Basic tutorial for using condor at Nevis

Login to your file server

For the purposes of illustration, I'm going to assume that your file server is `olga` (a machine name that does not exist at Nevis), and that your Nevis account name is `$USER` (which is correct if you log into your file server from a Nevis system).

```
> ssh $USER@olga.nevis.columbia.edu
```

Create a directory on your file server

Disks are divided into "partitions", and directories are created within partitions. All the Nevis systems have a /data partition. Let's go there, create our test directory, and go into the new directory:

```
> cd /nevis/olga/data
> mkdir $USER
> cd $USER
```

Copy the files used in this tutorial

```
> cp ~seligman/root-class/condor-example.* $PWD
```

Look at the list of files you've copied:

```
> ls -lh
```

Look at the batch cluster

This command will show you the systems on the cluster, and how much memory has been assigned to each batch queue.

```
> condor_status
```

It's a long list, so you may want to pipe it to the `less` command:

```
> condor_status | less
```

These are the systems on which your condor processes can execute.

Look at the files

Although there are exceptions, typically condor jobs require at least three files: the condor command file, a shell script executed by the command file, and a program executed by the shell script. Take a look at these files and read the comments:

```
> less condor-example.cmd
> less condor-example.sh
> less condor-example.py
```

The command file submits the shell script to be executed on some machine in the condor pool. The shell script sets up the environment for the program to execute. The program, when it executes, writes an output file. That output file is copied by condor to the directory from which you originally executed `condor_submit`.

Make sure they're executable

For a file to be executed as a program, it must be executable. I've already made sure that `condor-example.sh` and `condor-example.py` are executable programs via the following commands, but I suggest you type them in again to both be certain and to know how to do this when you start writing your own scripts.

```
> chmod +x condor-example.sh
> chmod +x condor-example.py
```

Let's try it

Submit your condor command file to the condor cluster:

```
> condor_submit condor-example.cmd
```

Quickly (before the program has a chance to finish), type

```
> condor_q
> condor_q -run
```

The first command shows all the jobs you submitted on this computer. The second command shows the jobs you submitted which are currently executing, and on which computer.

Within a minute or so, the job will complete and there'll be no result with your account ID from `condor_q`. Take a look at the contents of your directory:

```
> ls -lrth
```

The files are listed in ascending order by date. Note the new files at the bottom of the list. Compare these files names to the ones given in `condor-example.sh`. Can you see how `condor-example-test-0.root` got its name?

Multiple jobs

Edit the file `condor-example.cmd` and change the last line to read

```
queue 10
```

This means to submit 10 jobs. Save the file and execute the `condor_submit` command again. Note how the submitted jobs are "counted off" by periods. Type `condor_q` and `condor_q –run` to see which computers execute the jobs. When they're all done, look at the contents of your directory to see all the new files.

Run ROOT and look at the contents of `condor-example-test-9.root`. Does it contain the histogram you expect? Look at the mean and the histogram limits.

Aborting a job

It happens all the time: You submit 10,000 jobs, and then realize that something is wrong. Fortunately, you can quickly abort a cluster of condor jobs.

Do `condor_submit` again. The message that comes out looks something like this:

```
> condor_submit condor-example.cmd
Submitting job(s)..........
Logging submit event(s)..........
10 job(s) submitted to cluster 14.
```

The identifier for this particular cluster is "14" (you'll almost certainly see a different number). If you want to cancel all the jobs in that cluster at once, the command is:

```
> condor_rm 14
```

If you forget the cluster ID, you can always remind yourself with `condor_q`.

Clean up

Finished? Get rid of the files you no longer need:

```
> rm condor-example-test*
```

Or if you really want to wipe a directory that you're never going to use again:

```
> cd /nevis/olga/data
> rm –rf $USER
```

Optional: A couple of tricks

At this point you're done with the basics. Here are a couple of extra tricks you can do with python to improve this process a little bit.

If you've deleted your temporary directory in /data, create it again and **`cd`** to it. Copy over these example files:

```
> cp ~seligman/root-class/root-python-setup.* $PWD
```

Take a look at `root-python-setup.cmd`. It looks pretty much the same as that other condor command file, with one big difference: instead of executing a shell script that will execute another program, this command file will execute the python program directly.

Now look at `root-python-setup.py` and look at the comments. Two new things are happening in this program:

- The python program is setting up its own environment. This requires the "stupid python trick" I mention in the comments (causing the program to run itself again). This altering of an external environment is something python can do but C++ cannot, at least not without even more trickery than you see here.
- The python program is parsing its arguments; that is, it's looking for options and arguments instead of just assuming that the first argument has a particular meaning. This can be done in C++ as well. When I'm writing code that requires only a couple of parameters, I like to use "getopt" or "argparse" methods because they help tell a user what a program is doing. Which is clearer to you?

```
> condor-example.py 5 myfile-5.root
> root-python-setup.py --mean=5 --outputfile=myfile.root
```