|epcc|

MSc in High Performance Computing

MSc in High Performance Computing with Data Science

Programming Skills

Coursework

Adrian Jackson, Mark Bull, David Henty, Alan Simpson and Mike Jackson

28 August 2019

Contents

1 Introduction

In this coursework you are given a program that implements a scientific algorithm. You will refactor the program, write tests for it and analyse its performance. The aim is to use your programming skills, tools and techniques, throughout:

- Use version control.
- Write an automated build script to build and run the program.
- Write documentation on how users can build and run the program.
- Refactor the program to improve both its structure and its readability.
- Write unit or regression tests using a unit test framework.
- Design and run a performance experiment to assess the performance of the program.

The coursework consists of two assessments, one focused on refactoring and one focused on testing and performance.

The coursework must be done individually.

1.1 C or Fortran

There are two versions of the program available, one in C and one in Fortran. These implement the same algorithm and are, as far as the syntax of each language permits, identical. **You only need to work with one version for your coursework.**

If you are familiar with neither of these languages, then it is **strongly recommended** you choose the C version. It is expected that you may not be familiar with C. A valuable programming skill is being able to pick up and work with code that is written in a language you have not encountered before, learning about the language as you go. A summary of the key aspects of C syntax and how to compile and run C programs is provided online at the same location as this handout. The course organiser and lab demonstrators will be happy to answer any questions you have on the use of C.

The Fortran version is provided for those students who know Fortran already or who are learning this as part of EPCC's MSc programmes in High Performance Computing and Data Science. There are many resources online on Fortran syntax and how to compile and run Fortran programs.

1.2 Coursework files on LEARN

To access the LEARN area for this coursework:

- Log-in to LEARN at http://www.learn.ed.ac.uk OR via http://www.myed.ed.ac.uk, using your EASE username and password.
- Click My Courses.
- Click Programming Skills (2019-2020)[SV1-SEM1].
- Click Assessment.
- See the Coursework Materials section.

To download a file, right-click on the associated link and select Save As...

1.3 Questions about the coursework

Please e-mail the course organiser if you have any questions, especially those relating to C syntax, compilation and linking. Questions of interest to the whole class will be anonymised and the question, plus the answer, forwarded to the whole class.

2 Percolate

The program solves the following problem. Suppose we have a grid of squares (e.g. 5 × 5) in which a certain number of squares are “filled” (grey) and the rest are “empty” (white). The problem to be solved is whether there is a path from an empty (white) square on the top row to an empty square on the bottom row, following only empty squares. This is a bit like solving a maze. The connected squares form clusters. If a cluster touches both the top and bottom of the grid, i.e. there is a path of empty squares from top to bottom through the cluster, then we say that the cluster “percolates”. See Figure 1 for an example of a grid that has a cluster which percolates. See Figure 2 for an example of a grid with no cluster that percolates.

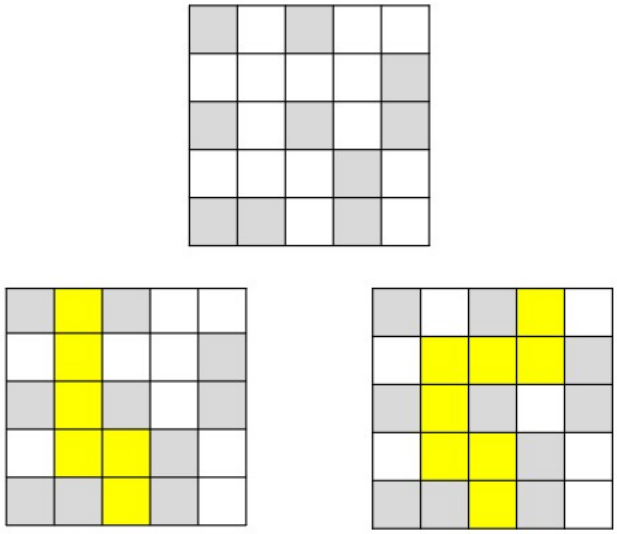

Figure 1: The grid at the top has a cluster which has paths through empty squares from the top to the bottom of the grid. Two examples of these paths are highlighted in the grids at the bottom. This cluster percolates.

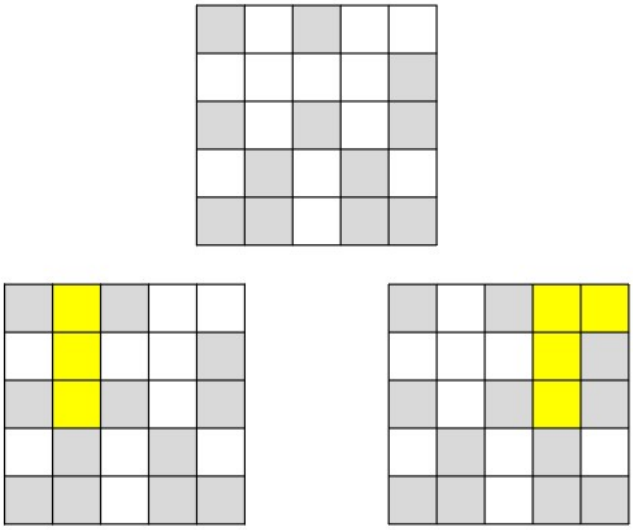

Figure 2: The grid at the top has no cluster with a path through empty squares from the top to the bottom of the grid. Two examples of the longest paths are highlighted in the grids in the bottom. This grid does not have a cluster that percolates.

To solve our problem, to see whether there is a path from an empty (white) square on the top row to an empty square on the bottom row, following only empty square, the largest clusters may be of interest.

In the grid of Figure 1 the largest cluster is 13 squares, and the second largest cluster is 2 squares. In the grid of Figure 2 the largest cluster is 9 squares and the second largest cluster is 2 squares.

Now, consider an L × L grid. What is the probability, P, of percolation if the grid is filled with a given density ρ (Greek letter, pronounced "rho") where 0.0 <= ρ <= 1.0 i.e. where the density ranges from empty to completely full. In the grid of Figure 1, L=5 and ρ=0.40 (10/25, as 10 squares are filled). In grid of Figure 2, L=5 and ρ=0.48 (12/25, as 12 squares are filled).

For a given ρ there are many possible grids so we can generate different random grids with the same density then compute the probability that a cluster percolates. The simplest cases are ρ = 0.0 (all squares are empty) as then P = 1.0 and if ρ = 1.0 (all squares are filled) then P = 0.0. But what about ρ = 0.5? For this we need to do many simulations and if, for example, we generate 100 grids and 72 of them percolate then we can assume that P = 0.72. So, what does the graph of P versus ρ look like? And, how does it depend on the grid size, L We can see this in Figure 3.

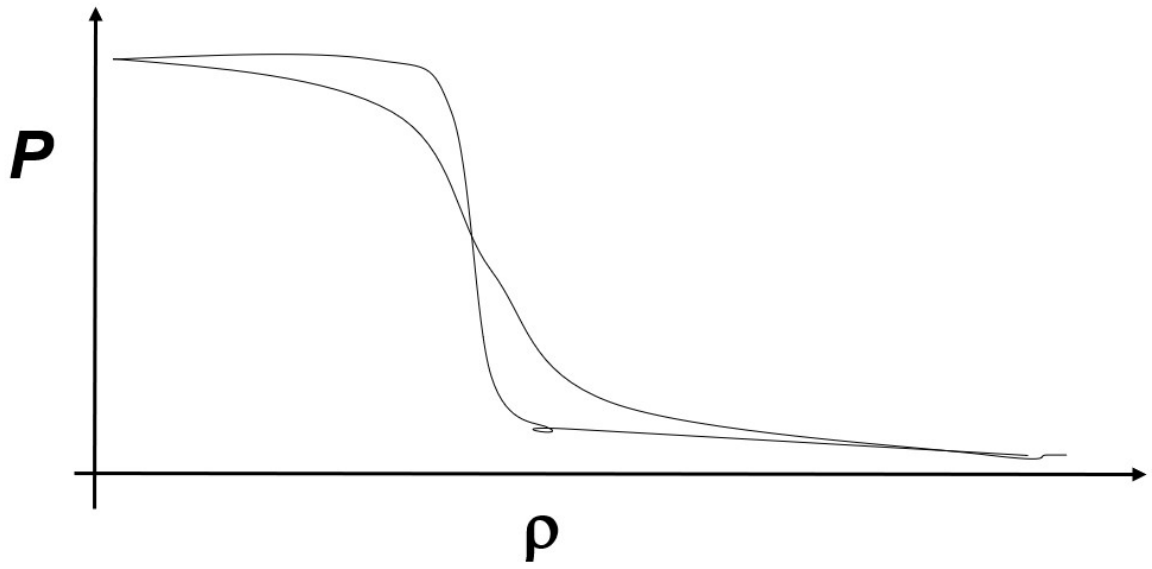

Figure 3: Two plots of density, ρ, against probability of percolation, P.

2.1 Solving Percolate

One solution to identify whether there is a cluster such that a path of empty squares from top to bottom through a cluster, that a cluster "percolates", is as follows. First, we initialise each of the empty squares with a unique positive integer (all the filled squares are set to zero) Then, we loop over all the squares in the grid many times and during each pass of the loop we replace each square with the maximum of its four neighbours. In all cases, we can ignore the filled (grey) squares. The large numbers gradually fill the gaps so that each cluster eventually contains a single, unique number. This then allows us to count and identify the clusters. If we find a part of a cluster on both the top and bottom row of the grid and it shares the same number, then we know that this is the same cluster and so we can conclude that the cluster percolates. An example is shown in Figure 4.

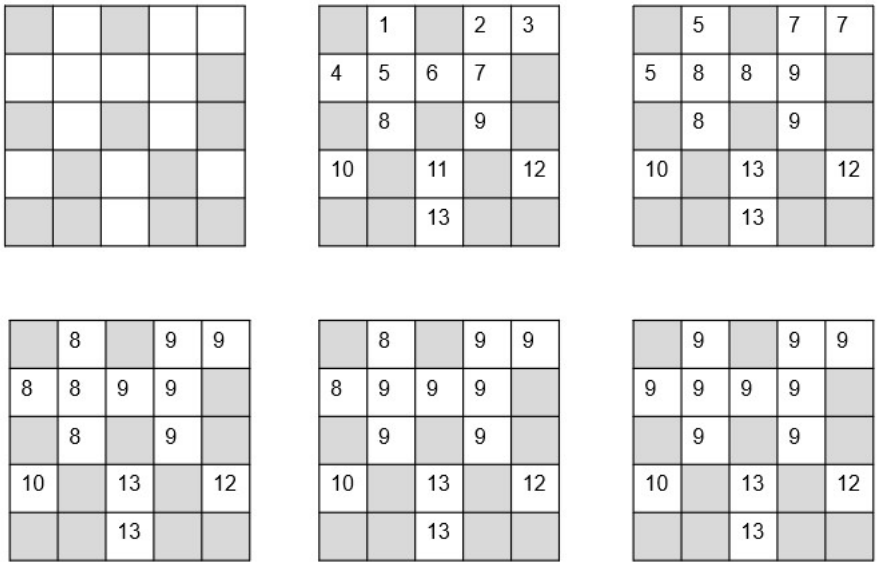

Figure 4: Solving Percolate.Top-left: our initial grid. Top-middle, grid initialised with unique numbers 1 to 13. Top-right: grid after first round of updates (8 squares are updated) Bottom-left: grid after second round of updates (5 squares are updated) Bottom-middle: grid after second round of updates (2 squares are updated) Bottom-right: grid after second round of updates (2 squares are updated) No squares would be updated on a subsequent round. All the filled (grey) squares can be assumed to be zero.

Our solution involves replacing each square with the maximum of its four neighbours, but what about squares at the edges of the grid? We don't want to write a lot of code to handle these as a special case, so a common solution to this problem is to define a "halo" around the grid. So, to solve Percolate for a L × L grid we use a (L+2) × (L+2) and set the halo squares to be filled (i.e. grey). We can do this and then set these values to zero as the zeroes will never propagate to other squares. See Figure 5.

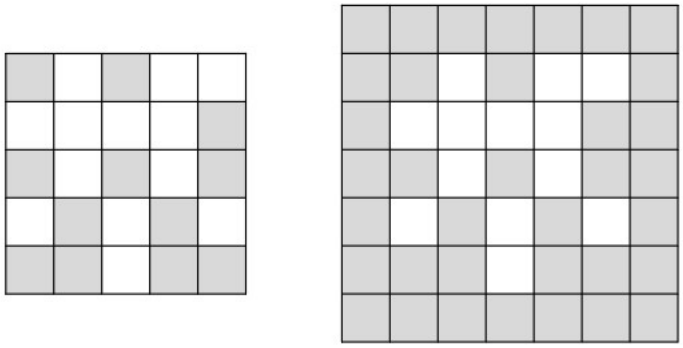

Figure 5: To process a 5x5 grid, we add a halo, with empty squares round the border of the grid, to produce a 7x7 grid.

See Figure 6 for the example of Figure 4 but with the use of a halo.

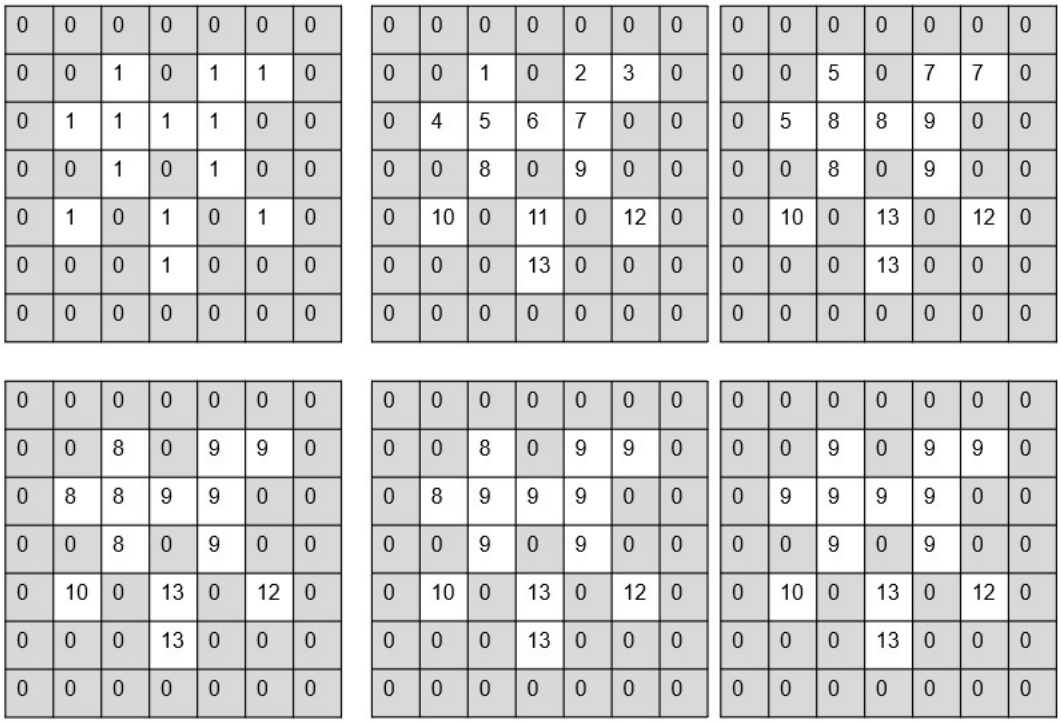

Figure 6: Solving Percolate using a halo

3 A program to Percolate

The program is available online at the same location as this handout. The program implements the above approach to solving the Percolate problem.

There are C and Fortran versions available.

The C version is in files called `coursework-percolate-c.zip` and `coursework-percolate-c.tar.gz` (both files contain the same contents).

The Fortran version is in files called `coursework-percolate-fortran.zip` and `coursework-percolate-fortran.tar.gz` (both files contain the same contents).

Download one of these files and unpack it.

3.1 Compiling the C version

Unpacking the bundle will produce a folder called `coursework-percolate-c`.

You can build the program as follows:

```
$ cd coursework-percolate-c
```

Compile the source code and assemble it but do not link it ("-c" flag) and add default debug information to compiled code ("-g" flag):

```
$ gcc -g -c arralloc.c uni.c percolate.c
```

Link the binaries into an executable file and also link the common maths library ("-l" flag for link, "m" for maths library):

```
$ gcc -g arralloc.o uni.o percolate.o -lm
```

3.2 Compiling the Fortran version

Unpacking the bundle will produce a folder called `coursework-percolate-fortran`.

You can build the program as follows:

```
$ cd coursework-percolate-fortran
```

Compile the source code and assemble it but do not link it ("-c" flag) and request that comments are not discarded ("-C" flag):

```
$ gfortran -c -C uni.f90 percolate.f90
```

Link the binaries into an executable file and request that comments are not discarded ("-C" flag):

```
$ gfortran -C uni.o percolate.o
```

3.3 Running the program

The program is run the same way for both the C and Fortran versions. Run the `a.out` executable file:

```
$ ./a.out

Parameters are rho=0.400000, L=20, seed=1564

rho = 0.400000, actual density = 0.425000

Number of changes on loop 1 is 199

Number of changes on loop 2 is 170

Number of changes on loop 3 is 149

Number of changes on loop 4 is 132

Number of changes on loop 5 is 118

Number of changes on loop 6 is 113

Number of changes on loop 7 is 78

Number of changes on loop 8 is 64

Number of changes on loop 9 is 63

Number of changes on loop 10 is 58

Number of changes on loop 11 is 54

Number of changes on loop 12 is 48

Number of changes on loop 13 is 40

Number of changes on loop 14 is 38

Number of changes on loop 15 is 30

Number of changes on loop 16 is 28

Number of changes on loop 17 is 25

Number of changes on loop 18 is 22

Number of changes on loop 19 is 11

Number of changes on loop 20 is 2

Number of changes on loop 21 is 0

Cluster DOES percolate. Cluster number: 225
```

```
Opening file <map.dat>

Writing data ...

...done

File closed

Opening file <map.pgm>

Map has 21 clusters, maximum cluster size is 121

Displaying all clusters

Writing data ...

...done

File closed
```

The program creates a grid. Each square is randomly assigned to be empty (1) or filled (0) according to the designed density distribution of filled squares. Each empty square is then assigned a unique number. The program iteratively updates the grid, updating each square is updated with the maximum of its neighbours. When an iteration results in no squares being updated, the iteration exits. The program then checks whether there is a cluster on the top row of the grid that has the same number as one on the bottom row i.e. it checks whether percolation has occurred.

The program writes out two files. The first is a data file which contains the values of each empty square when each square can be no longer updated i.e. the final state of the grid. This file is plain-text so you can view them as you would any plain-text file e.g.:

```
$ cat map.dat
```

The second file is a type of image file, a Portable Grey Map (PGM) file[1]. This file can be viewed with standard visualisation tools for example:

```
$ display map.pgm
```

or:

```
$ gimp map.pgm
```

The largest cluster will be coloured black. Other clusters will be shades of grey, the smaller the cluster the lighter the shade. The filled squares will be coloured white.

PGM files are plain-text so you can view them as you would any plain-text file e.g.:

```
$ cat map.pgm
```

[1] pgm format specification, http://netpbm.sourceforge.net/doc/pgm.html

These files do not include the halo as the use of a halo is an implementation detail.

3.4 Converting grids to PGM files

The plain-text file holds the final state of the grid. For example, consider the following file with the final state of the grid from our example:

```
  0   9   0   9   9

  9   9   9   9   0

  0   9   0   9   0

 10   0  13   0  12

  0   0  13   0   0
```

This grid has 4 clusters. The cluster with value 9 is the largest cluster, and the size of this cluster is, coincidentally, 9 squares. The second largest cluster is the cluster with value 13, and the size of this cluster is 2 squares.

The program converts the grid into a PGM file. An example corresponding to the above data is below. The first line "P2" states that this is a PGM file. The second line specifies the height and width of the grid. The third line has the maximum colour value. This is equal to the number of clusters. The colour values range from 0 to the number of clusters inclusive. Clusters are converted to colours as follows. If there are C clusters then the largest cluster is assigned value 0 (black), the 2nd largest cluster value 1 (a deep grey) and so on up to the Cth cluster, the smallest, which is assigned the colour value C-1. The colour value of C itself is used for the filled squares in the grid.

```
P2

          5          5

          4

  4   0   4   0   0

  0   0   0   0   4

  4   0   4   0   4

  2   4   1   4   3

  4   4   1   4   4
```

For our example there are 4 clusters so the maximum colour value is 4. The largest cluster (the cluster of values of 9 above) is assigned the colour value 0 (look at how the positions of the 9 values in the data file are the same as the positions of the 0 values in the PGM file). The second largest cluster (the cluster of values of 13 above) is assigned the colour value 1. The filled squares (with value 0 above) are assigned the maximum colour value of 4.

A variable allows this behaviour to change so that only a specific number of the largest clusters are shown.

3.5 Controlling program behaviour

The following variables in `percolate.c` and `percolate.f90` control the program's behaviour:

- `L`: map width and height. `L` >=1.
- `rho`: density, ρ. 0 <= `rho` <= 1.
- `seed`: seed for random number generator. Using the same seed allows the same random numbers to be generated every time the program is run. 0 <= `seed` <= 900,000,000.
- `datafile`: data file name.
- `percfile`: PGM file name.
- `MAX`: the number of clusters to output to the PGM file. 0 <= `MAX` <= `L` x `L`. If `MAX` is 1 then only the largest cluster is output. If it is 2 then the two largest are output etc. The default is all clusters are output.

4 Assessment 1 – Refactoring

Though the program works, and has no bugs that the authors are aware of, it is poorly commented, badly laid out, has cryptic variable names and is not modular. For your first assessment you are expected to refactor the program to address these limitations. Specifically, you are required to make the following changes to the program:

- Clean up and refactor the source code to make it more readable, better commented, and more modular.
- Extend the program to allow the user to specify the values for the grid size, data file name, PGM file name, rho, seed and maximum number of clusters in the PGM file via the command-line as command-line arguments or parameters.
- Write a build script which allows a user to:
    - Compile and link the source code into an executable.
    - Remove any auto-generated files created during compilation, linking or when running the program.
    - Any other tasks you think may be of use.
- Write documentation, in a plain-text file, on how a user can build and run the program and provide any other information you think a user would find useful.

4.1 Source code repository

You must create and use a source code repository for example, a Git repository in your own account on Cirrus or DICE. You can also use GitHub or BitBucket but, if you choose to do so, then please ensure your repository is **private**.

4.2 Execution

The program currently runs on both Cirrus and DICE and via the command-line. After your changes it must still be possible to run it on both Cirrus and DICE and via the command-line.

The program is runnable from the command-line in batch mode. In this context, "batch mode" means once the user presses RETURN the program runs to completion and outputs the results without prompting the user for further information or action. When implementing support for configuration via the command-line your program must still run batch mode in this way.

Non-compiling code or code that does not run successfully will receive less marks than code that compiles and runs successfully.

4.3 `arralloc.h`, `arralloc.c`, `uni.h`, `uni.c`, `uni.f90`

C creates dynamic arrays via its `malloc` function. This allocates memory for all the members of the array. Allocating memory for arrays using `malloc` can be difficult and time-consuming. The C version includes the files `arralloc.h` and `arralloc.c` which provides an implementation of an `arralloc` function to simplify the process of allocating memory in C. `arralloc` wraps up all the complexities of `malloc` and provides a nice clean interface for creating arrays.

The C files `uni.h` and `uni.c`, and the Fortran file `uni.f90`, contain an implementation of a random number generator.

You do not need to refactor these files. Treat these as third-party code that you are using as-is. You **can** move these files if you need to or want to create a different folder structure for the code, for example. Otherwise, for the C version you only need to focus on refactoring the files `percolate.h` and `percolate.c`. For the Fortran version you only need to focus on the refactoring the file `percolate.f90`.

4.4 Submitting your assessment

You are required to submit your source code repository with your source code, build files and documentation.

Submissions are done via the Turnitin submission tool from within LEARN.

Submit a single archive, `zip` or `tar.gz`, file containing your source code repository, build files and documentation. To make sure you submit your source code repository, archive the folder which contains your clone of your Git repository (i.e. the folder that includes the ".git" folder).

The filename of your submission must include your *exam number* which is the B number which appears on your matriculation card (this is not the same as your student number). You should also include the course identifier, `PS` for Programming Skills, and the identifier `Assessment1` for this coursework. So, for example, if your exam number is `B123456`, then you would name your submission file `B123456-PS-Assessment1.zip` or `B123456-PS-Assessment1.tar.gz`.

4.4.1 Deadline

The deadline for submission for your coursework for assessment 1, is **16:00 Tuesday 15th October (week 5) 2019** for all students.

Submission are allowed up to 7 calendar days after the deadline, with a 5% deduction per day or part-day late.

If you submit your coursework by the deadline then it will be marked, and you cannot submit an amended version to be marked later on.

If you do not wish the submitted version of your coursework to be marked, and intend submitting an amended version after the deadline, then it is your responsibility to let the course organiser know, *before the deadline*, that the submitted version should not be marked.

4.5 Marking

Your submission will be marked out of 100 as follows:

- Use of version control: 10
- Presentation and readability: 25
- Redesign and modularity: 25
- Command-line configuration: 10
- Build script: 20
- Documentation: 10
- **Total: 100**

The mark for this assessment forms 50% of your overall mark for the Programming Skills course.

4.5.1 Provisional marks

Your feedback will be returned no later than 16:00 Friday 1st November 2019.

You will then be asked to submit a short description of your views on your feedback. This is due by 16:00 Friday 8th November 2019. Your provisional marks will then be released. All marks are provisional until confirmed by the MSc course board in January 2020.

5 Assessment 2 – Testing and Performance

In this part of the coursework you will undertake two activities:

1. Write automated tests using an xUnit test framework.
2. Design and carry out a performance experiment and write a report describing your experiment and your results.

5.1 Automated tests

Do **one** of the following tasks **only**:

- Implement unit tests for the program using an xUnit test framework. If you choose to implement unit tests then these need to be written in C or Fortran i.e. the same language as the version of the program you are developing.
- Implement regression tests for the program using an xUnit test framework. If you choose to implement regression tests then these can be written in another language, e.g. Python.

If your test code requires additional packages or libraries that are not already available on Cirrus or DICE then you should document what these packages or software are, where to get them, and how to set them up.

5.2 Performance experiments

Design and carry out a performance experiment and write a report describing your experiment and your results. It is up to you what experiment you conduct, but some ideas include:

- Investigate the effects on performance of different compiler optimisation flags.
- Identify hotspots in the program which could be optimised in future.
- Investigate the behaviour and performance of the program when configured with different sized grids.
- Investigate the behaviour and performance of the program when configured with different proportions of empty and filled squares.
- Any other performance-related experiment you think are of interest,

Do **not** carry out all the experiments above – **you only need to design, execute and write up a single experiment**. Quality is more important than quantity. Credit will be given for intelligent and insightful analysis of your results, in preference to the quantity of data you present.

Your performance report should be structured as follows:

- **Introduction**: Short introduction to the rest of the report.
- **Method**: Description of how you carried out your experiment, what your set up was, how you collected data.
- **Results:** Your results, including a clear graphical presentation of any performance data.
- **Discussion**: Analysis and discussion of your results.
- **Conclusions**: Some brief conclusions and suggestions for future work.

Your report must a **maximum of 8 pages**. Any submission of longer than 8 pages will result in any pages after the first 8 pages not being read or marked.

5.3 Submitting your assessment

You are required to submit your updated source code repository with your test code, and your performance report.

Submissions are done via the Turnitin submission tool from within LEARN.

Submit a single archive, `zip` or `tar.gz`, file containing your updated source code repository and your performance report (as a PDF). To make sure you submit your source code repository, archive the folder which contains your clone of your Git repository (i.e. the folder that includes the ".git" folder) and make sure you include your performance report.

The filename of your submission must include your *exam number* which is the B number which appears on your matriculation card (this is not the same as your student number). You should also include the course identifier, `PS` for Programming Skills, and the identifier `Assessment2` for this coursework. So, for example, if your exam number is `B123456`, then you would name your submission file `B123456-PS-Assessment2.zip` or `B123456-PS-Assessment2.tar.gz`.

5.3.1 Deadline

The deadline for submission for your coursework for assessment 2, is **16:00 Wednesday 20th November (week 10) 2019** for all students.

Submission are allowed up to 7 calendar days after the deadline, with a 5% deduction per day or part-day late.

If you submit your coursework by the deadline then it will be marked, and you cannot submit an amended version to be marked later on.

If you do not wish the submitted version of your coursework to be marked, and intend submitting an amended version after the deadline, then it is your responsibility to let the course organiser know, *before the deadline*, that the submitted version should not be marked.

5.4 Marking

Your submission will be marked out of 100 as follows:

- Automated tests: 50
    - Choice of tests: 40
    - Presentation and readability: 10
- Performance report: 50
    - Description of performance experiments, results and analysis: 40
    - Presentation, quality of text and report: 10
- **Total: 100**

The mark for this assessment forms 50% of your overall mark for the Programming Skills course.

5.4.1 Provisional marks

Your feedback and provisional marks will be returned no later than 16:00 Wednesday 11th December 2019. All marks are provisional until confirmed by the MSc course board in January 2020.