A short manual to run simulations on the High Performance Cluster (HPC)

Bram Kuijper

address
Theoretical Biology Group, Centre for Ecological and Evolutionary Studies (CEES), University of Groningen, P.O. Box 14, NL-9750 AA Haren, the Netherlands, a.l.w.kuijper@rug.nl
errors, mistakes, additions?
This manual is far from perfect, so e-mail me any suggestions for improvement.

1 What is the high performance cluster?

The University's Center for Information Technology (CIT) has a number of computing clusters available to us, as you can see on their website: http://www.rug.nl/cit/hpcv/index. Each cluster is basically a large collection of computers that are meant to exclusively run simulations or do other forms of number crunching. We theobios are allowed to use their 200-core Linux cluster: http://www.rug.nl/cit/hpcv/faciliteiten/HPCCluster. This manual will briefly explain to you how to run programs on that cluster.

2 Brief description of cluster use

In order to be prepared, you'll have to perform the following steps to run something on the hpc-cluster:

1. copy the appropriate files from your computer to your home directory on the hpc, using a scp transfer program such as WinSCP;
2. log in with a ssh shell program such as putty to your home directory on the hpc computer;
3. recompile your simulations on the cluster computer, using a command-line C++ compiler such as g++ or pgCC;
4. when your source code contains errors, make edits using a console editor such as nano or vim or by making edits on your own computer and copying the edited file to the cluster;
5. write a file that submits jobs to the cluster using either of the previously mentioned editing possibilities;
6. submit the jobs to the computer using the command-line tool qsub;
7. check after a while if the jobs is done using the command-line tool qstat.

3 Requirements

1 An account on the cluster.
Request a HPC account by sending an e-mail to Kees Visser: kees.visser at rug.nl. Explain him briefly why you need a cluster account, say that you are from the Theoretical Biology group and ask him kindly to provide you an account. You'll subsequently receive an e-mail with a username and a password, which you need to change as soon as possible (see below). Let our secretary, Joke, know when you have a cluster account.

2 A scp transfer program.
On Windows, use WinSCP, available at: http://winscp.net/eng/index.php. Or use FileZilla: http://filezilla-project.org.

3 On Mac OS X, you can make scp connections within the 'network connections' options of Finder. If using Finder does not work for you you can either use the Mac

version of FileZilla, available here: http://filezilla-project.org, or resort to command-line use of the built-in scp program (see later).

4 On Linux, use either the built-in scp through the command-line or make a scp connection in nautilus (Gnome) or dolphin (KDE), so that you have a graphical interface.

5 A ssh shell program.
You will need a ssh shell program to log onto the HPC's main computer and to recompile your program (see later).

6 On Windows, use putty, which can be found here:
http://the.earth.li/~sgtatham/putty/latest/x86/putty.exe.
If you have installed WinSCP, putty is already included somewhere (just look for it in the program's menu bar).

7 On Mac OS X as well as Linux, there is a built-in ssh shell in every console. Just fire up a console and type ssh -v in order to be sure.

4 Step 1: logging into the cluster using ssh

Windows
Start up putty. Under Host Name type the following hostname: hpcibm1.service.rug.nl. Type an appropriate name in the Saved Sessions field, for example 'hpccluster' (this can be any name). Then click Save, so that this hostname is stored in putty for future use. Then press login and putty connects. When putty asks you if you want to accept a key, click yes. A black screen starts up and promts for 'login as:'. Now type the username that has been provided to you by Kees Visser (see his e-mail). Subsequently, type the provided password. If the password works, you are now logged into the cluster and you see something like the text below.

Mac OS X and Linux
Start a console and type:

```
$ ssh your_user_name@hpcibm1.service.rug.nl
```

Leave out the initial dollar sign, this is only indicating the reader the prompt used in a console window. Replace your_user_name by the user name provided to you by Kees Visser in the e-mail. Press enter. Type 'yes' when you get the question if you want to accept a key or not. Type the password provided to you in the e-mail. If this password does not work after a number of tries, contact Kees Visser again. If the password works, you'll get the following screen below:

```
Last login: Mon Nov 17 22:05:49 2008 from cc809020-a.groni1.gr.home.nl
username@hpcibm1:~$
```

Congratulations, you have just logged in to the cluster. Just to let you know, username@hpcibm1: $ is the commandline prompt and just indicates the place where you can type commands. You should not type this prompt.

Actually changing the password
Now you are at the command line of one of the HPC cluster's computers. To change your password type:

```
$ passwd
```

You'll have to type in your current password, after which the computer asks you to provide your own. Please, use a password that is a bit more difficult to crack by making use of capitals and normal letters as well as digits.
If the password is successfully changed, you can start your work.

Note
Linux is case-sensitive in everything: names, passwords, files, etc. Take care when you use a upper or lowercase character!

5 Step 2: transferring your files to the cluster

Windows: transfer using WinSCP
Open WinSCP. It should start with a dialog asking you for a hostname and your username and password. The hostname of the HPC is again: hpcibm1.service.rug.nl. The username is the username used previously to change the password, the password is your new password. Before pressing the Login button, save your settings first by pressing Save so that you don't have to retype everything the next time.

After you have logged in, you'll be asked once more to store a key. Do that. If everything went okay, you should now see the contents of your home directory on the HPC cluster, having the name /home/rugthbio/your_user_name. You should now be able to transfer files to and from the cluster, just by dragging folders and files from your own computer to the HPC cluster and back.

If however, WinSCP opens a location on the HPC cluster that is outside of your home directory, make sure to switch to somewhere within that directory so that you actually have access to a place to which you can copy things.

6 Step 3: recompile your simulations on the cluster

Recompiling your simulations is necessary, since the operating system version is different from your computer. This means that libraries and header files normally used by your program are still there, but they may differ. Recompiling is therefore mandatory.
First, I discuss the different compilers present on the HPC. Subsequently, I describe how to work with them in order to get a program that can be ran on the HPC.

What compilers can I use?
There are two C++ compilers installed on the HPC cluster, g++ and pgCC. g++ is the C++ compiler of the GNU Compiler Collection (GCC), a collection of open source compilers for various computer languages. g++ is available on almost any Linux system and recently became available on Windows as well, distributed through the MinGW-package (see below). The pgCC compiler is a proprietary compiler, sold by the Portland Group and supposedly produces much faster code than g++ (although I did not notice any difference). I advice you to start using g++ and if that works out successfully, step over to pgCC if performance is indeed different.

How to compile your source code?
First of all, forget the idea of a nice graphical interface like Borland or Visual C++ have. g++ and pgCC are just simple command line compilers. Second, these command-line compilers do not directly run the program after building, like Borland or Visual C++ can

do for you. Instead, the command line compiler now just delivers you an executable file that you can run yourself. Let's start:

6.1 Make a subfolder in the home directory on the HPC

You can do this in WinSCP and I think you are wise enough to figure out how. I show you the way I do it using the command line shell on the cluster (e.g. putty):

```
username@hpcibm1:~$ pwd
/home/rugthbio/username
username@hpcibm1:~$ mkdir simulation
username@hpcibm1:~$ cd simulation
username@hpcibm1:~$ pwd
/home/rugthbio/username/simulation
```

>From now on, I will not show the prompt username@hpcibm1: $ anymore in any listings, but I replace it by $. Do not type that $! It is just to indicate that you are working on the command line.

With the pwd command you can check in which directory you are right now, and it turns out that you are apperently in your own home directory /home/rugthbio/username. Subsequently, with the mkdir command, you create a new directory and you change to that directory using cd followed by the desired directory name. Last but not least, using pwd you see that the current working directory is your newly created folder.

6.2 Copy the source files to the subfolder on the HPC

Copying the necessary *.cpp and *.h files over is an easy task using a graphical client such as WinSCP, just drag it over to the desired folder on the HPC.

If you want to copy stuff from the Mac or Linux computer, and you have a console open on your own computer at the location of your source code, type the following:

```
$ scp my_source_code.cpp username@hpcibm1.service.rug.nl:~/simulations/.
```

The dollar sign indicates the prompt. The □ sign is a shortcut for your home directory. Now, scp will copy the file named source_code.cpp to the simulations directory within your home directory on the HPC. If you want to copy a whole folder at once, type scp -r instead of just scp

6.3 Compile your source code

If you are not logged in yet on the HPC through the ssh shell (i.e., through putty), do so (see above).
Once logged in through the ssh shell, change to the directory in which you have your simulations, using the cd command (see the intro of section 6.5 How to compile your source code?paragraph*.7). I assume that your *.cpp and *.h files are all present in the folder named simulations, within your home directory on the HPC cluster:

```
$ cd simulation
```

Now type the command to invoke the compiler. I will assume that you want to use the g++ compiler and not the pgCC compiler.

In my examples I will also assume that your program consists of one *.cpp file. If you have multiple *.cpp files, it is in most cases sufficient to just name all of them on the command line (see below). If you want to get sophisticated you can also use a program like make to make sure only the compilation steps absolutely required are done. For simple examples of how to use make see:
http://www.gnu.org/software/make/manual/html_node/index.html, especially chapter 2.

Given our previously mentioned assumptions we compile our program by typing (add more .cpp files after my_code.cpp if required):

```
$ g++ -Wall -o the_program my_code.cpp -lstdc++
```

This command compiles the source file named my_code.cpp into the program name the_program. The option -o tells the compiler how to name the program it is going to produce. Use -l to add system libraries to the compilation (in most cases the C++ standard library stdc++ will be sufficient). Note that I refrain from using spaces. You can use spaces on Linux however, although not the normal way. Everytime you use a space in a name, escape it by using a backslash, for example: my\ code.cpp. I also do not type the_program.exe. You can still do that if you want but it is not necessary under Linux.

If you don't see any errors, it means that g++ was able to compile your source without any problem. After the compiler is finished without any error, you should find a program file the_program in the current directory. Check for this using the ls command, with which you can list directory contents:

```
$ ls
my_code.cpp  myheader.h  random.h  the_program
```

and indeed, the last entry in the list shows that the_program is now created. If you want to see file modification times, use ls -l.

If the compiler did finish with errors, the program has not been created. Instead, you probably saw errors rolling over your screen that have now disappeared (especially if there are many errors). To debug your program, you want to have these error messages back. There are three ways to solve this problem. In most cases it is sufficient to scroll back in the terminal program you are using. If that doesn't work you can redirect the output of the compiler into a program which allows you to scroll up and down (take care to leave no spaces in 2>&1):

```
$ g++ -Wall -o the_program my_code.cpp 2>&1 | less errors.txt
```

For the technically inclined - this tells the shell to redirect (the > sign) error output (this is the 2) into standard output (the 1) and then use that as standard input for less (the |). You will end up with a screen full of error messages which you can scroll up and down using the arrow keys. To return to the shell prompt type 'q'. The third option is to store these messages in a file and open that with an editor:

```
$ g++ -Wall -o the_program my_code.cpp 2> errors.txt
```

You see that the command is almost the same as the previous one, except for the 2> errors.txt part. In this case the error output is redirected into a file instead of another program.

Now open the file errors.txt. You can use a shell editor for this, for example nano. I normally use vim, but good luck with learning that one.

```
$ nano errors.txt
```

nano can not only view but also edit text files for you, but we'll talk about that later. For now, you can scroll up and down using the cursor keys and so you can inspect the full list of compiler errors. You can ignore any warnings, these are not fatal. After you have inspected the file and know which errrors to repair in your *.cpp-file, close nano by typing Ctrl+X.

6.4 Figure out what the compilation errors mean

Each compilation error is characterized by a source code in which an error has been found an a line number that refers to the line number in the original *.cpp file. No matter how vague the error may look to you, at least looking at the line of code should be informative enough.

```
my_code.cpp: In function `int main(int, char**)':
my_code.cpp:7: error: `dafasd' undeclared (first use this function)
my_code.cpp:7: error: (Each undeclared identifier is reported only once for
  each function it appears in.)
```

Here an error has been found in line 7 of the file my_code.cpp. Note that sometimes, g++ also reports presumable errors in system files. For example, in this piece of source code:

```
#include <iostream>
#include <cstdlib>

using namespace std;

int main(int argc, char **argv)
{
      cout << random(124) << endl;
      return(1);
}
```

g++ complains with the following piece of code:

```
/usr/include/stdlib.h: In function `int main(int, char**)':
/usr/include/stdlib.h:423: error: too many arguments to function `long int random()'
```

Strange? You did not touch any file named /usr/include/stdlib.h! Indeed, you didn't. But the fact is that you wrongly called a standard function random(124) that has originally been defined in /usr/include/stdlib.h. A compiler cannot distinguish between a wrong way of calling a correctly defined function and a right way of calling an incorrectly defined function. In both occasions, g++ will think that the function is wrongly defined and refers to the source file of the function. Anyway, the proper way to call this function would be as

follows: random(). However, given that much better random number generators are at your disporsal, why work with the one from the standard library anyways?

Here is a short list of some common errors detected by g++. I can adjust the list if you provide any additional common error messages. Note again that warnings will not prevent the compilation of a normal executable file. But errors do:

old header files: Do not use iostream.h but use iostream. Do not use stdlib.h, but use cstdlib. For a list of header files which names have changed in C++, see: http://www.cplusplus.com/reference/clibrary/.

local instead of global header files. I have seen several cases where people use #include "iostream". This is incorrect. As soon as you use one of the system's header files (meaning: the one's that are not present in your own directory), you have to use #include <iostream> instead.

6.5 Repair errors in your source code

So now you have to repair the errors in your source file (i.e., the *.cpp file). I advise you to do that on your own computer and then to copy the repaired file by using WinSCP. However, you can also edit the file directly by using nano or vim on the HPC. Directly editing on the HPC works generally much faster, but is also more tough to learn. Dive into the manuals of nano or vim for that.

6.6 Compile again (and again...) until all errors are solved

Go through the previous steps repeatedly until all your errors are resolved.

7 Step 5. Run your program to test it

Now, finally has come the moment that you can test your simulation. Assuming that you are currently in the same directory as the program file, you can run your program on the HPC by executing the following command:

```
$ ./your_program
```

it should probably run now. Depending on the quality of the code, the program may also crash. The example below shows the Linux way of crashing:

```
$ ./your program
segmentation fault
```

If you encounter segmentation faults, and you have not been able to track the error by inserting assert or cout statements in the code, you should debug your program using a command-line Linux debugger such as gdb (the GNU debugger). I am not going to discuss the usage of gdb for now. Instead, I advise you to read chapter 10.3 of the Linux Programmer's Toolbox by John Fusco on the usage of gdb. I can provide you a PDF copy of that book on request.

If testing the program is successful and it does not crash, i.e., the program finishes and produces the desired output, you can think about putting the jobs on the batch job queueing system for execution by the cluster. Indeed, now we are getting almost there...

8 Step 6. Writing job files

To distribute jobs over the cluster, the HPC uses the program PBS. This program expects from you to upload a file with commands and then executes these commands somewhere on the cluster. So you have to write a file with all the desired commands in it. Here is an example of such a file:

```
#PBS -N MyImportantSimulation
#PBS -l walltime=06:00:00
#PBS -l ncpus=1
#PBS -j oe

cd simulations
./my_program
./my_program
./my_program
./my_program
```

The first part of the file is preceded by a hash character # and these are the commands that PBS needs to properly execute the desired jobs. The second part of the file (the part without the # character) are the actual commands. I will now explain this file line by line.

PBS commands

First of all, all the different options and other intricacies of the #PBS... commands can be obtained by opening the manual page of the program qsub, with which you will submit these jobfiles later on. To read this manual page, type:

```
$ man qsub
```

which results in a big page of information on the screen. You can scroll through with the cursors and if you want to search something, press a forward slash, followed by a search term: /search_term, in which search_term has to be replaced by the search term of interest. Closing the man page can be done by typing q. Now the list of command options used in the file above:

#PBS -N MyImportantSimulation
Designates the name ‘MyImportantSimulation’ to this batch file. The name can be anything, but is not trivial: later on you can only identify each batch job file by this name and not anymore by its filename. Therefore, choose a unique name.

#PBS -l ncpus=1
Allocates the number of cpus needed for your commands. If you have a multithreading job that needs multiple cpus, then increase this number. For evolutionary simulations, you probably will never use it. Using a single cpu also means that all commands in the second part of the file are only executed in a sequential fashion. Therefore, if you want to run multiple simulations simultaneously (the point of having a cluster), you should make multiple files. More on this later.
#PBS -l walltime=06:00:00
This indicates the maximum time needed in order to finish all the commands in your file (in this case 6 hours). It is necessary to put this in, so that the cluster knows in which queue to put you. If your job lasts very long, you get a specific queue so that other, shorter jobs don’t have to wait for you to finish. If your simulations exceed this maximum time, they

are aborted, so make sure this maximum time is correct. It is called walltime referring to wall clock time (i.e., the actual timescale used by humans as opposed to cpu time used by computers, which is completely different).
In fact walltime and ncpus are only two of a number of so-called resources you can specify with the -l option. See the manual page of pbs_resources for more information.

#PBS -j oe
This redirects both the normal output and the output in case of errors (think of the previously shown segfault) to a single output file. This output file is named after the name of your jobfile (but is different) and put in your home directory after the batch job is finished. Do check those files in the end, since if some runs contain segfaults or other ominous messages, there is something wrong with your simulation program.

Other commands
The second part of the jobfile above are the actual commands to run the simulations. We have already seen ./my_program, which is the command to execute my_program in its current directory. Note that PBS by default always starts in your home directory. In order to make sure PBS starts within the directory that contains your program files, you have to change directories using cd: in this case cd simulations to get to the desired directory (see above). Subsequently, you can start executing the individual programs. Using cd also means that you can traverse over different folders, just as in the old theobio cluster. However, it is now slightly different:

```
cd folder1
./my_program
cd ../folder2
./my_program
cd ../folder3
./my_program
```

note the use of the .. in front of folder2,3, etc... with which you go one folder up, to subsequently go one folder down into folder2.

Last but not least, something on using multiple cores: as far as I understand it, each file with commands is exclusively executed on a single cpu / core within the cluster. Thus, the second instance of my_program in the above command list, will only be executed after the first instance of my_program has finished. If you want to run multiple simulations simultaneously (which is the point of running things on the cluster after all), you should make multiple files, each having commands for a different core. Especially if you have many jobs, making many of these files can be tedious, so I advise you to use a scripting language such as Ruby, Python or even R code could produce such things.

9 Step 7. Submitting jobs

Now you have a jobfile, with the filename myjobfile.qsub in my home directory. Now I submit it to PBS using qsub:

```
$ qsub myjobfile.qsub
1671157.hpcibm1
```

you now see that I submitted myjobfile.qsub and that PBS notified me of its submission by printing the job's number: 1671157.hpcibm1. Now the job is running or scheduled to be run. You can check the job's status by printing:

```
$ qstat
```

but this will give you the status of all jobs at once. We can sort out our own records by using

```
$ qstat -u kuijper
1671160.hpcibm1  ...2008_10_32_54 kuijper        00:00:00 R long
1671161.hpcibm1  ...2008_10_32_54 kuijper        00:00:00 R long
1671162.hpcibm1  ...2008_10_32_54 kuijper        00:00:00 R long
1671163.hpcibm1  ...2008_10_32_54 kuijper        00:00:00 R long
1671164.hpcibm1  ...2008_10_32_54 kuijper        00:00:00 R long
1671165.hpcibm1  ...2008_10_32_54 kuijper        00:00:00 R long
1671166.hpcibm1  ...2008_10_32_54 kuijper        00:00:00 R long
1671167.hpcibm1  ...2008_10_32_54 kuijper        00:00:00 R long
```

in which you can replace kuijper with your own username on the HPC. The R's in the fifth column show that your job is running at this very moment. Other characters indicating status can be found by typing man qstat, with which you get qstat's help page.

10 Step 8. Bringing the output to your computer

You can see if your job is finished by checking with qstat -u username. If this does not list anything anymore, your jobs are finished and you can copy the results to your computer. Inspect the folder in which the job had to be executed for any output by using ls -l to list files and their sizes. If the list is too large use ls -l | less so that you obtain a browsable list. If there is no output in there, check the home directory. If there is still no output, you have a problem.

If you do have output you can copy it back to your computer in basically the same way as you did it the other way. If you simulations produce huge amounts of data it might pay though to first compress the output. You can check the amount of space a directory uses by typing du -sh directory, everything above a couple of 100 megabytes is worth compressing. Also if you have more than 100 files to copy, coping one single *.zip file back to your computer is much faster than copying all separate files, including the time of compressing and decompressing. You can use zip as a command line tool to do so. If you have a folder simulations to compress within your home directory type:

```
$ cd ~  # go to home directory
$ zip -r simulations.zip simulations
  adding: simulations/ (stored 0%)
  adding: simulations/myheader.h (stored 0%)
  adding: simulations/random.h (stored 0%)
  adding: simulations/the_program (deflated 58%)
  adding: simulations/my_code.cpp~ (deflated 18%)
  adding: simulations/my_code.cpp (deflated 18%)
```

Now you can copy simulations.zip to your computer and extract it using the conventional unzipping tools on your computer.

11 Some last tips

11.1 For beginners: try MinGW on your own computer first

If you are new to any command line stuff (i.e., a typical Windows user), a good advice is probably to install MinGW in combination with MSyS on your own computer first. These two programs install a Linux-like command line shell on Windows with a built-in gcc compiler, so you can get acquainted with the peculiarities of command line compiling. MinGW and MSyS are notorious for their incomprehensible websites, so here is the link to the download page: http://sourceforge.net/project/showfiles.php?group_id=2435. From the whole list on the download page, only download and install the 'Automated MinGW installer' and the 'MSyS base system'. I might embellish this manual with more info on those compilers later.

11.2 Use screen to get multiple ssh shells at once

Try to use screen, with this you can open multiple terminals at the same time and traverse through them so that you can work on multiple things at the same time (i.e., multitasking). Moreover, if your connection gets broken while you are working within screen, nothing gets lost. Just login again to the HPC and type screen -r to get back to where you were before the connection broke down. A short screen tutorial can be found here: http://www.kuro5hin.org/story/2004/3/9/16838/14935. screen is one of the most useful programs for working on the command-line.

11.3 Does your ssh shell lock up after mistakenly pressing Ctrl+S?

Type Ctrl+Q and it should be usable again. Ctrl+S is a relic from the old times in which people phoned in to the internet and serves as the 'pause' signal to a Linux terminal. Ctrl+Q ends this pause.

This document was translated from LATEX by HEVEA.