Working on the NewRiver Cluster

CMDA3634: Computer Science Foundations for Computational Modeling and Data Analytics
09 October 1017

NewRiver is one of the clusters offered to researchers by Virginia Tech's Advanced Research Computing facility (ARC). Some clusters are simply a stack of hundreds of the same kind of server. NewRiver is a more complicated system, comprising 173 nodes, with several classes of processor type and memory size, and in some cases, with attached graphical processing units (GPUs).

NewRiver can be thought of as a collection of smaller clusters; users can choose which subcluster to work on depending on their preferences and needs:

- Login: 2 nodes, where users can log in via ssh, transfer files with sftp or scp, submit jobs and retrieve results; **Computations should not be done on the login nodes!**
- General: 100 nodes, 2 Haswell processors, 128GB memory, for traditional HPC jobs using MPI;
- Big Data: 16 nodes, 2 Haswell processors, 512GM of memory. 43.2 TB of storage;
- Visualization: 8 nodes, 2 Haswell processors, 512GB memory, and NVIDIA K80 GPU attached, used for remote graphics rendering;
- GPU: 39 nodes, 2 Broadwell processors, 512 GB memory, and 2 NVIDIA K1200 GPUs attached;
- Interactive: 8 nodes, 2 Haswell processors, 256GB memory, and 1 NVIDIA K1200 GPU attached, accesible from web-based ETX interface;
- Very Large Memory: 2 nodes, 4 Ivybridge processors, 3 TB memory;

In a typical computation, a user starts on a local desktop machine, creating source code that is to be run on the cluster. The user must also prepare a job file, which contains instructions on where the program is to be run, and the commands that must be executed. The source code and job files are then transferred to a login node using sftp or scp. The user connects to the login node interactively with the ssh command, and submits the job file to the job scheduler, to request execution. The user's job enters a queue, to be executed when the scheduler finds the necessary resources available. When the job is completed, the results are returned to the user's directory on the login node. The results can be examined there, or transferred back to the user's home device.

1 Exercise: Login to NewRiver, Customize Your Prompt

If you gave me your PID information, then the ARC administrators should have created an account for you on the NewRiver cluster. Let's check how to access NewRiver, log into your account, and do some minor task.

Open a terminal on your laptop. (Since we will soon be talking to several different computers, I will try to indicate commands you give directly to your laptop by prefacing them with **Laptop:**). Use the**ssh** command to log into a NewRiver login node, *newriver1* or *newriver2*, with your PID:

```
Laptop: ssh PID@newriver1.arc.vt.edu
```

or, if you plan to use **emacs** or do any graphics work, include the **-X** switch:

```
Laptop: ssh -X PID@newriver1.arc.vt.edu
```

The name of the **ssh** command is short for "Secure Shell".

When prompted for your password, you need to enter password **and** a six-digit Two-Factor-Authorization code (2FA), separated by a **comma**:

```
password: PASSWORD COMMA 2FA-code
```

The two factor authorization requires you to open your DUO app and press the "key" symbol to get a numeric code. When NewRiver prompts for your password, you type your password, followed immediately by a comma, followed immediately by the six-digit code that was just supplied by DUO. You have to do this on every login, and you have to request a new code each time.

Your terminal window is now talking to NewRiver; you have been placed in a home directory on NewRiver, named **/home/PID**, where you can place files, or create directories. Commands you type now are sent to NewRiver, executed there, with results sent back to your screen. This continues until you break the connection with the **logout** command.

Note that the commands we give on the NewRiver login node will be "housekeeping" commads; creating directories, typing or editing files, submitting jobs. But the login nodes are not intended for big computations. These nodes are shared with many other users, (type "who" to see who and how many). If you start a big computation on this node, the work of other uses will be degraded, and a system administrator may send you a warning or even shut your session down.

NewRiver runs a version of the Unix operating system; you may already be familiar with the basic commands necessary to get around. One thing that is often confusing, though, is the question *"Which computer is this terminal talking to right now?"* We can try to answer this question by replacing the default prompt string on NewRiver with a customized version.

While logged into NewRiver, use **emacs** to edit (or create) the file **.bashrc**:

```
emacs .bashrc &
```

and type in the following single line

```
  PS1="\h: "
```

Save the file and quit **emacs**.

Your change to the prompt will take effect the very next time you log in. Because we are impatient, let's check right away. Typing the following command should update your prompt:

```
source .bashrc
```

and I expect your prompt will now be something like:

```
nrlogin1:
```

until you log off from NewRiver. This way, your prompt string will remind you about which computer you are talking to.

2 Exercise: Copy Example Files, Interactive GCC Compile and Run

A number of example files are available for you in my directory on NewRiver.

I suggest you make a corresponding subdirectory in your account, move there, and copy my files:

```
nrlogin1: mkdir 3634
nrlogin1: cd 3634
nrlogin1: cp ~burkardt/3634/* .      <- Don't omit the "period" at the end!
```

One file is *hello.c*. Since this is a tiny, simple file, we will go ahead and compile and execute it on the login node. But remember, in general, we should **not** use the login nodes to execute large jobs!

Type the following commands, which will clear out any old modules (explained later), pick a compiler, compile your program, and run it:

```
nrlogin1: module purge
nrlogin1: module load gcc
nrlogin1: gcc -o hello hello.c
nrlogin1: ./hello
```

3 Background: The SCP Command

Often, you will have a file on one computer but need it to be copied to another. The **scp** command, short for "Secure Copy", is one way to do this. (You might also be familiar with a similar program, **sftp**.)

It will be easiest to work with **scp** if you always run it from your laptop, assuming you have a version of Unix available there. The format of the command to copy an "original" to a "copy" is:

```
scp original_login:original_filename copy_login:copy_filename
```

This full version of the command actually lets you be on computer A, but copying a file from computer B to computer C. That's actually a rare case. If you're logged into one of the two computers involved in the transfer, then you can leave out the login information for that computer. If the copied file is to have the same name as the original, then *copy_filename* can be abbreviated to **"."**.

If you're logged in on NewRiver, it's not so easy to use the **scp** command there to copy files to your laptop. The reason is that your command needs to specify the login information and IP address of your laptop. Whereas we know the address of a NewRiver login node, we probably don't know how to specify the temporary IP address assigned to your laptop!

4 Exercise: Transfer Files with SCP

Let's use **scp** to transfer the file *hello.c* from its NewRiver location to your laptop. Assuming that on NewRiver, the file is in subdirectory 3634, your command might be:

```
Laptop: scp PID@newriver1.arc.vt.edu:3634/hello.c .
```

You can also copy files from other people's directories, if they let you. Because my file *quad.c* on NewRiver is world readable, you should also be able to copy it straight from my directory to your laptop:

```
Laptop: scp PID@newriver1.arc.vt.edu:~burkardt/3634/quad.c . <- Don't forget "period"
```

Copying files from your laptop to NewRiver is similar. In order to see this happen, it will be easier if we make a copy of *hello.c* called *goodbye.c*. Then if we transfer this file from your laptop back to NewRiver, it will stand out:

```
Laptop: cp hello.c goodbye.c
Laptop: scp goodbye.c PID@newriver1.arc.vt.edu:
```

In your **ssh** window that is connected to NewRiver, verify that there is now a new file called *goodbye.c*

5 Background: The Module Command

In order to serve a variety of users. NewRiver uses the **module** command to customize the software environment. When you log in, the default environment on NewRiver will be empty. The module command can be used to "load" a specific piece of software, that is to make it available to you. In many cases, before that software can be loaded, a specific compiler is needed, and perhaps a specific version of MPI. In more complicated situations, the software may also require that several software libraries be loaded as well.

At any time, you can ask to see what the module command has loaded for you:

```
nrlogin1: module list
```

At any time, you can return to the default "empty" module environment:

```
nrlogin1: module purge
```

At any time, you can ask what modules may now be loaded:

```
nrlogin1: module avail
```

The **module avail** command only lists software for which all the necessary support modules have already been loaded.

You can get a list of all the software modules:

```
nrlogin1: module spider
```

For any software, you can find out the versions available:

```
nrlogin1: module spider SOFTWARE
```

By specifying the version as well, you can find out the details of the specific compiler and MPI version needed:

```
nrlogin1: module spider SOFTWARE/VERSION
```

To actually load a particular module, the command is

```
nrlogin1: module load SOFTWARE
```

or, to specify a particular version:

```
nrlogin1: module load SOFTWARE/VERSION
```

In some cases, a **module load** request will fail, with possible responses including:

```
nrlogin1: module load fred
          unknown: "fred"       <- no such package
nrlogin1: module load gcc/7.0
          unknown: "gcc/7.0"    <- no such version
nrlogin1: module load openmpi
          These module(s) exist but cannot be loaded
          as requested: "openmpi" <- Need to load compiler first!
```

As long as we are just interested in compiling C programs, and possibly executing with MPI, the necessary module command is simple. Note that several packages can be included on a single module command:

```
nrlogin1: module load gcc openmpi
```

6 Exercise: Set the Environment with the Module Command

A useful Unix command is called **which**. You can put the word **which** in front of any command, and Unix will report the full name of the executable program that that command will invoke...or nothing, if the command doesn't mean anything to Unix.

Let's restore our default environment, that is, the way the computer is set up when we just logged in:

```
nrlogin1: module purge
```

Now, in our default environment, does the **mpicc** command mean anything?

```
nrlogin: which mpicc
         /usr/bin/which: no mpicc in (LONG LIST OF DIRECTORIES SEARCHED)
```

Now, for a change, let's load the Intel compiler and the Intel version of MPI, after which **mpicc** will mean something again.

```
nrlogin: module load intel
nrlogin: module load impi
nrlogin: which mpicc
         /opt/apps/intel15_3/impi/5.0/intel64/bin/mpicc <- Now "WHICH" recognizes MPICC
```

7 Exercise: Interactive MPI Compile and Run

Let's compile and run the file *hello_mpi.c*. I presume you have just done the previous exercise, so that the **mpicc** command will be recognized. Note that, once again, we are abusing the login node, because we are running a user program here. Again, though, it's really a small, quick program that will not hurt other users. But we will soon learn how not to have to do this ever again!

```
nrlogin: mpicc -o hello_mpi hello_mpi.c
nrlogin: mpirun -np 4 hello_mpi
```

The file *quad_mpi.c* is another MPI program. It approximates the integral $I = \int_0^1 \frac{1}{1+x}\,dx = \ln(2) \approx 0.6931471805$. If we use m MPI processes, then each process divides a part of the interval [0,1] into n subintervals, and uses a Riemann sum approximation. The individual estimates are combined using the MPI function: **MPI_Reduce()**. The value of m is determined by how many processes we create. The value of n is a command line parameter.

Compile the program. Compare the results using m=1 process and n=1000 intervals versus m=4 processes and n=250 intervals:

```
nrlogin: mpicc -o quad_mpi quad_mpi.c -lm
nrlogin: mpirun -np 1 quad_mpi 1000
nrlogin: mpirun -np 4 quad_mpi 250
```

8 Background: MPE and Jumpshot

It's not easy to monitor, benchmark, or debug a program running under MPI. Argonne National Laboratory supplies MPE2, the "MPI Parallel Environment", which offers tools that can sometimes help. MPE2 works by replacing the standard MPI library with an "instrumented" version, which automatically keeps track of various MPI events, such as "process 1 is sending to process 7". If the user program is compiled with the MPE library, then during the run, a data file is created containing the MPI transaction information. This data file can then be processed by other MPE2 tools to help the user to understand the patterns of communication and delay. A particularly useful tool is **jumpshot**, which plots a sort of graphical time history of the activity and communication of the processes.

MPE2 and jumpshot are installed on NewRiver. You can also install them on your own laptop system, and you may find it helpful to have these tools available locally. On NewRiver, if you compile and run an MPI program with the MPE2 environment, the data file created will have an extension of ".clog2". When you run jumpshot, and specify this data file, it will first need to convert it to a format it denotes with the extension ".slog2". After that process, you can get your picture of the communication history.

9 Exercise: Interactive MPE Run

If we want to use the **jumpshot** program to visualize our MPI execution, there are a number of issues:

- we need to load some special modules;
- we need to use the MPE2 version of the MPI compiler;
- we need to specify an extra "logging" option;
- if we plan to view the results with jumpshot on NewRiver, we must have logged in with the -X switch;
- if we plan to use jumpshot on our laptop instead, we need to copy the **.clog2* file back to our laptop.

Let's see how a run of **quad_mpi** would work under MPE2.

When in doubt, it's best to purge the old modules and start fresh. For MPE2, we can type

```
nrlogin: module purge
nrlogin: module load gcc openmpi jdk mpe2
```

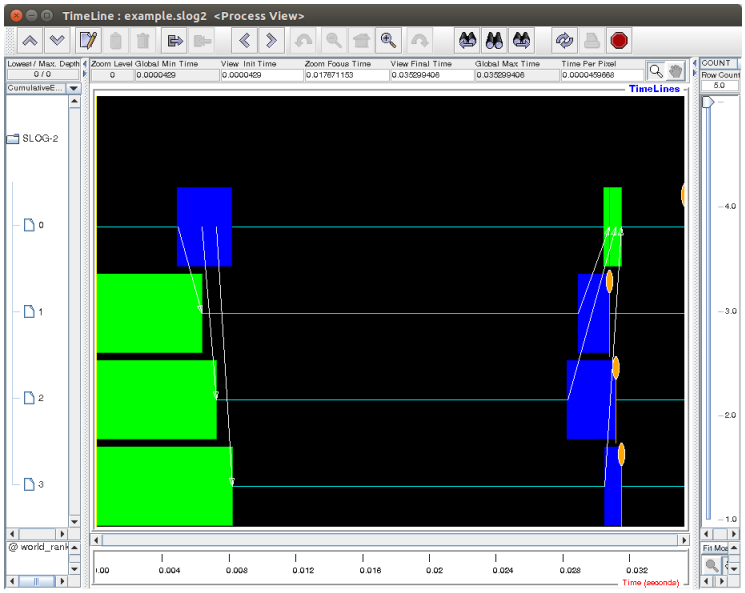

Figure 1: The Jumpshot display for the QUAD program run on 4 processes.

Instead of compiling and loading with **mpicc**, we need to use **mpecc**, which adds in some special code to monitor the parallel programs as they execute and communicate:

```
nrlogin1: mpecc -o quad_mpi quad_mpi.c -lm -mpilog
nrlogin1: mpirun -np 4 quad_mpi 250
```

After we run the program, we notice a *quad_mpi.clog2* file, which contains the timing informatation that **jumpshot** needs. To view the information on NewRiver, we then simply type

```
nrlogin1: jumpshot quad_mpi.clog2
```

To view on your laptop instead, we use **scp** to copy the file:

```
Laptop: scp PID@nrlogin1.arc.vt.edu:quad_mpi.clog2 .
```

and then run our local copy of jumpshot.

10 Background: Batch Jobs

NewRiver has 173 nodes, but generally, you only have direct, interactive access to the two login nodes, and we really shouldn't do any big computing there. Type

```
nrlogin1: who
```

right now in your NewRiver terminal and you will see many other people logged in. The NewRiver *compute nodes* are where real computational work is done, and generally, this work is done indirectly. Rather than you typing your commands interactively, you put them into a *batch file*, which also includes some information about the amount of memory and other resources you are requesting.

The batch file is sent to a queue manager, the queue manager throws your request into a pile of such files, and then, when it's your turn and the resources are available, it schedules your job to run. The output files that would have appeared on your terminal in an interactive job are instead stored in a file that is returned to you. This is how an expensive, shared resource must be most efficiently used.

Let's look at a typical batch file that could be used to compile and run the program in *quad.c*. We can call the file *quad.sh*. This file consists of three parts:

1. #! line, requesting the **bash** interpreter;

2. #PBS lines, for the scheduler, such as job size and time limits;
3. Unix commands you could have entered interactively.

Here is what the job file *quad.sh* looks like, for compiling and running *quad.c* through the batch system. You should be a little mystified by the first five or six lines, but you should recognize the rest:

```
#! /bin/bash

#PBS -l walltime=00:05:00       <-- limit of at most 5 minutes of time
#PBS -l nodes=1;ppn=1           <-- limit of one node please, and just one core
#PBS -W group_list=newriver     <-- Run on NewRiver
#PBS -q open_q                  <-- queue should be ‘‘open_q’’
#PBS -j oe                      <-- join output and error into a single file

cd $PBS_O_WORKDIR               <-- Run job where this batch file is

module purge
module load gcc

gcc -o quad quad.c -lm          <-- need -lm for math library (log, fabs)
./quad 1000                     <-- run the program
```

The really mysterious line is **cd $PBS_O_WORKDIR**. This simply says to the batch system that, before trying to run the job, it should move to the directory that contains the batch file **quad.sh**, where it will be able to find the file *quad.c*.

11 Exercise: Run A (Sequential) Batch Job

Verify that, in your NewRiver directory, you have copies of *quad.c* and *quad.sh*. Now we are ready to run the job in batch. Because it only asks for a small amount of computer resources, we should expect it to run quickly.

1. submit your job request using the command **qsub quad.sh**.
2. note the number in the system response, such as **123456.master.cluster**, (I'm pretending it's 123456!).
3. check your directory using the **ls** command.
4. check your job with the command **checkjob -v 123456**.
5. keep issuing **ls** commands until you see a file called *quad.sh.o123456*.
6. use **emacs** or **more** to display *quad.sh.o123456*. Did an estimate for the integral appear?

12 Background: Using MPI in Batch

The whole point of having a cluster is that we get to use lots of computing power.

Our previous job just used 1 core on one node. Each NewRiver node has at least 24 cores, and there are many nodes. In the batch file, the statement

```
#PBS -l nodes=?;ppn=?
```

specifies how many nodes we want and how many processors per node (cores actually). In the HPC subcluster of NewRiver, each node has 2 Haswell processors, and hence has a total of 24 cores available. A batch file asking for all the cores on such a node would include the line:

```
#PBS -l nodes=1;ppn=24
```

If more cores are needed then we have to ask for more nodes. An MPI batch file needing 100 cores would actually need 5 nodes.

```
#PBS -l nodes=5;ppn=24
```

It will now have access to 120 nodes, even if it only uses 100.

For our demonstration, we'll just ask for 4 cores in the batch file *quad_mpi.sh*, which should still enable us to see a speedup:

```
#! /bin/bash

#PBS -l walltime=00:05:00
#PBS -l nodes=1;ppn=4             <-- One node please, and just 4 cores
#PBS -W group_list=newriver
#PBS -q open_q
#PBS -j oe

cd $PBS_O_WORKDIR

module purge
module load gcc
module load openmpi           <-- Load a version of MPI.

mpicc -o quad_mpi quad_mpi.c -lm  <-- MPI-aware compiler
mpirun -np 4 ./quad_mpi 250       <-- run the program using 4 processes
                                  <-- mpiexec -n 4 ... will also work.
```

13 Exercise: Using MPI in Batch

Verify that you have copies of *quad_mpi.c* and *quad_mpi.sh*.

As before, submit your job request using the command **qsub quad_mpi.sh**. and wait for the appearance of an output file whose name might be something like *quad_mpi.sh.o123456.*.

14 Exercise: Using MPE in Batch

Suppose that you want to have **jumpshot** analyze your MPI program's execution? Then we pretty much just have to copy all the interactive commands for MPE into the corresponding batch file, submit the job, and when it is finished, use jumpshot to view the results on NewRiver or transfer the **.clog2* files to your laptop if you have a copy installed there.

The batch file *quad_mpe.sh* has the necessary commands set up. It looks like this:

```
#! /bin/bash

#PBS -l walltime=00:05:00
#PBS -l nodes=1;ppn=4
#PBS -W group_list=newriver
#PBS -q open_q
#PBS -j oe

cd $PBS_O_WORKDIR

module purge
module load gcc
module load openmpi
```

```
module load jdk
module load mpe2

mpecc -o quad_mpi quad_mpi.c -lm -mpilog
mpirun -np 4 ./quad_mpi 250
```

Submit this job, wait for it to complete, verify that the *quad_mpi.clog2* file was created. You may now run jumpshot interactively on NewRiver, or use **scp** to copy the data back to your laptop where you can use a local copy of jumpshot.

15 Background for the Future: Queues, Allocations, and Accessing GPU's

In the bach file examples we have looked at here, the batch script always included the following *queue request statement*:

```
#PBS -q open_q
```

Every job on an ARC cluster must be submitted to a specific queue; some queues are for large memory jobs, or for GPU usage, and the use of most queues requires that the user have a specific allocation that will be charged for computer time. The **open_q** provides free access to any ARC user, without requiring an allocation, and that's why it was used for these examples.

Later, to do GPU computing, you will need to access the NewRiver GPU subcluster. To do that, you will have to specify the appropriate queue, **and** the allocation that your instructor has set up for you. This will be discussed in a later presentation, but for now, you may be interested to know that this is done by replacing the queue request statement above by

```
#PBS -q p100_dev_queue
#PBS -A MATH3634
```

16 Summary

- **ssh** logs you into NewRiver;
- **scp** commands on the laptop transfer files back and forth;
- **sftp** is another file transfer program;
- **module** commands set up your environment;
- **gcc**, **mpicc** or **mpecc** compiles and loads programs;
- **mpecc** needs the -mpilog switch to do jumpshot logging;
- **mpirun -np 4** runs MPI programs on 4 processes;
- you need a batch file *batch.sh* to run programs on the compute nodes;
- request nodes and cores on the **-l nodes=?:ppn=?** line;
- **qsub** *batch.sh* submits the batch file;
- **checkjob -v 123456** checks on job 123456;
- Output comes back in a file *batch.sh.o123456*;
- *.clog2 files are created by MPE2 and can be processed by jumpshot.