Administrative Information and Advice

Dr. Christopher Marcotte — *Durham University*

Course Details

This module takes place over the course of 4 weeks (typically early November through early December), and is allocated 16 hours over 8 sessions of 2 hours each. Each session will be split between (roughly) a 1 hour lecture followed by a (roughly) 1 hour workshop, in which I will be present to discuss the exercises appropriate for the day, flanked by at least two demonstrators with experience in C and/or parallel programming.

Exercises

The bulk of your efforts in this course are likely to be the exercises, which are provisioned for each session. All exercises will be made available on the Blackboard Learn Ultra module page. The exercises will be documents with some text, and perhaps figures, interspersed with problems and/or solutions that look like this:

Exercise

This is problem text! Frequently it will feature math!

$$g(s) = \int_0^\infty f(t)e^{-st}\,\mathrm{d}t \tag{1}$$

Solution

This is solution text, referencing Equation 1! Solutions will often include code:

```
int main(void){
   return 0;
}
```

I encourage you to read and begin the exercises during the alloted class time, so that you have an opportunity to ask questions. As these are not assessed, I encourage you to work together so long as doing so doesn't hinder your learning.

Supercomputer Basics

Throughout the course, we will be using the supercomputing facilities here at Durham to complete exercises. For most of us, this will mean getting set up on Hamilton. If you are in the Astrophysics stream, then you may also have access to Cosma.

Registering for Hamilton access

Prior to the start of the course you will have recieved an email with this document, or otherwise informing you that you have been registered for Hamilton access. In case you have not, or can not access Hamilton, please register using the registration form. In this form, give the module name (PHYS51025) in the 'brief description of work' field and my name (Christopher Marcotte) in the 'Head of research group' field, so that ARC can identify you all as a group.

Logging into Hamilton

On Linux / macOS, you will have a terminal emulator with `ssh` already installed and available for use. On Windows, you may need to spend some time configuring your system. I recommend using Windows Terminal, which may be preinstalled on the lab machines.

The first command is the log in:

```
ssh USERNAME@hamilton8.dur.ac.uk
```

where `USERNAME` is your CIS username. You will then be prompted for your password. Please note: we will be using Hamilton8 for the course (Hamilton7 is old and being deprecated).

We will be logging in to Hamilton consistently often, so I encourage you to set up your `ssh` environment to make this process less tedious.

SSH Tips

Inputting your username and password every time can be tedious – unfortunately this is an archetypal example of the trade off between ease-of-use and security.

When you run `ssh`, it is reading from a configuration file located at ***`$HOME/.ssh/config`*** on your local machine. Modifying this configuration file will let us simplify the login process. You can modify this file by typing `nano $HOME/.ssh/config` at the terminal prompt. Adding a block to this file containing:

```
Host hamilton
  HostName hamilton8.dur.ac.uk
  User <CISusername>
```

where `<CISusername>` is replaced by your CIS username, will let you log in using just the command

```
ssh hamilton
```

which is, if nothing else, considerably shorter than the base command.

It is also possible to set up passwordless login by generating a public-private key pair. If you run the command

```
ssh-keygen -t rsa -b 4096 -C "<CISusername>@hamilton8.dur.ac.uk"
```

(again, with `<CISusername>` replaced with your CIS username) then you will generate a public-private key pair. You will need to supply a passphrase (*which should not be blank*) which you would then be prompted for instead of your password, on log in. Before you can log in to Hamilton with public key authentication, you need to copy the public key to Hamilton by executing

```
ssh-copy-id <CISusername>@hamilton8.dur.ac.uk
```

Now, when you log in, you'll be prompted for your *passphrase* instead of a password, and be authenticated. By setting up an `ssh-agent` on your local machine, you have have the passphrase also be remembered locally. To this end Github has advice (ignoring the Github-specific parts) and there is additional mac-specific advice here.

If this seems like a lot of mysterious work to avoid typing in a password… you are right! The ergonomics of a technology only improves if social conditions are just right.

Copying files to and from Hamilton

Hamilton does not mount any of the Durham shared drives, so you have to manually transfer any files you want. You can do this with `scp`. The basic syntax for using this program is `scp <source path> <destination path>`. As an example, consider the following scenario: On your local machine you have written a code file ***~/Downloads/myAmazingCode.c***, and you wish to copy it to Hamilton under your home directory ***~/*** in order to compile and run it. The command to do this is, from your local terminal:

```
scp ~/Downloads/myAmazingCode.c USERNAME@hamilton.dur.ac.uk:~/
```

This will securely copy your local file from your machine (source) to Hamilton (destination). Likewise, reversing the order of the arguments will copy files from Hamilton to your local machine:

```
scp USERNAME@hamilton.dur.ac.uk:~/myAmazingCode.c ~/Downloads/myAmazingCode.c
```

which will **overwrite** ***~/Downloads/myAmazingCode.c*** with the copy just downloaded from Hamilton.

Remote editing

I personally use `nano` to edit text files on remote servers – this is easy in the sense that I don't need to plan ahead. It is not, however, especially ergonomic if you are accustomed to UIs and autocomplete. If you prefer vim then you do not need to read this section.

If your personal machine is running Windows 10 or 11, and you intend to develop for this course locally, then I recommend installing the Windows Subsystem for Linux (WSL).

Versioning You may wish to develop locally and then run on Hamilton – one of the least tedious ways of accomplishing this is to use `git` versioning. That is, develop your code in a repository for this course, `git commit` your changes locally, and then `git pull` those changes while signed in on Hamilton. You will need a Github account, and I encourage you to sign up for their ***free*** student development pack to unlock otherwise unavailable features. Having an active, organized, and publicly accessible git repository is a great addition to your CV.

Compiling code

As is common with supercomputers, there are many different compiler versions available on Hamilton. These are managed with environment modules so that different Hamilton users can control which compilers and tools they use.

Often in this course we'll use the gcc compiler. To use `gcc`, we need to load a module at the Hamilton command line, or in our build script (more on that later):

```
module load gcc
```

This makes the GNU compiler tools available and loads a recent version of `gcc`. After executing these commands you can check the version of the compiler which is currently loaded with `gcc --version`, which will print something like:

```
gcc (GCC) 11.2.0
Copyright (C) 2021 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

The exercises will typically mention which modules are needed to complete the assignment. In almost all circumstances it will be `gcc`, though you may prefer `clang` or you may want to use the intel compiler `icc` occasionally, which will necessitate loading a different set of modules (`module load llvm` and `module load intel`, respectively).

Note: some modules conflict. If you try to load `gcc` and `icc`:

```
module load gcc
module load intel
```

You will be given an error which explains that the intel compiler conflicts with `gcc`:

```
intel/2021.4 conflicts with loaded module 'gcc/11.2'
```

If you only want the intel compiler, then running

```
module load intel
```

is sufficient, as it will load the last compatible `gcc` as well.

A list of available modules is available using

```
module avail
```

which will give a long detailed list of modules you can load. Once you have some modules loaded,

```
module list
```

will show you what you have loaded.

Finally, if you have found yourself mad with power and loaded too many or the wrong set of modules, then the command

```
module purge
```

will helpfully remove them all from the current working space.

Running code

The way most supercomputers work is that your computer connects to a login node, which contains your home directory, on which you may compile code and manipulate files, and then the login node disseminates your compute work to the compute nodes. You should not run significant compute workloads on the login node, since this will directly impact everyone else trying to use the cluster.

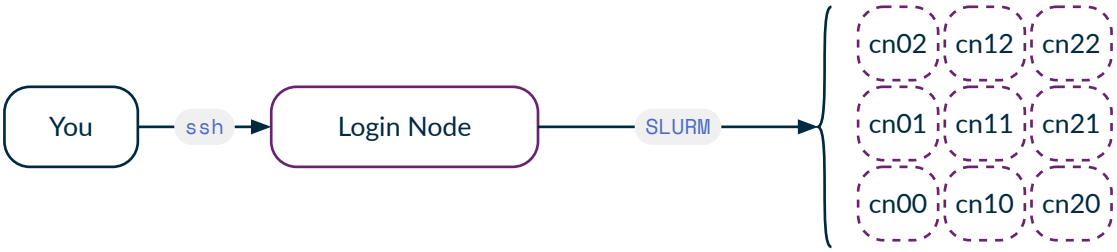

Figure 1: Your computer logs into a Hamilton login node (one of several, not depicted), which then manages your session and submits your jobs to the scheduler to run on the compute nodes.

To run code on the compute nodes we have to submit a job from the login node to the scheduler – the program responsible for allocating resources (compute node time) for all users – which on Hamilton is called SLURM. The hierarchy for this connection is (your computer, single) using ssh to connect to a (login node, several) using SLURM to distribute computations to the (compute nodes, many). A diagram of this relationship can be seen in Figure 1.

You submit a script to SLURM which details your compute resource requirements, loads modules, (optionally) compiles your code into an executable, and finally runs the executable. Below we show a basic SLURM script for a serial job:

serial.slurm

```sh
#!/bin/bash
#SBATCH --job-name="test"
#SBATCH -o myjob.%A.out
#SBATCH -e myjob.%A.err
#SBATCH -p test.q
#SBATCH -t 00:01:00
#SBATCH --nodes=1
#SBATCH --cpus-per-task=1
#SBATCH --mail-type=ALL
#SBATCH --mail-user=YOUREMAIL@durham.ac.uk

module load gcc

gcc test.c -o myExecutable

./myExecutable
```

Lines beginning with `#SBATCH` are read and interpreted by SLURM. First we name our job – it is always a good idea to choose a descriptive name. Then we define output and error files for the run – `%A` denotes the run ID, which would be filled in by SLURM. We likewise selected a particular queue on Hamilton – the testing queue, you can see available queues by running `sfree` on Hamilton at the command prompt – and set a time limit of 1 minute. We specified the number of nodes and the number of (logical) cores to devote to the task, and indicated that Hamilton should email `YOUREMAIL@durham.ac.uk` at the beginning, end, and incase of any errors (`mail-type=ALL`). If you run `man sbatch` on the login node, you will see a dizzying amount of detail for options. After the comments (lines beginning with `#`), we have actual commands to be executed on the compute node(s). First we load the relevant module, then we compile the code into an executable, then we run the executable.

Finally, we submit the script ***serial.slurm*** to the scheduler:

```
sbatch serial.slurm
```

which submits it to the queue, and will be run when there are available compute resources. You may check on the status of submitted jobs using the command

```
squeue -u $USER
```

which pulls your username from the environment variable `$USER`. If, like me, you find the best time to spot an error in your code is immediately after having submitted the job, then you can cancel a submitted job using

```
scancel jobid
```

where `jobid` is a number assigned by the scheduler and may be found by looking at the output of `squeue`.

Throughout this course you will be submitting and running a number of jobs on Hamilton. I recommend looking closely at the example jobs scripts helpfully provided by ARC.

Storage Management

The home directory (***$HOME***) is backed up but limited in size, and where you should keep code while it is being developed. The data directory (***$NOBACKUP=/nobackup/$USER***) is not backed up, per the name, but slightly more generous.

Additionally, Hamilton 8 has a directory for temporary data storage – ***$TMPDIR***, which is allocated per-node and per-job. As this is an introductory course, we are unlikely to use that storage directory.

You may run into a file number limit on Hamilton — one should verify that this is the case by running `quota` on the Hamilton login node. If it is indeed the case that you have used up your ***$HOME*** space, you may need to move (`mv`) your remote editing directory out of ***$HOME*** and into ***$NOBACKUP***, while linking (`ln -s`) back to maintain compatibility. You can perform this move and link manoeuvre by executing.

```
mv $HOME/.vscode-server $NOBACKUP/vscode-server
ln -s $NOBACKUP/vscode-server $HOME/.vscode-server
```

at the Hamilton command line. This only needs to be done once.

Aims

- Getting acquainted with the command line.
- Getting onto Hamilton using `ssh` and managing your account credentials.
- Learning about module management.
- Learning about data management.