MPP Coursework Assessment

1 Introduction

The purpose of this assessment is for you to demonstrate you can write a message-passing parallel program 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. For example, on 4 processes the Case Study splits the grid over only one of its two dimensions, e.g. in C there are 4 subdomains each of size $\frac{L}{4} \times L$. Your code should be able to split the grid over both dimensions, i.e. use 4 subdomains each of size $\frac{L}{2} \times \frac{L}{2}$.

You will be provided with simple solutions to the Case Study. However, I recommend that you still attempt to write your own parallel solution so that you fully understand the more straightforward problem, which uses a one-dimensional decomposition, before attempting the two-dimensional decomposition.

The aims of the assessment are to:

- write a working MPI program for the sandpile model (in C, C++ or Fortran) that can use a two-dimensional domain decomposition and uses non-blocking communication for all halo-swapping;
- use new boundary conditions that are non-periodic in the first dimension (first index "i") and periodic in the second dimension (second index "j") – we will provide a reference example showing how to implement this in serial;
- calculate the total number of sandpiles (i.e. the number of cells with height greater than zero) after each step (printing at appropriate intervals) and be able to terminate the calculation as soon as the number sandpiles that change their height reaches zero;
- demonstrate that your parallel program works correctly and that the performance improves as you increase the number of processes, including runs on multiple nodes.

2 Structure

The assessment comprises two parts with different deadlines:

Part 1: a short report containing a description of how you plan to design, implement and test your 2D-decomposed MPI program plus initial test results from the 1D-decomposed Case Study;

Part 2: your final MPI program plus a short report with results from your own 2D-decomposed solution.

Your tests must cover both **correctness** and **performance**.

Although you do not submit any code for Part 1, you should run your proposed correctness and performance tests against the Case Study exercise (i.e. the 1D-decomposition with non-periodic boundaries). You can either use your own solution or the simple example parallel solution from the course.

The reason for having two parts is to ensure that, after Part 1, you have a clear plan for your work when you start to develop your own parallel program for Part 2. For Part 1, you will also have developed and implemented the infrastructure for all the required tests. This means that the testing process for Part 2 will be straightforward as you can just re-run the same tests but on a different program. The performance results from Part 1 will also give baseline values to compare to your own 2D solution.

3 Part 1 (30% of course)

3.1 Report

Submit a short report, **a maximum of 4 pages** (excluding cover page, table of contents or appendices), including a description of the proposed design and implementation of your MPI program followed by brief results from tests applied to the Case Study exercise.

Your description should cover how you plan **to improve the Case Study exercise** to meet the **requirements set out in Section 1**.

Your report should cover:

- Which parts of the existing program need to be changed and why.
- What additional MPI functions you plan to use and why.
- How you plan to structure your program: what new functions will you write and how will your source code be split across different files?
- What correctness and performance tests you plan for the full program and brief results of these tests applied to the Case Study exercise.

You should present results from a number of tests designed to show that the Case Study program works correctly and that its performance improves as the number of processes increases. You should quantify the parallel performance using appropriate metrics such as speedup and parallel efficiency; you may also wish to consider weak scaling, i.e. how problem size affects scalability. For all performance results you must be clear what part(s) of the program were timed. Even if you use the supplied Case Study solution, you will still have to add some additional timing calls to enable you to measure the performance.

3.2 Marking Scheme

Marks will be allocated for the report as follows: proposed changes; use of MPI; program structure; testing; presentation (20% for each category). Note that the presentation component includes how easy the report is to read, its layout and the quality of figures and diagrams.

The deadline for Part 1 is **16:00 GMT (UK time) on Friday 31st October 2025**.

4 Part 2 (70% of course)

4.1 Report

Submit a short report, **a maximum of 3 pages** (excluding cover page, table of contents or appendices), focusing on the results from the correctness and performance tests of your 2D-decomposed program.

You should study performance for a range of process counts and problem sizes, focusing on the parallel scaling. Performance **must** be measured **on the ARCHER2 compute nodes**, including results on **multiple nodes**. In your report you must state **the Slurm job id of the log file** that you have included in your code submission (see Section 5.2).

The report should end with brief conclusions for your work as a whole. This should include comments on the overall performance of your code, and a brief reflection on the project process: what went well, what did not go so well and any lessons learned for the future.

4.2 Source Code

Ensure that your program is clearly written and easy to understand with appropriate use of comments, multiple source files, user 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.

All programs **must** use genuine 2D arrays declared using standard C/C++ syntax or dynamically allocated using `arraymalloc2d`. We **will not** accept codes that allocate 1D arrays such as `oldh[L*L]` and index them using a scheme like `oldh[i*L+j]`.

4.3 Marking Scheme

Marks will be allocated as follows, **independently** for the report and source code:

- Report (weighting of **30%**)

  Presentation, testing, performance analysis, conclusions; 25% for each category.

- Source code (weighting of **70%**)

  Presentation & structure, use of MPI, correctness, functionality; 25% for each category.

The deadline for Part 2 is **16:00 GMT (UK time) on Wednesday 26th November 2025**.

5 Submission Guidelines

5.1 Reports

Preference will be given to reports that present carefully chosen, clean and well explained results as opposed to large amounts of raw data. As well as written text you should use graphs, figures and tables as appropriate. If you want to include the raw data, this can be included in an appendix but this should only be a few pages long. The report **must be in PDF format** and must have a file name of the form "B1234567-MPP2526-cwork1.pdf" where you replace B1234567 with your own **exam number** and **not your UUN or matric number** and use "cwork2" for the second report. The report **must be written anonymously** but **must contain your exam number** on the title page. You **must not** include your name or UUN anywhere on the report. When uploading to Turnitin, your "Submission title" **must match the file name**, e.g. "B1234567-MPP2526-cwork1".

5.2 Code

You **must** submit a copy of the code you used to generate the results presented in your report - you **must not** make edits to your code after you have finalised and submitted your written report.

It is important to remember:

1. we have to mark multiple codes so it is **essential** they are all submitted using a common format;
2. you should assume the person marking your code **does not** have access to your report, so a short README **must** be supplied containing essential information on compilation and execution.

You should follow these guidelines:

- Your submission **must** include a markdown README file, with the title README.md, which contains a very brief description of the structure of your code.

  - Where are the source and header files? What functionality is contained in which source files?
  - Instructions on how to build the code on ARCHER2. Your code **must** build using the default Cray compilers (**not** GCC) with `-O3` optimisation.
  - Instructions on how to run the code on P processes, including any command-line arguments. The first argument **must** be the offset of the initial sandpile as supplied serial code. You should describe any restrictions, e.g. if the grid needs to divide equally amongst processes.

- As submitted, the code **must** be able to compile and run, without modification, on 16 processes with $L = 512$. Specifically, if your program is `sandpile`, issuing the command `srun -ntasks=16 ... ./sandpile 256` on the login node **must** run the code on 16 processes using the specified offset and a grid size of 512×512.
- This may be adjustable with additional arguments (e.g. you could accept L as an additional optional argument), but the default **must** be $L = 512$.
- If changing either the problem size or the number of processes requires modification of any source files, this **must** be clearly explained in the README.
- The code **must** produce an output file called "sand.ppm" using a call to "sandwrite" or "sandwritedynamic", although you may wish to suppress file output for large simulations (e.g. $L \geq 2048$) for benchmarking.
- The README should not include any performance or correctness results - these should all be included in your report.
- You **must** also include the Slurm job script and complete output from one run of the program on more than 128 MPI processes on the compute nodes (i.e. from a Slurm job using more than one node). The file name **must** contain the Slurm job id, e.g. `sandpile-3141593.out`.

5.3 Submission format

We strongly recommend using a version control system such as GitLab for development; the submission format is designed to be simple when using GitLab. Your remote repository **must** be private: you can obtain free, private GitLab repositories from the University at `https://git.ecdf.ed.ac.uk/`.

The code **must** be uploaded to Turnitin as a single .zip archive. The archive **must** have the following naming convention: `B1234567-MPP2526[-yyyy].zip` where B1234567 should be substituted by your exam number; `-yyyy` is an optional, hyphen-separated text string. For example, you may include the repository branch name here, which is the default if you download from GitLab.

The archive file **must** unpack to a directory with the same name as the archive. For example, issuing `unzip B1234567-MPP2526-main.zip` should create a directory `B1234567-MPP2526-main/` containing all the source code; no files should appear in the current directory. The README.md file **must** be at the top level of your source code, i.e. directly in the `B1234567-MPP2526-main/` directory. When you upload your zip file via Turnitin, your "Submission title" **must** match the file name, e.g. `B1234567-MPP2526-main`

Finally, please ensure that your name and UUN are not included anywhere in the source code, archive or README file. Only your anonymous exam number should be visible to the marker.

5.4 Late penalties

Standard penalties apply: 5% reduction for each elapsed calendar day (or part day) after the deadline; submissions more than seven days late will be awarded 0%.

6 Practical Sessions

The practical sessions will continue for the duration of the course. These are held so you can ask questions and discuss problems relating to your MPP coursework (or any part of the MPP course as a whole). This includes the content of your report as well any programming issues.

7 Notes

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

- You will **not** easily be able to use `MPI_Gather` for collecting the final height array in a 2D decomposition. You should initially 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 or inelegant to start with - you can return to them later when you have completed the more important parts of the program such as 2D halo-swapping. We will provide a simple example of how to do this for the 1D Case Study code.
- To ensure your code has uploaded correctly, we strongly advise you re-download the zip file from Turnitin, copy to ARCHER2, and follow your own README instructions to check that it compiles and runs as expected. This will catch simple errors like forgetting to include some source files.
- It is essential that your report contains tests that demonstrate your parallel program works correctly. I am equally interested in situations when the tests fail as when they pass. For example, what bugs did your tests uncover during development? If your program has some limitations, do the tests detect that it does not run properly if these limitations are not adhered to?
- 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) initialisation and IO sections. You should run on the backend compute nodes of ARCHER2 and check that the performance is stable and reproducible.
- 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 programs (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.
- 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 compiler optimisation. For example, you should do all your performance tests with programs compiled with the `-O3` option on ARCHER2.
- The supplied solution to the Case Study is very basic. As well as the mandatory aims listed in Section 1, there are other parts of the program you may wish to improve. For example, the problem size and number of processes are fixed at compile time, and it does not run correctly when L is not an exact multiple of the number of processes (this is not even checked).
- You should concentrate on writing an elegant and efficient MPI program, 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 program.
- If you cannot produce a working solution you should still submit any code you have written. The report should contain a description of how far you got and what the problems were. Correctness, performance tests and scaling analysis should be done with the Case Study solution. These should be more extensive than was submitted in Part 1.