Draft version June 4, 2018
Typeset using LaTeX **modern** style in AASTeX62

Running code on the Wesleyan Cluster

ABSTRACT

This document is intended as a quick reference guide for running code on the Wesleyan computing cluster. It is not an introduction to high-performance cluster computing (I would not be qualified to write such a document) but it does cover the details of computing power available, and how to get started using it.

1. CLUSTER BASICS

The Wesleyan computing center (located on the 5th floor of the Excley Science Tower) contains a range of computing capabilities, all available through the Wesleyan network. These capabilities include highly parallelized computing, GPU computing, high memory ($\sim$100 GB) computing, etc. A full list of the hardware setup is available online[1], along with a list of available software[2]. The cluster is managed by Henk Meij (hmeij@wesleyan.edu); I have always found Henk to be very responsive and helpful when setting up my code, and running through any sort of trouble shooting that I can't handle on my own. The first step in running code on the cluster is to contact Henk, and to request an account on the cluster.

The first point of contact with the computing cluster is through one of the login nodes. The primary login node is `cottontail`, although secondary nodes `cottontail2` and `swallowtail` are also available. These login nodes can be accessed through `ssh`

```
ssh -XY kflaherty@cottontail
```

You will be deposited into your home directory (`/home/kflaherty`) on the computing cluster, where you will store your code and data. Note that this home directory is not the same as the home directory on you local machine and your data and code must be copied over from your local machine to the computing clusters home directory

```
scp mydata.fits kflaherty@cottontail:/home/kflaherty/mycode/
scp mycode.py kflaherty@cottontail:/home/kflaherty/mycode/
```

[1] https://dokuwiki.wesleyan.edu/doku.php?id=cluster:126
[2] https://dokuwiki.wesleyan.edu/doku.php?id=cluster:73

There is a total of 10 TB of space allocated for home directories; if more space is needed for your data, contact Henk to make arrangements.

After connecting to the login node, you can access the computing nodes through the use of the OpenLava Scheduler. A job, which is one instance of running your code, is submitted to the computing cluster using the scheduler, which in turn distributes the computational work according to the requested configuration (e.g. the requested computing queue, the requested number of cores, etc.) onto any available cores. The computing cluster is set up with a number of different queues, which provide access to a collection of nodes, which are many up of multiple *chips*. The number of nodes, and the number of *chips* per node depends on the chosen queue. The exact queue that works best for you will depend on the computing capabilities needed for your code. The nodes are specified by `n` followed by a number (e.g. `n33 n14 n27`). When multiple cores from a single node are used by a piece of code, you will see the node name repeated multiple times (e.g. `n33 n33 n33 n33 n33` for 5 cores from node 33)

The main queues available on the Wesleyan computing cluster are:

1. `hp12`: The default queue. Consists of 32 nodes, with dual quad core chips with 12 GB of memory each
2. `mw256fd`: Contains 8 nodes with dual eight core chips, with 256 GB of memory each.
3. `mwgpu`: Contains four GPUs, 2,500 cores/GPU with GPU memory footprint of 5 GB.

The queue `hp12` is designed for highly parallelized, but small memory, jobs (`tinymem` is a queue that also handles small memory jobs). The queue `mw256fd`, the primary queue that I use, is designed for large memory jobs, while `mwgpu` is designed for GPU computations. All queues have access to the same software base, and are all accessible from any login node. As described below your choice of queue is specified when submitting a job to the scheduler.

2. SCRIPTS FOR RUNNING JOBS

The submit a job to the desired queue, a submission script must be created. This script contains a series of commands that setup the computing job, and then call you code to start the computation. Outlined below are the basic commands used in the submission script for either a serial or parallel job. Examples of these pieces of code are available in Henk's home directory (`/home/hmeij/jobs` within the `serial` and `parallel`) directories. Henk has also implemented checkpointing, which allows for a job to be restarted from a saved checkpoint if the cluster crashes. The checkpointing scripts include the same basic setup, with additional code (that doesn't need to be modified by the user) that handles the checkpointing functionality.

2.1. *Basic serial script*

The simplest type of job is one that requires only serial computations. Here you simply need to specify the requested queue, provide a name for your job, setup up the software environment, copy the data and code to the scratch directory, and run your code. This is all done within a shell script, with the associated commands for shell scripting available within this submission script.

The first line setups up the bash script

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

Note that the cluster uses the bash shell, rather than csh or tcsh. Any shell scripts called by you code must be setup in bash.

Next are a series of parameters for the scheduler:

```
#BSUB -q mw256fd
#BSUB -J test
#BSUB -o test.stdout
#BSUB -e test.stderr
#BSUB -N
```

The first line specifies the queue to which the code is sent (e.g. hp12, mwd26fd). The second line provides the name of the job, while the third and fourth lines specify the names for the output and error files. These two files are created once the job is complete and they contain outputs from you code, as well an error messages generated while running your code. The `test.stdout` file will also contain basic information on the name of the job, the home directory, the working directory, the job id, a snippet of the code, plus some other information about the run. The last line requests that the scheduler send you an email when the code has finished running.

Next you need to set up the software environment for your code. This can require defining path variables for various packages (in this case Python and MIRIAD).

```
export PYTHONHOME=/home/apps/python/2.6.1
export PYTHONPATH=/home/apps/python/2.6.1/site-packages
export PATH=$PYTHONHOME/bin:$PATH
. /home/apps/miriad/MIRRC.sh
export PATH=$MIRBIN:$PATH
```

The first three lines set up the python paths, specifically referencing python 2.6.1. Note that multiple versions of python exist on the cluster, but version 2.6.1 contains numpy, scipy, astropy and emcee which are all used in my own code. The last two lines set up the MIRIAD installation. First it calls the MIRIAD initialization script, which sets up the variable `$MIRBIN`, which is then added to the path. Note that the

MIRIAD installation requires a fortran library that is only available on the mw256fd queue.

The next step is to copy the data and code over to the working directory. Upon the start of a job a temporary scratch directory is created, with a name specified by the ID number associated with job. This directory is removed once the job is complete, but is a useful place for running the code in order to reduce the load on the home directory server. By default this happens in the `/sanscratch` directory, but if large amounts of data are being generated, then the `/localscratch5tb` scratch directory should be used[3].

```
cp -r mydata.fits /sanscratch/$LSB_JOBID
cp -r mycode.py /sanscratch/$LSB_JOBID
cd /sanscratch/$LSB_JOBID
```

The system variable `$LSB_JOBID` contains the job ID number that is generated when a job is sent to the Scheduler. This sequence of commands not only copies the data (`mydata.fits`) and code (`mycode.py`) into the scratch directory, but it also moves into that directory. This is important so that any read/write commands happen within the scratch directory. Otherwise these commands will be sent to your home directory, which will put unnecessary load onto the server that runs the home directory, slowing down operations on the home directory.

Once the setup is complete, you can call your actual code (`test.py`), making sure to copy over any results (`myresults.txt`) from the scratch directory back into the home directory.

```
python test.py
cp myresults.txt /home/kflaherty/mycode/
```

2.2. *Basic Parallel script*

Running a parallel code that distributes jobs over multiple cores (which may or may not be on the same node) follows the same basic flow as the serial script, with additional parameters for the scheduler, and additional PATH variables to provide access to the software that does the parallelization. Listed below are the commands that are included in addition to those already specified within the basic serial script.

The first addition is in the parameters for the scheduler. Since a parallel job requires resources from multiple cores you need to specify the necessary computing resources.

[3] The `/localscratch5tb` directory is only available to the mw256fd queue

```
#BSUB -n 8
#BSUB -R ``span[hosts=1]''
```

The first command specifies the number of cores requested for this job, in this case 8. The second command tells the scheduler that all of the cores must come from one node. This, in turn, limits the maximum number of cores that can be requested in the first line; in the hp12 queue there are 8 cores per node, while for the mw256fd queue there are 16 cores per node. Conversely, you can remove the second command and replace it with

```
#BSUB -R ``span[ptile=1]''
```

which tells the scheduler to use one core per node (e.g. use 8 nodes, instead of 8 cores within a single node). Here the limit is the number of nodes within a queue (32 for hp12, and 8 for mw256fd). I have always found that `openmpi` only works when all of the cores are within a single node, rather than being spread out over multiple nodes.

The next step is to set up paths to the `openmpi` code

```
export PATH=/share/apps/openmpi/1.2+intel-10/bin:$PATH
export LD_LIBRARY_PATH=/share/apps/openmpi/1.2+intel-10/lib:$LD_LIBRARY_PATH
```

From here your code can be called:

```
./lava.openmpi.wrapper python mpi_run_models.py
```

Calling your code is slightly different here that in the serial case. The call is done via a wrapper (`lava.openmpi.wrapper`) that sets up variables needed for parallel jobs. This wrapper takes the place of a direct call to e.g. `mpirun` that you would use on your local machine. The wrapper is available from Henk's directory (`/home/hmeij/jobs/parallel/`)

2.3. *Checkpointing*

Recently Henk has set up the cluster to take advantage of checkpointing within the code, based on the Berkeley Laboratory Checkpoint/Restart tool. What this means is that if the cluster crashes, you can restart from the last checkpoint, rather than having to restart from the very beginning. This is very useful if you e.g. are running a week long MCMC chain and the cluster crashes on day four. The details of how this works are complicated, but Henk has hidden most of this within new execution files, having worked closely with me and Jesse Tarnes ('16, within Seth's group) to make sure our code works with checkpointing. Information on both serial[4] and parallel[5] jobs can be found online.

Once you have copied over Henk's checkpointing scripts from his directory (`/home/hmeij/jobs/blcr/blcr_wrapper.serial`), you will notice some small differences from the original sample scripts. The first thing the script does is set up the code to operate from the scratch directory

```
export MYSANSCRATCH=/sanscratch/$LSB_JOBID
cd $MYSANSCRATCH
pre_cmd=``scp  $HOME/kflaherty/test.py .''
post_cmd=``scp  $MYSANSCRATCH/result.txt $HOME/kflaherty/``
```

These commands first set up the `$MYSANSCRATCH` system variable, move to the temporary directory and set up commands that will copy over your code at the start of the run, and copy back any results (`result.txt`) of the run. The operation of copying your code and data into the scratch directory is not executed right away, this just sets up the commands to be called later. Moving everything to the temporary directory, including your code, is important when checkpointing because the BLCR code has to recover everything that was used by the code in order to properly restart it. This includes every file created by the code while it is running. In the example above, I only have the file `test.py` that needs to be copied over at the beginning, but this can be expanded to include any pieces of code, data sets, support files, etc. that are needed.

Next you need to tell the wrapper that you are starting a new job:

```
mode=start
queue=mwd256fd
cmd=``python test.py''
```

where `queue` specifies the queue to which the code is sent (the same as specified with `BSUB -q`), while the second line specifies command used to call the code. If you are restarting a job then the code block above is replaced by:

```
mode=restart
queue=mw256fd
orgjobid=250
```

[4] https://dokuwiki.wesleyan.edu/doku.php?id=cluster:147
[5] https://dokuwiki.wesleyan.edu/doku.php?id=cluster:148

The queue must be the same as the queue used for the original job. The `orgjobid` parameter is the job ID number for the original job.

Finally, the time interval over which the checkpointing occurs is specified:

```
cpti=15m
```

This requests that a complete copy of the state of system is saved every 15 minutes. This copy includes all of the files that have been generated, all of the code used to generate them, and the state of the memory. These states are saved within `/sanscratch/checkpoints/$JOBID` where `$JOBID` is the unique ID number for the job. For any full run of your code a checkpoint every 15 minutes is overkill; more reasonable values are 12h (12 hours) or 2d (2 days).

You can check on the status of the script using two files within the checkpoints folder titled `cr_checkpoint.err` and `cr_mpirun.err`. The first of these keeps a log of errors associated with the checkpointing process. The second keeps tracks of any errors/warning output by your code during the course of its operation. This later file can be useful for checking on the progress of your code (e.g. if your code crashes without stopping, you will likely see an error message here).

Also, the blcr wrapper will pipe any text that would normally be written on the screen into a file in your home directory. It is located in `/home/kflaherty/.lsbatch` and the name will have a format `(random number).$JOBID.out`. If your code regularly prints text to the screen, then this is a good place to look for the text as the code is running.

Once the wrapper has been set up it can be sent to the scheduler in the same manner as any other script. As soon as a file titled `chk.PID` shows up, the code has completed its first checkpoint. From here on the code can be resumed from this point after it crashes.

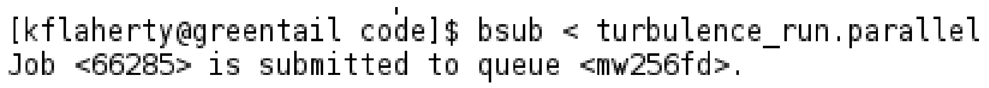
3. SUBMITTING A JOB

Regardless of whether a job is serial or parallel, or includes checkpointing, submitting the job to the scheduler is handled in the same way. This is done with the `bsub` command (Figure 1):

```
bsub < myscript
```

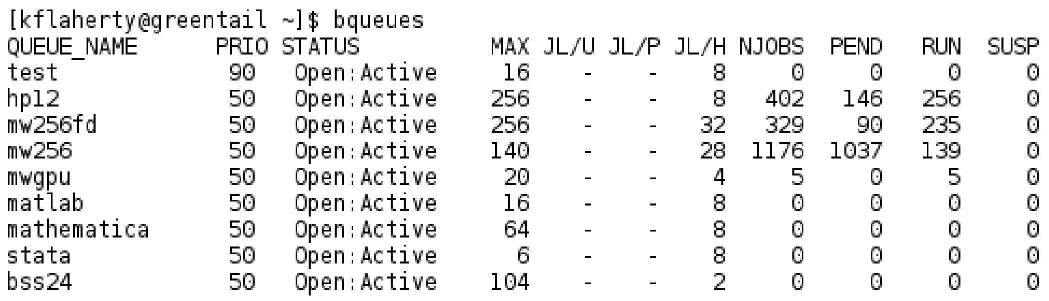
where `myscript` is the script containing the commands listed above. This pipes the `myscript` file into the `bsub` command, including all of the `bsub` parameters specified within the script.

Before submitting a job, you can check the status of the queues to e.g. determine how many other jobs are still pending on your desired queue. This is done with `bqueues` (Figure 2).



```
[kflaherty@greentail code]$ bsub < turbulence_run.parallel
Job <66285> is submitted to queue <mw256fd>.
```

Figure 1. Example of the `bsub` command. A script is submitted to the mw256fd queue and is assigned a job id number.

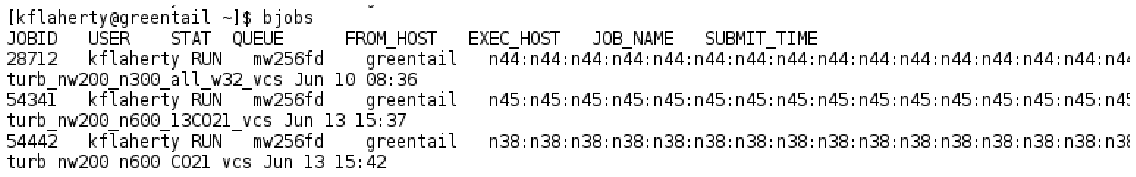


```
[kflaherty@greentail ~]$ bqueues
QUEUE_NAME      PRIO STATUS          MAX JL/U JL/P JL/H NJOBS  PEND   RUN  SUSP
test             90   Open:Active     16    -    -    8     0     0     0     0
hpl2             50   Open:Active    256    -    -    8   402   146   256     0
mw256fd          50   Open:Active    256    -    -   32   329    90   235     0
mw256            50   Open:Active    140    -    -   28  1176  1037   139     0
mwgpu            50   Open:Active     20    -    -    4     5     0     5     0
matlab           50   Open:Active     16    -    -    8     0     0     0     0
mathematica      50   Open:Active     64    -    -    8     0     0     0     0
stata            50   Open:Active      6    -    -    8     0     0     0     0
bss24            50   Open:Active    104    -    -    2     0     0     0     0
```

Figure 2. Example of the `bqueues` command. This lists all of the available queues, the maximum number of jobs, the allowable jobs per host, the number of submitted jobs, the number of pending jobs, the number of running jobs and the number of suspended jobs. Most of the queues are currently overloaded, although this tends to fluctuate with time.

This lists the name of all available queues, the status of the queue, the number of jobs per host, the number of total jobs submitted to the queue, the number of pending jobs, and the number of running jobs.

You can check on any submitted jobs using the `bjobs` command (Figure 3).



```
[kflaherty@greentail ~]$ bjobs
JOBID   USER    STAT  QUEUE      FROM_HOST   EXEC_HOST   JOB_NAME   SUBMIT_TIME
28712   kflaherty RUN   mw256fd    greentail   n44:n44:n44:n44:n44:n44:n44:n44:n44:n44:n44:n44:n44:n44:n44:n4
turb_nw200_n300_all_w32_vcs Jun 10 08:36
54341   kflaherty RUN   mw256fd    greentail   n45:n45:n45:n45:n45:n45:n45:n45:n45:n45:n45:n45:n45:n45:n45:n4
turb_nw200_n600_13CO21_vcs Jun 13 15:37
54442   kflaherty RUN   mw256fd    greentail   n38:n38:n38:n38:n38:n38:n38:n38:n38:n38:n38:n38:n38:n38:n38:n3
turb nw200 n600 CO21 vcs Jun 13 15:42
```

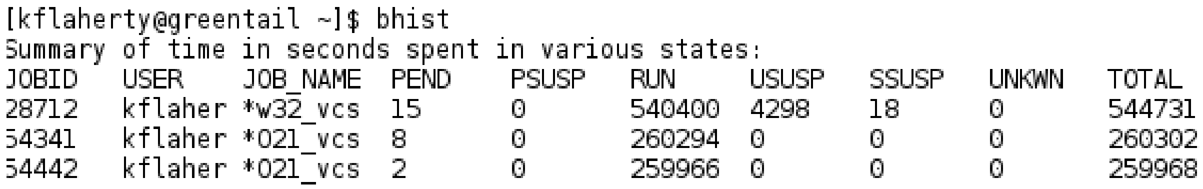
Figure 3. Example of a call to `bjobs`. I have submitted three jobs from greentail, all of which are running. Each job is taking up eight cores on different nodes.

This command will list any jobs that are running, or have been recently submitted. It lists the unique job ID number, the user that submitted the job, the status of the job (e.g. running vs pending), the queue that the job was sent to, the login node from which the job was submitted, the cores that are executing the job, the name of the job, and the time that the job that was submitted. A useful variant of bjobs is:

```
bjobs -u all -q mw256fd
```

This lists the jobs from all users that have been submitted to a particular queue. This can be useful for looking in detail at the load on a particular queue. The `bjobs` command can also be called with the `-m node` flag, where `node` is the node name (e.g. n44); `bjobs` will then show the jobs operating on a particular node.

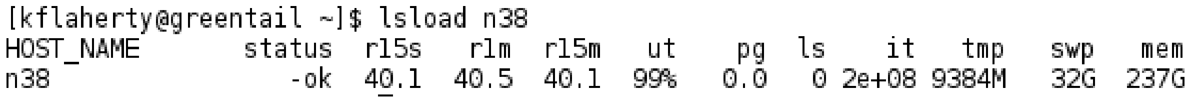
Similarly, the `bhist` command (Figure 4) lists any submitted jobs, including the amount of time they have spent in various states (e.g. running vs pending).



```
[kflaherty@greentail ~]$ bhist
Summary of time in seconds spent in various states:
JOBID   USER    JOB_NAME  PEND    PSUSP   RUN     USUSP   SSUSP   UNKWN   TOTAL
28712   kflaher *w32_vcs  15      0       540400  4298    18      0       544731
54341   kflaher *021_vcs  8       0       260294  0       0       0       260302
54442   kflaher *021_vcs  2       0       259966  0       0       0       259968
```

Figure 4. Example of a call to `bhist`. This shows the three jobs currently running, along with the number of seconds they have spent in various states.

Once a job is running, a number of commands can be used to check its status. The first is `lsload` (Figure 5), which lists the load on a particular node, in the form of the number of cores that are currently operating, and the amount of memory being used. The number of cores should be at least as large as the number of cores requested by your code (assuming you have requested that all cores be located within a single node). If not, then the code has likely crashed without stopping (see below).



```
[kflaherty@greentail ~]$ lsload n38
HOST_NAME        status  r15s   r1m  r15m   ut    pg  ls    it   tmp   swp   mem
n38                 -ok  40.1  40.5  40.1  99%   0.0   0 2e+08 9384M   32G  237G
```

Figure 5. Example of a call to `lsload`. The values *r15s*, *r1m*, *r15m* shows the core usage averaged over the previous 15 seconds, 1 miute and 15 minutes.

Another method for checking on the operation of your code is with the `top` command, which can be called as :

```
ssh <node> top -u <username> -b -n 1
```

where `<node>` is the node name (e.g. n42) and `<username>` is your username. This will execute `top` on all of your commands executing on the specified node (Figure 6). This contains information on the total memory usage (important to keep track of) and the fraction of CPU being devoted to a particular command.

If you need to terminate a job for any reason, then this can be done with the `bkill` command:

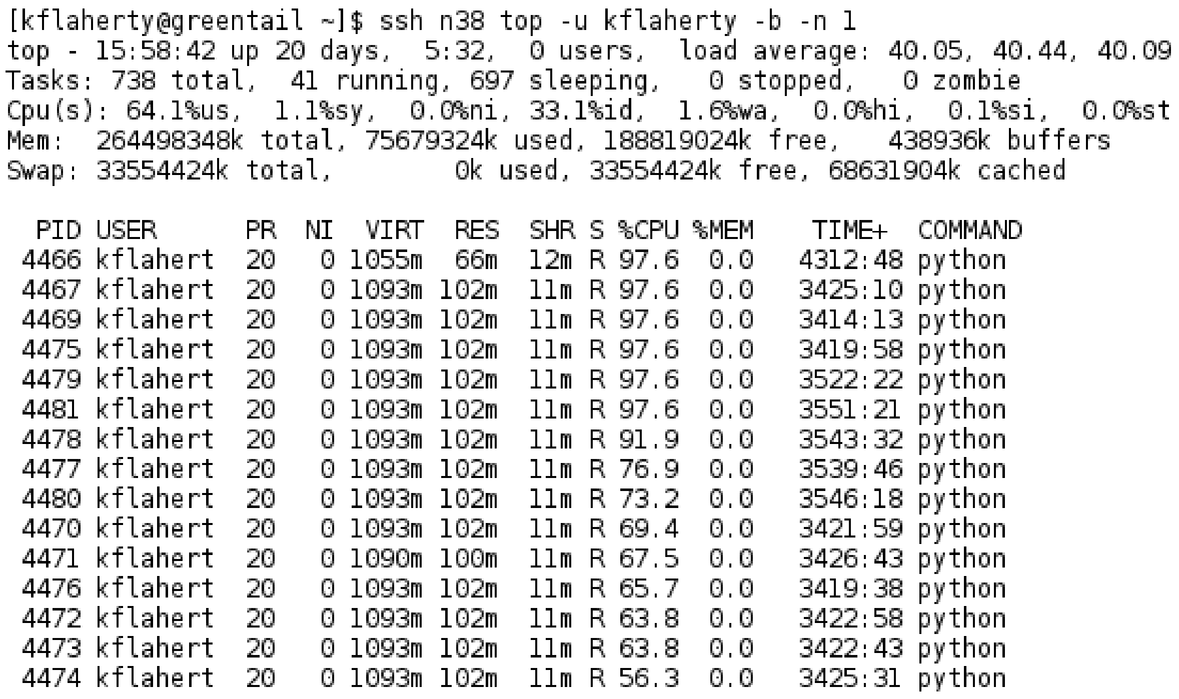
```
bkill <jobid>
```

where `<jobid>` is the unique ID number for the particular job that you want to kill. This may take a minute or two depending on the complexity of the job that is being killed.

4. MISC.

4.1. *Crashing without Stopping*

Most of the time if your code crashes the job will be killed, and error messages will be output to the `.stderr` and `.stdout` files. But I have run into some in-



```
[kflaherty@greentail ~]$ ssh n38 top -u kflaherty -b -n 1
top - 15:58:42 up 20 days,  5:32,  0 users,  load average: 40.05, 40.44, 40.09
Tasks: 738 total,  41 running, 697 sleeping,   0 stopped,   0 zombie
Cpu(s): 64.1%us,  1.1%sy,  0.0%ni, 33.1%id,  1.6%wa,  0.0%hi,  0.1%si,  0.0%st
Mem:  264498348k total, 75679324k used, 188819024k free,   438936k buffers
Swap: 33554424k total,        0k used, 33554424k free, 68631904k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 4466 kflahert  20   0 1055m  66m  12m R 97.6  0.0   4312:48 python
 4467 kflahert  20   0 1093m 102m  11m R 97.6  0.0   3425:10 python
 4469 kflahert  20   0 1093m 102m  11m R 97.6  0.0   3414:13 python
 4475 kflahert  20   0 1093m 102m  11m R 97.6  0.0   3419:58 python
 4479 kflahert  20   0 1093m 102m  11m R 97.6  0.0   3522:22 python
 4481 kflahert  20   0 1093m 102m  11m R 97.6  0.0   3551:21 python
 4478 kflahert  20   0 1093m 102m  11m R 91.9  0.0   3543:32 python
 4477 kflahert  20   0 1093m 102m  11m R 76.9  0.0   3539:46 python
 4480 kflahert  20   0 1093m 102m  11m R 73.2  0.0   3546:18 python
 4470 kflahert  20   0 1093m 102m  11m R 69.4  0.0   3421:59 python
 4471 kflahert  20   0 1090m 100m  11m R 67.5  0.0   3426:43 python
 4476 kflahert  20   0 1093m 102m  11m R 65.7  0.0   3419:38 python
 4472 kflahert  20   0 1093m 102m  11m R 63.8  0.0   3422:58 python
 4473 kflahert  20   0 1093m 102m  11m R 63.8  0.0   3422:43 python
 4474 kflahert  20   0 1093m 102m  11m R 56.3  0.0   3425:31 python
```

Figure 6. Example of calling `top` for a particular node. There are lots of python processes running right now.

stances where the code will crash, but the job will not kill itself. This can happen if *emcee* runs into a problem with a particular model and isn't able to proceed any further along its chains. If you have piped your output into `test.out` (described below), then you can see this happen when `test.out` stops growing. At this point you have to kill the job and examine the error file to discern what went wrong. Similarly, for checkpointed jobs you can look at `~/.lsbatch/[0-9]*.JOBID.out` or `/sanscratch/checkpoints/$JOBID/cr_mpirun.err` to see if it has registered any errors.

4.2. *MIRIAD on the Cluster*

Miriad runs the same as on our local machines, with one small exception having to do with how `uvmodel` handles polarized data and an unpolarized model. On my local machine, if the data has dimensions N_{base}xN_{chan}x3x2, where N_{base} is the number of baselines and N_{chan} is the number of channels and the last dimension splits the data into XX and YY, then when the model is run through `uvmodel` it results in a visibility file with dimensions $2N_{base}$xN_{chan}x3x1. The last dimension has been reduced while the first dimension has been doubled, with the intensity copied into twice as many baseline positions. On the cluster `uvmodel` does not do this. It maintains the original dimensions of the data (N_{base}xN_{chan}x3x2) and copies intensity into the XX,YY positions in the array. This requires slightly different handling of the model visibilities when they are loaded into python.

Also note that miriad is very talkative, and likes to tell you all about things that are going on when it is running. This can become burdensome when hundreds of thousands of models are run since all of this information will be dumped into the `.stderr` and `.stdout` files created at the end of each run (described below). To avoid keeping this useless information, I usually pipe the command into a file (e.g. `python code.py > test.out`. The file `test.out` will slowly grow with time, which a useful way to test that your code has not crashed. If you place this file in the scratch directory then it will deleted once the code is done running. When checkpointing is used but the output is instead placed in `~/.lsbatch/[0-9]*.JOBID.out`. When the job ends, this file becomes the `.out` file.