

Running code on the Wesleyan Cluster

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 contains a number of clusters all available through one scheduler. Each cluster consists of a collection of nodes, each with a multiple cores. The number of nodes range from 5 to 32 with the number of cores ranging from one to 32 (with hyperthreading turned on). A separate cluster holds the Home directory, with a total of 10 TB, as well as a scratch directory, with a total of 5 TB. More details on the exact hardware setup is available online¹ along with a list of available software² and basic, although slightly out of date, information about connecting to the cluster³

The first thing to do is contact Henk Meij to set up an account on the cluster. This will give you access to swallowtail, greentail, petaltail and cottontail (soon the be exclusively cottontail), which are the machines used to access the cluster. Once this is done you can ssh into a machine (I usually use greentail) you will be dumped into your home directory (e.g. `/home/kflaherty`). This is where you will house your code and store your results. A simple scp command⁴ can be used to copy the code from your personal machine to the cluster.

Listed below are sample codes useful for setting up a program that runs in serial and a program that runs in parallel, followed by some useful command line utilities. The example serial/parallel script include examples of code for specifying various parameters in the job submission program, setting up python and miriad, copying the data over to a local scratch directory, and running the code itself.

2. Scripts

Jobs are submitted to the queue of your choice as scripts containing a series of commands following a basic structure. These shell scripts set up the details of the job (e.g. name of the

¹<https://dokuwiki.wesleyan.edu/doku.php?id=cluster:126>

²<https://dokuwiki.wesleyan.edu/doku.php?id=cluster:73>

³https://wesfiles.wesleyan.edu/departments/SCIC/Public/Documentation/unix_cluster/JOB_ON_SWALLOWTAIL.pdf

⁴e.g. Starting from your home machine enter: `scp myfile.fits kflaherty@greentail:/home/kflaherty/code/`

job, number of cores needed), define any system variables (e.g. for running miriad) and call your actual workhorse code. Outlined below are the basic commands used in a script used for a serial job and for a parallel job. Examples of these pieces of code are available in Henk’s home directory */home/hmeij/jobs* in the *serial* and *parallel* directories. Note that these scripts do not include checkpointing, discussed below, and should be used as starting points to understand basic scripting and to make sure your code works as expected when ported to the cluster. Full production runs should use the wrappers that include checkpointing to ensure that any mid-run crashes do not severely disrupt your work.

2.1. Basic serial script

Here are some of the commands that go into a basic serial computing script.

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

(1)

Set up the bash shell. Any shell scripts called by your code must be setup in bash, not csh or tcsh.

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

(2)

Set up a number of parameters for the *bsub* command. The first line specifies the queue to which the code is sent. The second line gives the name of the job, while the third and fourth line specify names for output files and error output files. These last two files are created once the job is complete and they contain standard outputs from your code, as well as any error messages generated while running the code. The *test.stdout* file will 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 *.stderr* file contains error messages produced by your code while it is running.

```
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
```

(3)

These commands set up environment variables necessary to call supporting programs (don’t forget the space after the period in the fourth line above). In this case, I am using a couple of python packages as well as MIRIAD. 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 mw256[fd] queues. So even though we don't need the extra memory space available on this cluster, we need to use it to run MIRIAD.

$$cp -r mydata.vis /sancscratch/$LSB_JOBID \quad (4)$$

When your job starts, a temporary directory is created within the /sancscratch directory. Data I/O should be done from this directory to avoid over taxing the /home directory. Many people forget to do this, and the I/O on the /home directory slows down. This temporary directory is created using your unique job id number, accessed with the environment variable it *\$LSB_JOBID*. It is also erased as soon as your job is completed. Make sure to copy any data you need out of this directory before the end of your script! In your code use the name 'sancscratch' but if want to look into this directory from the command line (e.g. with an ls command to make sure things got transferred over) you want to use that name 'sancscratch_sharptail'.

$$python test.py /sancscratch/$LSB_JOBID \quad (5)$$

This is the command to call my actual code. In this case, my code is called *test.py* and it takes as input the temporary working directory.

That covers all of the basics of a serial code. This is a shell script, so any shell commands will also work here, but any heavy lifting should probably be left to your code to leave this script as simple as possible.

2.2. Basic Parallel script

Running a parallel code is similar to running a serial code, with some minor additions, which are outlined below.

$$\begin{aligned} \#BSUB -n 8 \\ \#BSUB -R "span[ptile = 1]" \\ \#BSUB -R "span[hosts = 1]" \end{aligned} \quad (6)$$

The first command specifies the number of hosts to send this job to, in this case 8. The next two commands are actually mutually exclusive, but are both included here for illustration. The

ptile=1 command tells the scheduler to one send one job to each node, and to use as many nodes as are specified (for reference the *mw256fd* queue has 5 nodes with 32 threads per node, so you would be limited to *-n 5*). The *hosts=1* command tells the scheduler to dump all of the jobs onto one node. The advantage/disadvantage of each depends on the exact nature of your code. Try both options and see which provides the best results.

```
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
```

 (7)

These commands set up various path variables pointing to the installation of openmpi.

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

 (8)

Calling your code is slightly different here than in the serial case. Here was call a wrapper (*lava.openmpi.wrapper*) that sets various variables (the machine file and number of parallel jobs) for this code. Basically instead of directly calling *mpirun*, as you would on your local machine, use this wrapper. You will need to copy over this wrapper from Hank’s directory (/home/hmeij/jobs/parallel/)

Everything else should be run as it was in the serial script. You still need to setup the paths for python and miriad, as well as copy the data to and from the scratch directory.

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 serial⁵ and parallel⁶ jobs can be found online. These resources include same wrappers to be used when calling a serial or parallel job, replacing the sample scripts in /home/hmeij/jobs. Here I will highlight some of the differences in the code, and how to use checkpointing.

⁵<https://dokuwiki.wesleyan.edu/doku.php?id=cluster:147>

⁶<https://dokuwiki.wesleyan.edu/doku.php?id=cluster:148>

3.1. Checkpointing scripts

Once you have copied over Henk’s scripts, you will notice some small differences from the original sample scripts. The first thing the script does is set the scratch directory as a system variable and move all code from the home directory to this scratch directory.

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

(9)

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. These commands are not executed at this moment, but they are set up to be executed 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.

The next few specify whether this is the start of a new run, or the restart of a run that crashed.

```
mode = start
queue = test
cmd=" python test.py /sanscratch/$LSB_JOBID"

#mode = restart
#queue = test
#orgjobid = 250
```

(10)

The first three lines are to be used when starting a new run. This sets up the correct queue as well as the command that is used to call the code. If instead you are restarting a prematurely terminated run, then comment out the first three lines and uncomment the next three lines. The variable `orgjobid` refers to the `JOBID` (found using the `bjobs` command when it was running) and must be adjusted to the proper value.

The final command, which likely occurs earlier in the script, sets the time interval between checkpointing.

$$cpti = 15m \tag{11}$$

This sets will define checkpointing to occur every 15 minutes. For any production run of your code this is overkill, and should only be used if you are testing the checkpointing process. More appropriate values are 12h, 18h or 1d depending on the full length of your run.

You can check on the status of the script using two files within the checkpoints folder (*/sanscratch/checkpoints/\$JOBID*) 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 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 on 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.

3.2. How to take advantage of Checkpointing

There are detailed descriptions online of how to recover from a crash, and here I will highlight the more important details. The first step is to uncomment the start block of the wrapper. Once this is done, the wrapper can be submitted to the queue like any other script.

$$bsub < blcr_wrapper \tag{12}$$

This will generate a temporary directory as before, but it will also create a file in the checkpoints directory (*/sanscratch/checkpoints/\$JOBID*). This folder will contain the checkpoint updates. 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. The process for restarting is straightforward. Simply comment out the start block, and uncomment the restart block, and resubmit the *blcr_wrapper* script. The restart does need to be submitted to the same queue as the original run, but doesn't need to land on the same node. It will take a few minutes to restart, but should get going soon enough.

4. Useful Commands

In addition to the scripts that set up the code, there are a number of useful command line utilities. Many of these have an associated man page that contains more information on their function as well as various flags that accompany the command.

bsub < *my.code* (Figure 1) Short for bsubmit, this is the main code used to submit a job to the queue. The file *my.code* contains the commands needed to run your code, as outlined in the previous section.

bqueues (Figure 2) This command lists information on the available queues including the name of the queues, its priority, status, the maximum number of jobs, the number of jobs per host, the number of total jobs on this machine, the number of pending jobs and the number of jobs currently running (and suspended). There are a number of different clusters to which your code can be sent. If you have a very memory intensive job (as in many GB of memory), then send it to mw256[fd]. If you can run the code on your own machine then hp12 should be good enough. If your code runs on a gpu then mwgpu is the queue for you. The queue hp12 is made up of 32 nodes, each with 12 GB of memory, while mw256[fd] are each made of 5 nodes with 256 GB of memory. The hp12 nodes (n1-n32) can each handle 16 jobs per node, while the mw256fd nodes (n38-n45) can each handle 32 jobs per node, while mw256 (n33-n37) can handle 28 jobs per node (the extra 4 jobs per node are reserved for GPU calculations).

bjobs (Figure 3) This command will list any current jobs that you are running. This shows the unique jobid number, the user that submitted the job, its status, the queue it is assigned to, the host from which the command was sent, the nodes that are executing the command, the name of the job and the time the job was submitted. To view the jobs submitted by all users, use the *-u all* keyword. Call this command with the *-q queue* flag, where *queue* is the specific queue of interest, to see all of the jobs for a specific queue. Specify the *-m node* flag to look at the jobs related to a particular node (e.g. n45).

bkill jobid This command kills a job that you are currently running. Most jobs, even when they fail, will exit on their own, although there are some exceptions.

lsload node (Figure 5) Check the load on a node. Usefully to make sure you aren't overloading the node, which is usually a sign of something wrong

ssh node top -u username -b -n 1 (Figure 6) This is another useful command for checking a load on a particular node. This command calls *top* for that node, with various flags to e.g. only list the programs run by one user. This also contains information on your total memory usage (important to keep track of)

4.1. Misc.

Miriad on the cluster: Miriad runs the same as on our local machines, with one small exception. This has to do with how *uvmodel* handles polarized data and an unpolarized model. On my local machine, if the data has dimensions $N_{base} \times N_{chan} \times 3 \times 2$, 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 $2N_{base} \times N_{chan} \times 3 \times 1$. 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 ($N_{base} \times N_{chan} \times 3 \times 2$) 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. Check out `/home/kflaherty/code/single_model.py`, under the `compare_vis` function.

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). These files could reach hundreds of MB! 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 is a useful way to test that your code has not crashed. If you place this file in the scratch directory then it will be deleted once the code is done running. When checkpointing is used, the file `test.out` may not appear, but the output is instead placed in `./lsbatch/[0-9]*.JOBID.out`. When the job ends, this file becomes the `.out` file.

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 instances 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` as described above, 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.

Generic file names: Also, if you output any files (e.g. model images that need to be run through miriad tasks) be careful about giving them generic names. When running a parallel job you could potentially end up with many files with the same name, which is a recipe for disaster. Take advantage of the python utility that generates a unique name to keep parallel processes separate (and clean up after yourself). An example of how to do this is in the `/home/kflaherty/code/single_model.py` code within the function `lnlike` (look for the `cleanup` keyword).

Shell type: The cluster uses `bsh`, which is different than my machine. If you are using a shell script to e.g. call miriad tasks make sure it references the correct shell (taken care of with the first line in the script).


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

Fig. 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

Fig. 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.

[illegible]

Fig. 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.

```
[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
```

Fig. 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.

```
[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

Fig. 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.

```
[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

Fig. 6.— Example of calling *top* for a particular node. There are lots of python processes running right now.