Advanced Research Computing Coursework Report

James Hughes
Word count: 2977

March 28, 2024

Contents

1 Introduction

Conway's Game of Life first made an appearance in public discourse in 1970, when Martin Gardner wrote a brief article introducing the concept in his column Mathematical Games in Scientific American[1]. The piece briefly described the game, stating its rules, and displaying various diagrams of interesting simple behaviours such as periodicity and indefinite movement across the board. It is notable that Gardner suggested that readers use 'counters and a board', or failing that, 'pencil and graph paper', to see a demonstration of the game in action. This is in stark contrast to the situation today, where plenty of computer simulations are easily accesible online [2], enabling the game's interesting behaviours at scale to be appreciated. This project seeks to take the ease of implementing this simulation with modern computing tools a step further, by treating it as an exercise in developing highly optimised and parallelised code, with a focus on maximising the performance of the simulation algorithm when run at scale.

The Game of Life consists of a large orthogonal grid of cells which exist in either one of two states: 'dead' or 'alive'. At each stage or 'tick' of the simulation, the state of each cell is updated according to its own current state, and the current state of the cells adjacent (including diagonally) to it. The current state of every cell persists to the next tick, except,

- an 'alive' cell with fewer than two, or more than three, 'alive' neighbours will become 'dead' in the next tick, and
- a 'dead' cell with exactly three 'alive' neighbours will become 'alive' in the next tick [1].

The evolution of the state of any cell within the simulation is thus bound by a deterministic rule that only depends on the current state of the simulation in a localised area. This makes it akin to physical simulations of the real-world, for instance discrete numerical techniques to solve PDEs that relate to fluid dynamics. Therefore, it is a suitable choice of problem to demonstrate the role of high-performance computing at the frontier of scientific research.

While the focus of this project was to produce highly optimised code, care was taken to balance this with traditional best practices in software development. Only once the codebase was implemented and verified in a readable manner was it then iteratively re-factored to be increasingly optimised. This

ensured that the maintainability and robustness of the earlier code could be preserved as much as possible, despite the tendency of optimisation to reduce these qualities. In summary, the strategy of development sought primarily to demonstrate sophisticated optimisation techniques, while not losing sight of computer scientist Donald Knuth's sage advice [3]:

> "Premature optimization is the root of all evil."

2 Algorithm Design and Early Development

The first consideration in carrying out this simulation computationally was the abstraction and storage of the simulation grid in memory. Conceptually, operations need to occur at two levels of abstraction:

- the level of matrix operations, including reading binary cell states and summing the adjacent cell states; and
- the level of simulation operations, such as evolving the simulation, storing how many ticks have passed, and displaying the result.

Therefore, both a `Matrix` and a `World` class were created to store methods and attributes relevant to these respective levels. `World` objects store two `Matrix` objects as members, to represent the current state of the simulation, and the next state. They also store an `age` attribute to keep track of the progress of the simulation, and whose parity determines which of the two `Matrix` members stores the current state.

One of the key routines to be prototyped was the function responsible for summing adjacent cell states:

```
1  Matrix matrix::count_neighbours(Matrix A) {
2      Matrix B = Matrix(A.n_rows - 2, A.n_cols - 2);
3
4      for (int i = 1; i < A.n_rows - 1; i++) {
5          for (int j = 1; j < A.n_cols - 1; j++) {
6              for (int p = -1; p < 2; p++) {
7                  for (int q = -1; q < 2; q++) {
8                      B(i - 1, j - 1) += A(i + p, j + q);
9                  }
10             }
11             B(i - 1, j - 1) -= A(i, j);
12         }
```

```
13          std::cout << std::endl;
14      }
15      return B;
16  }
```

This routine iterates across all of the entries of A (besides the edges) and then across all neighbours of said entry, summing all of the values to return the results in matrix B - which is easy to understand and debug. Similarly, the first prototype for the routine used to update cells according to the rules was written in an intuitive way:

```
1  int conway::evaluate_rules(Matrix cells_count, Matrix &
       cells_current,
2      Matrix &cells_next) {
3      for (int i = 1; i < cells_count.n_rows - 1; i++) {
4          for (int j = 1; j < cells_count.n_cols - 1; j++) {
5              if (cells_current(i, j) == 1) {
6                  if (cells_count(i, j) != 2 and cells_count(i,
       j) != 3) {
7                      cells_next(i, j) = 0;
8                  } else {
9                      cells_next(i, j) = 1;
10                 }
11             } else {
12                 if (cells_count(i, j) == 3) {
13                     cells_next(i, j) = 1;
14                 } else {
15                     cells_next(i, j) = 0;
16                 }
17             }
18         }
19     }
20 }
```

The routine evaluates a set of nested if statements, which first inspects the current state of the cell, and then checks the counts of the neighbouring cells accordingly. Both of these fundamental routines were kept separate from the handling of the periodic boundary. Within the `World` class, the grid of cells was represented as a matrix that had an extra row of 'ghost' cells on each edge. There was then a separate routine to handle the update of these edges to implement a periodic boundary, by copying each 'proper' edge and vertex of the grid to the corresponding opposite ghost edge or vertex. Therefore the evolution of the simulation by one tick is decomposed into updating this boundary first, and then updating the cells.

This modular quality of the code enabled precise verification early on in the project, and indeed developing a suite of unit tests via GoogleTest was the next step after prototyping. This ensured that these core routines behaved as expected, and handled edge cases, forming part of a testing led-development strategy. Creating the test suite involved the anticipation of other important functions that handled the input and output of matrices, for instance. This ensured that the later development was more efficient as the modular organisation of the code, and the role of each routine, had already been planned out. Moreover, the modularity contributed to a more maintainable code, particularly with the separate handling of the boundary, in that different boundary conditions (such as constant or reflective padding) could be easily implemented in future.

3 Optimisation and Parallelisation

Optimisation was first achieved by single-thread improvements. The focus was on the fundamental simulation routines since the performance of these becomes increasingly important as the scale of the simulation increases. A major improvement in this respect was changing the `count_neighbours` method to exploit the separability of the box blur convolution [4], in the following fashion:

```
Matrix matrix::count_neighbours(Matrix &A) {
    int n_rows = A.n_rows;
    int n_cols = A.n_cols;

    // Row convolution
    Matrix B(n_rows, n_cols);
    for (int i = 0; i < n_rows; i++) {
        for (int j = 1; j < n_cols - 1; j++) {
            B(i, j) = A(i, j - 1) + A(i, j) + A(i, j + 1);
        }
    }

    // Column convolution
    Matrix C(n_rows, n_cols);
    for (int i = 1; i < n_rows - 1; i++) {
        for (int j = 1; j < n_cols - 1; j++) {
            C(i, j) = B(i - 1, j) + B(i, j) + B(i + 1, j);
        }
    }
```

```
20      return C;
21  }
```

Note that in addition to the separation of the convolution into two one-dimensional kernels, the new routine included the target cell entry in the count, removing the need to subtract this value in line 17. This reduced the number of operations to compute each count from 10 to 6. Additionally, the first row convolution has the extra benefit of accessing both matrices A and B contiguously, since all entries of A accessed in each iteration lie in the same matrix row.

The performance gain was documented by the `time_count_neighbours` script, the results of which are found in Table 1. For the largest matrices in the table this optimisation had the effect of a speed up of roughly 8-fold, which increases even further for smaller matrices. It should also be noted that there appeared to be a cache-resonance effect for matrices of size larger than 500, significant slow-downs for sizes that are powers of two beyond this number (512, 1024 and so on.). In later timing scripts therefore, these sizes of simulation were avoided, so as not to confound different performance factors.

Incorporating a transposition of matrix `B` was also experimented with, both by including a standalone loop to compute a separate matrix `B_T`, as well as computing `B_T` directly from `A` instead of `B`. Theoretically this could enable both 1D convolutions to be computed as row convolutions, such that they both benefit from contiguous array access. However, this appeared to cause a decrease in performance. This is also a reasonable outcome considering that both the convolution operations and the transposition involve $O(n^2)$ operations with respect to the matrix size n, and so the gain from more efficient usage of the cache is offset by the extra number of total operations from the transposition itself, even at scale.

The `evaluate_rules` function was also optimised to produce the following code:

Matrix size	Prototype time (ms)	Separable convolution (ms)
2	0.0000658	0.0001272
3	0.003598	0.0001808
4	0.0097835	0.0004059
6	0.0160791	0.0001572
7	0.0210006	0.0002301
8	0.0255929	0.0002429
14	0.0476087	0.0004165
15	0.039723	0.0007573
16	0.0394387	0.0008217
30	0.134101	0.0005995
31	0.115235	0.0007182
32	0.0972259	0.0013144
62	0.431404	0.0018366
63	0.326085	0.0020772
64	0.325566	0.0038449
126	1.48509	0.0319895
127	0.925173	0.0282965
128	1.32786	0.0354224
254	5.63889	0.185919
255	6.37235	0.197116
256	6.23367	0.171525
510	19.1519	0.538269
511	20.3742	0.602934
512	22.1763	0.760738
1022	63.7401	2.37999
1023	67.6161	3.1337
1024	59.022	3.58696
2046	165.155	9.53187
2047	172.047	9.17258
2048	202.097	12.758
4094	608.048	53.6286
4095	554.764	66.2772
4096	580.508	70.9785
8190	1949.4	242.911
8191	2085.35	249.298
8192	1959.89	289.589

Table 1: Mean of 10 timed runs on square matrices, comparing two implementations of `count_neighbours`

.

```
1  int conway::evaluate_rules(Matrix &Cells_count, Matrix &
       Cells_current, Matrix &Cells_next) {
2      for (int i = 0; i < Cells_count.n_rows; i++) {
3          for (int j = 0; j < Cells_count.n_cols; j++) {
4              Cells_next(i, j) =
5                  (Cells_count(i, j) == 3) ||
6                  ((Cells_count(i, j) == 4) && (Cells_current(i
       , j) == 1));
7          }
8      }
9      return 0;
10 }
```

The main change is the removal of nested if statements, avoiding costly branch mis-predictions in the CPU, a problem which scales with the size of the simulation. This routine instead employs a masking technique which exploits the binary nature of the cell states. In fact, in this case the optimisation is arguably more readable; it essentially updates each cell to alive only if there are two adjacent alive cells (in which case the cell could be alive or dead), or if there are three and the cell is currently alive. On a more subtle note, C++ implements short-circuit evaluation of these boolean operators, meaning that the second expression evaluation can be bypassed when the first already determines the overall value of the boolean operator. With this in mind, the more likely condition of 'exactly 2 alive neighbours' is placed before 'exactly 3 alive neighbours and current cell alive', while the less likely condition of 'exactly 3 alive neighbours' is placed before 'current cell is alive', with the aim of minimising the number of boolean expressions that must be evaluated.

Further optimisation was achieved by parallelisation, using MPI and OpenMP. The main difference between the two interfaces is that unlike OpenMP, MPI 'ranks' do not have access to a shared memory space, instead having separate allocated memory that can be communicated across the ranks. Therefore MPI was used to implement a domain decomposition of the simulation, that is, dividing the simulation grid amongst the ranks to be separately evolved. The main challenge with this is implementing an organised collective communication such that cells on the boundaries of each 'chunk' can be correctly updated with respect to the values of the correct neighbouring cells located in other chunks.

In the first instance this was done using a 1D topology, dividing the simulation along the vertical dimension.

Incidentally, the pre-existing routine for updating the periodic boundary was helpful here. This routine updates every 'ghost' cell at the boundary of the matrix of cell values with the corresponding true inner cell value found on the opposite side of the matrix. Without parallelisation this simply implements the periodic boundary at the extremes of the simulation. But adding this as an extra step before communicating between ranks is also helpful, as shown in Figure 1. Without this step, vertices must be passed to multiple locations which can make it difficult to keep track of where cell values are communicated, as in Figure 1a. Moreover, there must be separate communications for the edges and vertices.

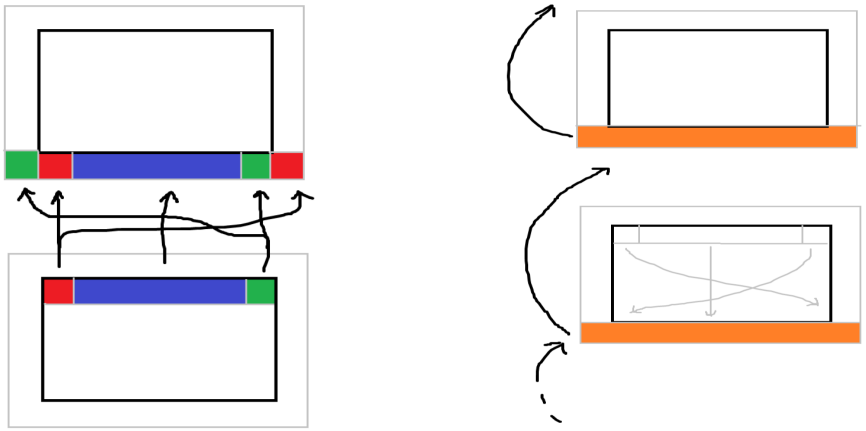
On the other hand, Figure 1b demonstrates how using the intermediate update step makes the code more interpretable. In turn, this made the later adaptation to a 2D topology easier. In this method, the communication can be done in just two MPI statements:

```
1 // Cycle edges
2 MPI_Sendrecv_replace(top_edge, N_COLS_TOTAL + 2, MPI_INT,
     rank_down, 42, rank_up, 42, MPI_COMM_WORLD, &status);
3 MPI_Sendrecv_replace(bottom_edge, N_COLS_TOTAL + 2, MPI_INT,
     rank_up, 42, rank_down, 42, MPI_COMM_WORLD, &status);
```

All ranks can store the top and bottom edge in buffers of the same name which are then simply cycled in the correct direction across adjacent ranks. The approach for 2D domain decomposition was similar, with Figure 2 illustrating how this is made more elegant by updating each rank's halo first, and then cycling the corresponding edges and vertices of all the halos. The challenge for the 2D decomposition was to embed the ranks in a coordinate system such that each rank could communicate to its correct adjacent ranks, but the two-step approach also made this more manageable.

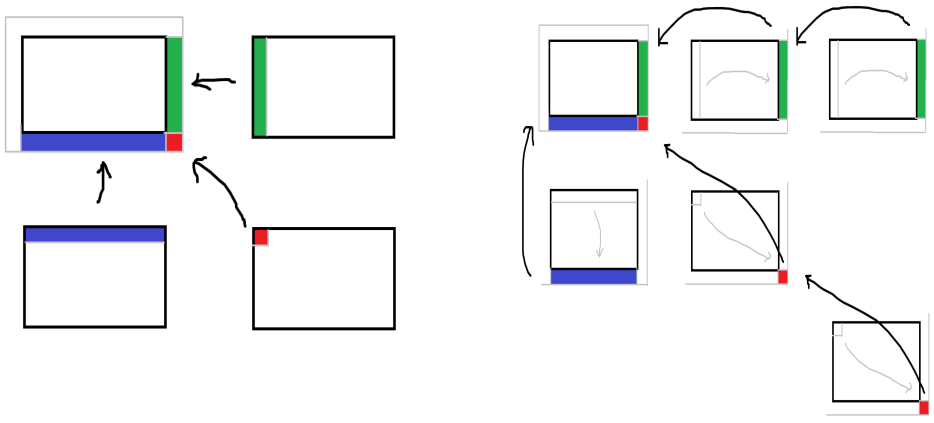
Once this worked for grids whose sizes were integer multiples of the topology sizes, it was extended to grids of any size. This involves allocating differently sized chunks for the ranks. In turn, if this is done properly then it does not affect the coherence communication of halos, since corresponding halo edges will still always have the same size. However, the scattering and gathering of the chunks at the start and end of the programme becomes more involved. The sizes of each chunk are stored in a 2D array, and then the partial sums or 'offsets' are computed from this, which enables the use of the `MPI_Scatterv` function to distribute across the differently sized buffers, and the same calculations are passed to the `MPI_Gatherv` function at the end.

The domain decomposition was then integrated with OpenMP function-



(a) Required communication (b) Two-step implementation

Figure 1: Illustration of the communication between chunks in the 1D topology.



(a) Required communication (b) Two-step implementation

Figure 2: Illustration of the communication between chunks in the 2D topology.

ality to employ threading within each rank. This consisted of enabling the convolution, rule evaluation, and periodic boundary update to be performed by mutliple threads sharing the relevant array. For this purpose, statements such as `#pragma omp parallel for collapse(2)` were used to distribute the serial routines, which mostly consisted of for loops, amongst the threads (with `collapse(2)` implementing this for nested for loops). Additionally, the setting of shared memory required the use of `#pragma omp barrier` statements. For instance, for the separable convolution algorithm, this was important because all entries of the intermediate row-convolved matrix are required to accurately perform the column convolution. The barrier prevented idle threads from starting the column convolution iterations until the row convolution was complete.

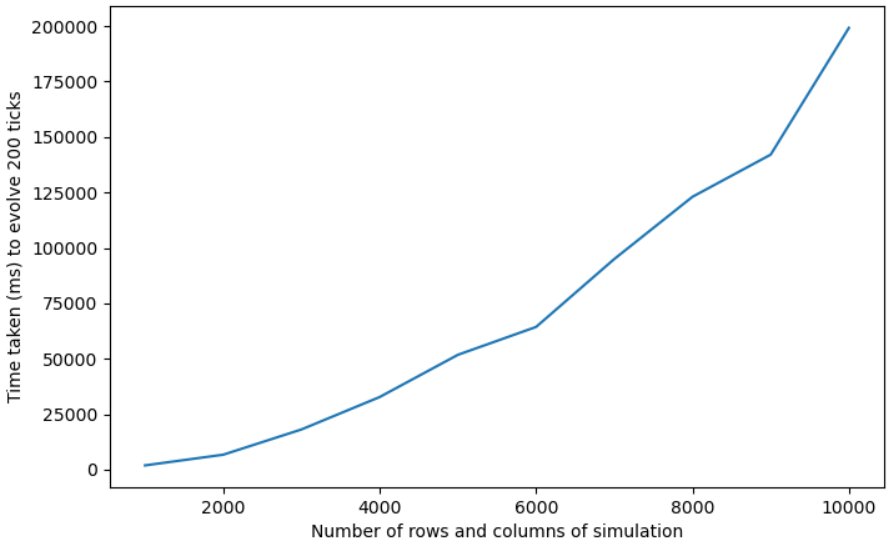


Figure 3: Timing plot for single thread code

Once the hybrid parallel program was written and verified, it could be investigated via timing scripts and compared with the single-thread algorithm. This was carried out using Tier 2 services on the CSD3 computing clusters. All timing scripts were performed on a single Cascade Lake node, each of which has 56 cores, enabling various configurations of domain decomposition and threading to be investigated. This was controlled via the configuration of

the SLURM submission script (found in the root of the repository), in which the `ntasks` parameter controlled the number of MPI ranks used, while the `cpus-per-rank` parameter configured the number of OpenMP threads assigned to each rank; moreover these were assigned efficiently by setting the environment variable `export I_MPI_PIN_DOMAIN=omp:compact` which ensured that the threads assigned to each MPI process were in close proximity, thus reducing the communication overhead.

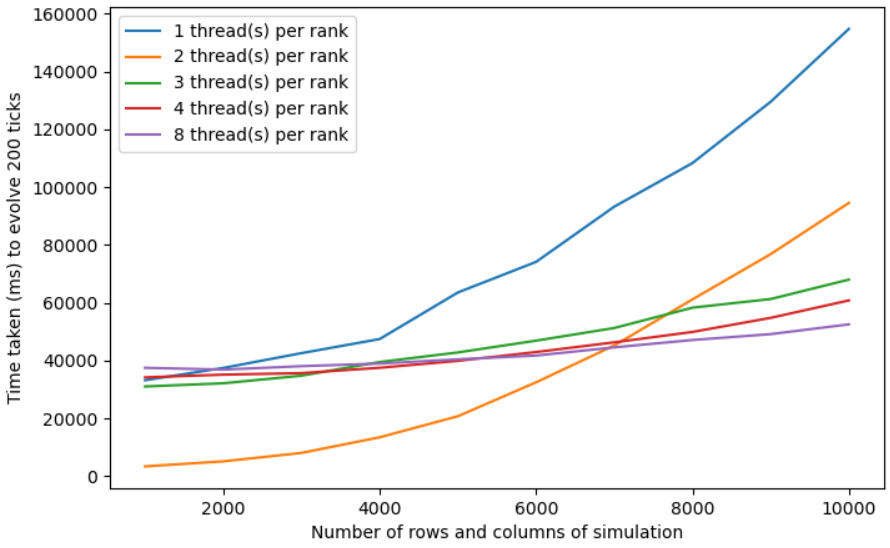


Figure 4: Timing plot for 2x2 domain decomposition

For comparison, Figure 3 shows how the time taken to simulate 200 ticks scaled with the number of rows of a square simulation, for the single-threaded implementation of the code. This plot shows the expected trend of time taken increasing as $O(n^2)$ with respect to the simulation size n. Moreover, note that for small simulations the time taken appears to be very small with respect to the scale of the timing for the full experiment. This is unlike the situation for the parallelised program. All subsequent timing experiments were done with the same maximum simulation age 200, as well as the same progression of simulation size. Moreover, the investigation of domain decompositions was limited to square layouts.

When inspecting Figure 4, we see an enormous increase in the time taken

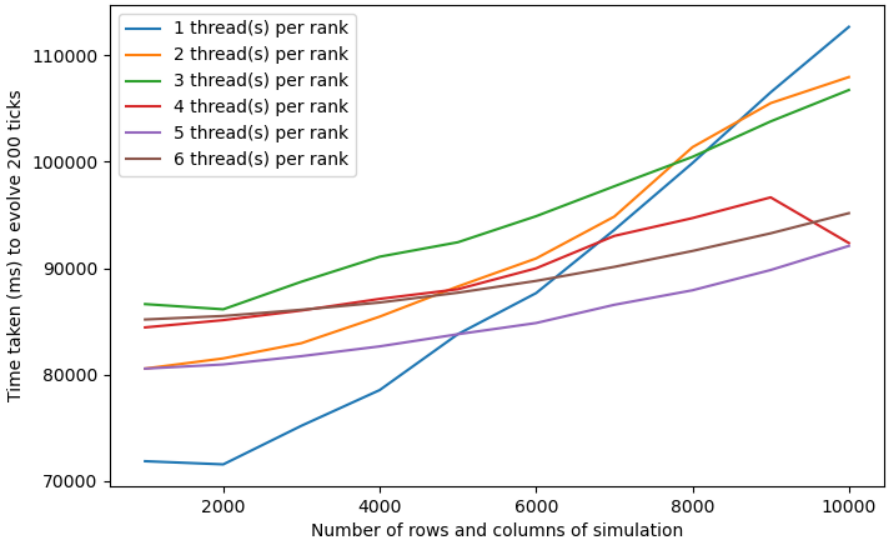


Figure 5: Timing plot for 3x3 domain decomposition

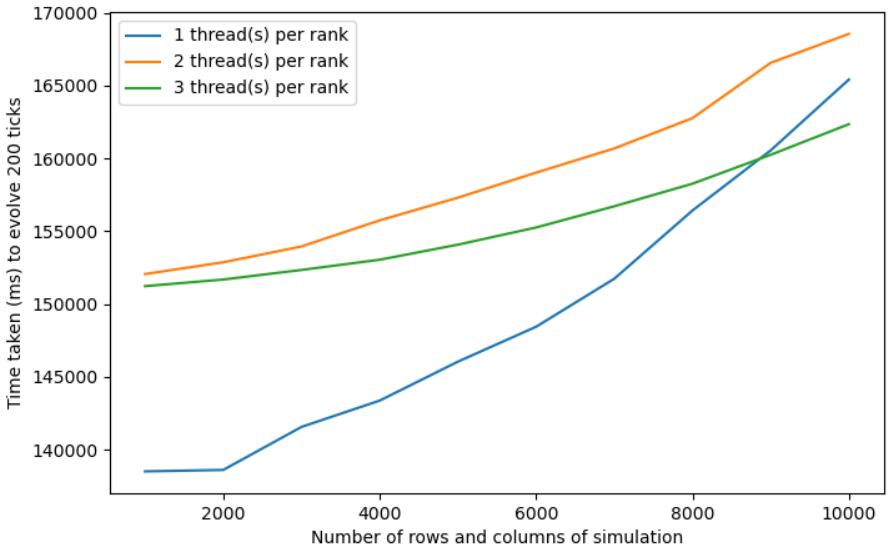


Figure 6: Timing plot for 4x4 domain decomposition

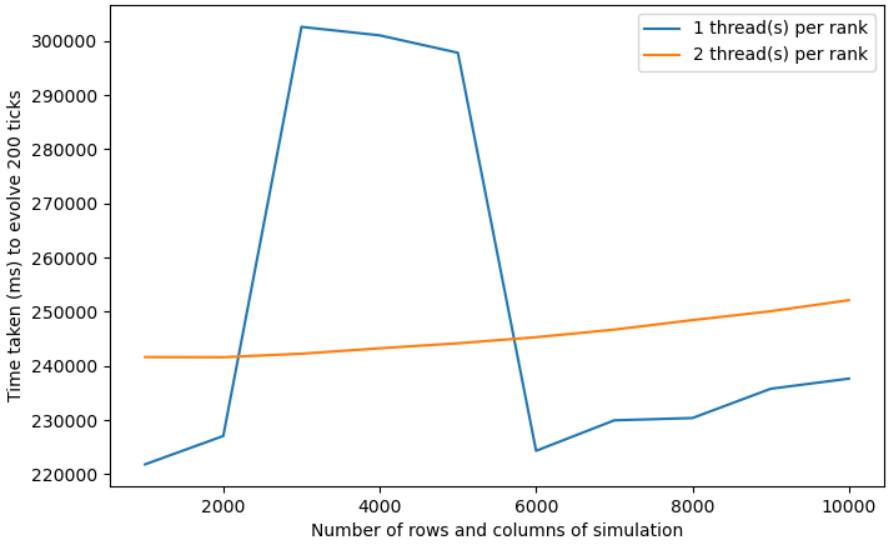


Figure 7: Timing plot for 5x5 domain decomposition

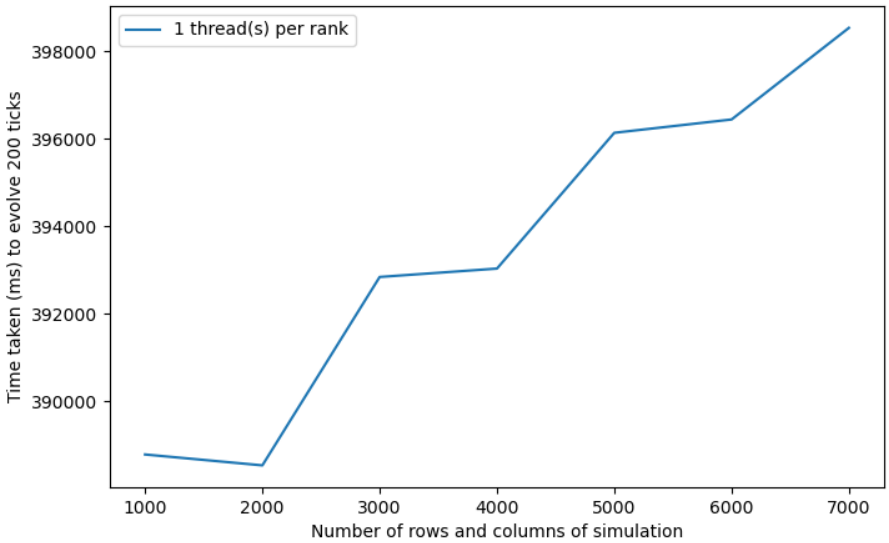


Figure 8: Timing plot for 6x6 domain decomposition

for the smallest 1000 by 1000 simulation for four single threaded MPI ranks – from 2 to 33 seconds. This represents the increase in 'fixed' computational cost caused by the collective communication, especially at the start and end of the simulation to distribute and recollect the simulation chunks across the ranks. However, we continue to observe an apparent $O(n^2)$ scaling law of the time taken with respect to the simulation size. Furthermore, in this implementation, the exact coefficient of this scaling law is considerably less than that of the single-threaded program, as evidence by the result for the largest simulation which is around 25% faster than before. This indicates that the domain decomposition has successfully increased the performance of the parts of the code that scale with respect to the size of the simulation. Distributing each MPI process across multiple threads causes further improvements, with speed-ups of roughly 38%, 60%, and 66% for 2, 4, and 8 threads per rank respectively on the largest simulation, though this indicates diminishing returns as the number of threads per rank increases.

As the number of ranks increases this trend continues with greater communication overhead apparent in the smaller simulations, but smaller scaling coefficients as the simulation grows in size. For large numbers of ranks the fixed computational cost becomes dominant in the experiments, with topologies of 5x5 and beyond taking longer to compute the smallest simulations than the single-thread algorithm takes to compute the largest simulation. This shows that in practice, the parallelisation parameters must be matched to the problem size; these topologies will likely become optimal for much larger simulations.

4 Summary

This project was an incredibly useful exercise in learning how to use the CSD3 compute resources, which comes with its own set of challenges such as setting up the correct module environment, configuring SLURM submission scripts correctly, and also time management to ensure productivity is maintained in the context of the job queuing system. Additionally, the focus on developing highly-optimised code required reviewing and practising the fundamentals of the C++ language, as well as the best practices studied in the C1 course. This was crucial in ensuring that the codebase remained manageable despite the complexity of the parallel code.

There are aspects of the code that could be extended upon in the future.

For instance, the single-thread, 1D, and 2D domain decomposition scripts could all be integrated to produce a single code that can run in any configuration specified at runtime. This would likely improve the performance compared to using the 2D decomposition for 1 by m chunk layouts, since the 1D implementation invokes far less fixed cost in the form of setting up the Cartesian grid of chunks. Additionally, a routine could be added to automatically find a suitable chunk topology given the raw number of mpi ranks and the size of the simulation. Similarly, the possibility of cache resonance in Table 1 could be investigated further, and a feature could be added to automatically pad the simulation with redundant cells when such problematic simulation sizes are used.

References

[1] M. Gardner, "The fantastic combinations of john conway's new solitaire game 'life'," *Scientific American*, vol. 223, no. 4, 1970.

[2] E. Martin, "Playgameoflife.com," accessed Mar. 27, 2024. [Online]. Available: https://playgameoflife.com/

[3] D. E. Knuth, "Structured programming with go to statements," *ACM Comput. Surv.*, vol. 6, no. 4, pp. 261–301, dec 1974. [Online]. Available: https://doi.org/10.1145/356635.356640

[4] Wikipedia.org, "Separable filter," https://en.wikipedia.org/wiki/Separable_filter, 2022.

A Statement on the use of auto-generation tools

Auto-generation tools, such as GitHub or Microsoft Copilot, were not used at any stage during the development of the code within the repository of this project. Similarly, none of the content of this report was produced – or proofread – by modern Large Language Models such as ChatGPT at any point.

B High-Performance Computing Resources

This work was performed using resources provided by the Cambridge Service for Data Driven Discovery (CSD3) operated by the University of Cambridge Research Computing Service (www.csd3.cam.ac.uk), provided by Dell EMC and Intel using Tier-2 funding from the Engineering and Physical Sciences Research Council (capital grant EP/T022159/1), and DiRAC funding from the Science and Technology Facilities Council (www.dirac.ac.uk).

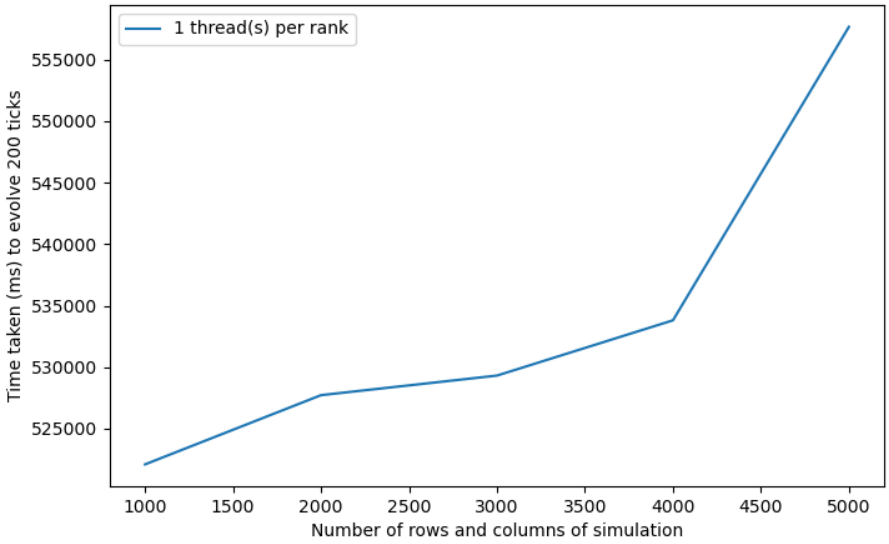


Figure 9: Timing plot for 7x7 domain decomposition