

Running batch jobs at TACC



Reservations

Use our summer school *reservation* ([intro_NGS](#)) when submitting batch jobs to get higher priority on the **Lonestar5** normal queue during this course:

```
sbatch --reservation=intro_NGS <batch_file>.slurm
```

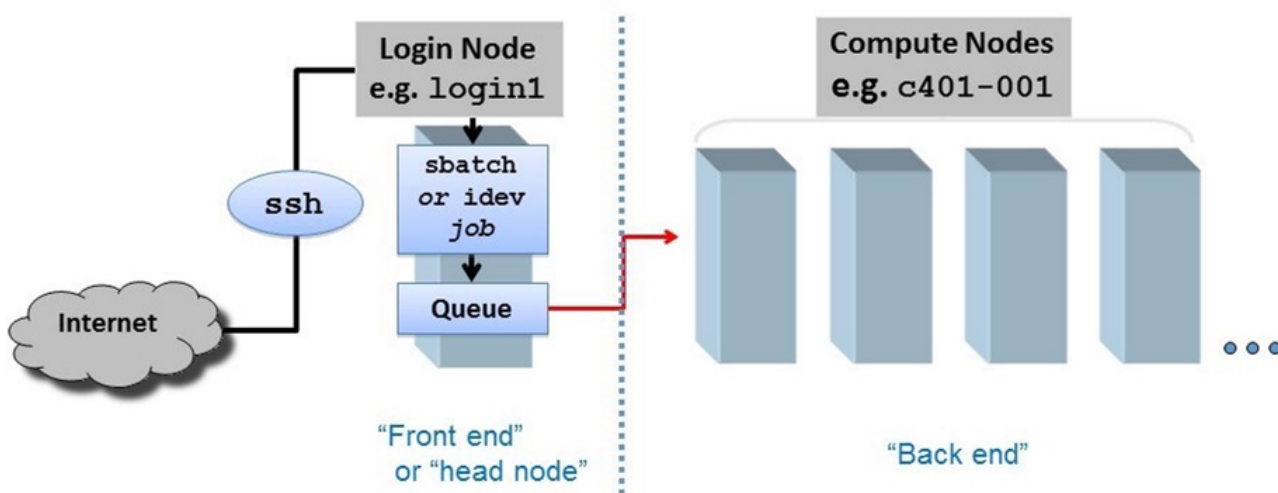
Note that a reservation is different from the TACC allocation/project for this class, which is **UT-2015-05-18**.

- [Compute cluster overview](#)
 - [Lonestar5 and Stampede2 overview and comparison](#)
- [Software at TACC](#)
 - [Programs and your \\$PATH](#)
 - [The module system](#)
 - [module spider](#)
 - [TACC BioContainers modules](#)
 - [loading a biocontainer module](#)
 - [installing custom software](#)
- [Job Execution](#)
 - [SLURM at a glance](#)
 - [Simple example](#)
 - [Job parameters](#)
 - [launcher_creator.py](#)
 - [job name and commands file](#)
 - [queues and runtime](#)
 - [allocation and SUs](#)
 - [wayness \(tasks per node\)](#)
 - [Wayness example](#)
- [Some best practices](#)
 - [Redirect task output and error streams](#)
 - [Combine serial workflows into scripts](#)
 - [Use one directory per job](#)
- [Interactive sessions \(idev\)](#)

Compute cluster overview

When you SSH into **ls5**, your session is assigned to one of a small set of *login nodes* (also called *head nodes*). These are not the *compute nodes* that will run your jobs.

Think of a node as a computer, like your laptop, but probably with more cores and memory. Now multiply that computer a thousand or more, and you have a cluster.



The small set of login nodes are a shared resource (type the **users** command to see everyone currently logged in) and are *not* meant for running interactive programs – for that you submit a description of what you want done to a batch system, which farms the work out to one or more compute nodes.

On the other hand, the login nodes *are* intended for copying files to and from TACC, so they have *a lot* of network bandwidth while compute nodes have limited network bandwidth.

So follow these guidelines:

- **Do not perform substantial computation on the login nodes.**
 - They are closely monitored, and you will get warnings from the TACC admin folks!
 - Code is usually developed and tested somewhere other than TACC, and only moved over when pretty solid.
- **Do not perform significant network access from your batch jobs.**
 - Instead, stage your data onto \$SCRATCH from a login node before submitting your job.

Lonestar5 and Stampede2 overview and comparison

Here is a comparison of the configurations and **ls5** and **stampede2**. As you can see, **stampede2** is the larger cluster, launched in 2017.

	ls5	stampede2
login nodes	6 20 cores each 128 GB memory	6 28 cores each 128 GB memory
standard compute nodes	1,252 24 cores per node (48 virtual) 64 GB memory	4,200 KNL (Knights Landing) <ul style="list-style-type: none">• 68 cores per node (272 virtual)• 96 GB memory 1,736 SKX (Skylake) <ul style="list-style-type: none">• 48 cores per node (96 virtual)• 192 GB memory
large memory nodes	10 total 2 w/1 TB memory, 48 cores 8 w/512 GB RAM, 32 cores	--
batch system	SLURM	SLURM
maximum job run time	48 hours	96 hours on KNL nodes 48 hours on SKX nodes

Note the use of the term *virtual core* above. Compute cores are standalone processors – mini CPUs, each of which can execute separate sets of instructions. However modern cores may also have *hyper-threading* enabled, where a single core can appear as more than one virtual processor to the Operating system (see <https://en.wikipedia.org/wiki/Hyper-threading> for more on hyper-threading). For example, **Lonestar5** nodes have 2 *hyperthreads (HTs)* per core. So with 2 HTs for each of the 24 physical cores, each node has a total of 48 *virtual cores*.

User guides for **ls5** and **stampede2** can be found at:

- <https://portal.tacc.utexas.edu/user-guides/stampede2>
- <https://portal.tacc.utexas.edu/user-guides/lonestar5>

Unfortunately, the TACC user guides are aimed towards a different user community – the weather modelers and aerodynamic flow simulators who need very fast matrix manipulation and other high performance computing (HPC) features. The usage patterns for bioinformatics – generally running 3rd party tools on many different datasets – is rather a special case for HPC. TACC calls our type of processing "*parameter sweep jobs*" and has a special process for running them, using their **launcher** module.

Software at TACC

Programs and your \$PATH

When you type in the name of an arbitrary program (**ls** for example), how does the shell know where to find that program? The answer is your **\$PATH**. **\$PATH** is a pre-defined environment variable whose value is a list of directories. The shell looks for program names in that list, in the order the directories appear.

To determine where the shell will find a particular program, use the **which** command:

Using which to search \$PATH

```
which rsync
which cat
```

The module system

The **module** system is an incredibly powerful way to have literally thousands of software packages available, some of which are incompatible with each other, without causing complete havoc. The TACC staff builds the desired package from source code in well-known locations that are NOT on your **\$PATH**. Then, when a module is loaded, its binaries are added to your **\$PATH**.

For example, the following **module load** command makes the **bwa** aligner available to you:

How module load affects \$PATH

```
# first type "bwa" to show that it is not present in your environment:
bwa
# it's not on your $PATH either:
which bwa

# now add bwa to your environment and try again:
module load bwa
bwa
# and see how it's now on your $PATH:
which bwa
# you can see the new directory at the front of $PATH
echo $PATH

# to remove it, use "unload"
module unload bwa
bwa
# gone from $PATH again...
which bwa
```

module spider

These days the TACC module system includes hundreds of useful bioinformatics programs. To see if your favorite software package has been installed at TACC, use **module spider**:

```
module spider fastqc
module spider samtools
module spider bedtools
```

TACC BioContainers modules

It is quite a large systems administration task to install software at TACC and configure it for the module system. As a result, TACC was always behind in making important bioinformatics software available. To address this problem, TACC moved to providing bioinformatics software via **containers**, which are *virtual machines* like **VMware** and **Virtual Box**, but are lighter weight: they require less disk space because they rely more on the host's base Linux environment. Specifically, TACC (and many other High Performance Computing clusters) use **Singularity** containers, which are similar to **Docker** containers but are more suited to the HPC environment (in fact one can build a **Docker** container then easily convert it to **Singularity** for use at TACC).

TACC obtains its containers from **BioContainers** (<https://biocontainers.pro/> and <https://github.com/BioContainers/containers>), a large public repository of bioinformatics tool **Singularity** containers. This has allowed TACC to easily provision thousands of such tools!

These BioContainers are not visible in TACC's "standard" module system, but only after the master **biocontainers** module is loaded:

```

# Make sure the non-biocontainers version of bwa is not loaded
module unload bwa
# Verify that bwa is not available
bwa

# Verify that these programs are not in the standard TACC module system
module spider kallisto
module spider bowtie2
module spider minimap2
module spider multiqc
module spider GATK
module spider velvet

# Load the Biocontainers master module (this takes a while)
module load biocontainers

# Now look for those programs
module spider kallisto
module spider bowtie2
module spider minimap2
module spider multiqc
module spider GATK
module spider velvet

```

Notice how the **BioContainers** module names have "ctr" in their names, version numbers, and other identifying information.

loading a biocontainer module

Once the **biocontainers** module has been loaded, you can just load the desired tool module, as with the **kallisto** pseudo-aligner program below.

```

# Load the Biocontainers master module
module load biocontainers

# Load the default kallisto biocontainer
module load kallisto

# Verify you can now execute kallisto
kallisto

```

Note that loading a BioContainer does not add anything to your **\$PATH**. Instead, it defines an *alias*, which is just a shortcut for executing the command. You can see the **alias** definition using the **type** command. And you can ensure the program is available using the **command -v** utility.

```

# Note that kallisto has not been added to your $PATH
which kallisto

# Instead, an alias has been defined. Use type to see its definition
type kallisto

# Ensure kallisto is available with command -v
command -v kallisto

```

installing custom software

Even with all the tools available at TACC, inevitably you'll need something they don't have. In this case you can build the tool yourself and install it in a local TACC directory. While building 3rd party tools is beyond the scope of this course, it's really not that hard. The trick is keeping it all organized.

For one thing, remember that your **\$HOME** directory quota is fairly small (10 GB on **Is5**), and that can fill up quickly if you install many programs. We recommend creating an installation area in your **\$WORK** directory and installing programs there. You can then make symbolic links to the binaries you need in your **\$HOME/local/bin** directory (which was added to your **\$PATH** in your **.bashrc**).

See how we used a similar trick to make the **launcher_creator.py** program available to you. Using the **ls -l** option shows you where symbolic links point to:

Real location of launcher_creator.py

```
ls -l ~/local/bin
```

```
/work/projects/BioITeam/common/bin/launcher_creator.py
```

! \$PATH caveat

Remember that the order of locations in the **\$PATH** environment variable is the order in which the locations will be searched. In particular, the **module load** command adds to the *front* of your path. This can mask similarly-named programs, for example, in your **\$HOME/local/bin** directory.

Job Execution

Job execution is controlled by the **SLURM** batch system on both **ls5** and **stampede2**.

To run a job you prepare 2 files:

1. a **commands file** containing the commands to run, one command per line (**<job_name>.cmds**)
2. a **job control file** that describes how to run the job (**<job_name>.slurm**)

The process of running the job involves these steps:

1. Create a commands file containing **exactly one command per line**.
2. Prepare a job control file for the commands file that describes how the job should be run.
3. You **submit** the job control file to the batch system. The job is then said to be **queued** to run.
4. The batch system **prioritizes** the job based on the number of compute nodes needed and the job run time requested.
5. When compute nodes become available, the job tasks (command lines in the **<job_name>.cmds** file) are **assigned** to one or more compute nodes and **begin to run** in parallel.
6. The job **completes** when either:
 - a. you cancel the job manually
 - b. all tasks in the job complete (successfully or not!)
 - c. the requested job run time has expired

SLURM at a glance

Here are the main components of the **SLURM** batch system.

	ls5, stampede2
batch system	SLURM
batch control file name	<job_name>.slurm
job submission command	sbatch <job_name>.slurm
job monitoring command	showq -u
job stop command	scancel -n <job name>

Simple example

Let's go through a simple example. Execute the following commands to copy a pre-made **simple.cmds** commands file:

Copy simple commands

```
mkdir -p $SCRATCH/core_ngs/slurm/simple
cd $SCRATCH/core_ngs/slurm/simple
cp $CORENGS/tacc/simple.cmds .
```

What are the tasks we want to do? Each task corresponds to one line in the **simple.cmds** file, so let's take a look at it using the **cat** (concatenate) command that simply reads a file and writes each line of content to **standard output** (here, your Terminal):

View simple commands

```
cat simple.cmds
```

The tasks we want to perform look like this:

```
echo "Command 1 on `hostname` - `date`" > cmd1.log 2>&1
echo "Command 2 on `hostname` - `date`" > cmd2.log 2>&1
echo "Command 3 on `hostname` - `date`" > cmd3.log 2>&1
echo "Command 4 on `hostname` - `date`" > cmd4.log 2>&1
echo "Command 5 on `hostname` - `date`" > cmd5.log 2>&1
echo "Command 6 on `hostname` - `date`" > cmd6.log 2>&1
echo "Command 7 on `hostname` - `date`" > cmd7.log 2>&1
echo "Command 8 on `hostname` - `date`" > cmd8.log 2>&1
```

There are 8 tasks. Each is a simple **echo** command that just outputs string containing the task number and date to a different file.

Use the handy [launcher_creator.py](#) program to create the job submission script.

Create batch submission script for simple commands

```
launcher_creator.py -j simple.cmds -n simple -t 00:00:05 -w 8 -a UT-2015-05-18 -q development
```

You should see output something like the following, and you should see a [simple.slurm](#) batch submission file in the current directory.

```
Project simple.
Using job file simple.cmds.
Using development queue.
For 00:00:05 time.
Using UT-2015-05-18 allocation.
Not sending start/stop email.
Launcher successfully created. Type "sbatch simple.slurm" to queue your job.
```

Submit your batch job like this, then check the batch queue to see the job's status.

Submit simple job to batch queue

```
sbatch --reservation=intro_NGS simple.slurm
showq -u
```

If you're quick, you'll see a queue status something like this:

```
SUMMARY OF JOBS FOR USER: <abattenh>

ACTIVE JOBS-----
JOBID   JOBNAME   USERNAME   STATE   NODES  REMAINING  STARTTIME
=====
2916562  simple    abattenh   Running 1      0:00:57   Sat Jun 13 13:52:32

WAITING JOBS-----
JOBID   JOBNAME   USERNAME   STATE   NODES  WCLIMIT   QUEUETIME
=====

Total Jobs: 1      Active Jobs: 1      Idle Jobs: 0      Blocked Jobs: 0
```

If you don't see your **simple** job in either the ACTIVE or WAITING sections of your queue, it probably already finished – it should only run for a second or two!

Notice in my queue status, where the STATE is Running, there is only one node assigned. Why is this, since there were 8 tasks?

Every job, no matter how few tasks requested, will be assigned at least one node. Each **ls5** node has 24 physical cores, so each of the 8 tasks can be assigned to a core.

Exercise: What files were created by your job?

ls should show you something like this:

```
cmd1.log  cmd3.log  cmd5.log  cmd7.log  simple.cmds  simple.o2916562
cmd2.log  cmd4.log  cmd6.log  cmd8.log  simple.e2916562  simple.slurm
```

The newly created files are the **.log** files, as well as error and output logs **simple.e2916562** and **simple.o2916562**.

filename wildcarding

Here's a cute trick for viewing the contents all your output files at once, using the **cat** command and filename wildcarding.

Multi-character filename wildcarding

```
cat cmd*.log
```

The **cat** command actually takes a list of one or more files (if you're giving it files rather than *standard input* – more on this shortly) and outputs the concatenation of them to *standard output*. The asterisk (*****) in **cmd*.log** is a multi-character wildcard that matches any filename starting with **cmd** then ending with **.log**. So it would match **cmd_hello_world.log**.

You can also specify single-character matches inside brackets (**[]**) in either of the ways below, this time using the **ls** command so you can better see what is matching:

Single character filename wildcarding

```
ls cmd[12345678].log
ls cmd[1-8].log
```

This technique is sometimes called *filename globbing*, and the pattern a *glob*. Don't ask me why – it's a Unix thing. Globbing – translating a glob pattern into a list of files – is one of the handy things the **bash** shell does for you. (Read more about [Wildcards and special filenames](#).)

Exercise: How would you list all files starting with **simple**?

```
ls simple*
```

Here's what my **cat** output looks like. Notice the times are all nearly the same because all the tasks ran in parallel. That's the power of cluster computing!

```
Command 1 on nid00022 - Sat Jun 13 13:52:52 CDT 2020
Command 2 on nid00022 - Sat Jun 13 13:52:47 CDT 2020
Command 3 on nid00022 - Sat Jun 13 13:52:58 CDT 2020
Command 4 on nid00022 - Sat Jun 13 13:52:53 CDT 2020
Command 5 on nid00022 - Sat Jun 13 13:52:41 CDT 2020
Command 6 on nid00022 - Sat Jun 13 13:52:49 CDT 2020
Command 7 on nid00022 - Sat Jun 13 13:52:40 CDT 2020
Command 8 on nid00022 - Sat Jun 13 13:52:48 CDT 2020
```

echo

Lets take a closer look at a typical task in the **simple.cmds** file.

An echo command

```
echo "Command 3 `date`" > cmd3.log 2>&1
```

The **echo** command is like a print statement in the **bash** shell. Echo takes its arguments and writes them to one line of *standard output*. While not always required, it is a good idea to put the output string in double quotes.

backtick evaluation

So what is this funny looking ``date`` bit doing? Well, `date` is just another Linux command (try just typing it in). Here we don't want the shell to put the string "date" in the output, we want it to *execute* the `date` command and put the *result* text into the output. The backquotes (```, also called *backticks*) around the `date` command tell the shell we want that command executed and its output substituted into the string. (Read more about [Quoting in the shell](#).)

Backtick evaluation

```
# These are equivalent:
date
echo `date`

# But different from this:
echo date
```

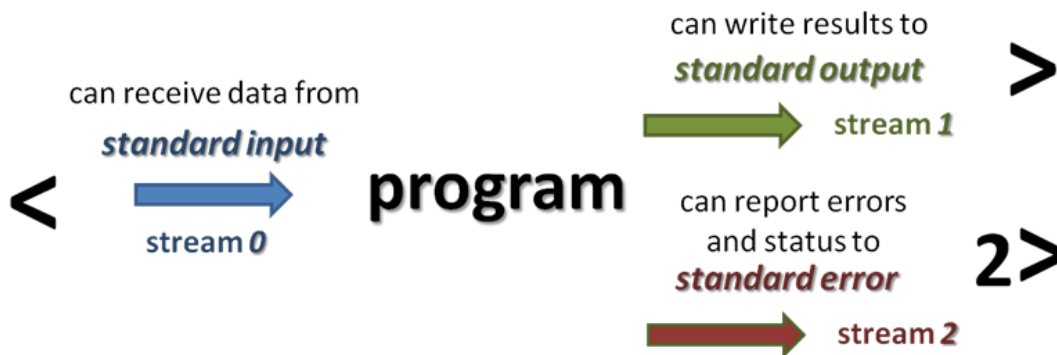
output redirection

There's still more to learn from one of our simple tasks, something called *output redirection*.

```
echo "Command 3 `date`" > cmd3.log 2>&1
```

Normally `echo` writes its string to *standard output*. If you invoke `echo` in an interactive shell like Terminal, standard output is displayed to the Terminal window.

the three standard program streams



Usually we want to *separate the outputs* of all our commands. Why is this important? Suppose we run a job with 100 commands, each one a whole pipeline (alignment, for example). 88 finish fine but 12 do not. Just try figuring out which ones had the errors, and where the errors occurred, if all the normal output is in one intermingled file and all the error in another intermingled file!

So in the above example the first `>` says to redirect the *standard output* of the `echo` command to the `cmd3.log` file. The `'2>&1'` part says to redirect *standard error* to the same place. Technically, it says to redirect *standard error* (built-in Linux stream 2) to the same place as *standard output* (built-in Linux stream 1); and since *standard output* is going to `cmd3.log`, any *standard error* will go there also. (Read more about [Standard I/O streams](#).)

So what happens when output is generated by tasks in a batch job? Well, you may have noticed the files with names like `simple.e2916562` and `simple.o2916562` were created by your job. `simple.o2916562` contains all *standard output* and `simple.e2916562` contains all *standard error* generated by your tasks that was not redirected elsewhere, as well as information relating to running your job and its tasks.

Job parameters

Now that we've executed a really simple job, let's take a look at some important job submission parameters. These correspond to arguments to the `launcher_creator.py` script.

A bit of background. Historically, TACC was set up to cater to researchers writing their own C or Fortran codes highly optimized to exploit parallelism (the HPC crowd). Much of TACC's documentation is aimed at this audience, which makes it difficult to pick out the important parts for us.

The kind of jobs we biologists generally run are relatively new to TACC. They even have a special name for them: "*parametric sweeps*", by which they mean the *same program* running on *different data* sets.

In fact there is a special software module required to run our jobs, called the `launcher` module. You don't need to worry about activating the launcher module – that's done by the `<job_name>.slurm` script created by `launcher_creator.py` like this:


```
module load launcher
```

The [launcher](#) module knows how to interpret various job parameters in the `<job_name>.slurm` batch **SLURM** submission script and use them to create your job and assign its tasks to compute nodes. Our [launcher_creator.py](#) program is a simple Python script that lets you specify job parameters and writes out a valid `<job_name>.slurm` submission script.

launcher_creator.py

If you call [launcher_creator.py](#) with no arguments it gives you its usage description:

launcher_creator.py usage

```
usage: launcher_creator.py [-h] -n NAME -t TIME_REQUEST [-j JOB_FILE]
                          [-b SHELL_COMMANDS] [-B SHELL_COMMANDS_FILE]
                          [-q QUEUE] [-a [ALLOCATION]] [-m MODULES]
                          [-M MODULES_FILE] [-w WAYNESS] [-N NUM_NODES]
                          [-e [EMAIL]] [-l LAUNCHER] [-s]
```

Create launchers for TACC clusters. Report problems to rt-other@ccbb.utexas.edu

optional arguments:

-h, --help show this help message and exit

Required:

-n NAME, --name NAME The name of your job.
-t TIME_REQUEST, --time TIME_REQUEST
The time you want to give to your job. Format:
hh:mm:ss

Commands:

You must use at least one of these options to submit your commands for TACC.

-j JOB_FILE, --jobs JOB_FILE
The name of the job file containing your commands.
-b SHELL_COMMANDS, --bash SHELL_COMMANDS
A string of shell (Bash, zsh, etc) commands that are executed before any parametric jobs are launched.
-B SHELL_COMMANDS_FILE, --bash_file SHELL_COMMANDS_FILE
A file containing shell (Bash, zsh, etc) commands that are executed before any parametric jobs are launched.

Optional:

-q QUEUE, --queue QUEUE
The TACC allocation for job submission.
Default="development"
-a [ALLOCATION], -A [ALLOCATION], --allocation [ALLOCATION]
The TACC allocation for job submission. You can set a default ALLOCATION environment variable.
-m MODULES, --modules MODULES
A list of module commands. The "launcher" module is always automatically included. Example: -m "module swap intel gcc; module load bedtools"
-M MODULES_FILE, --modules_file MODULES_FILE
A file containing module commands.
-w WAYNESS, --wayness WAYNESS
Wayness: the number of commands you want to give each node. The default is the number of cores per node.
-N NUM_NODES, --num_nodes NUM_NODES
Number of nodes to request. You probably don't need this option. Use wayness instead. You ONLY need it if you want to run a job list that isn't defined at the time you submit the launcher.
-e [EMAIL], --email [EMAIL]
Your email address if you want to receive an email from Lonestar when your job starts and ends. Without an argument, it will use a default EMAIL_ADDRESS environment variable.
-l LAUNCHER, --launcher_name LAUNCHER
The name of the launcher script that will be created.
Default="<name>.slurm"
-s
Echoes the launcher filename to stdout.

Because it is a long help message, we may want to pipe the output to [more](#), a *pager* that displays one screen of text at a time. Type the [spacebar](#) to advance to the next page, and [Ctrl-c](#) to exit from [more](#).

Get usage information for launcher_creator.py

```
# Use spacebar to page forward; Ctrl-c to exit
launcher_creator.py -h | more
```



The `launcher_creator.py` script does not handle every job control parameter you might ever want to set. For that, make a copy of the default script, found at `$LAUNCHER_DIR/extras/batch-scripts/launcher.slurm`, and edit it appropriately.

To read more about the `launcher` module:

```
module load launcher
module help launcher
more $LAUNCHER_DIR/README
```

job name and commands file

Recall how the `simple.slurm` batch file was created:

Create batch submission script for simple commands

```
launcher_creator.py -j simple.cmds -n simple -t 00:00:05 -w 8 -a UT-2015-05-18 -q developmen
```

- The name of your *commands file* is given with the `-j simple.cmds` argument.
- Your desired *job name* is given with the `-n <job_name>` argument.
 - The `<job_name>` (here `simple`) is the *job name* you will see in your queue.
 - By default a corresponding `<job_name>.slurm` batch file is created for you.
 - It contains the name of the commands file that the batch system will execute.

queues and runtime

TACC resources are partitioned into *queues*: a named set of compute nodes with different characteristics. The major ones on `Is5` are listed below. Generally you use `development` (`-q development`) when you are writing and testing your code, then `normal` once you're sure your commands will execute properly.

queue name	maximum runtime	purpose
<code>development</code>	2 hrs	development (short queue wait times)
<code>normal</code>	48 hrs	normal priority (queue waits can sometimes be long)
<code>largemem</code>	48 hrs	large memory jobs

- In `launcher_creator.py`, the queue is specified by the `-q` argument.
 - The default queue is `development`. Specify `-q normal` for `normal` queue jobs.
- The *maximum runtime* you are requesting for your job is specified by the `-t` argument.
 - Format is `hh:mm:ss`
 - Note that your job will be terminated *without warning* at the end of its time limit!

allocation and SUs

You may be a member of a number of different projects, hence have a choice which allocation to run your job under.

- You specify that allocation name with the `-a` argument of `launcher_maker.py`.
- If you have set an `$ALLOCATION` environment variable to an allocation name, it will be used if you are a member of only one TACC project.

The `.bashrc` login script you've installed for this course specifies the class's allocation as shown below. Note that this allocation will expire after the course, so you should change that setting appropriately at some point.

ALLOCATION setting in .bashrc

```
# This sets the default project allocation for launcher_maker.py
export ALLOCATION=UT-2015-05-18
```

- When you run a batch job, your project allocation gets "charged" for the time your job runs, in the currency of **SUs** (System Units).

- SUs are related in some way to node hours.



Jobs tasks should have similar expected runtimes

Jobs should consist of tasks that will run for approximately the same length of time. This is because the total node hours for your job is calculated as the run time for your *longest running* task (the one that finishes last).

For example, if you specify 64 commands and 99 finish in 2 seconds but one runs for 24 hours, you'll be charged for 64 x 24 node hours even though the total amount of work performed was only ~24 hours.

wayness (tasks per node)

One of the most confusing things in job submission is the parameter called *wayness*, which controls how many tasks are run on each computer node.

- Recall that there are 24 physical (48 virtual) cores and 64 GB of memory on each compute node
 - so technically you can run up to 48 commands on a node, each with ~1.3 GB available memory
 - you can run fewer tasks, and if you do, each task gets more resources
- Because bioinformatics programs generally require more memory and fewer cores, [launcher_creator.py](#) sets a 24 cores/node maximum.

tasks per node (wayness)	cores available to each task	memory available to each task
1	24	64 GB
2	12	32 GB
3	8	21.3 GB
4	6	16 GB
6	4	10.6 GB
8	3	8 GB
12	2	5.3 GB
24	1	2.6 GB

- In [launcher_creator.py](#), *wayness* is specified by the `-w` argument.
 - the default is 24 (one task per core)
- A special case is when you have only 1 command in your job.
 - In that case, it doesn't matter what *wayness* you request.
 - Your job will run on one compute node, and have all 24 cores available.

Your choice of the *wayness* parameter will depend on the nature of the work you are performing: its computational intensity, its memory requirements and its ability to take advantage of multi-processing/multi-threading (e.g. `bwa -t` option or `tophat -p` option).

Wayness example

Let's use [launcher_creator.py](#) to explore *wayness* options. First copy over the `wayness.cmds` commands file:

Copy wayness commands

```
cds
mkdir -p core_ngs/slurm/wayness
cd core_ngs/slurm/wayness
cp $CORENGS/tacc/wayness.cmds .
```

Exercise: How many tasks are specified in the wayness.cmds file?

`wc --help`

Find the number of lines in the `wayness.cmds` commands file:

ALLOCATION setting in .bashrc

```
wc -l wayness.cmds
```

The file has 24 lines, representing 24 tasks.

The `wayness.cmds` commands file consists of a number of identical lines that look like this:

```
sleep 3; echo "Command $LAUNCHER_JID of $LAUNCHER_NJOBS ($LAUNCHER_PPN per node) ran on node `hostname` core $LAUNCHER_TSK_ID." > cmd.$LAUNCHER_JID.log 2>&1
```

The `wayness` commands take advantage of a number of environment variables the `launcher` module system sets automatically for each task:

- `$LAUNCHER_JID` – the task number of the running task (from 1 to total number of tasks)
- `$LAUNCHER_NJOBS` – total number of tasks specified by the job
- `$LAUNCHER_TSK_ID` – number of the core running the task (0 to number of tasks - 1)
- `hostname` – Linux program that returns the name of the current compute node

For more information, see <https://github.com/TACC/launcher>

Create the batch submission script specifying a `wayness` of 8 (8 tasks per node), then submit the job and monitor the queue:

Create batch submission script for wayness example

```
launcher_creator.py -j wayness.cmds -n wayness -w 8 -t 00:00:20 -a UT-2015-05-18
sbatch --reservation=intro_NGS wayness.slurm
showq -u
```

Exercise: With 24 tasks requested and `wayness` of 8, how many nodes will this job require? How much memory will be allocated to each task?

3 nodes (24 tasks x 1 node/8 tasks)
16 GB (64 GB/node * 1 node/8 tasks)

Exercise: If you specified a `wayness` of 2, how many nodes would this job require? How much memory could each task use?

12 nodes (24 tasks x 1 node/2 tasks)
32 GB (64 GB/node * 1 node/2 tasks)

Look at the output file contents once the job is done.

```
cat cmd*log

# or, for a listing ordered by command number (the 2nd field, a number)
cat cmd*log | sort -k 2,2n
```

You should see something like output below.

```
Command 1 of 24 (8 per node) ran on node nid00020 core 0.
Command 2 of 24 (8 per node) ran on node nid00020 core 1.
Command 3 of 24 (8 per node) ran on node nid00020 core 2.
Command 4 of 24 (8 per node) ran on node nid00020 core 3.
Command 5 of 24 (8 per node) ran on node nid00020 core 4.
Command 6 of 24 (8 per node) ran on node nid00020 core 5.
Command 7 of 24 (8 per node) ran on node nid00020 core 6.
Command 8 of 24 (8 per node) ran on node nid00020 core 7.
Command 9 of 24 (8 per node) ran on node nid00021 core 8.
Command 10 of 24 (8 per node) ran on node nid00021 core 9.
Command 11 of 24 (8 per node) ran on node nid00021 core 10.
Command 12 of 24 (8 per node) ran on node nid00021 core 11.
Command 13 of 24 (8 per node) ran on node nid00021 core 12.
Command 14 of 24 (8 per node) ran on node nid00021 core 13.
Command 15 of 24 (8 per node) ran on node nid00021 core 14.
Command 16 of 24 (8 per node) ran on node nid00021 core 15.
Command 17 of 24 (8 per node) ran on node nid00022 core 16.
Command 18 of 24 (8 per node) ran on node nid00022 core 17.
Command 19 of 24 (8 per node) ran on node nid00022 core 18.
Command 20 of 24 (8 per node) ran on node nid00022 core 19.
Command 21 of 24 (8 per node) ran on node nid00022 core 20.
Command 22 of 24 (8 per node) ran on node nid00022 core 21.
Command 23 of 24 (8 per node) ran on node nid00022 core 22.
Command 24 of 24 (8 per node) ran on node nid00022 core 23.
```

Notice that there are 3 different host names, each of which ran 8 tasks:

```
cat cmd*log | awk '{print $11}' | sort | uniq -c
```

should produce this output (read more about [piping commands to make a histogram](#))

```
8 nid00020
8 nid00021
8 nid00022
```

Some best practices

Redirect task output and error streams

We've already touched on the need to redirect *standard output* and *standard error* for each task. Just remember that funny redirection syntax:

```
my_program input_file1 output_file1 > file1.log 2>&1
```

Combine serial workflows into scripts

Another really good way to work is to "bundle" a complex set of steps into a shell script that sets up its own environment, loads its own modules, then executes a series of program steps. You can then just call that script, probably with data-specific arguments, in your commands file. This multi-program script is sometimes termed a *pipeline*, although complex pipelines may involve several such scripts.

For example, you might have a script called [align_bwa.sh](#) (a **bash** script) or [align_bowtie2.py](#) (written in **python**) that performs multiple steps needed during the alignment process:

- quality checking the input **FASTQ** file
- trimming or removing adapters from the sequences
- performing the alignment step(s) to create a **BAM** file
- sort the **BAM** file
- index the **BAM** file
- gather alignment statistics from the **BAM** file

The BioITeam maintains a set of such scripts in the `/work/projects/BioITeam/common/script` directory. Take a look at some of them after you feel more comfortable with initial NGS processing steps. They can be executed by anyone with a TACC account.

Use one directory per job

You may have noticed that all the files involved in our job were in one directory – the batch submissions file, commands file, log files our tasks wrote, and the launcher job output and error files. Of course you'll probably need input files too 🍌 as well as output files.

Because a single job can create a lot of files, it is a good idea to *use a different directory* for each job or set of closely related jobs, maybe with a name similar to the job being performed. This will help you stay organized.

Here's an example directory structure

```
$$SCRATCH/my_project
  /original      # contains or links to original fastq files
  /fastq_prep    # run fastq QC and trimming jobs here
  /alignment     # run alignment jobs here
  /gene_counts   # analyze gene overlap here
  /test1         # play around with stuff here
  /test2         # play around with other stuff here
```

Command files in each directory can refer to files in other directories using relative path syntax, e.g.:

Relative path syntax

```
cd $$SCRATCH/my_project/fastq_prep
ls ../original/my_raw_sequences.fastq.gz
```

Or create a symbolic link to the directory and refer to it as a sub-directory:

Symbolic link to relative path

```
cd $SCRATCH/my_project/fastq_prep
ln -s ../original fq
ls ./fq/my_raw_sequences.fastq.gz
```

relative path syntax

As we have seen, there are several special "directory names" the bash shell understands:

- "dot directory" (.) refers to "here" or "the current directory"
- "dot dot directory" (..) refers to "one directory up"
- "tilde directory" (~) refers to your **home** directory

Try these relative path examples:

Relative path exercise

```
cd $SCRATCH/core_ngs/slurm/simple
ls ../wayness
ls ../../
ls -l ~/.bashrc
```

Interactive sessions (idev)

So we've explored the TACC batch system. What if you want to do some interactive-style testing of your workflow?

Interactive sessions are available through the **idev** command as shown below. **idev** sessions are configured with similar parameters to batch jobs.

Start an idev session

```
idev -p development -m 20 -A UT-2015-05-18 -N 1 -n 24 --reservation=intro_NGS
```

Notes:

- **-p development** requests nodes on the **development** queue
- **-m 20** asks for a 20-minute session (120 minutes is the maximum for development)
- **-A UT-2015-05-18** specifies the TACC allocation/project to use
- **-N 1** asks for 1 node and **-n 24** requests access to 24 cores
- **--reservation=intro_NGS** gives us priority access to TACC nodes for the class. You normally won't use this argument.

When you ask for an **idev** session, you'll see output as shown below. Note that the process may pause while it waits for available nodes.

```
-> Defaults file      : ~/.idevrc
-> System             : ls5
-> Queue              : development  (cmd line: -p      )
-> Nodes              : 1             (cmd line: -N      )
-> Total tasks        : 24            (cmd line: -n      )
-> Time (minutes)     : 20           (cmd line: -m      )
-> Project            : UT-2015-05-1 (cmd line: -A      )

-----
                Welcome to the Lonestar 5 Supercomputer
-----

No reservation for this job
--> Verifying valid submit host (login2)...OK
--> Verifying valid jobname...OK
--> Enforcing max jobs per user...OK
--> Verifying availability of your home dir (/home1/01063/abattenh)...OK
--> Verifying availability of your work dir (/work/01063/abattenh/lonestar)...OK
--> Verifying availability of your scratch dir (/scratch/01063/abattenh)...OK
--> Verifying valid ssh keys...OK
--> Verifying access to desired queue (development)...OK
--> Verifying job request is within current queue limits...OK
--> Checking available allocation (UT-2015-05-18)...OK
Submitted batch job 1579644

-> After your idev job begins to run, a command prompt will appear,
```

```
-> and you can begin your interactive development session.
-> We will report the job status every 4 seconds: (PD=pending, R=running).

->job status:  PD
->job status:  R

-> Job is now running on masternode= nid00011...OK
-> Sleeping for 7 seconds...OK
-> Checking to make sure your job has initialized an env for you...OK
-> Creating interactive terminal session (login) on master node nid00011.

Warning: Permanently added '[nid00011]:6999,[10.128.0.12]:6999' (RSA) to the list of known hosts
```

Once the **idev** session has started, it looks quite similar to a login node environment, except for these differences:

- the **hostname** command on a login node will return a login server name like **login2**
 - while in an **idev** session **hostname** returns a compute node name like **nid00011**
- you cannot submit a batch job from inside an **idev** session, only from a login node
- your **idev** session will end when the requested time has expired
 - or you can just type **exit** to return to a login node session