Prepared for submission to University of Cambridge

C2: Advanced Research Computing - Coursework Assignment

Adnan Siddiquei

University of Cambridge

E-mail: as3438@cam.ac.uk

Contents

1 Introduction

Conway's Game of Life...

2 Selection of Solution Algorithm and Prototyping

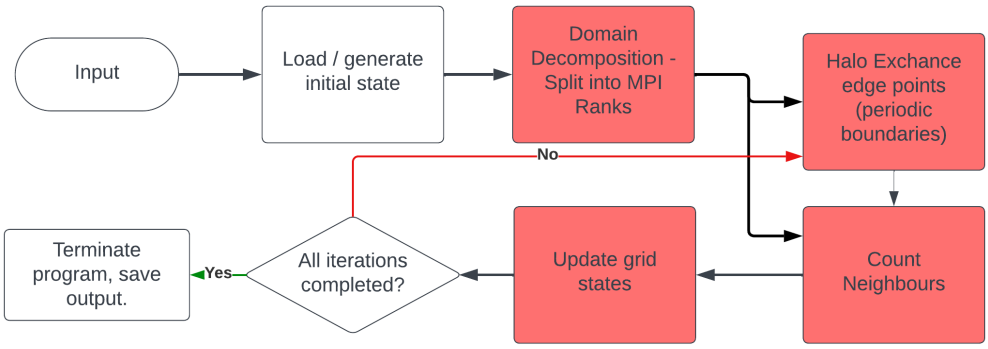

Figure 1: A high-level flow chart depicting the algorithm. The key points for efficiency considerations are highlighted in red, the considerations are discussed in the main text.

Fig.(1) shows the high-level algorithm being used to simulate Conway's Game of Life. The key points of consideration with regard to performance are highlighted in red. There are 3 domain decomposition methods available (column, row and block decomposition) as well as multiple ways to count neighbours and update grid states. As indicated in Fig.(1), communication overheads between MPI ranks can be hidden behind computation by using non-blocking communication and counting whichever neighbours are available at the time. Likewise, the actual method of counting neighbours and updating the grid can be done in a variety of ways, each with their own merits. All these considerations are discussed thoroughly in this section, and then they are investigated in the next section.

2.1 Domain Decomposition and Neighbour Counting Algorithm

Domain decomposition can be done by either strips or blocks. Strips has the advantage that it is simpler to implement but will have more memory needing to be passed between boundaries, with the opposite being true for blocks. Therefore, the choice of domain decomposition will need to be tested to see which is more efficient as the data size and number of MPI ranks increases. However, the overheads of communication can be effectively nullified if the communication is hidden behind computation, as such, this also needs to be explored to see what proportion of the communication can be hidden.

The neighbour counting algorithm has 2 possible implementations, a simple convolution and a separable kernel.

- **Simple convolution.** Loop through each cell, and count the number of neighbours. This is equivalent to a 3x3 convolution with the below kernel.

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{2.1}$$

  This requires a total of $8N^2$ addition operations (the $9N^2$ multiplication operations that would usually occur can be ignored because the kernel only contain 1s and 0s so we can skip the multiplication operation in this special case).

- **Separable kernel.** If the following kernel is used instead,

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{2.2}$$

  then the kernel becomes separable. This means that the kernel can be split into two 1D identical kernels $\begin{bmatrix} 1 & 1 & 1 \end{bmatrix}$ and applied in two passes which yields $2N^2$ addition operations per pass, yielding a total of $4N^2$ operations [1]. After another N^2 addition operations, to correct for replacing the central 0 with a 1, the total number of operations becomes $5N^2$. Therefore, the separable kernel method yields a theoretical speedup of $8/5 \approx 1.6$. However, due to strided memory access on the vertical pass, the speedup will be less than this. Although, the vertical pass can be made more efficient by transposing the grid and making the data contiguous, it will need to be investigated whether the additional overhead of transposing the grid yields any speedup.

The simple convolution method would prefer column-wise decomposition as it would reduce the number of cache misses when counting the 6 neighbours that are not horizontally adjacent to the current cell. This is because shorter rows would reduce the distance in memory between a cell and it's vertical neighbour. Theoretically, the separable kernel should not have this issue if the separable convolution can be applied in two horizontal passes with a transpose in between.

2.2 Hiding Communication Behind Computation

To the extent of hiding communication behind computation, the separable convolution allows naturally for the horizontal pass to be done first, and then the vertical pass to be done after the vertical halos have been exchanged. Doing the vertical pass first is also possible but likely to be

less efficient due to the strided memory access. This would, however, only hide communication if the horizontal halos arrive before the vertical halos which is not guaranteed. Given that the horizontal halo exchange involves the exchange of non-contiguous data (assuming the grid is represented as a 1D array in row-major order), unless the horizontal halos are smaller than the vertical halos (which would be the case in row decomposition), the communication will likely not be hidden behind computation. There are other factors such as physical positioning of the MPI ranks and the network topology that will also affect the communication overheads between MPI ranks, so it is hard to predict how much time could be consistently saved by hiding communication in this method.

The alternative option for hiding communication is to count the neighbours with the simple convolution for all the cells that are not on the boundary, and then count the neighbours for the cells on the boundary once the halo exchange has been completed (note, this method is later referred back to as the 'SimpleIO' method). The downside of this is that it involves a lot of non-contiguous memory access, which is not ideal for performance, but how this method compares to the previous method as the simulation scaled will need to be investigated.

2.3 Updating Grid

Given the neighbour count and the current state of the grid, there are a few possible ways to update the grid [2].

- **`if` statements.** This is the most straightforward implementation. The minimum number of `if` statements required to implement the rules of Conway's Game of Life is 3, and it is unlikely the CPU will be able to optimise this to any degree with branch prediction, so this method will likely be the slowest.
- **Bitwise operations.** This method removes all `if` statements and replaces them with inline bitwise (`&&` and `||`) operations.
- **Lookup table.** Given that there is only 18 states a cell can be in (dead or alive, with 0 to 8 neighbours), a lookup array can be used to update the grid with the new state of each cell, given one of the current 18 states.

The efficacy of these methods will need to be tested to see which is the most efficient.

2.4 OpenMP with MPI

The final consideration is to assess how to use OpenMP with MPI. Not all loops will be infinitely parallelisable, more threads won't always yield a linear speedup. Likewise, more MPI ranks yields more communication overheads, and as such there will be trade-offs in this respect. The trade-off between OpenMP and MPI will need to be investigated empirically to see what the best combination is.

3 Development, Experimentation, Profiling and Optimisation

This section discusses the experimentation and profiling of the different methods discussed in the previous section (Section (2)). Most of the experimentation was done on a Macbook M1 Pro (10 cores; 16GB RAM; 128 KB of L1 data cache; 192 KB of L1 instruction cache; 64 Byte cache line size; 4 MB of L2 cache), which is why the number of OpenMP threads for local experimentation was limited to 10.

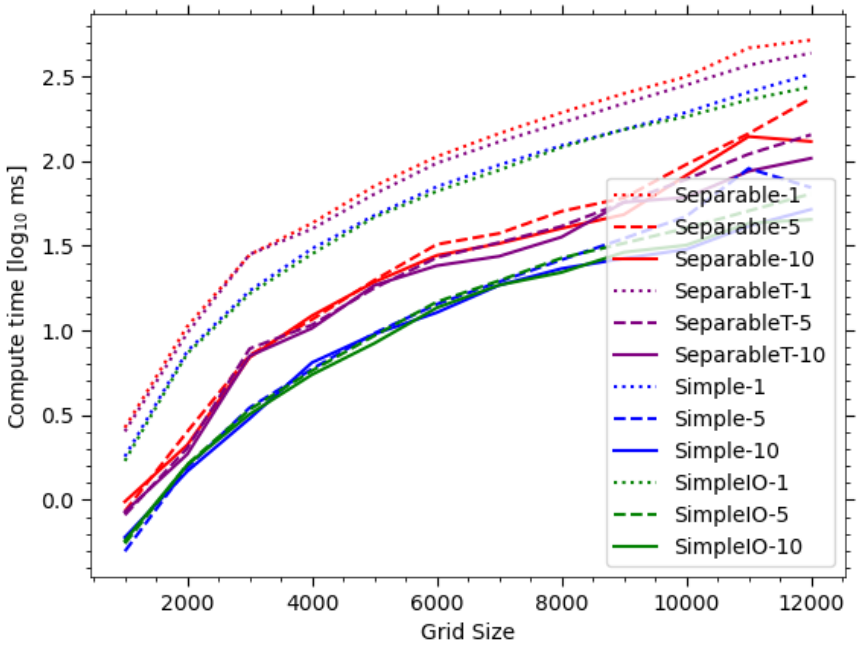

Figure 2: A figure showing the compute time for 4 of the convolution methods for counting neighbours. The colour of the line indicates the method used, and the line style (dotted, dashed, solid) indicates the number OpenMP threads used (1, 5 and 10 respectively). Only 1 MPI rank was used. The x-axis shows the size of one dimension of the square grid. The Simple and Separable methods are as described in (2.1) and the SimpleIO method is as described in (2.2)) The SeparableT method is the Separable method done in 2 horizontal passes with a transpose in between, except the time on the graph is just the compute time to repeast the first horizontal pass twice (i.e, it is the time for the actual SeparableT method minus the tranpose compute time). This was done for simplicity, to assess whether a tranpose operation needed coding. The compute time is averaged over 3 runs.

3.1 Profiling the Convolution Methods

Fig.(2) shows the compute time across 4 different convolution methods for counting neighbours and how they scale with `grid_size` and `OMP_NUM_THREADS`. The first observation is that both the simple methods are faster than both of the separable methods, which is at face value, unexpected. Furthermore, there is a negligible difference in speed between the Simple and SimpleIO method which is a positive outcome which means the SimpleIO method is a good candidate for hiding communication overheads, as the vast majority of the computation can be done while waiting for the communication to complete. It is worth noting that all of these methods show diminishing returns as the number of OMP threads increases.

Fig.(3) shows that as the number of OMP threads increases, the compute time decreases, but with diminishing returns. The jump from 1 to 2 threads yields a 2x increase but jumps following this do not yield speed increases in the same proportion. At about 8 threads, the returns on increasing the number of threads become negligible, which is useful information for optimising the MPI rank to OMP thread ratio.

The analysis thus far has yielded the following conclusions: the SimpleIO method is a good candidate for counting neighbours and hiding MPI communication overheads; and 8

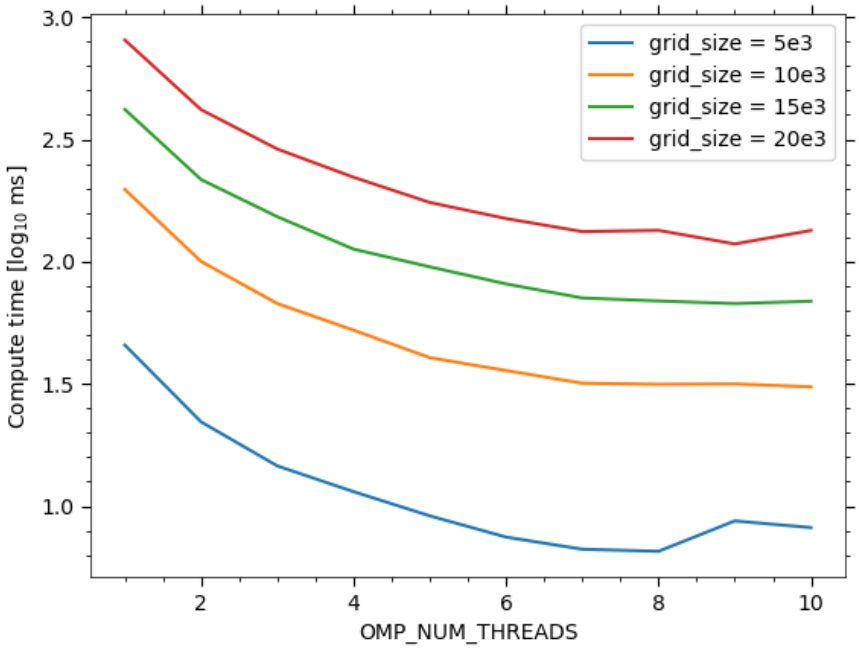

Figure 3: A figure showing how the compute time varies for the SimpleIO method as the number of OMP threads is changed. 1 MPI rank was used, and compute time is averaged over 3 runs.

may be a reasonable cap on the optimal number of OMP threads.

3.2 Profiling the Update Methods

Fig.(4) explores the update methods discussed in Section (2.3), with a clear conclusion that the lookup method is the fastest, and as expected, the `if` method is the slowest. This makes sense, due to the (effectively) random nature of the simulation, the CPU will have little success in branch predictions, further slowing the `if` method down.

References

[1] Steve Eddins, *Steve on Image Processing with MATLAB*. Available at: https://blogs.mathworks.com/steve/2006/10/04/separable-convolution/ [Accessed: 8-Mar-2024].

[2] Srdjan Delić, *Branchless programming — Why your CPU will thank you*. Available at: https://sdremthix.medium.com/branchless-programming-why-your-cpu-will-thank-you-5f405d97b0c8 [Accessed: 17-Mar-2024].

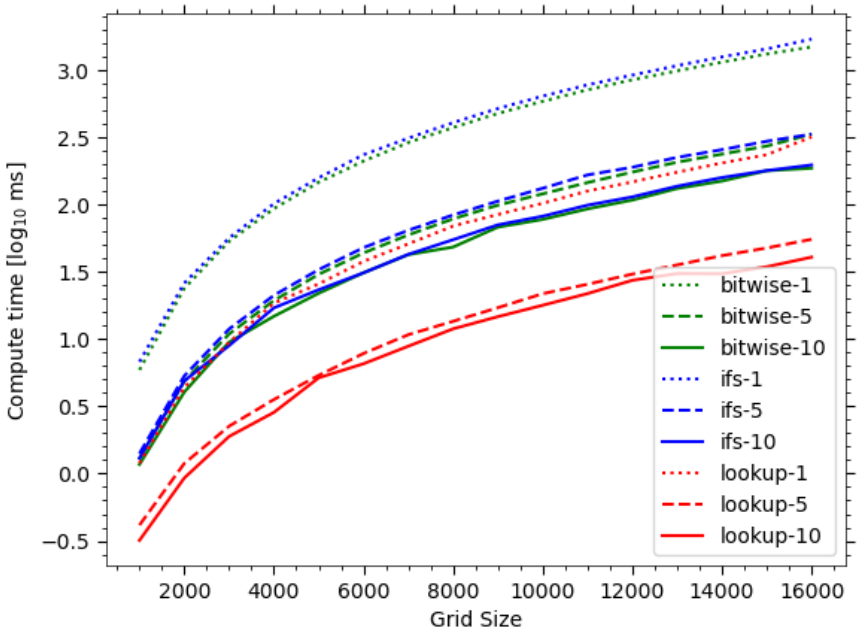

Figure 4: A figure showing the compute time for the 3 the update methods discussed in Section (2.3). The colour of the line indicates the method used, and the line style (dotted, dashed, solid) indicates the number OpenMP threads used (1, 5 and 10 respectively). Only 1 MPI rank was used. The compute time is averaged over 3 runs.