Using Jinx

How to use Jinx

Finally we are ready to give you the go ahead for using the *Jinx* cluster. We apologize for the long wait.

You can now run your jobs on *Jinx* though, with some caveats.

Logging Into Jinx

The Jinx cluster consists of up to 30 nodes. One node is the designated login-node, called jinx-login.

This is the only node you can connect to from the georgia tech network.

To connect to Jinx, run (replace gtusername with your actual GT username):

```
ssh gtusername@jinx-login.cc.gatech.edu
```

You can verify that you are logged in correctly by running the `hostname` command, which should print:

```
$ hostname
jinx-login
```

Compiling

The login node (jinx-login) is shared between all users, and has a few restrictions in place. You should use it only for dispatching jobs, and never to execute any problems.

For compiling our code, we will request an interactive session with a single node.

To do so, we run:

```
qsub -q class -I
```

qsub is the command used for submitting jobs to the job scheduling system TORQUE.

With this particular command, we are requesting the default allocation (1 node for 10 minutes) of queue `class` as an interactive session (-I).

Once you enter this command, you will see something like the following:

```
qsub: waiting for job 459111.jinx-login.cc.gatech.edu to start
```

and after a (hopefully) short wait, it will allocate a node to you and you see this:

```
qsub: job 459111.jinx-login.cc.gatech.edu ready

Adding user gtusername to group authjobs
-bash-4.1$
```

You are now logged into one of Jinx's nodes. Running `hostname` should now print which node you are logged into:

```
-bash-4.1$ hostname
jinx4
```

So you can now run any command on this node. Beware: your session will automatically terminate after your

allotted time (10 minutes) has expired.

We can now load the OpenMPI environment on this node and then compile our program:

1) load OpenMPI module:

```
module load openmpi-x86_64
```

Compile our program:

```
$ make
```

You can now also test-run your program on this node:

```
$ mpirun -np 6 ./prog1  <commandline-input1> <commandline-input2>
<your program output will be here>
```

This will use only your local node however.

Now that we made sure everything compiled and works, let's exit the interactive session and start setting up

scripts for (much) larger batch submissions.

```
exit
```

Submitting Batch Jobs

Jinx is using the scheduling system TORQUE/PBS. For submitting batch jobs, we will have to write a job script that

defines the resources we request and what commands to run when the job is actually executed.

The script is a bash script with additional #PBS commands inserted at the top:

Let's look at an example:

```
#!/bin/sh

#PBS -q class
#PBS -l nodes=4:sixcore
#PBS -l walltime=00:10:00
#PBS -N cse6220-prog1

# commands to run go here!
```

Here we use the #PBS commands to specify the resources we want. In this example we are requesting an allocation of 4 nodes of type "sixcore" for a maximum of 10 minutes.

We have to also define the queue we are submitting to, which for us will be the queue "class".

The "-N" parameter simply gives our job a name, you are free to choose whatever you want here.

You can look at the hardware description here to see what hardware each node in Jinx contains: https://support.cc.gatech.edu/facilities/instructional-labs/jinx-cluster

BEWARE: Do **NOT** follow their "Compiling Programs on the Jinx Cluster" or "How to Run Jobs on the Jinx Cluster" manuals, since they will not work. MPI on Jinx is currently mis-configured,

so we have to call mpirun with some tweaks to get it working correctly.

The nodes of type "sixcore" on Jinx each have two processors with 6 cores each, making a total of 12 cores. Additionally, the CPUs use hyperthreading, so the systems can run up to

24 threads at the same time. However, we will not use hyper-threading for our experiments, since it will increase the performance only by little.

Requesting 4 nodes thus allows us to run up to 48 processes, one on each core.

Next, we want to actually run something with our script, so we add the following:

```
# change to our project directory
cd $HOME/cse6220-prog1

# hardcode MPI path
MPIRUN=/usr/lib64/openmpi/bin/mpirun

# loop over number of processors (our 4 nodes job can run up to 48)
for p in 6 12 24 48
```

```
do
    $MPIRUN -np $p --hostfile $PBS_NODEFILE ./poly constants.txt values.txt
done
```

Note, that we first have to `cd` into the directory of our project.

Also note, that we have to use $MPIRUN instead of mpirun. This is because Jinx is currently mis-configured, so we have to explicitly supply the whole path to the mpirun executable.

We are passing the list of allocated nodes to mpirun via --hostfile $PBS_NODEFILE.

Everything after ./prog1 are commandline parameters that are passed to the prog1 executable.

Let's say we called this script pbs_script.sh.

We can now submit the script to the Jinx scheduler via:

```
qsub pbs_script.sh
```

We can now monitor our job with

```
qstat -u gtusername #Reports jobs for user gtusername
qstat -a  #Reports all jobs
```

Once the job finishes, you will have two new files: cse6220-prog1.e459116 and cse6220-prog1.o459116 (the number at the end will be different)

these contain the commandline standard output and standard error.

The standard output file should contain something like this:

```
$ cat cse6220-prog1.o459116
Adding user gtusername to group authjobs
<your program output will be here>
Removing user gtusername from group authjobs
```