Introduction to Computing
V - Linux and High-Performance Computing

Jonathan Mascie-Taylor
(Slides originally by Quentin CAUDRON)

Centre for Complexity Science,
University of Warwick

Outline

main() as a Function

```c
#include <stdio.h>

int main(int argc, char * argv[])
{
  printf("%s, %s", argv[1], argv[2]);
  return 0;
}
```

`main()` as a Function

```c
#include <stdio.h>

int main(int argc, char * argv[])
{
  printf("%s, %s", argv[1], argv[2]);
  return 0;
}
```

Remember that in **C**, arrays index from 0. Notice that `argv[]` is a character array, and that we are printing `argv[1]` and `argv[2]`. This is because `argv[0]` is always reserved for the name of the program itself.

main() as a Function

```
#include <stdio.h>

int main(int argc, char * argv[])
{
  printf("%s, %s", argv[1], argv[2]);
  return 0;
}
```

Remember that in **C**, arrays index from 0. Notice that `argv[]` is a character array, and that we are printing `argv[1]` and `argv[2]`. This is because `argv[0]` is always reserved for the name of the program itself.

The array is of type `char *`, and so, we pass `%s` to `printf`. This has a downside - we can't naturally use the values as numerical values.

More Flexibility in Parameters

Modify your code so that `main(int argc, char * argv[])` takes arguments.

- Standard Map : change your value of K to
  `double K = atof(argv[1]);`
- Julia Set : change your value of $\mathrm{Im}(c)$ to
  `double cIm = atof(argv[1]);`

Because the argument is a string, we need to use `atof(myString)` (alphanumeric to float) to convert user input into something we can store in a float or double.

Calling the Program with Arguments

Windows Users :

- Open a command prompt (Start → "cmd")
- Browse to your directory (using `cd` and `dir`)
- `myprogram.exe 1.265`

Mac / Linux Users :

- Open a terminal
- Browse to your directory (using `cd` and `ls`)
- `./myprogram 0.28`

Recursion

We can also create functions which repeatedly call themselves. This is called recursion.

An example of this is Euclid's algorithm for the greatest common divisor of two numbers. Eculid's Algorithm is defined as

- $\gcd(a, 0) = a$
- $\gcd(a, b) = \gcd(b, a \bmod b)$.

Exercise: Code up a version of Euclid's algorithm that works from the command line. (For example, `gcd.exe 10 5` should print 5.)

Overview of Warwick HPC

Warwick have several high-performance computing and distributed computing facilities which you can access.

- CSC Linux Desktop
- Cluster of Workstations
- Apocrita
- Minerva

Minerva

396 nodes :

- 2× hexa-core 2.66 GHz
- 24 GB RAM

Total cores: 4752

12 GPU nodes :

- 2× NVIDIA Tesla M2050
- 48 GB RAM

Minerva has restricted access and is fairly expensive for the department. To access you need to apply via the CSC website.

Apocrita

150 nodes :

- 12 cores
- 24 GB RAM

Total cores: 1800

Apocrita is run by Queen Mary, University of London and like Minerva has restricted access.

Cluster of Workstations

Approximately 1000 cores on various computers in Physics, CSC and Complexity.

The CoW's performance per core isn't much greater than that of a decent laptop, but you can submit to many cores, and leave jobs running for a long time.

Cluster of Workstations

Approximately 1000 cores on various computers in Physics, CSC and Complexity.

The CoW's performance per core isn't much greater than that of a decent laptop, but you can submit to many cores, and leave jobs running for a long time.

The CoW is accessed via its frontend, Godzilla, which serves to access the file servers and to allow users to submit compute jobs to Torque, the queue for distribution among the CoW's computers, via a graphical desktop or a terminal.

Cluster of Workstations

Approximately 1000 cores on various computers in Physics, CSC and Complexity.

The CoW's performance per core isn't much greater than that of a decent laptop, but you can submit to many cores, and leave jobs running for a long time.

The CoW is accessed via its frontend, Godzilla, which serves to access the file servers and to allow users to submit compute jobs to Torque, the queue for distribution among the CoW's computers, via a graphical desktop or a terminal.

Godzilla is **not** for computationally-intensive jobs. It is just a front-end. You **must** submit your intensive jobs to the queue for proper processing, or you risk being banned from the CoW.

Connecting to Godzilla

Set up your SCP client to connect to `godzilla.csc.warwick.ac.uk`. This is a protocol for transferring files to Godzilla and back. Once you are connected, copy your code onto Godzilla, and then close the client.

Connecting to Godzilla

Set up your SCP client to connect to `godzilla.csc.warwick.ac.uk`. This is a protocol for transferring files to Godzilla and back. Once you are connected, copy your code onto Godzilla, and then close the client.

Using your CSC login and password, open Putty and connect to Godzilla and try locating your code. You can also use NX Client to access Godzilla.

Alternatively, use a terminal to ssh into Godzilla :

```
ssh usercode@godzilla.csc.warwick.ac.uk
```

Bash Commands

From Putty try to learn how to navigate around your directory.

- `ls` - list. Shows you all files in the current directory
- `cd` - change directory. Use `cd hello` to get into a directory called "hello". If you want to go back up a level, try `cd ..`
- These can be combined : `cd ../../hello`
- `cp` - copy a file. Takes two arguments : `cp fromhere tohere`
- `mv` - move a file. Takes two arguments, and can be used to rename files : `mv oldname newname`

Other Commands

- `man <command>` - bring up the manual on a certain command
- `mkdir <dirname>` - create a directory
- `rmdir` - remove directory - see also `rm` for files
- `cat` - quick way to view the contents of a file

You can use a graphical text editor such as `gedit` or `kwrite`. For editing directly in the terminal, try `pico` or `nano`. Great for quick edits, you can use the shortcut to close the file once you're done - `CTRL + x`. For more functionality, pick a camp : `vim` or `emacs`.

Compiling under Linux

The compiler we will use is the GNU C Compiler, or gcc. You can call it from the terminal to compile like this :

```
gcc mycode1.c mycode2.c ...  -o myoutput
```

Compiling under Linux

The compiler we will use is the GNU C Compiler, or gcc. You can call it from the terminal to compile like this :

```
gcc mycode1.c mycode2.c ...  -o myoutput
```

This will generate a file called myoutput, which the system will be able to run, from the source code in mycode.c. If you have included `math.h`, you'll need to add `-lm` to the end to link the maths library.

Compiling under Linux

The compiler we will use is the GNU C Compiler, or gcc. You can call it from the terminal to compile like this :

```
gcc mycode1.c mycode2.c ...  -o myoutput
```

This will generate a file called myoutput, which the system will be able to run, from the source code in mycode.c. If you have included `math.h`, you'll need to add `-lm` to the end to link the maths library.

Executable files are run like this :

```
./myoutput
```

... but not directly on Godzilla !

The dot in the above means "this directory", so `./myoutput` is "in this directory, run `myoutput`".

Torque

```
#!/bin/bash
#PBS -l nodes=1:ppn=1,pvmem=50mb,walltime=00:10:00
#PBS -V
cd $PBS_O_WORKDIR

jobid=`echo $PBS_JOBID | awk -F. '{print $1}'`

./dla 200 300 results.$jobid$.csv
```

This script will run `myoutput` on the CoW. To submit the script, you need to send it to the queue : `qsub -q taskfarm myscript`.

In order for it to be run, you may need to allow the script to be executed : `chmod +x myscript`.

nice

Complexity has about 20 desktop computers you can also use. E.g.

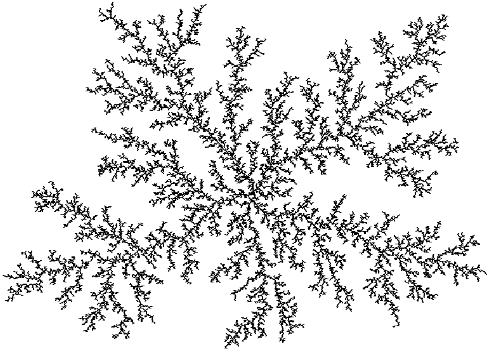
- `adobo.complexity.warwick.ac.uk`
- `bulalo.complexity.warwick.ac.uk`
- `caldereta.complexity.warwick.ac.uk`

These can be used for computation - but only with `nice`.

E.g. `nice -n 10 command-name`

Diffusion-Limited Aggregation

"Diffusion-Limited Aggregation is the process whereby particles undergoing a random walk due to Brownian motion cluster together to form aggregates of such particles."



The DLA Process

In order to build up the Brownian trees that occur as a result of diffusion-limited aggregation, one can code up a DLA process on a lattice. You begin this by creating one non-diffusing particle in the middle of the lattice.

The DLA Process

In order to build up the Brownian trees that occur as a result of diffusion-limited aggregation, one can code up a DLA process on a lattice. You begin this by creating one non-diffusing particle in the middle of the lattice.

You then randomly add a particle which diffuses along the lattice until it coalesces with the existing particle mass, at which point, it stops and we create another new particle.

The DLA Process

In order to build up the Brownian trees that occur as a result of diffusion-limited aggregation, one can code up a DLA process on a lattice. You begin this by creating one non-diffusing particle in the middle of the lattice.

You then randomly add a particle which diffuses along the lattice until it coalesces with the existing particle mass, at which point, it stops and we create another new particle.

The continuation of this process generates a branching fractal structure, with dimension ~ 1.71.

Computational Tips

Instead of looking at the neighbourhood of each particle, to see if it is near the main aggregate, it may be easier to consider a three-state lattice : empty, aggregate and "sticky patch". The latter is equal to the aggregate plus its nearest neighbours.

Computational Tips

Instead of looking at the neighbourhood of each particle, to see if it is near the main aggregate, it may be easier to consider a three-state lattice : empty, aggregate and “sticky patch”. The latter is equal to the aggregate plus its nearest neighbours.

Thus, consider a lattice site to be empty if its state is 0, aggregate if 1 and sticky if 2. Then, when your particle performs its random walk, check if it is on a site of state 2. If so, it doesn’t move - you change the neighbouring lattice sites to 2 and the central position to 1.

Computational Tips

Instead of looking at the neighbourhood of each particle, to see if it is near the main aggregate, it may be easier to consider a three-state lattice : empty, aggregate and "sticky patch". The latter is equal to the aggregate plus its nearest neighbours.

Thus, consider a lattice site to be empty if its state is 0, aggregate if 1 and sticky if 2. Then, when your particle performs its random walk, check if it is on a site of state 2. If so, it doesn't move - you change the neighbouring lattice sites to 2 and the central position to 1.

When you view your results in MATLAB, ignore 2-states (use `mod(A,2)`).

Exercise

- Draft up some code that allows you to generate random numbers - remember to `#include` the relevant headers
- Allow the user to input one argument as they call the program, using `argv` - this will be the number of particles required
- Generate an integer lattice with resolution 500×500, initialise it to zeros expect one point in the middle, where it is equal to two
- Declare two integers, `px` and `py`, the coordinates of the particle
- Loop over the number of particles `argv[1]`, using `atoi()` to interpret it as a number
- For each particle, give it a random initial condition on the lattice
- While the particle is not on a lattice site equal to 2, perform a random jump into one of its eight neighbouring sites, and check that it hasn't escaped the lattice (use periodic boundaries)
- Once it's on a sticky boundary, set this part of the lattice to 1 and its neighbours to 2
- After looping over all particles, print the resulting lattice to file

Submit to the CoW

Copy your code for generating a DLA cluster onto the CoW. Then write a Torque submission script with 20 MB of memory and one minute of walltime, then submit your job to the queue.

Submit to the CoW

Copy your code for generating a DLA cluster onto the CoW. Then write a Torque submission script with 20 MB of memory and one minute of walltime, then submit your job to the queue.

```
qsub -q taskfarm mytorquescript.pbs
```

Note that we're specifying *which* queue we want to submit to.

Once submitted, you can check on progress using `qstat -u[your login code]`.

Submit to the CoW

Each job has a number, visible when you call `qstat`.

Once finished, you will have some new files in the directory. There will be `mytorquescript.pbs.e12345` and similarly, a `.o12345` file. The `.o` file is output that should have reached the terminal. Because you aren't running the code in a terminal, output is redirected to this file. The `.e` file is an error file, and should hopefully be empty. If not, it might contains clues as to why an error occurred.

Once your job has finished running, you should have a results file in addition to the `.o` and `.e` files. Use your SCP client to copy it back to your laptop, import it in Matlab and visualise your DLA cluster.

Batch Submission

One of the advantages of the CoW is access to a large number of cores. If you had to run a simulation multiple times (say, to calculate an average), you could submit a batch job to the CoW, and whenever a processor was free, it was start one of your simulations.

Batch Submission

One of the advantages of the CoW is access to a large number of cores. If you had to run a simulation multiple times (say, to calculate an average), you could submit a batch job to the CoW, and whenever a processor was free, it was start one of your simulations.

To run a batch job, we add the line
`#PBS -t 1-20`
near the top of the Torque script (generally, underneath `#PBS -V` though this isn't necessary).

Batch Submission

Since we are using random numbers we need to make sure each node seeds with a different value - since all the nodes will probably be starting at the same time `time(0)` will mean all the nodes use the same random numbers.

To get around this we use the machine random number generator - we need to add this line to our PBS file:

```
seed=‘od /dev/urandom --read-bytes=4 -tu | awk ’{print
$2}’‘
```

- Edit your Torque script to request a batch job, with twenty simulations, feeding it the seed generated above in the forth argument.
- Submit your script, and wait for the results to roll in.

DLA Density

As a further exercise, use Matlab to import all of your DLA cluster results and visualise the average.

```
L = dir('results*');
```

This returns all of the files whose name begins with `results`. Windows users, you can also use `L = ls('results*')`.

Declare an array of the right size :

```
A = zeros(500);
```

Loop over the length of `L`, loading each file and viewing it :

```
B = load(L(i).name);
A = A + mod(B,2);
```

Then, see what the average cluster looks like.

```
imagesc(A);
```

Final Words

Good luck!