Supercomputing Durham
Hamilton Quick Start Guide

Tobias Weinzierl

November 18, 2019

This document is written for students, colleagues, ... that want to use Durham's supercomputer facilities as fast as possible. It is a quick start guide that helps you to use the system within a few minutes. It does not replace any official documentation.

Durham's supercomputer is called *Hamilton*. There are actually multiple Hamilton generations on campus with different features. This document does not distinguish between them and in particular does not tackle particular parts of Hamilton (housing GPUs, e.g.). It just refers to the standard/default Hamilton components comprising standard Intel processors.

As a students of the courses Numerical Algorithms, Scientific Computing, GPU and cluster programming, ... you should use Hamilton whenever

- you make any statement on performance in coursework, theses or papers. Never use your own system—in particular notebook adopt their speed to their temperature, i.e. you cannot make any performance claims. If you run benchmarks on standard School workstations, others might directly or remotely log into this workstation and 'steal' some of your CPU power. Your results then are messed up.
- you want to experiment with parallel codes. You might want to use school facilities or your local computer for prototyping, but at the end of the day, all code has to work on Hamilton. Notably buggy MPI codes tend to work on standard computers (MPI there is pretty forgiving) but might not run on Hamilton anymore.
- you conduct longer experiments (of a few hours) or your experiments really grab lots of memory. If you run such codes on a standard workstation within the School, you might throttle down your colleagues which is not a very nice way (and you might have to wait pretty long; see remarks on performance above).

If you use Hamilton for theses/papers/reports, it is good style to add a brief acknowledgement. As with all supercomputers, you find on the homepage the exact wordking how to acknowledge the machine. With grants and supercomputers, you are supposed to stick exactly the their wording.

1 How to access

Access to Hamilton is free for Durham researchers and students via your university login. Go to `https://www.dur.ac.uk/cis/local/hpc/hamilton/account` and fill out the registration. As your supervisor has to approve the request and some further validation has to be done, it might require a few days before your access is granted. From hereon, I use `uvwx12` as login.

The remainder of this guideline assumes you are familiar with a Linux shell. Obviously, appropriate terminals are available in any Linux distribution. If you use Windows (newer than Windows 10), you can use the Powershell. The Powershell is also available from the AppHub in the managed desktop environment at Durham.

2 Getting your code to Hamilton

Once you have access to Hamilton, you can log into the supercomputer via ssh:

```
> ssh uvwx12@hamilton.dur.ac.uk
The authenticity of host 'hamilton.dur.ac.uk (129.234.250. ...
RSA key fingerprint is 71:36:d9:4e:31:da:1a:1a:c1:d7:31:ed: ...
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'hamilton.dur.ac.uk' (RSA) to the ...
uvwx12@hamilton.dur.ac.uk's password:
Last login: Fri Jun 26 16:18:36 2015 from 129.234.250.135
University of Durham Hamilton HPC
> ls
```

You should see that your directory there is absolutely empty. Actually, Hamilton does not share anything from your Durham home directory. It is an independent system. Exit, i.e. type in `exit`, and create a small text file `helloworld.c` with the following content:

```
#include "stdio.h"

void main() {
  printf( "Hello HPC@Durham from Hamilton" );
}
```

This is now our test application. We copy the source code over to Hamilton and then log into the system again.

```
> scp helloworld.c uvwx12@hamilton.dur.ac.uk:~/
uvwx12@hamilton.dur.ac.uk's password:
helloworld.c                          100%   83     0.1KB/s   00:00
> ssh uvwx12@hamilton.dur.ac.uk
```

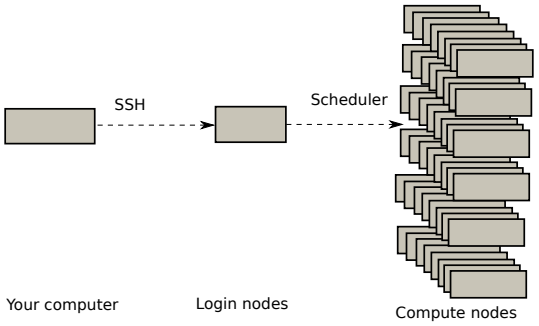
On Hamilton, you have the standard GNU compilers at hand. As it is an Intel-based system, the Intel compilers however give you way better performance. While the GCC is used with `gcc`, Intel's compiler is invoked with `icc`. However, if we type in `icc`, we get the message `icc: Command not found`. On systems such as Hamilton, there are tons of different software versions installed to please all different groups of users. By default, only a very limited set of tools (compilers, e.g.) is available, but everybody is allowed to load all the tools she needs on demand. For this, the tools are grouped into *modules*. For our purpose, we need the Intel module to translate the code with the Intel compiler:

```
> module load intel/xe_2017.2
> icc -O3 helloworld.c
> ls
a.out
> ./a.out
Hello HPC@Durham from Hamilton
```

Your code now is ready to run on the supercomputer. It actually has already executed once. However, this run is not really what you should do ...

3 Running your code

Actually, each supercomputer consists of two parts: there are *login nodes* and there are *compute nodes*. So far, we have only worked on the login nodes. A login node is a standard workstation that you can access via scp. Actually there are multiple login nodes, but you'll never see this. Whenever you do your ssh, the system automatically gives you the login nodes with the lowest number of users. The system tries to allow you to work smoothly.



If login nodes are only few and if you share them with other users, you've nothing won so far. You could have used a standard workstation at the School. It is the compute nodes you are interested in. There is a massive amount of them and you can get them exclusively. But not directly ... you have to use something we call a *scheduler*. On Hamilton, the scheduler is called Slurm.

To use a scheduler, we have to create a script that tells Hamilton everything how to call our code, i.e. to do an experiment, without any other context such as your current login configuration. So create a new file `my.slurm-script` and add the following things:

```
#!/bin/csh
#SBATCH --job-name="my-first-script"
#SBATCH -o myscript.%A.out
#SBATCH -e myscript.%A.err
#SBATCH -p test.q
#SBATCH -t 00:05:00
#SBATCH --exclusive
#SBATCH --nodes=1
#SBATCH --cpus-per-task=24
#SBATCH --mail-user=youremail@durham.ac.uk
#SBATCH --mail-type=ALL
source /etc/profile.d/modules.sh
module load intel/xe_2017.2
./a.out
```

This is a plain script file using bash[1]. The `source` statement first initialises our system before we load the Intel compiler's settings. Remember that Slurm has to be able to do all the experiments without any knowledge about your actual compile and user environment, i.e. it will start from your plain home directory. Every module you need, every file you wanna copy around, all the preparatory and clean-up steps have to be written into your Slurm script. The last line in the file then invokes the code. In this example, we do nothing with the output which is piped into a file starting with `myscript`. Slurm then adds the job number (each and every job gets a unique number) and then a `out` postfix. It is reasonable to add `%A`, i.e. the job number. We ensure this way that one Slurm run does not overwrite data from another one.

The lines in the header starting with # contain meta data. These are the lines you always need, but there are lots of additional options. In our example, we want to have one node running one MPI instance (`--cpus-per-task=24`, i.e. a single task grabs all the cores available). There are multiple queues that you can choose. `test.q` is the one for very small jobs that do not run long; the queue we use for testing. We know that our code does not run longer than five minutes—actually we allow Slurm to kill it if it lasts longer. Quite stupid for this simple example. But we want to have this one node of Hamilton exclusively

[1] If you add `#!/bin/csh` instead, you obtain a csh. All my examples here use bash. If you use the C-shell, you can omit the line starting with `source`.

(`--exclusive`) for us[2]; otherwise we still won't be able to do performance measurements as Slurm might add other users to 'our' nodes.

Save the file and type in

```
> module load slurm
> sbatch my.slurm-script
Submitted batch job 220948
```

You have now handed over the responsibility to run your experiment to Slurm. Feel free to hand in as many jobs as you want (great for lots of parameter studies)—Slurm will try to get them through as quickly as possible. To allow it to do so, it pays off to set the maximal run time as small as possible.

You also should get emails by Slurm now that notify you when a job has started or terminated, respectively. Throught the `mail-type` command, you can clarify if you want to see only some of the messages. Consult the Internet for details please.

4 Checking your experiment status

To find out when your job has finished, you can check output files written for example. Or you can again ask Slurm:

```
> squeue -u uvwx12
216690    par7.q runcaste   uvwx12  R    5:53:04    6 cn[6002-...
217105    par7.q 17_box03   rsvx72  R    2:57:46    1 cn6082
217207    par7.q 17_new_p   rsvx72  Q       0:00    1 cn6080
217229    par7.q 17_new_p   rsvx72  Q       0:00    1 cn6105
```

If you haven't logged out since your submission, you may skip the load command. You can also omit the two arguments. Then you see all the jobs currently in the system. For most users, only few columns of the table are of interest: The first one is the job number that you can use to cancel your job manually. The third one is the name of your script. Makes it easier to find it. The fourth column is the user name. Also useful sometimes. The Rs and Qs mean either Queued or Running. There's also PD (for pending), CG (completed and tidying up) and other stati. The next column tells you how long this jobs is already running. Once your job is not enlisted anymore, it has finished.

Once it has finished, you can do a simple `ls` and you should find a file with the extension `.out`. If you open it, you see what your application has actually done:

```
>less ....out
Hello HPC@Durham from Hamilton
```

That's all. That's it.

5 FAQ & Useful SLURM commands

- *I am running out of quota.* See remark below on the work directory.
- *My jobs don't run. They hang in the queue.* Try to reduce the total runtime of your job. If you don't need more than one node and your job does not last longer than one hour, use the `test.q` queue. Otherwise, use the `par7.q` queue. Do not use `seq7.q` unless you have a job that requires an enormous amount of main memory. But please check which queues are available at the moment.

[2] For proper runtime measurements, you need the exclusive flag. However, some queues (such as the test queue) might not support exclusive usage.

- *How do I find out which queues are available and what their maximum runtime is?* Type in `sinfo`.
- *Can I find out when I can expect my job to start?* You can find out when your job is supposed to run with `squeue --start -u $USER`. This is an estimated time. Your job might be started earlier if resources become available and the scheduler can squeeze in your job. To allow it to do so, ensure that you specify a job runtime and choose it as short as possible: the shorter your job the higher the probability that it fits earlier into the schedule.
- *Can I find out how much CPU time I usually burn?* Type in `sacct --format=JobID,JobName,MaxRSS,Elapsed` to get an overview over your recently completed runs. You also find details about its resource requirements. Please consult `https://slurm.schedmd.com/sstat.html` for example for available statistics.

6 GPGPU programming

Hamilton's fifth generation hosts a couple of graphics cards that you can programme via CUDA. To use them, you have to use the `gpu5.q` queue and you have to use the CUDA modules. Furthermore, please use the `gres` option to specify that you need the GPU.

```
#SBATCH -p gpu5.q
#SBATCH --gres=gpu:1
#SBATCH -N 1
#SBATCH --ntasks-per-node=1
#SBATCH -t 00:05:00
#SBATCH --exclusive

module purge
module load cuda/current
module load slurm
srun ./myfancyCUDAapplication
```

- The module environment does not set the right library path in the script. It is the `srun` command that sets the right path automatically. If it is absolutely necessary and you have to work without `srun`, please set the library path manually: `setenv LD_LIBRARY_PATH /usr/local/Cluster-Apps/cuda/5.0.35/lib64:$LD_LIBRARY_PATH`
- It currently seems to be impossible to use both GPUs at the same time. Thus, `gres` always has to be set to one card.
- Some people try to use

  ```
  #SBATCH -N 3
  #SBATCH --ntasks-per-node=2
  ```

  to access multiple GPUs. This snippet however launches six independent processes (two per node) that cannot interact with each other. It is not the classic case where you want one code to use two GPUs at the same time.

7 Further things to get used to

- The copying forth and back of source codes and files is cumbersome. At one point, it often makes sense to use some version control (CVS, SVN, git) and to have one checkout on Hamilton.

- If you do performance measurements, never run them on the login nodes. We got that. Also, don't do any IO in great quantities. Writing something with printf to the terminal or writing files is really slow on supercomputers. Restrict to the absolute mandatory things when you want to know how fast your code is.
- If you have to write (large) files, don't write to the standard directory, i.e. your Hamilton home. Write to `/ddn/data/uvwx12`. Every user has such a directory and using this one is by magnitudes faster than your home. You also are allowed to hold more stuff in there. Just ensure that you back up your stuff somewhere else. The data directory is, different to home, not backed up automatically.
- Familiarise yourself with Slurm. Google is your friend.
- Remember: Never do any larger tests or performance test on the login nodes. The measurements you get there are crab.
- Use the command `sinfo` to find out which machine is used by a queue. Afterward, consult the Hamilton webpages to see what this machine is: how many cores there are per node, e.g.
- Switch to the queue `par7.q` is you have jobs that run longer than the given time limit of the `test.q` queue. Note: Slurm might not reject a job if it exceeds the quotas. It might just mark them as invalid, i.e. mark that they harm the Quality of Service (QoS). Use `squeue` from time to time.

8 Quick jump into parallel runs

This section is required if you run parallel jobs. Otherwise ignore it.

MPI:

- If you want to use MPI, you have to load the corresponding module first. MPI is not available by default (similar to the Intel compiler). To find out which modules do exist, type in `module avail`. Then, first load your compiler and then the right MPI variant (typically `intelmpi/intel/...` for the Intel compiler). You have to do this to get access to the compile command (it is typically `mpiCC` or `mpiicc`), but you also need it to get access to `mpirun`, i.e. you have to load the module in your terminal and you have to insert a load command into your Slurm script file.
- Use `mpirun`. Do not use `mpiexec`.
- If you do MPI on your local machines, you typically have to specify how many ranks you want to get (`mpirun -np 10`, e.g.). With Slurm, you don't have to. Simply run `mpirun`. Slurm's meta data is used then to determine the right number of ranks on the right compute nodes.
- My code deadlocks if I place multiple MPI ranks on one node. I've seen this issue a couple of times and usually adding the statement `export I_MPI_FABRICS="shm:dapl"` fixes it.
- All the Slurm meta data refers to MPI only. If you use OpenMP, TBB or a hybrid code, such as both MPI and OpenMP, the parameters in the Slurm script affect only the MPI part. OpenMP at al is a different story ...

OpenMP:

- If you want to use OpenMP, you have to compile your code accordingly and then to set the environment variable: `export OMP_NUM_THREADS=24`. Remember: You have to set this variable within your Slurm script before you execute your code. Setting is simply on your terminal does not help. Slurm will ignore the settings of your terminal.
- If you use MPI plus OpenMP, the OpenMP variable `OMP_NUM_THREADS` applies to each and every MPI task. If you set it to `z` with the above example, you will book `x` nodes a 24 cores. Each node will host `y` MPI ranks. Each rank will run `z` OpenMP threads.

9 Parallel debugging

Correctness tests Debugging parallel applications is pain. I strongly recommend to use the tool MUST from time to time. MUST can find many MPI-related bugs automatically. It is straightforward to use:

- Retranslate your code with the argument `-g`. This adds debug information to your executable.
- Load MUST in your script file. Run `module avail`, search for `must`, and type in the corresponding load command.
- Run your SLURM script. Replace MPI's run command with
  `mustrun --must:nocrash --must:mpiexec mpirun -n mynumberofranks ./mycode myarguments`
- If your code crashes, you might want to omit the `nocrash` option.

Once Slurm tells you that your code has terminated, you will find a HTML file in your executable's directory that reports on all MPI-related bugs.

Debugging with DDT The preferred way to debug parallel applications on Hamilton is Allinea's DDT. Log into Hamilton with the `-Y` command, i.e. enable X-forwarding. Then, allocate your compute nodes manually and start the DDT. To do this, type in

```
# also load modules
module load ddt
salloc -N 1 -p test.q ddt
```

There's also a manual way to do this:

```
salloc -N 1 -p test.q
squeue -u $USER
```

The first command allocates you a compute node (don't forget to free it later on via `scancel`). The following queue command tells you exactly what this compute node is called. As you have reserved your node, log into this node via ssh. But don't forget to activate X-forwarding again. In the script below, I assume that I allocated `cn7001` successfully:

```
ssh -X cn7001
module load ...
module load ddt
ddt
```

After the loads, you have to reload all of your modules again. It is a different machine after all. Once this is done, we load the parallel debugger's module and we start up DDT.

The Allinea GUI is very self-explaining. For MPI jobs, select `Run`, choose your application plus its arguments, and pick the required number of MPI ranks. Through the Detail button, you can select Intel MPI.

Please note that you have to compile with `-g` to allow DDT to look up your symbols. If you use the Intel compiler, please use `-debug` on the command line when you translate as well.

10 Parallel top

On standard Linux systems, you can always type in `top` to get a first idea how well your code is behaving on the machine, i.e. whether it exploits all cores. In the SLURM context, this is obviously not that easy—once the job is submitted, it seems to be gone to the scheduler. You can however get an idea about your job nevertheless:

1. Log into Hamilton with X-forwarding, i.e. use `ssh -Y`.
2. Load the SLURM module.
3. Wait for your job to start. You can either poll the SLURM queue with `squeue -u $USER` or you can rely on SLURM's email notification.
4. Once your job is running, type in `sview`.
5. A GUI should open. Search for your job in the jobs tab and find out which nodes your job is running on.
6. Switch to `Visible Tabs` and select the `Nodes` option there.
7. Switch to the `Nodes` tab and search for your node of interest. A double click the node number opens another window with details about the job load, e.g.

11 Job farming

In many cases, it makes sense to think about combining multiple jobs into one SLURM submission. Two reasons for this exist:

1. If you are **not** interested in runtime measurements but want to get as many results as possible asap, and/or if you run a code that does **not parallelised**, then you waste CPU cores if you book a node for your job.
2. The bigger a single job, the higher its priority. So if you run 10 OpenMP jobs, it is better to grab 10 nodes with 24 cores each than to submit 10 individual SLURM jobs with one node each.

Reason 1: Open your SLURM script and enlist all your jobs as follows:

```
#!/bin/csh
#SBATCH --nodes=1
#SBATCH -p test.q
./myExecutable anArgument anArgument &
./myExecutable anArgument anArgument &
./myExecutable anArgument anArgument &
./myExecutable anArgument anArgument &
wait
```

This script books a node and issues all four program runs on this node. Do not forget the & postfix. In the present example, you exploit at least four cores instead of only one. Please note that you should not use more than 24 executables here in one script.

Reason 2: Create a SLURM script along the following lines:

```
#!/bin/csh
#SBATCH -p par7.q
#SBATCH --nodes=2
#SBATCH --cpus-per-task=12
module load slurm
echo "" > farming.script
echo "0   ./myExecutable anArgument anArgument " >> farming.script
echo "1   ./myExecutable anArgument anArgument " >> farming.script
echo "2   ./myExecutable anArgument anArgument " >> farming.script
echo "3   ./myExecutable anArgument anArgument " >> farming.script
srun --multi-prog farming.script
```

This script books two nodes and places two application on each node. To do this, it pipes instructions for all four invokations into a (temporary) file `farming.script`. This file then is scheduled through `srun`.