Introduction to HPC workshop tutorial

Useful Links

- Interactive learning
- Documentation
- Youtube

Category:
Published: March 7, 2023

Tutorial 1 - Logging into the HPC

Objective:	To log into the HPC cluster using a ssh client.
Prerequisite:	An active Apocrita account, and have provided your public ssh key

Instructions:

1. Follow the relevant instructions below for your installed operating system to launch a terminal within an ssh client:
   - Windows
   - Linux
   - MacOSX
2. Log into the HPC by executing the following command, having replaced the path to the private key with your own path, and replacing *abc123* with your QMUL username:

   ```
   ssh -i /path/to/private/key abc123@login.hpc.qmul.ac.uk
   ```

   This will attempt a login connection to the HPC cluster at address *login.hpc.qmul.ac.uk* for username *abc123*.

   Note: Your password will not be displayed on the screen as it is typed. If you type it incorrectly too many times in a short space of time, your account will be blocked for 1 hour - let us know if this happens.

   Once logged into the HPC, you should see the welcome message and a prompt similar to the following:

   ```
   [abc123@frontend8 ~]$
   ```

   Verify you see the prompt with your username.

   You are now logged onto the frontend server. All of the following exercises will be performed in this login session.

Windows Users

- On Windows 10 and above, type cmd (short for command) into the search box, and press Enter. You may need to firstly enable the OpenSSH client if SSH commands do not work by default.
- Alternatively, download a suitable SSH client, for example MobaXterm (https://mobaxterm.mobatek.net/download-home-edition.html). Download the portable edition to get started but if you have rights to install software on your machine, you

may also select the installer edition. Open the MobaXterm application and click the Start local terminal button.

Linux Users

Launch the in-built terminal application:

- Centos7: Applications > System Tools > Terminal.
- Ubuntu: Open the dash menu (Windows key) and type terminal.

MacOSX Users

- Launch the in-built terminal application by clicking the Finder icon in the dock > Go > Utilities > Terminal.

Help

- More details are available at https://docs.hpc.qmul.ac.uk/intro/login/

Tutorial 2 - Application Modules

Objectives:	To view, load, unload and list available application modules. Experiment by loading different versions of R.
Prerequisite:	An active connection to the HPC cluster.
Approx time:	5 minutes.

Instructions:

1. List all the available software packages by executing:

   ```
   module avail
   ```

   Output will be paginated if it contains more than one screenful. Use the space bar to continue to the end.

2. List the available versions of *R* by executing:

   ```
   module avail R
   ```

   You should see several versions listed, with one version designated as the default. If an application is not available as a module, you should see no output.

3. Load the default version of the *R* application into your environment by running:

   ```
   module load R
   ```

   Note: To load a specific version of an application, use the full module name e.g. `module load R/3.5.3`

4. List the currently loaded modules by running:

   ```
   module list
   ```

   The output will now show that R is loaded in your environment. Observe that some applications such as *R*, or *samtools* also load other modules as dependencies. These are also unloaded automatically during a `module unload` of the parent application.

5. Run `R --version` to confirm which version of R has been loaded. This is an in-built command within the R application, rather than a feature of modules. Other applications may also display the version number in the output from the `-h` or `--help` options.
6. With the default R version loaded, attempt to load the *R/3.5.3* module - note that a module conflict error appears. This is by design - you cannot have two versions of the same application loaded at once, and one needs to be unloaded first.
7. Let's unload the existing *R* module and then load version *R/3.5.3*:

   ```
   module unload R
   module load R/3.5.3
   ```

   Note: You may instead use the `module switch R/3.5.3` command to achieve the same outcome in one step.

   Run the `R --version` command to confirm that you are now running R version 3.5.3.
8. Unload all modules at once by executing:

   ```
   module purge
   ```
9. Confirm you have no modules loaded by running:

   ```
   module list
   ```

Note: When requesting a new version or software package, you may be asked to test a development version before it is released. These test installations will be made available through development modules.

After executing `module load use.dev`, the development modules appear in the `module avail` output and can be loaded as usual using the `module load` command.

Tutorial 3 - Batch Job Submission

Objectives:	To create a job script using a text editor.
	Submit the job to the HPC cluster.
Prerequisites:	An active HPC session.
	A text editor (i.e. *vim* or *nano*).
Approx time:	5-10 minutes.

Many Linux users decide to use `vim` as a text editor because of its efficiency to insert, delete, find and replace text only using the keyboard. On the HPC cluster, `vim` is readily available on the frontend and cluster nodes. For more information on how to use Vim, click here.

There are other editors available such as Nano, which is perhaps easier for beginners and is available via a module on the HPC cluster. To use Nano, run `module load nano` before starting this tutorial. For more information on how to use Nano, click here.

Instructions:

1. Open a text editor to create a job script. We will be requesting 1 core, 1GB RAM and 1 hour maximum runtime.
2. Add the following text and save the file as *example.sh*:

```
#!/bin/bash
#$ -cwd
#$ -j y
#$ -pe smp 1
#$ -l h_rt=1:0:0
#$ -l h_vmem=1G
#$ -m bea

# this simple job will print the hostname and time before
# sleeping for 2 minutes and printing the time again
hostname
date
sleep 2m
date
```

3. Submit the job script to the HPC cluster for processing by executing:

   `qsub example.sh`

4. Make a note of the job number (JOBID) - this is a unique identifier which can be used to check the job status and consumed resources.

5. Check the status of your queued / running jobs by executing:

   `qstat`

   If you see no output from the command, maybe your job has already completed running, or maybe exited immediately due to an error in the job script.

   Note: Running `qstat -j JOBID` for a currently running job will provide a high level of detail about job JOBID - this should be replaced with the actual job number.

6. Once the job has completed running, confirm that it is no longer visible in the output of the `qstat` command. Examine the job output file, named `example.sh.oJOBID`, replacing JOBID with the actual job number.

7. View the contents of the job completion email to find helpful information such as exit status and resources used.

8. Modify the resources in the *example.sh* script, increasing the core request to 4 and the **total** memory to 8GB. Add a job name with the `#$ -N NAME` parameter and repeat steps 3-5 and observe the change of output from `qstat`.

   Hint: The `h_vmem` memory request is per core.

   If your job does not start immediately, and is queued, you may see the position in the whole cluster queue with our `showqueue` utility script.

9. Check your list of queued jobs with `qstat` and cancel any jobs submitted in this section that you no longer require.

   Note: you can cancel a queued / running job by running `qdel JOBID`.

Tutorial 4 - Interactive Job Submission

Objective:	To run a short R script within an interactive session.
Prerequisite:	An active connection to the HPC cluster.
Approx time:	5 minutes.

An interactive job will provide you with a terminal session on a compute node with the requested resources, in which you can interactively run commands and see output. This is useful for running very quick tasks, or troubleshooting issues.

Jobs that request up to 1 hour will become eligible for the short queue and will often start running immediately. All other jobs may request up to 10 days runtime with the `h_rt=240:0:0` parameter.

Jobs are killed automatically when the runtime has been exceeded. Therefore, we recommend setting the runtime to the maximum of 10 days for any jobs that will exceed 1 hour. For example, if your job will take 24 hours, you should still request 10 days to cover any unexpected overheads, such as storage bottlenecks.

Instructions:

1. Request an interactive session on the HPC cluster by executing:

   ```
   qlogin
   ```

   Note: By default, all jobs request 1 core, 1GB RAM and 1 hour unless otherwise specified. For example, `qlogin -pe smp 2 -l h_vmem=2G` will request 2 cores and 2GB per core, while keeping the default 1 hour maximum runtime.

2. If there are available resources, the scheduler will provide an interactive session running on a compute node. Notice the change of prompt and the node the job is executing on, which will no longer be a frontend node. Type `qstat` and find your running session in the list of your jobs, named QLOGIN.

3. Load an *R* module and run

   ```
   Rscript /data/teaching/workshop/maths/4/countdown.R
   ```

   to execute a trivial R script from within the interactive session. Observe the output which should show decrementing numbers from 5 to 0.

4. Use `exit` to end the interactive job session.

Notice the change of prompt which will be a frontend node. Please do not start any computational work when connected to a frontend. All jobs must be submitted via `qsub` or `qlogin`.

Tutorial 5 - Using Resources Effectively

Objective:	To inspect resource usage of various jobs with different resource requirements.
Prerequisites:	An active HPC session. A text editor (i.e. *vim* or *nano*).
Approx time:	10-15 minutes.

In this task, we will use BLAST to find zebrafish proteins that are similar to a small set of mouse proteins. We will demonstrate 2 sub-optimal jobs, and finally demonstrate a good-practice approach.

Instructions:

1. Open a text editor to create a job script.
2. We are going to request 4 cores, but run a job that only uses 1 core, as an example

of bad practice.

Add the following text and save the file as *blast_job.sh*:

```
#!/bin/bash
#$ -cwd
#$ -j y
#$ -pe smp 4
#$ -l h_rt=1:0:0
#$ -l h_vmem=1G

# Load the blast+ module
module load blast+
blastp  -query /data/teaching/workshop/maths/5/mm-first.faa \
        -db /data/teaching/workshop/maths/5/zebrafish.1.protein.faa \
        -out mm1.txt
```

Note: The trailing backslash character breaks a single line command over multiple lines, to aid readability.

3. Submit the job script to the HPC cluster for processing with `qsub blast_job.sh`.
4. Use the `qstat` command to check the status of the job. When it enters running state, make a note of which cluster node it is running on, as we will use this in the next step.
5. Run `ssh -t NODE top -Hu ${USER}` to log into the compute node running your job and observe the process table output to see that the process *blastp* is only using up to 100% CPU (1 core) and one thread, meaning the job is using fewer cores than requested, because we requested 4 cores. Press `q` to return to the prompt.

   Note: Replace NODE with the actual node your job is running on.

   If this job runs to completion, the efficiency column of the output of the `jobstats` command will likely be around 25%.
6. Add `-num_threads 8` to the end of the *blast_job.sh* script. The complete `blastp` command should look like:

```
blastp  -query /data/teaching/workshop/maths/5/mm-first.faa \
        -db /data/teaching/workshop/maths/5/zebrafish.1.protein.faa \
        -out mm8.txt \
        -num_threads 8
```

   Note: Don't forget to add a backslash character to the `-out` parameter.

   This will instruct blast to use 8 threads during execution, running on only 4 cores, resulting in a slower performance for your job as each core is overloaded. This is another example of bad practice.
7. Submit the job script to the HPC cluster and run the command to observe the process table output on the node running your job, to see that *blastp* is now running on 8 threads, even though we only requested 4 cores. Note each thread is only using around 50% cpu.

   At this point you may want to delete the running job to free up resources. Use `qdel JOBID` to kill your job, replacing JOBID which the job identifier shown by the output of the `qstat` command
8. Replace `-num_threads 8` with `-num_threads ${NSLOTS}` in the job script and re-submit the job.

We are now using the `NSLOTS` variable which is assigned at job runtime to the number of cores requested, which is the correct method when running multi-threaded applications on the HPC cluster.

9. Submit the job script to the HPC cluster and run the `top` command again, (noting that the new job may be running on a different node this time). Observe the process table output on the node running your job, to see that *blastn* is now correctly running on 4 threads.
10. When the job has completed, examine the output file `mm8.txt`.
11. Confirm the exit status of your completed job by either:

    a) substituting JOBID with your job number in the following command:

    ```
    qacct -j JOBID
    ```

    Note: you can use `grep` to only view the exit status:

    ```
    qacct -j JOBID | grep exit_status
    ```

    b) Running the `jobstats` command and finding the row corresponding to the jobid of your completed job.

Tutorial 6 - Array Job Submission

Objective:	To create and submit an array job to the HPC cluster.
Prerequisites:	An active HPC session.
	A text editor (i.e. *vim* or *nano*).
Approx time:	10 minutes.

Instructions:

1. Using a text editor, make a job script file called *array.sh* containing the following:

    ```
    #!/bin/bash
    #$ -cwd
    #$ -j y
    #$ -pe smp 1
    #$ -l h_rt=1:0:0
    #$ -l h_vmem=1G
    #$ -t 1-3

    # Sleep for 1 minute to give you a chance to run qstat
    sleep 60
    # Display the task id of the current job
    echo "This task is numbered:" ${SGE_TASK_ID}
    ```

    This simple job will submit 3 tasks, each requesting 1 core and 1GB RAM. Each task will simply sleep for 60 seconds and print the task id.
2. Submit the job script to the HPC cluster for processing with `qsub array.sh`.
3. Check the status of your running job, with `qstat`, which should display three items, each with the same job number - these are called tasks.
4. Check the contents of your job output files by executing:

    ```
    cat array.sh.oJOBID.TASKNUMBER
    ```

Note: Replace JOBID with the actual job number of the example job you submitted in step 3 and TASKNUMBER with the number of the array, in this case 1, 2 or 3.

The output of the job files should consist of two lines; the first being the task number and the second being the corresponding line from the input file.

Warning: When using an array job it is important to make sure that you don't have email notifications turned on as this can result in a very high volume of emails being sent out as a task array runs.

---

5. Create an array to rotate a set of images. Perform this initial setup to copy the images to your local directory:

   ```
   cp /data/teaching/workshop/maths/6/image*.png .
   cp /data/teaching/workshop/maths/6/list_of_images.txt .
   ```

   Check the contents of the list, with `cat list_of_images.txt` and confirm that there are 5 lines in the file with `wc -l list_of_images.txt`

6. Prepare a job script `image_array.sh` that uses the `${SGE_TASK_ID}` variable and `sed` to process each image listed in `list_of_images.txt` as a separate task. For example, task 1 will process the first image in the list only. There are 5 lines in the file, so we will specify 5 tasks in the array.

   ```
   #!/bin/bash
   #$ -cwd
   #$ -j y
   #$ -pe smp 1
   #$ -l h_rt=1:0:0
   #$ -l h_vmem=1G
   #$ -t 1-5

   # Load imagemagick to allow image rotation
   module load imagemagick

   # Assign the nth line to INPUT_FILE
   INPUT_FILE=$(sed -n "${SGE_TASK_ID}p" list_of_images.txt)
   # Add an echo line for debugging (optional)
   echo "Rotating" $INPUT_FILE
   # Rotate the image and rename the output
   convert $INPUT_FILE -rotate 90 "${INPUT_FILE%.png}"_rotated.png
   ```

   Submit this job to the cluster.

7. Check that the job produces 5 new image files with the _rotated suffix.

   Note: in the scenario that you wanted to ensure that no more than two tasks run at once, you could use `#$ -tc 2` in your script.

   Note: The `-t` parameter can accept a variety of options, including a range (1-10), a range with a set step between values (1-10:2) or up to two solitary values (6 or 6,11).

Tutorial 7 - Storage Types

Objective:	To check the types of storage available on the HPC cluster.

Prerequisites:	An active HPC session.
	A text editor (i.e. *vim* or *nano*).
Approx time:	5 minutes.

Instructions:

1. List your available storage by executing:

   ```
   qmquota
   ```

2. Make a note of the free space available and the number of files remaining for your home and scratch directories.

   Note: When a storage quota has been exceeded, you will not be able to write files to that location. A daily automated warning email will be sent until the quota is no longer exceeded.

   The most common form of storage quota exceeded is the home directory. If you are running jobs in the home directory which produce a lot of output, or require large files, you may experience failing jobs as a result of the storage quota. If this occurs, we recommend using another suitable storage location as shown in qmquota.

3. Change to your home directory and verify your current location, by executing:

   ```
   cd ${HOME}
   pwd
   ```

   Note: By using the variable `${HOME}` you can provide the entire path of your home directory without having to type the entire location. This can be used in job scripts.

4. Start an interactive session, as shown in tutorial 4, by executing:

   ```
   qlogin
   ```

5. Demonstrate writing to files in the different storage locations by executing:

   ```
   echo "This is the home directory" > ${HOME}/Location1.txt
   echo "This is the scratch space" > /data/scratch/${USER}/Location2.txt
   echo "This is the temporary work directory" > ${TMPDIR}/Location3.txt
   ```

   Make a note of your job number when the session begins.

6. Check your current location using `pwd` and display the contents of the first file, in your home directory.

7. Change to your scratch directory, located at `/data/scratch/${USER}` and check the content of the second file.

8. Change to your temporary working directory and verify the location by executing:

   ```
   cd ${TMPDIR}
   pwd
   ```

   Note: Notice how the full location path contains the job number. The `${TMPDIR}` variable can be used to access this location in a job script despite the fact that the location will change for every job.

9. Check the contents of the third file.

   Warning: The temporary work directory exists only while the job is running and is automatically deleted at the end of the job, along with its contents. If you want to keep the contents after the job completes, then make sure your job script copies it to a safe location. Usually you would use scratch for working data, but ${TMPDIR} is useful for data you want to automatically discard after the job. It is also helpful in some edge cases where data i/o is so heavy that the job requires local, non-networked disk storage.

10. Return to your home directory and exit the interactive session.

Tutorial 8 - Application Packages

Objective:	To install Python, R and Julia application packages.
Prerequisites:	An active HPC session.
	A text editor (i.e. *vim* or *nano*).
Approx time:	10 minutes.

Although this tutorial focuses on using a virtual environment for Python packages, alternative methods are also available, such as Anaconda. For more information, see our Anaconda docs here.

Python Packages

Instructions:

1. Request an interactive session with the default resources:

   ```
   qlogin
   ```

2. Load the default version of the Python module:

   ```
   module load python
   ```

3. Confirm the Python module has been loaded with `module list`.

4. Create a Python virtual environment by executing the following commmands:

   ```
   virtualenv test-env
   source test-env/bin/activate
   ```

   You can use `virtualenv` to set up your own virtual Python environment over which you have full control. This allows you to use a specific Python version and its own set of packages. Once the virtual environment is set up, you need activate it when you log in and then you can use `python`, `pip` and `easy_install`.

5. Notice the change of prompt, which prepends the virtual environment name before the original prompt, for example: `(test-env) abc123@nxv1`.

6. Install the *numpy* package inside the virtual environment using `pip`:

   ```
   pip install numpy
   ```

   If you would like to install multiple packages, it would be better to define each package into a `requirements.txt` file and instruct `pip` to install all packages in this file. For more information on using a requirements file, click here.

7. Exit out of the virtual environment by executing `deactivate` then use `exit` to end the interactive job session. Confirm you are back on the frontend server and your prompt has changed.

8. Open a text editor to create the following job script and save the file as *python.sh*:

```
#!/bin/bash
#$ -cwd
#$ -j y
#$ -pe smp 1
#$ -l h_rt=1:0:0
#$ -l h_vmem=1G

module load python

source test-env/bin/activate
python /data/teaching/workshop/maths/8/numpy-test.py
```

9. Submit the job script to the HPC cluster for processing by executing:

   ```
   qsub python.sh
   ```

   Observe the output which should show 2 two-dimensional arrays and their elementwise and matrix products.

   Note: If you see the import error "No module named numpy", you have either not installed the `numpy` package, or not loaded the virtualenv.

R Packages

Instructions:

1. Request an interactive session with the default resources:

   ```
   qlogin
   ```

2. Load the default version of the R module:

   ```
   module load R
   ```

3. Confirm the R module has been loaded with `module list`.
4. Launch the R application and install the *rbenchmark* package:

   ```
   R
   > install.packages("rbenchmark")
   ```

   Note: The `>` character in the above code block indicates the change of prompt - do not enter this character in the R application.

   Answer "yes" if you do not already have a personal library setup with the deafault install location of `~/R/x86_64-pc-linux-gnu-library/X.Y` where X and Y form the version of R you have loaded.

   Type the number of the nearest CRAN mirror and press Enter.

   You should see the package being downloaded and installed. Check that the output contains `DONE (rbenchmark)`, which indicates the package was installed successfully. To test it installed successfully, execute `library(rbenchmark)`, which should return without error. Alternatively, execute `benchmark()` to call the function, which should return with a warning that no expressions could be evaluated.

5. Exit out of the R application by executing `q()` and optionally answering `y` to save the command history into file `~/.Rhistory`.
6. Use `exit` to end the interactive job session and confirm you are back on the frontend server and your prompt has changed.
7. Open a text editor to create the following job script and save the file as *R.sh*:

```
#!/bin/bash
#$ -cwd
#$ -j y
#$ -pe smp 1
#$ -l h_rt=1:0:0
#$ -l h_vmem=1G
#$ -m bea

module load R

Rscript /data/teaching/workshop/maths/8/fibonacci.r
```

8. Submit the job script to the HPC cluster for processing by executing:

   ```
   qsub R.sh
   ```

   Observe the output which should show the first 10 numbers in the Fibonacci sequence and a benchmark result for the first 10, 15 and 20 numbers.

   Note: If you see the "there is no package called 'rbenchmark' " error, the `rbenchmark` package has not been installed correctly.

Julia Packages

1. Request an interactive session with the default resources:

   ```
   qlogin
   ```

2. Load the default version of the Julia module:

   ```
   module load julia
   ```

3. Confirm the Julia module has been loaded with `module list`.
4. Launch the Julia application, invoke the application's builtin package manager `Pkg` and create a new Julia environment called julia-test:

   Note: Pressing the right square bracket on the `julia>` prompt will invoke the application's builtin package manager `Pkg`, you will not need to press Enter after typing `]`.

   ```
   julia
   julia> ]
   (v1.0) pkg> activate julia-test
   ```

   Note: The `julia>` and `pkg>` strings indicate the change of prompt and should not be entered into the Julia application.

5. Install the `DataFrames` package and verify it has installed successfully:

   ```
   (julia-test) pkg> add DataFrames
   (julia-test) pkg> status
   ```

   Note: The *DataFrames* package may take a couple of minutes to download and install.

6. Exit out of the `Pkg` tool by pressing Backspace on an empty prompt (holding Backspace also works), exit out of the Julia application by executing `exit()` then use `exit` to end the interactive job session. Confirm you are back on the frontend server and your prompt has changed.
7. Open a text editor to create the following job script and save the file as *julia.sh*:

```
#!/bin/bash
#$ -cwd
#$ -j y
#$ -pe smp 1
#$ -l h_rt=1:0:0
#$ -l h_vmem=1G
#$ -m bea

module load julia
julia /data/teaching/workshop/maths/8/example.jl
```

8. Submit the job script to the HPC cluster for processing by executing:

   ```
   qsub julia.sh
   ```

   Observe the output which should show a 4x2 data frame with the values 1-4 in column A and strings "M" and "F" in column B.

   Note: If you see the "Package DataFrames not found in current path" error, either the *DataFrames* package has not been installed correctly or the *julia-test* environment has not been loaded.

Tutorial 9 - Compiling and running programs on Apocrita

Objective:	Compile and run C, C++ and Fortran programs from source.
Prerequisites:	An active HPC session.
	A text editor (i.e. *vim* or *nano*).
Approx time:	10 minutes.

Instructions:

1. Change into your scratch folder and copy the source files of the programs into your scratch folder.

   ```
   cd /data/scratch/${USER}
   cp -r /data/teaching/workshop/maths/compiling .
   ```

2. You should now have three files in the directory `compiling/` in your scratch folder:

   ```
   ls -lh compiling/
   ```

3. Request an interactive session on a compute node and go to the directory with the source code:

   ```
   qlogin
   hostname
   cd /data/scratch/${USER}/compiling
   ```

4. Load a module for the Intel compiler suite and see that we have the the compilers available:

   ```
   module load intel/2022.2
   which icc icpc ifort
   ```

   Note: Other compiler suites are available, as are multiple versions of each. In this tutorial we focus on the Intel compilers but further details are available on our docs page.

5. Compile the C, C++ and Fortran source codes to executable programs:

   ```
   icc hello-c.c -o hello-c
   icpc hello-cpp.cpp -o hello-cpp
   ifort hello-fortran.f90 -o hello-fortran
   ```

   We can see that we now have three new programs as well as the original source files:

   ```
   ls -l
   ```

   Note: If we want to check that our programs work correctly after we've compiled them we can run them in this interactive session (using `./hello-c` and so on). This is valuable if, for example, there's a data input stage where we want to check that the files are read correctly before the main computation starts.

6. Now we can log out of our interactive session and confirm we're on a frontend server:

   ```
   logout
   hostname
   ```

7. Back on the frontend go to the directory with the programs we've created:

   ```
   cd compiling
   ```

8. Create a job submission file `hello.sh` with the following content:

   ```
   #!/bin/sh
   #$ -cwd
   #$ -pe smp 1
   #$ -l h_vmem=0.5G
   #$ -l h_rt=1:0:0
   #$ -N hello
   #$ -o hello.out
   #$ -j y

   module load intel/2022.2
   ./hello-c
   ./hello-cpp
   ./hello-fortran
   ```

   Note: We often need the module `intel/2022.2` loaded to be able to run the programs we've created as well as to compile the source. With these simple programs we don't (try running again after a `module unload intel`), but with other programs you may see errors. We recommend that you always load the module used to compile a program before trying to run that program.

   Note: Usually we tend to recommend 1GB as the minimum RAM size, but in this case the RAM usage is very tiny, so we're using this opportunity to show that decimal values of RAM can be used if you ever need to.

9. Submit the job request:

   ```
   qsub hello.sh
   ```

   Once the job has finished (check with the `qstat` command) you'll see our programs' output in the file `hello.out` - note that we don't have the job id suffix this time, because we manually specified an output file name.

   ```
   cat hello.out
   ```