1 **Introduction**

The purpose of this assignment is for you to demonstrate that you can write a message-passing parallel code for a simple two-dimensional grid-based calculation that employs a two-dimensional domain decomposition and uses non-blocking communications. This will involve extending and enhancing the MPI program you developed for the Case Study exercise. The basic aims of the coursework are to:

\_ write a working MPI code (in C, C++ or Fortran) for the percolation problem that can use a twodimensional

domain decomposition and uses non-blocking communication for halo-swapping;

\_ use periodic boundary conditions in the horizontal direction (first index “i”) – we have provided an

example of how to implement this in the serial version;

\_ calculate the average of the map array during the iterative loop, and print at appropriate intervals;

\_ be able to terminate the calculation when all clusters have been identified, i.e. when there are no

changes in the map array between steps;

\_ demonstrate that the parallel code works correctly and that the performance improves as you increase

the number of processes.

2 Report

Submit a short report, a maximum of 10 pages (excluding any cover page, table of contents or appendices), including a description of the design and implementation of your MPI program followed by results such as the tests you have done to investigate correctness and performance. **You should present results from a number of tests designed to show that the parallel code works correctly and that its performance improves as the number of processes is increased. You should quantify the parallel performance using appropriate metrics; you may also wish to consider how performance is affected by the problem size.**

**3 Source Code**

You should also submit your complete MPI program including a README file briefly describing it – see the separate document MPP-Code-Submission.pdf for full details. Ensure that your program is clearly written and easy to understand with appropriate use of comments, multiple source files, functions or subroutines, meaningful variable names etc. Preference will be given to simple, elegant and robust programs rather than code that is unnecessarily complicated or difficult to understand. Use MPI features described in the lectures (derived datatypes, virtual topologies, etc.) where appropriate.

**4 Notes**

Here are a few points that you should take into account.

1）It is essential that your report contains tests that demonstrate your parallel program works correctly.

2）Think carefully about what sections of your program you time to measure the parallel performance. There are a number of choices (e.g. time the entire program from start to finish), but the average time per step is usually a good measure as we are not particularly interested in the performance of the (serial) initialization and IO sections. You should run on the backend compute nodes of Cirrus and check that the performance is stable and reproducible.

3）Be careful to do sensible performance tests and not to burn huge amounts of CPU time unnecessarily: consider how long you need to run to give a reasonable assessment of performance. When benchmarking large HPC codes (as opposed to doing actual computations), it is normally not necessary to run to completion; perhaps only a limited number of steps is required. Conversely, for small problem sizes, you may need to run for additional steps to obtain a reasonable run time.

4）You will not easily be able to use MPI\_Scatter and MPI\_Gather for distributing and collecting the map array in a 2D decomposition. Implement these operations in a simple manner so you can start developing a working MPI program – it does not matter if these phases are inefficient.

5）This is not an exercise in serial performance optimisation. However, to get realistic parallel performance numbers it is necessary to use a reasonable level of serial compiler optimisation. For example, you should compile your codes with the -O3 option on Cirrus.

6）You should concentrate on writing an elegant and efficient MPI code, performing a solid investigation of its performance and writing a good report. However, if time is available, you are welcome to investigate enhancements to your parallel code (some suggestions are given in the Case Study).

7） If you cannot produce a working solution you should still submit any code you have written. The report should describe its design and implementation, a description of how far you got and what the problems were. Correctness and performance tests should be done with your Case Study solution.

8）This code should be one of the following:

• A working message-passing parallel version of the percolation problem using a twodimensional domain decomposition.

• Or, a working message-passing parallel version of the percolation problem using a onedimensional domain decomposition. Note that you should submit this only if you are unable to develop a working version with a 2D domain decomposition. In this case:

o You should explain in your report the issues you faced writing the 2D version.

o You should also submit whatever code you have for the 2D implementation.

o Any correctness and performance results in your report should come from the working 1D version, i.e. your Case Study solution.

9）• Instructions on how to run the code on P processes, including any command line arguments. The first argument must be the random number seed as in the supplied serial code.

10）

o As submitted, your code must at a minimum be able to run using 16 processes on a grid size of 288 x 288 with the random number seed specified at run time.

o By default, the code should run with L = 288 x 288 and  = 0.411. Specifically, if your program is called percolate, issuing the command mpirun -n 16 ./percolate 1564 on the login node must run the code on 16 processes using a random number seed of 1564, a grid size of 288 x 288 and density  = 0.411.

o This may be adjustable with additional arguments (e.g. you could accept L and/or  as additional optional arguments), but the defaults must be L = 288 and  = 0.411.

o If changing either the problem size or the number of processes requires modification of any source files, this must be clearly explained.

o The code must produce an output file called “map.pgm” with a call to “percwrite”, although you may wish to suppress file output for large simulations (e.g. L > 1024).

• Any restrictions on the running of the code.

o For example: if the number of processes, P, must evenly divide the grid size,

Report (weighting of 60%)

– Content 70%: code description, testing, performance data, performance quantification, analysis of results, completeness and originality, conclusions; 10% for each category.

– Presentation 30%: written communication, layout, quality of figures; 10% for each category.

Source code (weighting of 40%)

– **Presentation and structure, use of MPI, correctness, functionality;** 10% for each category.