How to use the cluster

This note is an addition to Mike's notes titled "How to cluster," and much of it is a direct clone of those notes. I made an effort to provide notes on our brand-new cluster (CMT), more examples of serial jobs, GitHub usage, as well as a brief note on how to use vim on the cluster. If found, report errors in this note to Sasanka. The most updated version of this note can be found in How to use the cluster.

Available clusters

The following is our group's own cluster

1. CMT (stands for 'Condensed Matter Theory' I guess)

    1. 4 cores, 700 GB RAM
    2. Time available - 7 days.
    3. Accessed via UTD netid and password

Logging into the cluster

(Linux and Mac) Open the terminal and type the following

$ ssh -Y (YOUR NET ID)@ganymede2.circ.utdallas.edu

The password is your usual password for outlook or Galaxy.

If you are NOT using CometNet, to log in, you need to first connect to the UTD VPN GlobalProtect. For details on how to install GlobalProtect, see the following link - *https://atlas.utdallas.edu/TDClient/30/Portal/Requests/ServiceDet?ID=167*

Before submitting a job to the cluster

Modify the .bashrc file (only need to do it once)

If you are using the cluster for the **first time**, you need to modify the .bashrc file to set up a few basic variables/load a few modules. The following is taken from Mike's .bashrc file

Set the path, which tells you where executables are, to include the folder $HOME/bin:
```
export PATH="$HOME/bin:$PATH"
```

Set the python path, which tells python where importable files are, to include the folder $HOME/py_include
```
export PYTHONPATH=$PYTHONPATH:$HOME/py_include/
```

Set various paths for C++ libraries for including and linking
```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$HOME/lib
export C_INCLUDE_PATH=$C_INCLUDE_PATH:$HOME/include
export CPLUS_INCLUDE_PATH=$CPLUS_INCLUDE_PATH:$HOME/include
```

Load any modules you regularly use, you don't have to use all of these, and can add others as necessary
```
module load anaconda2
module load git
module load matlab
```

Put your email in the following line, this way, an email will be sent to you when your job is finished/cancelled/timeout/error
```
export MYEMAIL=your_email@utdallas.edu
```

On topo or any other clusters without an explicit scratch directory, I encourage you to make your own and have the shell variable $SCRATCH reference it. For instance, I use this on topo
```
export SCRATCH=$HOME/data
```

Note that Lonestar has a very specific format where it wants you to put things in your .bashrc file. Please read commented statements in the .bashrc file for details.

Create local copies of various helper scripts from Github (only need to do it once, the first time you use the cluster)

Mike has written a number of scripts to make job submission on clusters as easy as possible, as well as various other scripts that he uses for programming. Here we will copy those scripts onto the cluster.

$ cd $HOME

Clone the folder containing templates for serial job submission, as well as basic things like plotting in python. Pick the one that is relevant to your specific cluster.

For Ganymede
$ git clone https://mkolodru@bitbucket.org/mkolodru/templates_ganymede.git templates

For topo
$ git clone https://mkolodru@bitbucket.org/mkolodru/templates_topo.git templates

For ls5
$ git clone https://mkolodru@bitbucket.org/mkolodru/templates_ls5.git templates

(The resulting files will end up in the folder $HOME/templates)

(Clone the folder containing various executables that are useful (including “initfolder” and “replace.py”) into your $HOME/bin folder. If you already have things in $HOME/bin, make sure to back them up, because cloning may overwrite them.)

$ git clone https://mkolodru@bitbucket.org/mkolodru/mkolodru_bin.git bin

Cloning a folder from Github

In this section, we will clone an existing folder from GitHub.
Once logged in to the cluster, under the home directory, do the following

$ git clone https://github.com/Psasanka1729/test_multiply.git

Now, if you type $ ls from your terminal, you should be able to see the folder “test_multiply”.

Synchronising code among GitHub, cluster and your local computer

It is recommended to synchronise your codes among GitHub, cluster and your local computer.

Creating a repository on GitHub is simple: Creating a repository.
(Tracks all the files, and they are ready to commit to GitHub)
$ git add –all

(If you only want to track the python files, you can use something like the following)
$ git add *.py

(Use the following command to commit to GitHub)
$ git commit -m "some message"

For simplicity, we will not discuss GitHub branches and how to manage them here, all the commits will be done to the main branch, if you do not understand this, no worries, just use this guide as it is.

Recently, GitHub changed its policy and users will need a token to push codes to GitHub.
See this link for How to create a personal access token?

After you successfully generate a token, you can use the following to push code from the cluster or your personal computer to GitHub.
$ git push -f https://YOUR_TOKEN@github.com/GITHUB_USERNAME/test_multiply.git

Replace YOUR_TOKEN with the token you just generated and GITHUB_USERNAME with your GitHub username.

To copy the code from GitHub to the cluster or your local computer, first, go inside the “test_multiply” folder, (if you are not there already), then you may do the following
$ git pull https://github.com/GITHUB_USERNAME/test_multiply.git

[NOTE] I have recently found that these python codes were written for python version 2.7 and can create issues for version >= 3.10. If that is the case then use the codes in the following repository. These are essentially the same but written for python version > 3.10.

$ git clone https://github.com/psasanka1729/ganymede2.git templates
$ git clone https://github.com/psasanka1729/ganymede2.git bin

Creating a run and submitting a job to the cluster

We will follow what Mike describes in his guide for “Using the cluster”.

If you are not already, you should be inside the “test_multiply” folder. You may use $ pwd to check it, it should show something like this in your terminal - */home/sxd190113/test_multiply*
Next, do the following

(Make a directory named “build”)
$ mkdir build

(Move to the directory build that was just created)
$ cd build

(Inside this build, we will have different runs. Create a directory with some name that you will be able to reference later, for now, we will name it “run1_09242022”)
$ mkdir run1_09242022

(Move inside the run1_09242022 folder)
$ cd run1_09242022

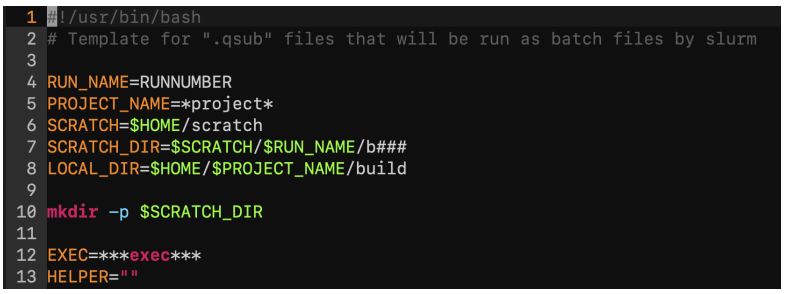
(Next up, we need to copy the files that will be used to generate code that may be run on the cluster. This consists of all the files starting with “RUNNUMBER” in the templates folder:)

$ cp ~/templates/RUNNUMBER* ./

Next, we need to edit the two files we just copied. You can use any text editor that you like, the instructions here are for **vim.** For a quick guide on how to use vim, check the appendix of this document.

(Open the .template file)
$ vim RUNNUMBER.template

The file will look like this



Now, change the line from “EXEC=***exec***” to “EXEC=test_multiply.py”, and then save the file. This will tell the code to run the “test_multiply.py” file.

(Open the builder file)
$ vim RUNNUMBER_builder.py

You may see something like this

```
#! /usr/bin/env python

import subprocess
import numpy
import os

partition_info=['normal',16] # = [partition,ncores]
# partition_info=['debug',16] # = [partition,ncores]
time_str='24:00:00'
project_name=os.getcwd().split('/')[-3]
myemail=os.environ["MYEMAIL"]

```

The “time_str” in your file may look different. This tells how long your code will run. “partition_info = [‘normal’,16]”, means the code will run in the “normal” node Ganymede, if you want to use the “CMT”, replace the word “normal” with “cmt”. For “normal”, the time limit is 96 hours *i.e.* time_str = ‘96:00:00’, for “CMT”, it is 7 days *i.e.* time_str=’7-00:00:00’.

Next, we will rename all the files RUNNUMBER to run1_09242022 using the initfolder command
$ initfolder RUNNUMBER run1_09242022

[NOTE] It is important that this new name is the same as the name of the folder you are in.

You will notice that there are now three files:

1. run1_09242022.sbatch.template: This will be used as a template for run1_09242022.sbatch, which is used to submit to the slurm scheduler. You do not need to modify this file.
2. run1_09242022.template: This will be used as a template to generate the shell scripts run1_09242022.qsub, run1_09242022.qsub, etc. which will be used to run the actual code. They are designed so that each .qsub file will run on one core. I call each one of these a “version” of the run.
3. run1_09242022_builder.py: Executable python file to actually do the work in turning files (1) and (2) into what is actually needed for submitting to the queue

Next, run the builder file which will create the qsub files.
$ ./run1_09242022_builder.py

Doing so will generate the following files: run1_09242022.sbatch (see above), run1_09242022_*.qsub (see above), run1_09242022.task, versionmap.dat, and run1_09242022.log. The .task file is simply a list of the .qsub files, letting the .sbatch file know what to run. The .log file is a log of this particular run. In the builder file (see below), you should modify the log string to make sure there is a useful description of what this run is doing. The version map file is a table showing what parameter values correspond to what versions of the code (vnum=0 corresponds to run1_09242022_0.qsub, etc.). The .log file will also keep track of the git commit info (git commit is automatically called in running the builder file) so that you can get back to this code base whenever you want to.

Next, copy the python or any other file you want to run using the following

(If you are running a python file)
$ cp ../../print_product.py ./

(If you are running a Julia file)
$ cp ../../print_product.jl ./

(If you are running a matlab file)
$ cp ../../print_product.m ./

[NOTE] Below is a section where you can test your code on the login node. I do not recommend doing this on ganymede because they have been very strict recently about people running their code on the login node. You can skip ahead this part or test your code on the “debug” node. Nevertheless, I retain this part as a part of Mike’s original notes.

Test your job in the login node (optional step)
We are ready to submit the job to the cluster, however, if you want a quick check of whether your code is free from errors/bugs, you can run the code in the login node for a quick check.

$ bash run1_09242022_0.qsub

DO NOT let this run for more than about 10 seconds! It can slow the login node down significantly.

Once everything is working, submit the job to the cluster using the following
$ sbatch run1_09242022.sbatch

Submit serial jobs to the cluster

Serial job: Example 1

Let us say we want to submit the following job:
Multiply the numbers of the two lists : x = [1,2,3,4,5] and y = [6,7,8,9].

We will create a new run for this. Create a folder named run2_24092022 inside the folder **build**. Do the same steps as before.

$ cp ~/templates/RUNNUMBER* ./

The following are examples of two programs to do the job in python and Julia. You can either modify the older print_product.py file or create a new one with a different name.

Python
print_product.py

```
# Usage: print_product.py num1 num2 outfile
import sys
num1=float(sys.argv[1])
num2=float(sys.argv[2])
fout=open(sys.argv[3],'w')
fout.write(str(num1*num2)+'\n')
```

Julia
print_product.jl

```
# Usage: print_product.jl num1 num2 outfile
num1=parse(Float64,ARGS[1])
num2=parse(Float64,ARGS[2])
fname=ARGS[3]
open(fname, "w") do f
        write(f, string(num1*num2,'\n'))
end
```

First, we will modify the template file
$ vim RUNNUMBER.template

RUNNUMBER.template

```
#!/usr/bin/bash
# Template for ".qsub" files that will be run as batch files by slurm
RUN_NAME=RUNNUMBER
PROJECT_NAME=*project*
SCRATCH_DIR=$SCRATCH/$RUN_NAME/b###
LOCAL_DIR=$HOME/$PROJECT_NAME/build
mkdir -p $SCRATCH_DIR

EXEC1 = print_product.py  < - Name of the program we want to run.

HELPER = " " <- Helper files are what I call non-executables that need to be in the same
directory for the executables to be run, leave this empty if you do not have any helper file.

cd $LOCAL_DIR/$RUN_NAME
cp $EXEC1 $SCRATCH_DIR/

if [ ! -z "$HELPER"] # Check that HELPER isn't empty then
cp $HELPER $SCRATCH_DIR/ fi
cd $SCRATCH_DIR/

{ time python $EXEC1 *111* *222* *ppp*; } > temp.out 2> error.err (For  python)

(*111*, *222* and *ppp* are just placeholders and will be replaced by the parameters that we are
sweeping over. Our code takes three arguments, two numbers and one file name. *111* and
*222* are the two numbers and *ppp* is the file name (check the print_product.py serial code))

if [ "$(pwd)" == $SCRATCH_DIR ]; then
        echo "Removing files"
        rm $EXEC1
        if [ ! -z "$HELPER"] # Check that HELPER isn't empty
        then
                rm $HELPER
        fi
fi
```

Save and exit, if in vim $:wq . Next, we will modify the builder file,
$ vim RUNNUMBER_builder.py

RUNUMBER_builder.py

```
#! /usr/bin/env python
import subprocess import os
import numpy
partition_info=['CMT',16]  (using CMT)
# = [partition, ncores per node]

time_str='7-00:00:00' # Format = dd-hh:mm:ss (change time string accordingly)
project_name=os.getcwd().split('/')[-3]
myemail=os.environ["MYEMAIL"]

#Log the current submission
logstr=’’’RUNNUMBER:
Codebase:
Cluster: ganymede
gittag: ‘’’

#Do the git-ing
cmd='git commit -a -m "Commit before run RUNNUMBER" > /dev/null'
print(cmd)
subprocess.call(cmd,shell=True)

cmd='gittag.py > temp.temp'
subprocess.call(cmd,shell=True)
logstr+=open('temp.temp','r').readline()
subprocess.call('rm temp.temp',shell=True)

open('RUNNUMBER.log','w').write(logstr)

#Setup the versionmap and qsub files
vmap_file=open('versionmap.dat','w')
vmap_file.write('vnum\tL\n')

task_file=open('RUNNUMBER.task','w')
template_file='RUNNUMBER.template'
template_contents=open(template_file,'r').read()

vnum=0

for num1 in range(5):  (<- list x has five elements.)
```

```
        for num2 in range(4): (<- list y has four elements.)

                qsub_file=template_file.replace('.template','_'+str(vnum)+'.qsub')
                        fout=open(qsub_file,'w')
                        contents=template_contents.replace('###',str(vnum))
                        contents=contents.replace('*project*',project_name)

                        contents=contents.replace('*111*',str(x[num1]))
                        (when the loop runs, x[num1] will loop over all elements of x, and *111*
                        will be replaced with elements of x.)
                        contents=contents.replace('*222*',str(y[num2]))
                        (when the loop runs, y[num2] will loop over all elements of y, and *222*
                        will be replaced with elements of y.)

                        out_file_base='data_'+str(num1)+'_'+str(num2)+'_*lll*.out'
                        contents=contents.replace('*ppp*',out_file_base.replace('*lll*','python'))
                        vmap_file.write(str(vnum)+'\t'+str(num1)+'\t'+str(num2)+'\n')
                        task_file.write('bash RUNNUMBER_'+str(vnum)+'.qsub\n')
                        fout.write(contents)
                        fout.close()
                        vnum+=1

n_cores=vnum+0
n_nodes=int(numpy.ceil(float(vnum)/partition_info[1]))

# Finally output sbatch file
contents=open('RUNNUMBER.sbatch.template','r').read()
contents=contents.replace('*nnn*',str(n_cores)) # The total number of processors
contents=contents.replace('*NNN*',str(n_nodes)) # The total number of nodes
contents=contents.replace('*ttt*',time_str) # The wall clock time per processor
contents=contents.replace('*partition*',partition_info[0]) # Partition
contents=contents.replace('*myemail*',myemail) # Email
contents=contents.replace('*project*',project_name) # Project name
open('RUNNUMBER.sbatch','w').write(contents)
```

Save and exit, if in vim using $:wq .

Next, (same as before) we will rename all the files RUNNUMBER to run2_09242022
$ initfolder RUNNUMBER run2_09242022

Next, run the builder file

$ ./run2_09242022_builder.py

Copy the python program into the folder using
$ cp ../../print_product.py ./

Submit the job to the cluster using
$ sbatch run1_09242022.sbatch

View your run results or run errors

The results of the runs are stored in the scratch directory.
$ cd (change directory to home directory)
Change directory to the run
$ cd scratch/run2_09242022
List the resulting files (optional)
$ ls -t (-t lists the files/directories chronologically)
We should see directories such as b0, b1 etc. Let us view the results in b0.
$ cd b0
List the files and open with vim or copy the files to your local computer.
$ vim [filename.txt]

In case the job fails and results in error, the errors will be written in the file error.err under the directories b0, b1 etc. You can view the errors using the following command
$ vim error.err

Serial job: Example 2

Sum the first N positive integers, where N = 10, 100, 1000, 10000.

Python program to do the job.
sum_integer.py

```
# usage sum_integer
import sys
N = float(sys.argv[1])
fout=open('sum_'+str(N)+'.txt')
my_sum = sum([i for i in range(1,N+1)])
fout.write(str(my_sum)+'\n')
```

RUNNUMBER.template

```
#!/usr/bin/bash
# Template for ".qsub" files that will be run as batch files by slurm
RUN_NAME=RUNNUMBER
PROJECT_NAME=*project*
SCRATCH_DIR=$SCRATCH/$RUN_NAME/b###
LOCAL_DIR=$HOME/$PROJECT_NAME/build
mkdir -p $SCRATCH_DIR

EXEC1 = sum_integer.py

HELPER = " "

cd $LOCAL_DIR/$RUN_NAME
cp $EXEC1 $SCRATCH_DIR/

if [ ! -z "$HELPER"] # Check that HELPER isn't empty then
cp $HELPER $SCRATCH_DIR/ fi
cd $SCRATCH_DIR/

{ time python $EXEC1 *111*; } > temp.out 2> error.err

if [ "$(pwd)" == $SCRATCH_DIR ]; then
        echo "Removing files"
        rm $EXEC1
        if [ ! -z "$HELPER"] # Check that HELPER isn't empty
        then
                rm $HELPER
        fi
fi
```

RUNUMBER_builder.py

```
#! /usr/bin/env python
import subprocess import os
import numpy
partition_info=['CMT',16]  (using CMT)
# = [partition, ncores per node]
```

```
time_str='7-00:00:00' # Format = dd-hh:mm:ss (change time string accordingly)
project_name=os.getcwd().split('/')[-3]
myemail=os.environ["MYEMAIL"]

#Log the current submission
logstr=’’’RUNNUMBER:
Codebase:
Cluster: ganymede
gittag: ‘’’

#Do the git-ing
cmd='git commit -a -m "Commit before run RUNNUMBER" > /dev/null'
print(cmd)
subprocess.call(cmd,shell=True)

cmd='gittag.py > temp.temp'
subprocess.call(cmd,shell=True)
logstr+=open('temp.temp','r').readline()
subprocess.call('rm temp.temp',shell=True)

open('RUNNUMBER.log','w').write(logstr)

#Setup the versionmap and qsub files
vmap_file=open('versionmap.dat','w')
vmap_file.write('vnum\tL\n')

task_file=open('RUNNUMBER.task','w')
template_file='RUNNUMBER.template'
template_contents=open(template_file,'r').read()

vnum=0

Ns = [10,100,1000,10000]

for L in xrange(4): (number of parameters to be sweeped over.)
        qsub_file=template_file.replace('.template','_'+str(vnum)+'.qsub')
        fout=open(qsub_file,'w')

        contents=template_contents.replace('###',str(vnum))
        contents=contents.replace('*project*',project_name)
        contents=contents.replace('*111*',str(Ns[L]))
        out_file_base = 'data_'+str(L)+'_*lll*.out'
```

```
        contents=contents.replace('*ppp*',out_file_base.replace('*lll*','python'))
        vmap_file.write(str(vnum)+'\t'+str(L)+'\n')
        task_file.write('bash RUNNUMBER_'+str(vnum)+'.qsub\n')
        fout.write(contents)
        fout.close()

        vnum+=1


n_nodes=int(numpy.ceil(float(vnum)/partition_info[1]))

# Pad to an even multiple of cores per node
n_cores=n_nodes*partition_info[1]
for j in range(vnum,n_cores):
        task_file.write('echo "Fake run"\n')

# Finally output sbatch file
contents=open('RUNNUMBER.sbatch.template','r').read()
contents=contents.replace('*nnn*',str(n_cores)) # The total number of processors
contents=contents.replace('*NNN*',str(n_nodes)) # The total number of nodes
contents=contents.replace('*ttt*',time_str) # The wall clock time per processor
contents=contents.replace('*partition*',partition_info[0]) # Partition
contents=contents.replace('*myemail*',myemail) # Email
contents=contents.replace('*project*',project_name) # Project name
open('RUNNUMBER.sbatch','w').write(contents)
```

Do the same steps as before and submit the job to the cluster.

Appendix

This is an addendum to Mike's note. I share a few tweaks to configure vim to be used by humans.

How to use vim? (INCOMPLETE)

Linux and mac come with a very basic .vimrc file. We will modify this file first which will make our coding easy. To do this, you may copy the .vimrc file from the following

```
set encoding=utf-8
set nocompatible
filetype on
filetype plugin on
filetype indent on
syntax on
set number
set cursorline
set expandtab
```