Linux and Lonestar 5

- Overview:
- Objectives:
- Tutorial:
    - Logging into lonestar5
    - Setting up your lonestar profile and other variables
    - Understanding what your .bashrc file actually does.
    - Editing files
    - Stringing commands together and controlling their output
    - Understanding TACC
        - Diagram of Lonestar5 directories: What connects to what, how fast, and for how long.
        - Understanding "jobs" and compute nodes.
            - Further sbatch reading
            - Using launcher_creator.py
        - Running a job
        - Interrogating the launcher queue
        - Evaluating your first job submission
        - Moving beyond the preinstalled commands on TACC
            - TACC modules
            - Transferring files to and from lonestar with a Mac/Linux machine
            - Transferring files to and from lonestar with Windows

Overview:

This portion of the class is devoted to making sure we are all starting from the same starting point on lonestar. This tutorial is adapted from a previous version which allowed for set up on the now decommissioned lonestar4. Portions of this tutorial were adapted from previous versions which can be found here, here, here, here, here, and here. Collective thanks to all those that contributed to those works which now appear in a single version. Anyone wishing to use this tutorial is welcome.

Objectives:

1. Log into lonestar5.
2. Change your lonestar profile to the course specific format.
3. Refresh understanding of basic linux commands with some course organization.
4. Review use of the nano text editor program, and become familiar with several other text editor programs.

Tutorial:

• Logging into lonestar5

Start a new terminal window. For MACs this is done by clicking on the magnifying glass on the right hand side of the toolbar at the top of the page and type "terminal". For windows this should be done by connecting through cygwin. Log into lonestar using your account information.

This brings us to our first "code block". There will be 3 types of code blocks used throughout this class:

1. Visible
    a. These are code blocks that you would have no idea what to type without help.
    b. These will typically be associated with longer/more detailed text above the text box explaining things.
2. Hinted
    a. These are code blocks that you can probably figure out what to type with a hint that goes beyond what the tutorial is requesting. Access the hint by clicking the triangle or hint hyperlink.
    b. These will always contain an additional hidden code block incase you don't find the hint as clever as we did.
3. Hidden
    a. These code blocks represent things that either there is a good chance you know how to do already, something too straightforward to warrant a hint, or are there to give you the answer if the hint doesn't help. Access the answer by clicking "expand source" on the right hand side of the code block.

Text inside of code blocks represent "right" answers, and should either be typed EXACTLY into the terminal window as they are, or copy pasted with a noteable exception. Things that exist within <> symbols represent something that you need to replace before sending it to the terminal. We try to put informative text within the brackets so you know what to replace it with. If you are ever unsure of what to replace the <> text with, just ask.

Using what we have just taught you about code blocks, log into lonestar. Since this is your first code box, it is probably worth expanding even if you know how to log into lonestar already.

How to log into lonestar

```
ssh <username>@ls5.tacc.utexas.edu
```

When prompted enter your password, and answer "yes" to the security question.

Logging into remote computers

As a matter of internet safety, the terminal window knows you are entering a password and may not want your neighbor to see what it is. For this reason, even as you type to enter your password, nothing will be displayed on the screen. While backspace will work if you know you made a mistake, we often find it better to just hit enter and try again.

If you have never logged into lonestar from the computer you are currently using before, you will be issued a security warning. The same will be true if you log into any of the other TACC resources, or any other remote computer. If you ever see a security warning logging into somewhere that you use commonly you should answer no and try to figure out why you were warned. Otherwise type "yes" to bypass the security check.

- Setting up your lonestar profile and other variables

There are many flavors of Linux/Unix shells. The default for TACC's Linux (and most other Linuxes) is **bash** (**b**ourne **a**gain **sh**ell), which we will use throughout.

Whenever you login via an interactive shell as you did above, a well-known script is executed by the shell to establish your favorite environment settings. We've set up a common profile for you to start with that will help you know where you are in the file system and make it easier to access some of our shared resources. If you already have a profile set up on lonestar that you like, we want to make sure that we don't destroy it but it will be important to make sure that we change it temporarily. Use the ls command to check if you have a **profile** already set up in your home directory.

Use ls to check if particular file exists

```
cdh
ls .profile
ls .bashrc
```

If you already have a **.profile** or **.bashrc** file, use the mv command to change the name to something descriptive (for example "**.profile_pre_bdib_backup**"). Otherwise continue to creating a new files.

Use mv to change your .profile file to a backup copy

```
mv .profile .profile_pre_bdib_backup
mv .bashrc .bashrc_pre_bdib_backup
```

The BioITeam has several useful programs, libraries, and scripts globally available on the head node, but these useful things are not available from any of the compute or interactive nodes. We will explain more about this soon, but for the time being just know that there are things that you only sometimes have access to currently, and we want you to have access to them all the time so we have to copy some things into specific locations to make sure everyone is working with the same set up throughout the course. After you have finished taking the course you may find additional useful things in the BioITeam locations, and the things that you copy may get updated from time to time. On the last day of the course we'll go through how to sync the things you have copied and how to access additional community tools that we won't use in this course, so if you like foreshadowing, you are welcome.

We will explain more about what the different areas of tacc are shortly, but for now we are going to execute some commands that will make it a bit more useful. First we will make 2 new directories on the $WORK partition to serve as locations to copy things from the BioITeam. Using the **mkdir** command, create a new directory named **src** and a directory inside of the src directory named **BioITeam**.

Create new directories on your work partition named src and BioITeam

```
cd $WORK
mkdir src
mkdir src/BioITeam
```

You may have noticed that we executed the **mkdir** commands sequentially. This is done to make sure that the directory exists before trying to put a new directory inside of it. This leads us to an interesting and important thing to consider. How should we name files and folders? In general you will want to adopt a consistent pattern of naming, and it should be your own and something that makes sense to you. The most important thing to get used to is the convention of using **.** or _ in names rather than spaces in names, and limiting your use of any other punctuation. Spaces are great for mac and windows folder names when you are using visual interfaces, but on the command line, a space is a signal to start doing something different. Imagine instead of a **Bio ITeam** folder you wanted to make it a little easier to read and wanted to call it "**Bio I Team**" certainly everyone would agree its easier to read that way, but because of the spaces, bash will think you want to create 3 folers, 1 named **Bio** another named **I** and a third named **Team**. Now this is certainly behavior you can use when appropriate to your advantage, but generally speaking spaces will not be your friend. Early on in my computational learning I was told "A computer will always do exactly what you told it to do. The trick is telling it to do what you want it to do".

This is hidden away to keep you from accidentally thinking that this is a good idea. If for some reason you encounter spaces in the file names or directories that you are working with, (assumably because a colleague sent you some data, and not because you thought it was a good idea personally) spaces can be "escaped" like many other special characters. Imagine someone sent you directory name "This is really annoying to use, but I don't know it yet" to change into that directory you would have to type:

```
cd this\ is\ really\ annoying\ to\ use\ but\ I\ don\'t\ know\ it\ yet
```

Notice that the apostrophe also had to be escaped, which should help show you not to use other punctuation.

Now that we have the directories created to copy BioITeam materials into lets copy the **bin**, **python2.7**, **lib**, **local**, and **perl5** directories from the **/corral-repl/utexas/BioITeam** directory to your $WORK/src/BioITeam directory. Remember, that you want to copy them recursively so you get all the contents of those folders as well.

List of commands to copy

```
cd $WORK/src/BioITeam
cp -r /corral-repl/utexas/BioITeam/bin .
cp -r /corral-repl/utexas/BioITeam/python2.7 .
cp -r /corral-repl/utexas/BioITeam/lib .
cp -r /corral-repl/utexas/BioITeam/local .
cp -r /corral-repl/utexas/BioITeam/perl5 .
cp -r /corral-repl/utexas/BioITeam/breseq .
```

Some of these copy commands may take a few minutes to complete (the bin directory specifically) and you may see some permissions errors such as the following. This is expected and not concerning.

cp: cannot open `/corral-repl/utexas/BioITeam/bin/smrtanalysis-2.0.1/analysis/lib/python2.7/networkx-1.1-py2.7.egg/networkx/drawing/nx_pydot.pyc' for reading: Permission denied

When the last of the above commands has finished, copy our predefined **GVA2016.bashrc** file from the `/corral-repl/utexas/BioITeam/scripts/` folder to your $HOME folder as **.bashrc** before using the chmod command to change the permissions to read and write for the user only.

Copy the course provided .profile file and change its name and permissions

```
cp /corral-repl/utexas/BioITeam/scripts/GVA2016.bashrc .bashrc
cp /corral-repl/utexas/BioITeam/scripts/GVA2016.profile .profile
chmod 700 .bashrc
chmod 700 .profile
```

The **chmod 700 <FILE>** command marks the file as readable/writable/executable only by you. The **.bashrc** script file will not be executed unless it has these permissions settings.

Notice that when you do a normal **ls** to list the contents of your home directory, this file doesn't appear. That's because it's a hidden "dot file" – a file that has no filename, only an extension. To see these hidden files use the **-a** (all) switch for **ls**:

How to see hidden and not hidden files in linux

```
ls -a
```

To see even more detail, including file permissions, add the **-l** (long listing) switch:

How to see details about hidden and not hidden files in linux

```
ls -la
```

Since **.bashrc** is executed when you login, to ensure it is set up properly you should first logout:

How to leave Lonestar by logging out

```
exit
```

then log back in:

Go log back in to Lonestar

```
ssh <username>@ls5.tacc.utexas.edu
```

If everything is working correctly you should now see a prompt like this: tacc:~$

In order to make navigating to the different file systems on lonestar a little easier ($SCRATCH and $WORK), you can set up some shortcuts with these commands that create folders that "link" to those locations. Run these commands when logged into Lonestar with a terminal, from your home directory.

Creating a shortcut to the main Lonestar working directories

```
cdh
ln -s $SCRATCH scratch
ln -s $WORK work
ln -s $BI BioITeam
```

- Understanding what your .bashrc file actually does.

Let's look at what your **.bashrc** profile actually does. Use the **cat** command to print contents of the **.bashrc** file to the screen.

Print the contents of the .profile file to the screen

```
cat .bashrc
```

This will print several lines of text to the terminal window. Let's look at what some of these lines do with a little more information:

- `lines that start with #`
    - Any line begins with a # symbol, it is "commented out". Anything after a # symbol will not be executed by any program. Programers commonly make use of behavior to leave notes for others, or even themselves at a later date as to what particular lines of a script are actually doing.
- `Section 1 has multiple lines involving "module load <NAME>"`
    - This loads different modules by default. We have included ones that we will use throughout the course and that you will commonly make use of. After we review the use of the nano text editor we'll go into more depth with TACC modules. But for now trust us when we say that not having to load a bunch of modules everytime you log into TACC is a good thing.
- `Section 2 has multiple lines starting with "export"`
    - The **export** lines define shell variables for example **BI** and **PATH**. You've already seen how using **$BI** can come in handy accessing our shared course directory. As for **PATH**, that is a well-known environment variable that defines a set of directories where the shell will look when you type in a program's name. Our shared profile adds the common course directories that we copied at the start of this tutorial and your local **~/local/bin** directory (which does not exist yet) to the location list. You can see the entire list of locations by doing this:

      How to see where the bash shell looks for programs

      ```
      echo $PATH
      ```

As you can see, there are a lot of locations on the path. That's because when you load modules at TACC (see above), that mechanism makes the programs available to you by putting their installation directories on your $PATH.

- `umask 002`
    - The **umask** command is used to set the default permissions of newly created files and directories limiting the need to use the **chmod** co mmand. **umask** functions as the inverse of chmod meaning that it *subtracts* the values from the default permissions. In this case the command **umask 002** is the equivalent of the command **chmod 775** for directories, and **chmod 664** for files. in summary, having this command in your **.profile** gives all new files you create read and write access to both you and your group while giving read only access to everyone else.
- `PS1='tacc:\w$ '`
    - The **PS1='tacc:\w$ '** line is a special setting that tells the shell to display the current directory as part of its prompt. It saves you typing **pw** all the time to see where you are in the directory hierarchy. Try using the mkdir command to make a new directory called **tmp** and change into that directory to see what it does to your prompt.

      See how your prompt reflects your current directory

      ```
      mkdir tmp
      cd tmp
      ```

    - Your prompt should have changed from: "`tacc:~$`"to now be "`tacc:~/tmp$`". Your prompt now tells you you are in the **tmp** subdirecto ry of your home directory (**~**). See if you can figure out how to return to your home directory without expanding the code block. Expand the following code block to see the different ways of returning to your home directory

      How to return to your home directory

      ```
      cd
      cdh
      cd $HOME
      cd ~
      ```

Editing files

There are a number of options for editing files at TACC. These fall into three categories:

- Linux text editors installed at TACC (**nano**, **vi**, **emacs**). These run in your terminal window. **vi** and **emacs** are extremely powerful but also quite complex, so **nano** may be the best choice as a first local text editor.
- Text editors or IDEs that run on your local computer but have an SFTP (secure FTP) interface that lets you connect to a remote computer (**Notepa d++** or **Komodo Edit**). Once you connect to the remote host, you can navigate its directory structure and edit files. When you open a file, its contents are brought over the network into the text editor's edit window, then saved back when you save the file.
- Software that will allow you to mount your home directory on TACC as if it were a normal disk e.g. MacFuse/MacFusion for Mac, or ExpanDrive for Windows or Mac ($$, but free trial). Then, you can use any text editor to open files and copy them to your computer with the usual drag-drop.

We'll go over `nano` together in class, but you may find these other options more useful for your day-to-day work so feel free to go over these sections in your free time to familiarize yourself with their workings to see if one is better for you.

Komodo Edit is another free, full-featured text editor with syntax coloring for many programming languages and a remote file editing interface. It has versions for both Macintosh and Windows. Download the appropriate install image here.

Once installed, start Komodo Edit and follow these steps to configure it:

- Configure the default line separator for Unix
    - On the **Edit** menu select **Preferences**
    - Select the **New Files** Category
    - For **Specify the end-of-line (EOL) indicator for newly created files** select **UNIX (\n)**
    - Select OK
- Configure a connection to TACC
    - On the **Edit** menu select **Preferences**
    - Select the **Servers** Category
    - For **Server type** select **SFTP**
    - Give this profile the **Name** of **Lonestar**
    - For **Hostname** enter **ls5.tacc.utexas.edu**
    - Enter your TACC user ID for **Username**
    - Leave **Port** and **Default path** blank
    - Select OK

When you want to open an existing file at Lonestar, do the following:

- Select the **File** menu -> **Open** -> **Remote File**
    - Select your **Lonestar** profile from the top **Server** drop-down menu

    - Once you log in, it should show you all the files and directories in your lonestar $HOME directory
- Navigate to the file you want and open it
    - Often you will use the **work** or **scratch** directory links to help you here

To create and save a new file, do the following:

- From the Komodo Edit **Start Page**, select **New File**
    - Select the file type (Text is good for commands files)
- Edit the contents
- Select the **File** menu -> **Save As Other** -> **Remote File**
    - Select your **Lonestar** profile from the **Server** drop-down menu
    - Once you log in, it should show you all the files and directories in your lonestar $HOME directory
- Navigate to where you want the put the file and save it
    - Often you will use the **work** or **scratch** directory links to help you here

Notepad++ is an open source, full-featured text editor for Windows PCs (not Macs). It has syntax coloring for many programming languages (Python, Perl, shell), and a remote file editing interface.

If you're on a Windows PC download the installer here.

Once it has been installed, start **Notepad++** and follow these steps to configure it:

- Configure the default line separator for Unix
    - In the **Settings** menu, select **Preferences**
    - In the **Preferences** dialog, select the **New Document/Default Directory** tab.
    - Select **Unix** in the **Format** section
    - **Close**
- Configure a connection to TACC
    - In the **Plugins** menu, select **NppFTP**, then select **Focus NppFTP Window**. The top bar of the **NppFTP** panel should become blue.
    - Click the **Settings** icon (looks like a gear), then select **Profile Settings**
    - In the **Profile settings** dialog click **Add new**
    - Call the new profile **lonestar**
    - Fill in **Hostname** (ls5.tacc.utexas.edu) and your TACC user ID
    - **Connection type** must be SFTP
    - Close

To open the connection, click the blue **(Dis)connect** icon then select **lonestar** connection. It should prompt for your password. Once you've authenticated, a directory tree ending in your home directory will be visible in the **NppFTP** window. You can click the the **(Dis)connect** icon again to Disconnect when you're done.

Since much of the editing we'll do will be in your SCRATCH area at TACC, rather than having to navigate around TACC's complex file system tree, it helps to create symbolic links to your WORK and SCRATCH directory in your home directory. Then you'll be able to get there just by clicking on the **scratch** or **w ork** folder in the **Notepad++** Remote directory tree. See below for how to do this.
Want your Lonestar files to appear like any other place on your hard drive? You can do this using MacFuse/MacFusion on a Mac.

Want to edit files on TACC without having to use `nano`? You might want to use TextWrangler, a text editor that can edit files over ssh.

Editing Text Files on TACC: TextWrangler

TextWrangler is a recommended FreeWare text editor for MacOS X. (It even keeps with the theme TACC has going with naming its clusters!) You can use it to directly edit text files on Lonestar with OSXFuse/MacFusion using a nice GUI. It is a much more powerful text editor than TextEdit, and won't trip you up by wrapping lines etc., if you learn to use it.

Even if you cannot install OSXFuse/MacFusion, TextWrangler allows you to edit a remote file via SSH. To do this:

1. Select *File > Open from FTP/SFTP Server...
2. Type `ls5.tacc.utexas.edu`, your username, and your password into the appropriate boxes.
3. Check the You need to check the SFTP box.
4. Click connect.
5. You will now have a file browser window. You can create new files and edit existing files on lonsetar, but won't be able to drag-and-drop copy files.

Tip: Files beginning in a dot (.) like (.profile_user) are "hidden" and won't show up when you are navigating in Finder (if using OSXFuse/MacFusion). There is a way to turn on showing these files in finder, but it can get annoying because they will show up everywhere. If you use the TextWrangler "open" command to open a file, there is a box that you can check to show these files.

Connecting to TACC Like a Hard Drive: MacFuse/MacFusion

Here are the steps for an installation:

1. Download and install FUSE for OS X.
    - Check the option to install the "compatibility layer"
2. Download MacFusion.
    - Move the app that gets downloaded to your Applications folder
3. Restart your computer.
4. Open the MacFusion application.
5. Click the + menu in the window and select SSHFS. Enter your login information for lonestar. Choose connect. The remote file system will appear in Finder (depending on your settings it may be on the desktop or inside the computer shortcut in the side of a Finder window). You can also click on the mounted volume within MacFusion and choose "Reveal" from the gear menu.

Copying Files To and From TACC: SFTP Clients

If you can't get OSXFuse/MacFusion to work, you can still copy files back and forth between your computer and TACC using a secure FTP (SFTP) client. Some examples of free programs for Mac are:

- Cyberduck
- Fetch

As we will be using nano throughout the class, it is a good idea to review some of the basics. **nano** is a very simple editor available on most Linux systems. If you are able to use ssh, you can use nano. To invoke it, just type:

How to start the nano text editor

```
nano
```

You'll see a short menu of operations at the bottom of the terminal window. The most important are:

- **ctl-o** - write out the file
- **ctl-x** - exit **nano**
  You can just type in text, and navigate around using arrow keys. A couple of other navigation shortcuts:
- **ctl-a** - go to start of line
- **ctl-e** - go to end of line

Be careful with long lines – sometimes **nano** will split long lines into more than one line, which can cause problems in our commands files, as you will see.

• Stringing commands together and controlling their output

In a linux shell, it is often useful to take output of one command save it to a new file rather than having it print to the screen. It uses a familiar metaphor: "pipes". The linux operating system expects some "standard input pipe" and gives output back through a "standard output pipe". These are called "stdin" and "stdout" in linux. There's also a special "stderr" for errors; we'll ignore that for now. Usually, your shell is filling the operating system's stdin with stuff you type - the commands with options. The shell passes responses back from those commands to stdout, which the shell usually dumps to your screen. The ability to switch stdin and stdout around is one of the key reasons linux has existed for decades and beat out many other operating systems. Let's start making use of this. Change to the scratch directory and make a new folder called "piping" and put list of the full contents of the $BI folder to a new file called **whatsHere**.

Redirecting STDOUT

```
cds
mkdir piping
ls -1 $BI > whatsHere
cat whatsHere
```

When you execute the `ls -1 > whatsHere` command, you should have noticed nothing happening on the screen, and when you cat the **whatsHere** file, you should notice the output you would have expected from the `ls -1 > whatsHere` command. Often it is useful to chain commands together using the output of the first command as the input of the second command. Commands are chained together using the "|" character (shift \ above the return key). Use redirection to put the first 2 lines of the $BI directory contents into the `whatsHere` file.

Piping one command's output to another, and then redirecting STDOUT to a file

```
ls -1 $BI| head -2 > whatsHere
cat whatsHere
```

Again, you should see your answer only showing up after the cat command. Note that by using a single > you are overwriting the existing contents and that there is no warning that this is happening beware of this in the future as linux doesn't have an "undo" feature. We will make use of the redirect output (stdout) character (**>**), and the "pass output along as input" "**|**" throughout the course. Not all shells are equal - the bash shell lets you redirect stdout with either > or 1>; stderr can be redirected with 2>; you can redirect both stdout and stderr using &>. If these don't work, use google to try to figure it out. The web site stackoverflow is a usually trustworthy and well annotated site for OS and shell help.

• Understanding TACC

Now that we've been using lonestar for a little bit, and have it behaving in a way that is a little more useful to us, let's get more of a functional understanding of what exactly it is and how it works.

Diagram of Lonestar5 directories: What connects to what, how fast, and for how long.

Lonestar is a collection of 1,252 computers with 24 cores connected to three file servers, each with unique characteristics. You need to understand the file servers to know how to use them effectively.

	$HOME	$WORK	$SCRATCH
Purged?	No	No	Files can be purged if not accessed for 10 days.
Backed Up?	Yes	No	No
Capacity	5GB	1TB	Basically infinite.
Commands to Access	cdh cd $HOME/	cdw cd $WORK/	cds cd $SCRATCH/
Purpose	Store Executables	Store Files and Programs	Run Jobs

Executables that aren't available on TACC through the "module" command should be stored in $HOME.

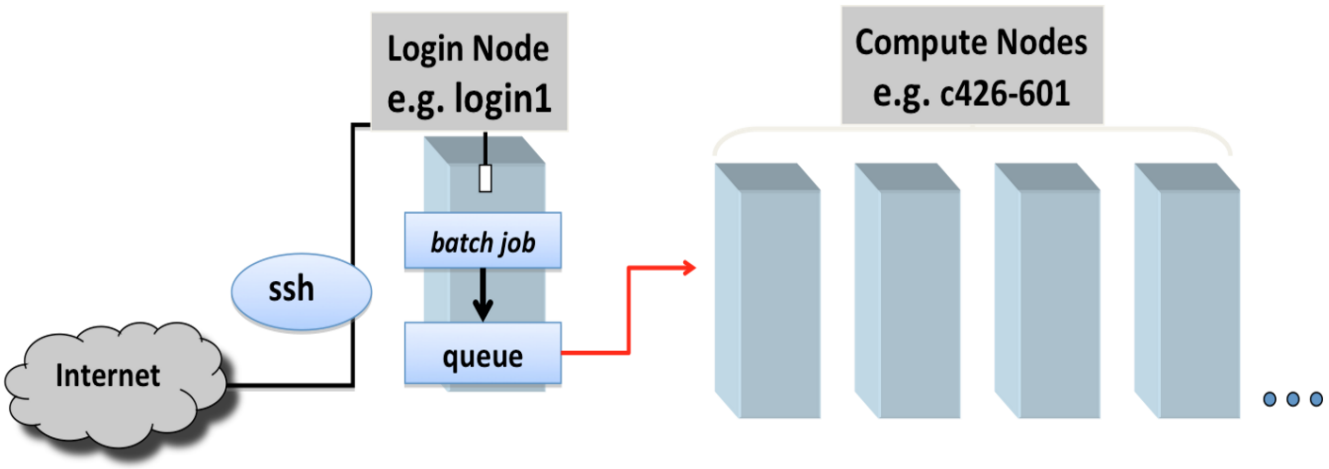
If you plan to be using a set of files frequently or would like to save the results of a job, they should be stored in $WORK.

If you're going to run a job, it's a good idea to keep your input files in a directory in $WORK and copy them to a directory in $SCRATCH where you plan to run your job.

Example command for copying data from a $WORK directory to $SCRATCH

```
cp $WORK/my_fastq_data/*fastq $SCRATCH/my_project/
```

Understanding "jobs" and compute nodes.



When you log into lonestar using **ssh** you are connected to what is known as the login node or "the head node". There are several different head nodes, but they are shared by everyone that is logged into lonestar (not just in this class, or from campus, or even from texas, but everywhere in the world). Anything you type onto the command line has to be executed by the head node. The longer something takes to complete, or the more it will slow down you and everybody else. Get enough people running large jobs on the head node all at once (say a classroom full of Big Data in Biology summer school students) and lonestar can actually crash leaving nobody able to execute commands or even log in for minutes -> hours -> even days if something goes really wrong. To try to avoid crashes, TACC tries to monitor things and proactively stop things before they get too out of hand. If you guess wrong on if something should be run on the head node, you may eventually see a message like the one pasted below. If you do, its not the end of the world, but repeated messages will become revoked TACC access and emails where you have to explain what you are doing to TACC and your PI and how you are going to fix it and avoid it in the future.

Example of how you learn you shouldn't have been on the head node

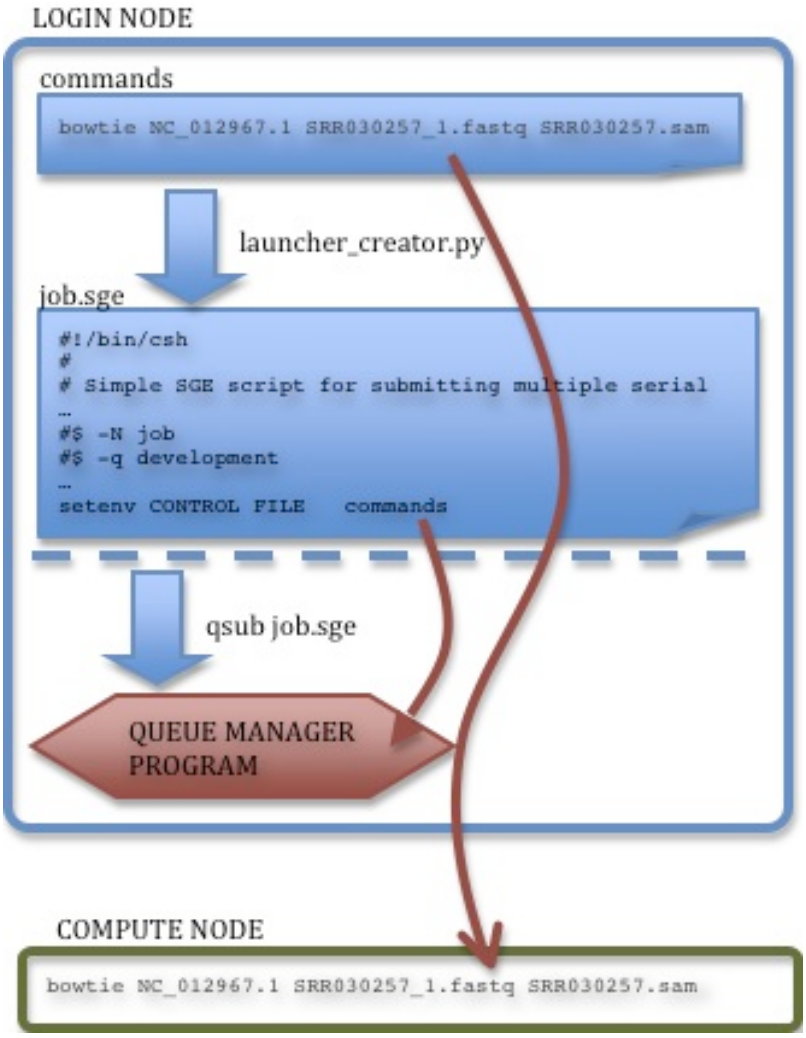
```
Message from root@login1.ls4.tacc.utexas.edu on pts/127 at 09:16 ...
Please do not run scripts or programs that require more than a few minutes of
CPU time on the login nodes.  Your current running process below has been
killed and must be submitted to the queues, for usage policy see
http://www.tacc.utexas.edu/user-services/usage-policies/
If you have any questions regarding this, please submit a consulting ticket.
```

So you may be asking yourself what the point of using lonestar is at all if it is wrought with so many issues. The answer comes in the form of compute nodes. There are 1,252 compute nodes that can only be accessed by a single person for a specified amount of time. These compute nodes are divided into different queues called: normal, development, largemem, etc. Access to nodes (regardless of what queue they are in) is controlled by a "Queue Manager" program. You can personify the Queue Manager program as: Heimdall in Thor, a more polite version of Gandalf in lord of the rings when dealing with with the balrog, the troll from the billy goats gruff tail, or any other "gatekeeper" type. Regardless of how nerdy your personification choice is, the Queue Manager has an interesting caveat: you can only interact with it using the **sbatch** command. "sbatch <filename.slurm>" tells the que manager to run a set job based on information in filename.slurm (i.e. how many nodes you need, how long you need them for, how to charge your allocation, etc). The Queue manager doesn't care **WHAT** you are running, only **HOW** to find what you are running (which is specified by a `setenv CONTROL_FILE commands` line in your filename.slurm file). The **WHAT** is then handled by the file "commands" which contains what you would normally type into the command line to make things happen.

Further sbatch reading

- Complete explanation of the options in sbatch

To make things easier on all of us, there is a script called launcher_creator.py that you can use to automatically generate a .slurm file. This can all be summarized in the following figure:



Using `launcher_creator.py`

We have created a Python script called `launcher_creator.py` that makes creating a .slurm file a breeze. Before learning to work with interactive compute nodes during the class, we will show you how you will most often do your analysis. Run the launcher_creator.py script with the `-h` option to show the help message so we can see what other options the script takes:

How to display all available options of the launcher_createor.py script

```
launcher_creator.py -h
```

Short option	Long option	Required	Description
-n	name	Yes	The name of the job.
-a	allocation		The allocation you want to charge the run to.
-q	queue	Default: Development	The queue to submit to, like 'normal' or 'largemem', etc.
-w	wayness		*Optional* The number of jobs in a job list you want to give to each node. (Default is 12 for Lonestar, 16 for Stampede.)
-N	number of nodes		*Optional* Specifies a certain number of nodes to use. You probably don't need this option, as the launcher calculates how many nodes you need based on the job list (or Bash command string) you submit. It sometimes comes in handy when writing pipelines.
-t	time	Yes	Time allotment for job, format must be hh:mm:ss.
-e	email		*Optional* Your email address if you want to receive an email from Lonestar when your job starts and ends.
-l	launcher		*Optional* Filename of the launcher. (Default is `<name>.sge`)
-m	modules		*Optional* String of module management commands. `module load launcher` is always in the launcher, so there's no need to include that.
-b	Bash commands		*Optional* String of Bash commands to execute.
-j	Command list		*Optional* Filename of list of commands to be distributed to nodes.
-s	stdout		*Optional* Setting this flag outputs the name of the launcher to stdout.

We should mention that `launcher_creator.py` does some under-the-hood magic for you and automatically calculates how many cores to request on lonestar, assuming you want one core per process. You don't know it, but you should be grateful that this saves you from ever having to think about a confusing calculation.

Running a job

Now that we have an understanding of what the different parts of running a job is, let's actually run a job. Move to your scratch directory, make a new folder called "my_first_job" (**Remember not to use spaces in file/folder names**), make a new file called "commands" inside of that directory using nano, and put 4-12 lines with 1 command on each line in that file, being sure to remember to pipe the output to 1 or more files.

how to make a sample commands file

```
# remember that things after the # sign are ignored by bash
cds  # move to your scratch directory
mkdir my_first_job  # make a new folder called "my_first_job"
cd my_first_job  # move into the new folder to make it easier to create a file there
nano commands

# the following lines should be typed into the nano editor so they will be saved to the new file "commands"
cat commands > commands.out  # this will print the contents of the file you are currently editing to a new file
called commands.out
date > date.out  # this will create a file with todays date on it
pwd > current_directory.out  # this will create a file with the current directory in it
echo "my name is <YOURNAME>" >> name.out  # Note that this time we used the append symbol >> not the write
symbol > as we plan to put multiple things into the same file. be sure to replace the <> signs with your name
echo "This is the final result of my first script. It worked how I thought it would, or hopefully have the
resources to figure out why it didn't" >> name.out  # this will add another line of text to the name.out file.
# feel free to add up to 7 more lines to your commands file here using the cat/ls/pwd/mkdir/other commands that
you know.
# beware using cd commands here as it will change your directory as if you were doing it on an interactive node
and may cause you to reference files that don't exist
# write and exit nano now ctrl-o ctrl-x
launcher_creator.py -n "my_first_job" -t 00:02:00 -a "UT-2015-05-18" # this will create a my_first_job.slurm
file that will run for 2 minutes
sbatch my_first_job.slurm  # this will actually submit the job to the Queue Manager and if everything has gone
right, it will be added to the development queue.
```

Interrogating the launcher queue

Here are some of the common commands that you can run and what they will do or tell you:

Command	Purpose	Output(s)
showq -u	Shows only your jobs	Shows all of your currently submitted jobs, a state of: "qw" means it is still queued and has not run yet "r" means it is currently running
scancel <job-ID>	Delete a submitted job before it is finished running **note**: you can only get the job-ID by using showq -u	There is no confirmation here, so be sure you are deleting the correct job. There is nothing worse than deleting a job that has sat a long time by accident because you forgot something on a job you just submitted.
showq	You are a nosy person and want to see everyone that has submitted a job	Typically a huge list of jobs, and not actually informative

If the queue is moving very quickly you may not see much output, but don't worry, there will be plenty of opportunity once you are working on your own data.

Evaluating your first job submission

Based on our example you may have expected 4 new files to have been created during the job submission, but instead you will find 3 extra files as follows: <job_name>.e(job-ID), <job_name>.pe(job-ID), and <job_name>.o(job-ID). When things have worked well, these files are typically ignored. When your job fails, these files offer insight into the why so you can fix things and resubmit.

Many times while working with NGS data you will find yourself with intermediate files. Two of the more difficult challenges of analysis can be trying to decide what files you want to keep, and remembering what each intermediate file represents. Your commands files can serve as a quick reminder of what you did so you can always go back and reproduce the data. Using arbitrary endings (.out in this case) can serve as a way to remind you what type of file you are looking at. Since we've learned that the scratch directory is not backed up and is purged, see if you can turn your 4 intermediate files into a single final file using the cat command, and copy the new final file, the .slurm file you created, and the 3 extra files to work. This way you should be able to come back and regenerate all the intermediate files if needed, and also see your final product.

make a single final file using the cat command and copy to a useful work directory

```
# remember that things after the # sign are ignored by bash
cat *.out > first_job_submission.final.output  # Remember that the * wildcard will take things in alpha order,
if you want you can list each file separately to control what order they go into the new file.
mkdir $WORK/BDIB_GVA_2016
mkdir $WORK/BDIB_GVA_2016/Day1
mkdir $WORK/BDIB_GVA_2016/Day1/first_tacc_job  # each directory must be made in order to avoid getting a no
such file or directory error
cp first_job_submission.final.output $WORK/BDIB_GVA_2016/Day1/first_tacc_job
cp *.slurm $WORK/BDIB_GVA_2016/Day1/first_tacc_job
cp *<job-ID> $WORK/BDIB_GVA_2016/Day1/first_tacc_job  #your job-id is the string of numbers following the .o
and .e filenames
```

Moving beyond the preinstalled commands on TACC

If (or when) you looked at what our edits to the .bashrc file did, you would have seen that the last lines were a series of "`module load XXXX`" commands, and a promise to talk more about them later. I'm sure you will be thrilled to learn that now is that time... As a "classically trained wet-lab biologist" one of the most difficult things I have experienced in computational analysis has been in installing new programs to improve my analysis. Programs and their installation instructions tend (or appear) to be written by computational biologists in what at times feels like a foreign language, particularly when a particular when things start going wrong. Luckily TACC (and the BioITeam) help get around a large number of these problems by preinstalling many programs if you know where to look.

TACC modules

Modules are programs or sets of programs that have been set up to run on TACC. They make managing your computational environment very easy. All you have to do is load the modules that you need and a lot of the advanced wizardry needed to set up the linux environment has already been done for you. New commands just appear.

To see **all** modules available in the current context, type:

list all modules

```
 module avail
```

Remember you can hit the "q" key to exit out of the "more" system, or just keep hitting return to see all of the modules available. The "`module avail`" command is not the most useful of commands if you have some idea of what you are looking for. For example imagine you want to align a few million next generation sequencing reads to a genome, but you don't know what your options are. You can use the following command to get a list of programs that may be useful:

List all modules containing a particular term

```
module keyword alignment
```

Note that this may not be an inclusive list as it requires the name of the program, or its description to contain the word "alignment". Looking through the results you may notice some of the programs you already know and use for aligning 2 sequences to each other such as blast and clustalw. Try broadening your results a little by searching for "align" rather than "alignment" to see how important word choice is. When you compare the two sets of results you will see that one of the new results is:

```
bowtie: bowtie/1.1.2, bowtie/2.2.6
    Ultrafast, memory-efficient short read aligner
```

This may sound much better, but you still only have limited information about it. To learn more about a particular program, try the following 2 commands:

Get more information on particular module

```
module spider bowtie

module spider bowtie/2.2.6
```

In the first case, we see information about what versions of bowtie lonestar has available for us, but really that is just the same information as we had from our previous search. This can be particularly useful when you know what program you want to use but don't know what versions are available. In the second case we now have more detailed information about the particular version of interest including websites we can go to to learn more about the program itself.

Once you have identified the module that you want to use, you install it using the following command:

```
module load bowtie/2.2.6
```

While not strictly necessary, using the "`/2.2.6`" text is a very good habit to get into as it controls what version is to be loaded. In this case the "2.2.6" version is the default version and `module load bowtie` will behave identically to `module load bowtie/2.2.6` but that will not always be the case, particularly if in the future TACC installs a new version of bowtie. Since the module load command doesn't give any output, it is often useful to check what modules you have installed with either of the following commands:

```
module list
module list bowtie
```

The first example will list all currently installed modules while the second will only list modules containing bowtie in the name. If you see that you have installed the wrong version of something, a module is conflicting with another, or just don't feel like having it turned on anymore, use the following command:

```
module unload bowtie
```

You will notice when you type module list you have several different modules installed already. These come from both TACC defaults (TACC, linux, etc), and several that are used so commonly both in this class and by biologists that it becomes cumbersome to type "`module load python`" all the time and therefore we just have them turned on by default by putting them in our profile to load on startup. As you advance in your own data analysis you may start to find yourself constantly loading modules as well. When you become tiered of doing this (or see jobs fail to run because the modules that load on the compute nodes are based on your .bashrc file plus commands given to each node), you may want to add additional modules to your .bashrc file. This can be done using the "nano .bashrc" command from your home directory.

Transferring files to and from lonestar with a Mac/Linux machine

Lonestar is tremendously powerful and capable of doing many things, but as most of you are probably being slightly frustrated by, it doesn't have much in the way of a GUI (graphical user interface), and does not have the same scrolling capabilities we are used to on our own computers, let alone actually visualizing graphs and more meaningful representations of our data. In order to do these types of things, we have to get our data off of lonestar and onto our own computers. On our diagram of lonestar we showed a boundary of what could be copied and moved within TACC and list the **scp** command as a way of moving files to other computers outside of TACC. **scp** works the same was as the **cp** command, it just includes more detailed information on the path of where the file is, or where the file is going. Here we will transfer our recently created "**first_job_submission.final.output**" file from lonestar to the computer you are sitting at as an example. First navigate to your work directory to find your final output file, and determine what the full path to that location is.

The **cd** command is used to change directories, the **ls** command lists what is in each directory, and the TAB key can be pressed in either to autocomplete paths, or double pressed to display all possible paths. The **pwd** command displays the full path.

Check your work, or get the answer

```
cd $WORK/BDIB_GVA_2016/Day1/first_tacc_job
ls
pwd
```

Next we'll transfer the file to a new "temp" directory on the computer using the scp command. Open a new terminal window and use the following code (make sure to replace <> and everything between them:

Use scp to transfer files from lonestar to your computer

```
mkdir temp
cd temp
ls
scp <username>@ls5.tacc.utexas.edu:<pwd_from_other_window>/first_job_submission.final.output .  # remember, the
. at the end signifies current location.
ls
```

There should be no files in the temp directory the first time you use the ls command, and after the **scp** command you should see your output file in that directory. Wildcards can be used to transfer files meeting specific conditions, or entire folders can be copied back and forth. Lets transfer the entire **my_firs t_job** directory to this same folder. You will need to do things on both the TACC terminal and desktop terminal. Add the -r option to the scp command to "recursively" transfer a folder and all contents.

On the TACC terminal

```
cds
pwd
```

On the DESKTOP terminal

```
scp -r <username>@ls5.tacc.utexas.edu:<pwd_from_other_window>/my_first_job .
ls
ls my_first_job
```

You should now see the my_first_job directory and all its contents on your desktop.

Files can be moved to lonestar in the same way, just by adding the "`<username>@ls5.tacc.utexas.edu:`" location information to the destination portion of the command.

Transferring files to and from lonestar with Windows

SSH Secure File Transfer (Windows) is available as part of the SSH Secure Shell client which can be downloaded from Bevoware.

This concludes the the linux and lonestar refresher tutorial.

Big Data In Biology Genome Variant Analysis Course 2016 home.