|epcc|

THE UNIVERSITY OF EDINBURGH

ARCHER Course: Object Oriented Fortran
Percolate Practical

Adrian Jackson

January 13, 2016

1 Introduction

The purpose of this practical is to gain experience of taking a real scientific problem and implementing it as a program in Fortran. We will then using this program to try out the object oriented features in F2003. The goal is to complete the code provided, and then create different versions using the different functionality we will discuss in the lectures.

You are provided with:

- A set of template routines in Fortran, some complete, some incomplete.
- A set of shell scripts to build executable programs from these.

You will take these and:

- Develop the Percolate code, to fill in missing code to complete the implementation.
- Implement a module structure in the percolate code
- Define derived types for use in the program
- Implement Fortran classes
- Compare performance between the different versions

2 Download and extract the exercise files

Use *wget* (on ARCHER) to get the exercise files archive from the EPCC webserver:

```
guestXX@archer:~> wget tinyurl.com/archer140116/percolate.tar.gz
--2016-01-13 13:10:46--  http://tinyurl.com/archer140116/percolate.tar.gz
Resolving tinyurl.com... 104.20.87.65, 104.20.88.65
Connecting to tinyurl.com|104.20.87.65|:80... connected.
HTTP request sent, awaiting response... 301 Moved Permanently
Location: http://www.archer.ac.uk/training/course-material/2016/01/oofortran_culham/Exe
--2016-01-13 13:10:47--  http://www.archer.ac.uk/training/course-material/2016/01/oofor
Resolving www.archer.ac.uk... 193.62.216.12
Connecting to www.archer.ac.uk|193.62.216.12|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 3183 (3.1K) [application/x-gzip]
Saving to: 'percolate.tar.gz.1'
```

```
100%[===================================================================================

2016-01-13 13:10:47 (367 MB/s) - 'percolate.tar.gz.1' saved [3183/3183]
```

To unpack the archive:

```
guestXX@archer:~> tar -xzvf percolate.tar.gz
percolate/
percolate/Fortran/
percolate/Fortran/uni.f90
percolate/Fortran/build
percolate/Fortran/percolate.f90
percolate/Fortran/perctest.f90
```

3 The Percolate problem

You will be implementing a program to solve the following problem. Suppose we have a grid of squares (e.g. 5 x 5) in which a certain number of squares are filled (green) and the rest are empty (white). The problem to be solved is whether there is a path from an empty space (white) on the top row to an empty space on the bottom row, following only empty spaces. This is a bit like solving a maze. See figure 1.

Connected spaces form clusters. If a cluster touches both top and bottom - i.e. there is a path of empty squares from top to bottom through the cluster - we say the cluster "percolates". This is the case in the second grid of figure 1.

To solve this problem, the largest clusters may be of interest. In the first two grids of figure 1, the largest clusters are 13 and respectively, and the second largest are 2 and 2 respectively.

Now, consider an $L \times L$ grid. What is the probability P of percolation if the grid is filled with a given density ρ (Greek letter, pronounced *rho*) where $0.0 <= \rho <= 1.0$ i.e. where the density ranges from empty to completely filled.

In the first two grids of figure 1, $L = 5$ and $\rho = 0.40$ (10/25) and 0.48 $(12/25)$ respectively.

For a given ρ there are many possible grids so to solve the problem we can generate different random grids with the correct density then compute the probability that a cluster percolates.

The simplest cases are $\rho = 0.0$ as then $P = 1.0$ and if $\rho = 1.0$ then $P = 0.0$. But what about, say, $\rho = 0.5$? For this we need to do many simulations and if we generate 100 grids and 72 of them percolate then $P = 0.72$.

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 2.

4 Solving Percolate

One solution is to initialise each of the empty cells with a unique positive integer. Then, loop over all the cells in the grid many times and during each pass of the loop to replace each cell with the maximum of its four neighbours. In all cases, we can ignore the filled (green) cells.

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 and look for percolation if the same number appears at top and bottom of the grid.

An example is shown in figure 3.

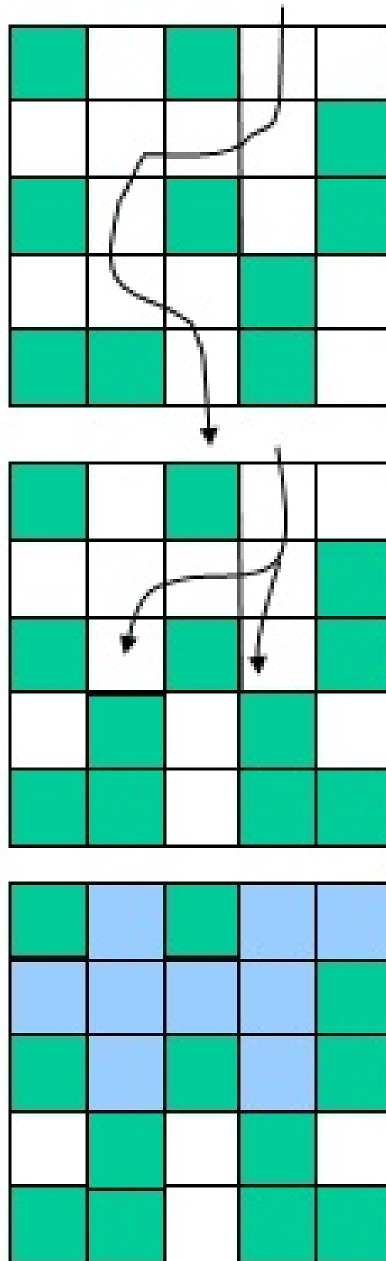

Figure 1: Three grids. The top grid percolates as there is a path from top to bottom.

Our solution involves replacing each cell with the maximum of its four neighbours, but what about cells 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 $L \times L$ grid we use a $(L+2) \times (L+2)$ and set the halo cells to be filled (i.e. green). We can do this as the zeroes will never propagate to other cells. See figure 4.

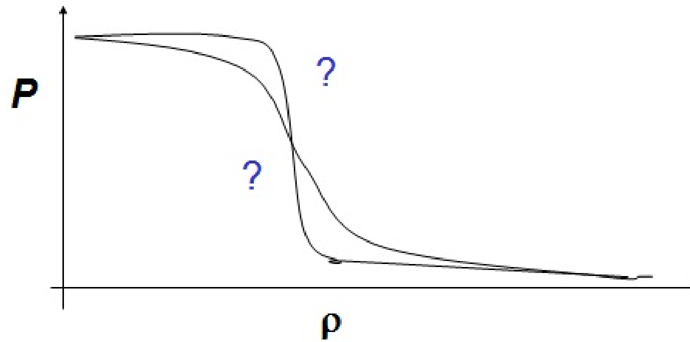

Figure 2: Plotting density, ρ, against probability of percolation, P.

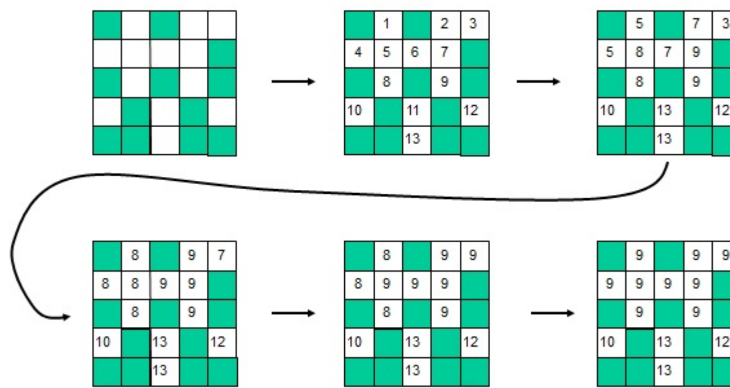

Figure 3: Solving Percolate. In this example no cluster percolates since the top cluster has value 9 and the bottom cluster has value 13.

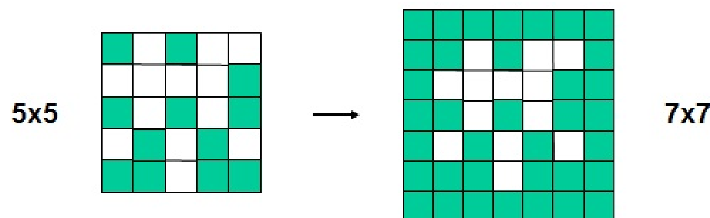

Figure 4: Using a halo to handle the edges of the grid.

5 Explore and build the code

This task aims to familiarise you with the command line interface and tools used on typical HPC systems, which are usually based on Linux or Unix.

Examine the files. Open some of the source files in an editor, have a look at the content. Initially the code will compile and run:

For Fortran:

```
$ cd Fortran
$ ./build
$ ./percolate
```

Try opening the image file, and have a play around.

The program produces a visual output as a Portable Grey Map (PGM) file which can be viewed with standard visualisation tools e.g.

```
$ display map.pgm
```

Initially, this will be a meaningless chequerboard because you haven't written the code yet and the output routine will incorrectly report the number and size of clusters.

6 Develop the Percolate code

Develop the Percolate code template file to fill in the incomplete functions. Currently all the functionality is in a single file, future practicals will restructure this to a more sensible configuration, but for now you need to focus on completing four routines:

- `fill` to fill the $L \times L$ grid with ones and zeros with the correct density. We look at this in more detail in the next section.
- `init` to initialise the grid for the `update` routine. This should set each non-zero cell to be a unique integer.
- `update` to look for clusters until they are all found, by looking at each non-zero cell, replacing it with the maximum of its four nearest neighbours and returning the number of cells that have changed.
- `test` to check if percolation has occurred. This returns `true` if percolated; `false` otherwise

The main program will stop calling `update` when there are no more changes in the cells.

There is also a routine, `write` which also checks some statistics and prints the number of clusters and the size of the largest.

You can also control how many clusters are displayed - choose only 2 or 3 for nicer images.

For an example of program behaviour, see figure 5.

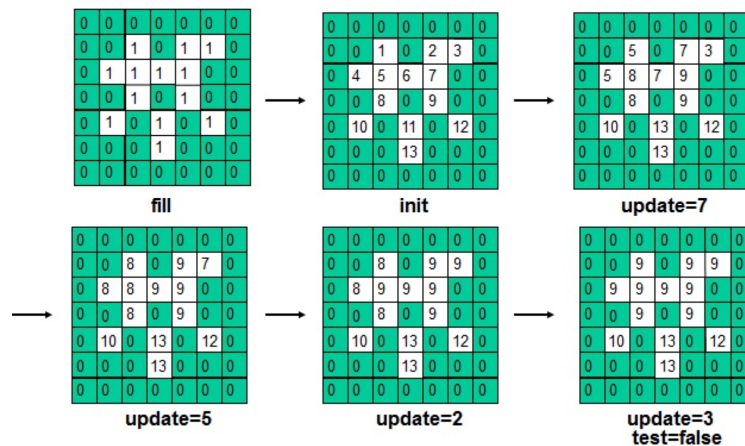

Figure 5: An example of Percolate program behaviour.

6.1 Filling the map - implementing `fill`

`fill` should always set the boundary, or halo, cells to 0.

`fill` should fill the map with probability ρ, For example, if $\rho = 0.6$, then 60% of the interior cells should be 0, the rest 1. One way we can do this is to check against the value of a random number, r where $0.0 <= r < 1.0$. So, if $\rho = 0.6$, we decide whether a square is filled or empty whether or not $r <= 0.6$ or $r > 0.6$.

There is a random generator in `uni.c`, `uni.f90` and `Uni.java`.

To compute the actual density we can do: $number_of_zeros(L \times L)$ and compare this to ρ

Implement `fill`, run the code and visualise the output for different values of ρ. Empty cells (the clusters) will appear black and only the interior is displayed.

7 Modules

Now you have a working program we want to transform it into a well structured and designed program. Split the currently source code you have into a number of modules that are used by the main program to calculate the solution.

If you use code from a module in a program or a different module then you need to `use` that module to make that functionality available in those modules/programs.

Remember, when you move code into a module file you will need to change the compilation script (we are using the `build` script to compile the code) to build the separate module and then link it with the main program. Compilers also require that modules be built before any program or module that uses these modules (you need to ensure the order of compilation is correct).

8 Derived Types

Implement a derived type for the map data type. Consider is any other variables should be packages with this type, and if any other derived types are needed. Which module will you put this derived type in? Or should it have it's own module.

Run the program to ensure it still completes correctly and see if the performance has changed at all.

If you have type you can also implement a 3d type that performs the same operations but on a 3d domain. This will involve making 3d versions of the other procedures as well. You do not have to implement an output routine for the 3d case if you do not have time.

9 Classes

Consider what procedures should be added to your derived type(s). If you have implemented a 3d type as well as a 2d type then use polymorphism in the procedures to enable different types to be passed to the same procedure.

10 Abstractions/Extensions

If you have not already done so you now need to implement a 3d type. Create an abstract class that defines the deferred procedures implementations need to provide and then extend it for the 2d and 3d types.

11 Run performance experiments

Run some experiments on how Percolate performs for various values of L and plot a graphs of P versus ρ for these values.

Also gather data and plot graphs to show performance of the program e.g. iterations per second versus L.

Fortran has timing routines `dtime()` and `etime()` which you could use.

You could consider using dynamic arrays, via the `ALLOCATE` function.