. However, what if your results are large files? Or your program does megabytes or gigabytes of temporary storage while it runs? There are many ways your cluster system could be configured to handle these situation. Some systems may have a high speed shared space that provides the necessary performance to handle many concurrent jobs of this type. In most cases though, you will want to copy your job to a *local scratch* space.

Accomplishing this is quite simple, assuming as in previous example all the necessary data is in the submission directory, we can modify our script to copy out and stage our data into a local scratch space (`/tmp`).

`blast.sh` :

```
#!/bin/bash

cd ${PBS_O_WORKDIR}

# setup and copy workdir
LOCAL_WORKDIR=/tmp/${USER}/${PBS_JOBID}
mkdir -p ${LOCAL_WORKDIR}
cp -r * ${LOCAL_WORKDIR}
cd ${LOCAL_WORKDIR}

/opt/bio/ncbi/bin/blastall -p blastp -i sv.fasta -d /library/yzhang/nr/nr -o
blast.out -O blast.seq

# copy back data
cp -r * ${PBS_O_WORKDIR}
cd ${PBS_O_WORKDIR}

# cleanup
rm -rf ${LOCAL_WORKDIR}
```

Example :

```
[jdpoisso@umms-amino blast_test]$ qsub blast.sh
1382907.umms-amino.ccmb.med.umich.edu
[jdpoisso@umms-amino blast_test]$ qstat 1382907
```

```
Job id                    Name             User            Time Use S Queue
------------------------- ---------------- --------------- -------- - -----
1382907.umms-amino        blast.sh         jdpoisso        00:04:10 R default
[jdpoisso@umms-amino blast_test]$ ls
blast.sh  sv.fasta
<----job finishes---->
[jdpoisso@umms-amino blast_test]$ ls
blast.out  blast.seq  blast.sh  blast.sh.e1382907  blast.sh.o1382907  sv.fasta
[jdpoisso@umms-amino blast_test]$
```

Note: The example script here does not take into account any potential caveats that could occur due to copying files back and forth, such as running out of disk space, or failing to copy files back to their origin. A modified version of the script above that includes checks for these factors is included in the scripting appendix. (Draft Note – Write a scripting appendix)

As you can see, the job may be submitted and run, and no change appears to the submission directory. This is because the relevant data (in this case the sv.fasta file) had been copied to a local scratch space on the whatever node the program was assigned. When the program completed, the data was copied back, and all the results appear in your submission directory.

Using these examples you have a simple framework and knowledge base you can use to submit jobs to the cluster. You should be able to submit jobs and have them run. However, you may notice certain problems when running your jobs using modified versions of these examples. Your jobs may end prematurely. They may not thread or distribute properly. They may run extremely slowly. They may crash for lack of memory or disk space. This is because so far there has been no discussion of how to request **resources**.

Job submission – Requesting resources

Resources are the criteria by which the scheduler determines when and where your job should run. By specifying how long your job needs to run, how much memory it needs to run and how many processors or nodes in needs to make available. Your job may also require licenses which need to be explicitly requested in order to be tracked and ensure your job gets the necessary licenses.

Depending on your cluster, there may different levels of enforcement for resources. Most systems rely on an honor system. Taking your resource requests at face value with no regard to how many resources you will in fact use. Lying, however, may not go unpunished, since the system uses the resources you've requested to determine the best place to run your job. Requesting one thing and then doing another may get your job exiled to a node which does not have the capability to properly run your job, causing it to slow, crash, or even refuse to run.

Another concept important in regards to resource requesting and allocation are **queues**. If you've read previous sections, **queues** were mentioned a few times as being involved with resource requests and resource allocation. To the scheduling system a **queue** is nothing more than a series of rules, which can restrict nodes, apply default resources, or set a job's priority. Each job submitted to the system must be submitted to a **queue**, which applies its rules to the job, usually in concert with any requests applied with that job. If the rules and the requests conflict or are at odds, the job may be outright refused, or may be submitted to the system but never allowed to run.

Getting Started – Discovering Queues

The first and ideal source for queue information and details about the queues on a cluster system should be your cluster's support staff. They will be able to brief you on the details of particular

queues, and their intended uses, and any limitations. If further information is needed you can use these commands to divine queue information from the scheduling system itself.

To view all the queues configured in your cluster system -

Example:

```
[jdpoisso@umms-amino ~]$ qstat -Q
Queue              Max   Tot   Ena   Str   Que   Run   Hld   Wat   Trn   Ext T
----------------   ---   ---   ---   ---   ---   ---   ---   ---   ---   --- -
casp                 0     0   yes   yes     0     0     0     0     0     0 E
ambyroy              0     0   yes   yes     0     0     0     0     0     0 E
default              0   891   yes   yes     0   891     0     0     0     0 E
interactive          0     0   yes   yes     0     0     0     0     0     0 E

urgent               0   238   yes   yes     0   238     0     0     0     0 E

[jdpoisso@umms-amino ~]$
```

From this output we can see that the cluster **umms-amino** has five queues configured. It also lists out an overview of the queues, the number of jobs, the status of the queues, and the how many jobs are in various job states. With this information we get a basic summation of the activity on the system, and which queues are heavily in use.

For our purposes, the only important information are the names of the queues. The other information, though useful, does not tell us any specific details on what rules the queue uses. To do this, we need to run a "`qstat -Q -f`" on the queues.

Example:

```
[jdpoisso@umms-amino ~]$ qstat -Q -f default
Queue: default
    queue_type = Execution
    Priority = 1
    total_jobs = 889
    state_count = Transit:0 Queued:0 Held:0 Waiting:0 Running:889 Exiting:0
    resources_default.nodes = 1
    resources_default.walltime = 01:00:00
    mtime = 1270501564
    resources_assigned.mem = 83886080000b
    resources_assigned.nodect = 889
    enabled = True
    started = True
[jdpoisso@umms-amino ~]$
```

Using "`qstat -Q -f`" on the "default" queue, we're able to see several pertinent details about the queue and the rules it uses. For instance we can see the "Priority" of the queue, which is a number assigned to all jobs submitted to the queue to determine when they should run. The higher "Priority" of your job, the sooner it will run. The other morsel of valuable information in are the lines regarding the default resources. From this information you can see that each job submitted to this queue is given a single node by default, and a walltime of one hour. Note, these values are just defaults, meaning that your resource requests may override them. In this queue you will also notice there are no job minimums, and no job maximums, these will show up in this output if they are present. Also if there is a set amount of resources limited available to the queue, that also will be listed. (Draft Note: Construct an example)

Note: Not all queue rules may be listed in the output of “qstat -Q -f” especially for aspects like node restrictions, and whether the queue is limited to specific users. This is why first and foremost you should consult with your support staff or reference any cluster specific documentation for details regarding the queues.

Requesting Resources - Reference

If you've read this document up to this point you should be able to submit simple jobs using default resources. For most serious jobs, the default resources are insufficient, and to effectively use the cluster you need to learn how to specifically request resources. In TORQUE, there are two methods you can use to request resources, **directives** and **command line arguments**.

In subsequent examples both forms will be demonstrated. Both forms may also be freely mixed, however, the **command line arguments** take priority over **directives**. For this most people find it useful to define a complete set of directives with all the defaults necessary for the job, and use command line arguments to adjust certain variables such as walltime or queue according to the specific needs of their input set.

All examples in this section will be using a script similar to that at the end of the preceding section. However, only the first lines of the script will be demonstrated in the **directive** examples. The practium section will then submit jobs using the two methods and you can compare it against the baseline to see the differences.

`blast.sh` :

```
#!/bin/bash

echo "Script begins here"
cd ${PBS_O_WORKDIR}

# setup and copy workdir
LOCAL_WORKDIR=/tmp/${USER}/${PBS_JOBID}
mkdir -p ${LOCAL_WORKDIR}
cp -r * ${LOCAL_WORKDIR}
cd ${LOCAL_WORKDIR}

/opt/bio/ncbi/bin/blastall -p blastp -i sv.fasta -d /library/yzhang/nr/nr -o
blast.out -O blast.seq

# copy back data
cp -r * ${PBS_O_WORKDIR}
cd ${PBS_O_WORKDIR}

# cleanup
rm -rf ${LOCAL_WORKDIR}
```

Job baseline for **`blast.sh`**:

```
Job Id: 2498949.umms-amino.ccmb.med.umich.edu
    Job_Name = blast.sh
    Job_Owner = jdpoisso@umms-amino.ccmb.med.umich.edu
    job_state = R
    queue = default
    server = umms-amino.ccmb.med.umich.edu
    Checkpoint = u
```

```
ctime = Mon Aug 23 10:20:59 2010
Error_Path = umms-amino.ccmb.med.umich.edu:/home/jdpoisso/blast_test/blast
  .sh.e2498949
exec_host = compute-4-5/0
Hold_Types = n
Join_Path = n
Keep_Files = n
Mail_Points = a
mtime = Mon Aug 23 10:21:00 2010
Output_Path = umms-amino.ccmb.med.umich.edu:/home/jdpoisso/blast_test/blas
  t.sh.o2498949
Priority = 0
qtime = Mon Aug 23 10:20:59 2010
Rerunable = True
Resource_List.nodect = 1
Resource_List.nodes = 1
Resource_List.walltime = 01:00:00
session_id = 11639
substate = 42
Variable_List = PBS_O_HOME=/home/jdpoisso,PBS_O_LANG=en_US.iso885915,
  PBS_O_LOGNAME=jdpoisso,
  PBS_O_PATH=/usr/kerberos/bin:/usr/java/latest/bin:/usr/local/bin:/bin
  :/usr/bin:/opt/bio/ncbi/bin:/opt/bio/mpiblast/bin/:/opt/bio/hmmer/bin:
  /opt/bio/EMBOSS/bin:/opt/bio/clustalw/bin:/opt/bio/tcoffee/bin:/opt/bi
  o/phylip/exe:/opt/bio/mrbayes:/opt/bio/fasta:/opt/bio/glimmer/bin://op
  t/bio/glimmer/scripts:/opt/bio/gromacs/bin:/opt/bio/gmap/bin:/opt/bio/
  tigr/bin:/opt/bio/autodocksuite/bin:/opt/ganglia/bin:/opt/ganglia/sbin
  :/opt/openmpi/bin/:/opt/maui/bin:/opt/torque/bin:/opt/torque/sbin:/opt
  /rocks/bin:/opt/rocks/sbin:/opt/sun-ct/bin:/home/jdpoisso/bin,
  PBS_O_MAIL=/var/spool/mail/jdpoisso,PBS_O_SHELL=/bin/bash,
  PBS_O_HOST=umms-amino.ccmb.med.umich.edu,
  PBS_SERVER=umms-amino.ccmb.med.umich.edu,
  PBS_O_WORKDIR=/home/jdpoisso/blast_test,PBS_O_QUEUE=urgent
euser = jdpoisso
egroup = jdpoisso
hashname = 2498949.umms-amino.ccmb.med.umich.edu
queue_rank = 2463766
queue_type = E
etime = Mon Aug 23 10:20:59 2010
submit_args = blast.sh
start_time = Mon Aug 23 10:21:00 2010
Walltime.Remaining = 359
start_count = 1
fault_tolerant = False
```

Requesting Queues

Directive:
#PBS -q <queue_name>

Example:

```
#!/bin/bash
#PBS -q urgent

echo "Script begins here"
```

```
<- omitted ->
```

Command Line:
qsub -q <queue_name> *script*

Example:
```
qsub -q default blast.sh
```

Practicum:
```
[jdpoisso@umms-amino blast_test]$ qsub blast.sh
2498963.umms-amino.ccmb.med.umich.edu
[jdpoisso@umms-amino blast_test]$ qsub -q urgent blast.sh
2498965.umms-amino.ccmb.med.umich.edu
[jdpoisso@umms-amino blast_test]$ qstat -f 2498963
Job Id: 2498963.umms-amino.ccmb.med.umich.edu
    Job_Name = blast.sh
<- output omitted ->
    queue = urgent
<- output omitted ->
[jdpoisso@umms-amino blast_test]$ qstat -f 2498965
Job Id: 2498965.umms-amino.ccmb.med.umich.edu
    Job_Name = blast.sh
<- output omitted ->
    queue = urgent
<- output omitted ->

[jdpoisso@umms-amino blast_test]$
```

Requesting Walltime

Directive:
#PBS -l walltime=<hours:minutes:seconds>

Example:
```
#!/bin/bash
#PBS -l walltime=10:00:00

echo "Script begins here"
<- omitted ->
```

Command Line:
qsub -l walltime=<hours:minutes:seconds> *script*

Example:
```
qsub -l walltime=12:00:00
```

Practicum:
```
[jdpoisso@umms-amino blast_test]$ qsub blast.sh
2499949.umms-amino.ccmb.med.umich.edu
[jdpoisso@umms-amino blast_test]$ qsub -l walltime=12:00:00 blast.sh
2499950.umms-amino.ccmb.med.umich.edu
[jdpoisso@umms-amino blast_test]$ qstat -f 2499949
```

```
Job Id: 2499949.umms-amino.ccmb.med.umich.edu
    Job_Name = blast.sh
<- output omitted ->
    Resource_List.walltime = 10:00:00
<- output omitted ->
    Walltime.Remaining = 3598
<- output omitted ->
[jdpoisso@umms-amino blast_test]$ qstat -f 2499950
Job Id: 2499950.umms-amino.ccmb.med.umich.edu
    Job_Name = blast.sh
<- output omitted ->
    Resource_List.walltime = 12:00:00
<- output omitted ->
    Walltime.Remaining = 4319
<- output omitted ->
[jdpoisso@umms-amino blast_test]$
```

Requesting Multiple Nodes and cpus

Directive:
#PBS -l nodes=<Number of Slices>:ppn=<Size of Slices>

Example:
```
#!/bin/bash
#PBS -l nodes=2:ppn=4

echo "Script begins here"
```

Command Line:
qsub -l nodes=<Number of Slices>:ppn=<Size of Slices> *script*

Example:
```
qsub -l nodes=2:ppn=4 blast.sh
```

Praticum:
```
[jdpoisso@umms-amino blast_test]$ qsub blast.sh
2500607.umms-amino.ccmb.med.umich.edu
[jdpoisso@umms-amino blast_test]$ qsub -l nodes=2:ppn=4 blast.sh
2500608.umms-amino.ccmb.med.umich.edu
[jdpoisso@umms-amino blast_test]$ qstat -f 2500607
Job Id: 2500607.umms-amino.ccmb.med.umich.edu
    Job_Name = blast.sh
<- output omitted ->
    exec_host = compute-6-18/5+compute-6-18/4+compute-6-18/3+compute-6-
18/2+co mpute-6-17/5+compute-6-17/4+compute-6-17/3+compute-6-17/2
<- output omitted ->
    Resource_List.nodect = 2
    Resource_List.nodes = 2:ppn=4
<- output omitted ->
[jdpoisso@umms-amino blast_test]$ qstat -f 2500608
Job Id: 2500608.umms-amino.ccmb.med.umich.edu
    Job_Name = blast.sh
<- output omitted ->
    exec_host = compute-6-15/5+compute-6-15/4+compute-6-15/3+compute-6-
```

```
    15/0+co mpute-6-13/5+compute-6-13/4+compute-6-13/3+compute-6-13/0
<- output omitted ->
    Resource_List.nodect = 2
    Resource_List.nodes = 2:ppn=4
<- output omitted ->
[jdpoisso@umms-amino blast_test]$
```

Note – **exec_host** : The exec host line includes the location of each *task* assigned to the job. It takes the form of *node_name* / *cpu_number*, adding each node name and cpu number together until you reach the total number of tasks selected. Tasks is equal to the number of nodes (**nodes**) multiplied by the processors per node (**ppn**).

Note: The preferred method of requesting multiple cores and nodes is to use the nodes : processors per node (nodes=X:ppn=X) method. There are other methods and features that can be used when requesting multiple tasks. Some of these methods are discussed in the advanced section, while others are deprecated or unique to certain cluster configurations. If the method of selection is relevant to your cluster, you should reference any cluster specific documentation available.

Requesting Multiple Nodes and CPUs - Advanced

The default method used when allocating nodes and processors is to break the request into *slices*, where the size of each slice is equal to the value of **ppn**, and the number of slices is equal to the value of **nodes**. On some clusters, the scheduler will not necessarily separate different slices on to separate nodes, rather it will try to assign as many slices onto individual nodes as it can according to resource availability. This can lead to situations where you may request 4 nodes with 2 processors per node, but you are assigned 2 nodes with 6 processors on one node, while the second has only 2 processors assigned.

Example:

```
[jdpoisso@umms-amino blast_test]$ qstat -f 2500610
Job Id: 2500610.umms-amino.ccmb.med.umich.edu
   Job_Name = blast.sh
    exec_host = compute-6-14/7+compute-6-14/6+compute-6-14/5+compute-6-
    14/4+co mpute-6-14/3+compute-6-14/0+compute-6-12/3+compute-6-12/0
<- output omitted ->
   Resource_List.nodect = 4
   Resource_List.nodes = 4:ppn=2
<- output omitted ->
```

This behavior originated to accommodate, a system in which there are many different kinds of jobs running with different requirements. When a cluster is in general use, some nodes may have many processors free, while another may only have a few free if any. In these cases it make take a great deal of time for the appropriate number of nodes with the appropriate number of processors to become available, where if the job were just assigned slices as available, it could run much sooner. Depending on the type of job you are running this may be a very bad idea. Due to factors, such as memory issues or disk access behavior you may want to restrict a node to *only* the number of tasks specified, or you may be willing to wait until the scheduler can free nodes exclusively for your job.

Explicitly Assigning Slices to Separate Nodes

Directive:

#PBS -W x=nmatchpolicy:EXACTNODE

Example:

```
#!/bin/bash
#PBS -l nodes=4:ppn=2
#PBS -W x=nmatchpolicy:EXACTNODE

echo "Script begins here"
```

Command Line:

qsub -l <resource_request> -W x=nmatchpolicy:exactnode *script*

Example:

```
qsub -l nodes=4:ppn=2 -W x=nmatchpolicy:exactnode blast.sh
```

Practicum:

```
[jdpoisso@umms-amino blast_test]$ qsub blast.sh
2500749.umms-amino.ccmb.med.umich.edu
[jdpoisso@umms-amino blast_test]$ qsub -l nodes=4:ppn=2 -W
x=nmatchpolicy:exactnode blast.sh
2500750.umms-amino.ccmb.med.umich.edu
[jdpoisso@umms-amino blast_test]$ qstat -f 2500749
Job Id: 2500749.umms-amino.ccmb.med.umich.edu
    Job_Name = blast.sh
<- output omitted ->
    exec_host = compute-4-31/3+compute-4-31/0+compute-4-30/3+compute-4-
      30/2+co mpute-4-29/3+compute-4-29/2+compute-4-26/3+compute-4-26/1
<- output omitted ->
    Resource_List.nodect = 4
    Resource_List.nodes = 4:ppn=2
<- output omitted ->
[jdpoisso@umms-amino blast_test]$ qstat -f 2500750
Job Id: 2500750.umms-amino.ccmb.med.umich.edu
    Job_Name = blast.sh
<- output omitted ->
    exec_host = compute-4-2/3+compute-4-2/0+compute-4-1/3+compute-4-
      1/0+comput e-3-31/3+compute-3-31/0+compute-3-27/3+compute-3-27/0
<- output omitted ->
    Resource_List.nodect = 4
    Resource_List.nodes = 4:ppn=2
<- output omitted ->
[jdpoisso@umms-amino blast_test]$
```

Explicitly Restricting a Node to a Single Job

Note: Multiple slices may still be assigned to a single node, you may combine with the method specified prior.

Directive:

#PBS -W x=naccesspolicy:singlejob

Example:

```
#!/bin/bash
```

```
#PBS -l nodes=4:ppn=2
#PBS -W x=naccesspolicy:singlejob

echo "Script begins here"
```

Command Line:

qsub -l <resource_request> -W x=naccesspolicy:singlejob *script*

Example:

```
qsub -l nodes=4:ppn=2 -W x=naccesspolicy:singlejob blast.sh
```

Practicum:

```
[jdpoisso@axiom qsub]$ qsub blast.sh
1581.axiom.localdomain
[jdpoisso@axiom qsub]$ qsub -l nodes=3:ppn=2 -W
x=naccesspolicy:singlejob,nmatchpolicy:exactnode script.sh
1582.axiom.localdomain
[jdpoisso@axiom qsub]$ qstat -f 1581
Job Id: 1581.axiom.localdomain
    Job_Name = blast.sh
<- output omitted ->
    exec_host = compute-0-14/7+compute-0-14/6+compute-0-14/5+compute-0-
      14/4+compute-0-14/3+compute-0-14/2+compute-0-14/1+compute-0-14/0
<- output omitted ->
    Resource_List.nodect = 4
    Resource_List.nodes = 4:ppn=2
<- output omitted ->
[jdpoisso@axiom qsub]$ qstat -f 1582
Job Id: 1582.axiom.localdomain
    Job_Name = blast.sh
<- output omitted ->
    exec_host = compute-0-13/5+compute-0-13/4+compute-0-13/3+compute-0-
      13/2+compute-0-13/1+compute-0-13/0
<- output omitted ->
    Resource_List.nodect = 3
    Resource_List.nodes = 3:ppn=2
<- output omitted ->
```