Game of Life: Project Report

Brief description of the code:

The implemented Game of Life code is comprised of 3 steps: pre-processing, MPI Paralleled iteration, and post-processing as shown below.

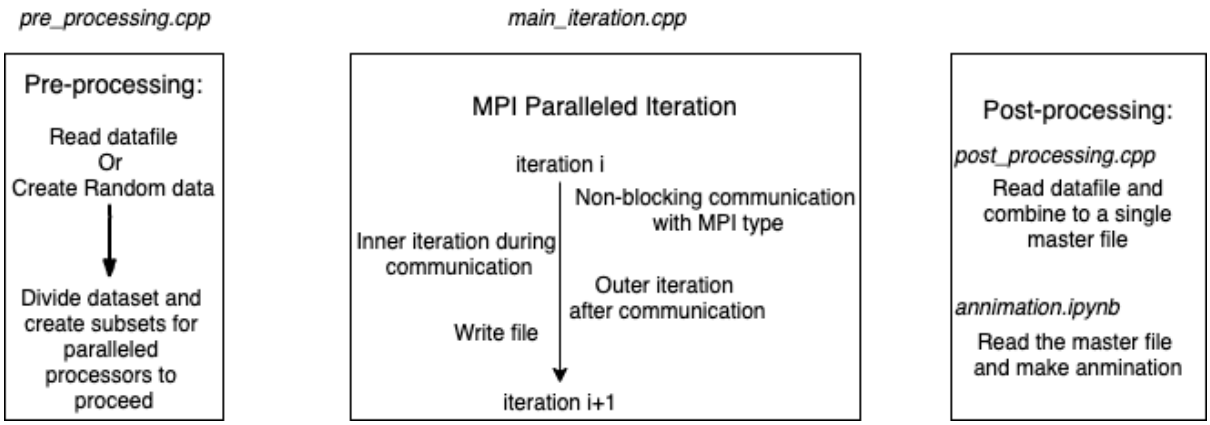

Figure 1: Framework of the coursework submission

Some characteristic of the main algorithm includes:

- Do partial iteration when waiting for the non-blocking communication to finish;
- Take in 1/0 at execution time to instruct Periodic or Non-periodic;
- Read all data except number of iterations from the files generated by *pre_processing.cpp*;
- Capable to handle situations where small number of processors are involved.

Discussion of the algorithm:

1. Domain decomposition strategy

   Two domain decomposition strategies are being proposed: grid and striped. Grid decomposition has higher efficiency when the matrix has similar numbers of rows and columns, while opposite for striped decomposition. Under the presumption that the number of rows and columns will be similar, this project chose grid decomposition.

2. Iteration logic

   There are two iteration logic being proposed: add padding, and separate outer cells.

   Add Padding logic creates a (n+2)*(m+2) field, that receives data from outside and update onto the padding area. This logic is straightforward and all the effective data (non-padding area) can be updated in one go.

   Separate Outer Cells logic iterates the inner cells and outer cells separately. Outer cells will be calculated with a different function that takes the received data as inputs.

   The efficiency is similar. But Separate Outer Cells logic can do iteration in 2 steps, which is more efficient when MPI non-blocking communication method is used.

3. Generation switch

   As the code is written in a for loop, the updated generation should be the to-be-updated generation of next iteration. There are some ways doing it efficiently: use vector.switch, or use pointer that switches the address where it's pointing. This project implemented the latter one.

4. Communication method

   MPI non-blocking communication is used. Isend and Irecv with waitall are used.

   To improve the efficiency, the processors shouldn't be idling when waiting the communication to complete. So, the inner iteration that doesn't need the information from other processors can be conducted during waiting.

   After all the sends and receives done, the outer cells can be updated.

Analysis and discussion with HPC:

A modified version of main_iteration.cpp that can generate random initial condition independently is fed into HPC to solve large matrix. The algorithms of the HPC version is same with *main_iteration.cpp*, except the initialisation method. Thus, it's reliable to examine *main_iteration.cpp*'s capacity with the HPC version. After running through several controlled groups, some data and observations are summarised as following (original data included in the datasheet.xlsx):

1. Runtime V.S Number of Proccessors

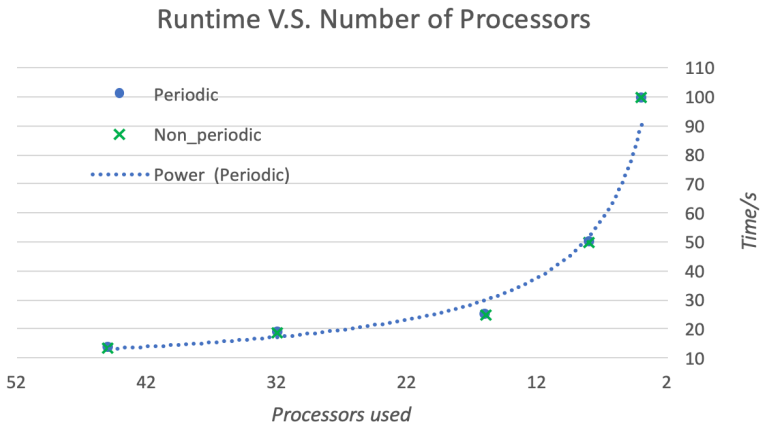

*Figure 2: Runtime V.S. Number of processors (30000*30000, 50 iterations; Periodic and Non-Periodic)*

For same matrix size and number of iterations, programme runtime with different processors is compared against each other (Figure 2).

It's found that the runtime has a power increment as the number of processors decreases. When the number of processors decreases from 8 to 4, the runtime doubles. But this trend is less significant when more and more processors are involved, which implies that the balance

between coordinating more processors and the size of dataset handled by each processor is important to consider: when more processors are involved, initialisation and communication work are increasingly complex and time consuming, while the dataset that needs to be dealt with by each processor becomes smaller and less time-consuming.

The time consumed by periodic and non-periodic boundary condition is close, which implies that the implemented algorithm requires similar computation for both conditions.

2. Runtime V.S Number of Cells

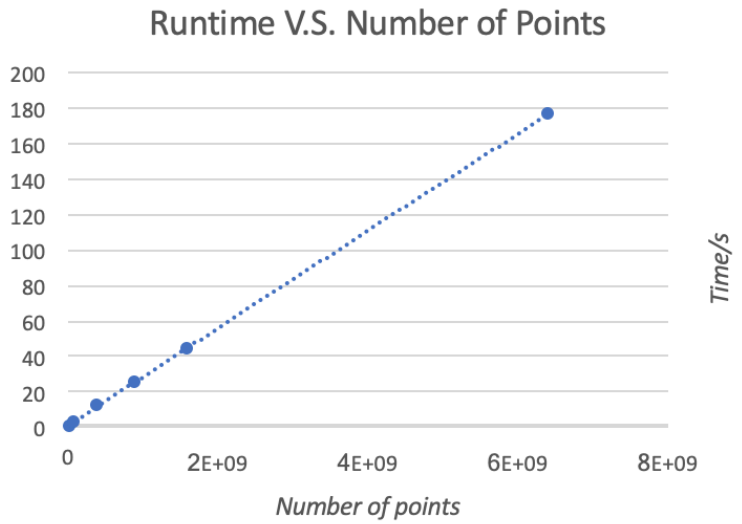

Figure 3: Runtime V.S. Number of points (50 iterations, 16 processors)

Matrices with different size (from 5k*5k to 80k*80k) are fed into HPC with same number of processors and iterations as shown in Figure 3.

The result is sensible that the time consumed increases linearly with the number of points involved.

3. The influence by Data distribution (proportion of alive cells)

*Table 1: Statistic of different proportion of True (alive) at initial condition (20k*20k, 16 processors, 50 iterations)*

	Peak memory used	Average NCPU	Average time	Longest time
1% alive	0	12.33	6.84	6.85
10% alive	0	13.1	8.1	8.14
25% alive	1	14.43	11.26	11.35
33% alive	1	14.45	11.61	11.73
50% alive	1	16.01	11.85	12.13
66% alive	0	13.88	7.7	7.75
75% alive	0	12.1	7.45	7.47
90% alive	0	11.89	7.26	7.32

99% alive	0	12.44	6.82	6.85

When varying the percentage of true (alive) cells, it's found from the visualised simulation result that when the initial condition is trending to be extreme, i.e., 99% alive or 1% alive, the cells would trend to extinct. Thus, a healthy society (a simulation field that more cells would remain alive) should have proper portion of alive cells, so that cooperation and resource can be well-balanced.

From the statistical perspective, table 1 implicates that generally the running time is longest when half of the cells are alive and half are dead. When the percentage of alive cells deviated from 50%, the memory used, average NCPU used, and the average time consumed are larger. It's an interesting observation. As the implemented algorithm iterates through and make changes onto every point each time, the status of the cell shouldn't influence the time of operation. The fact that operation time varies implicates that changing data values does consume more resource comparing to keep it original value. So, when the alive cells are approximately 1% or 99% percent of the total cells, the cells converge to a stable status quickly, thus less resource will be used to modify the cell condition.

4. Runtime V.S Number of Iterations

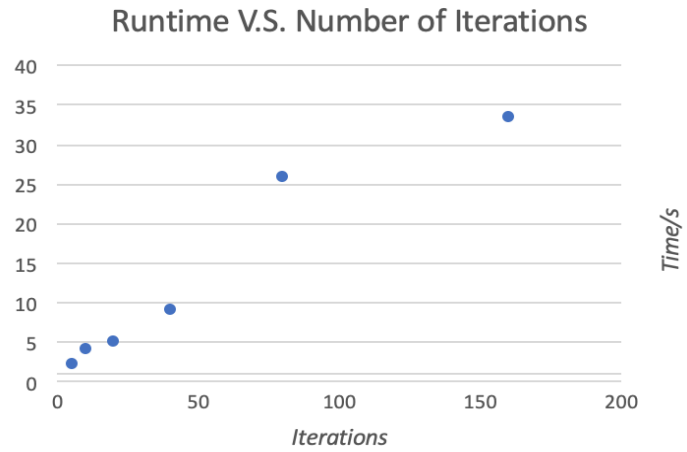

*Figure 4: Runtime V.S. Number of iterations (20000*20000, 16 processors)*

When comparing the number of iterations, the runtime increases linearly in general with the number of iterations, subject to fluctuation. The fluctuation can be caused by the initialisation time and processor factors. Meanwhile, this implies that after a certain iteration, the systems fall into a relatively stable status: only a small portion of cells will change their status. So, it consumes less time and resource to conduct the iterations in the latter part.

Conclusion

By paralleling the code, MPI does reduces the time if properly implemented. But the scope of parallelisation should be properly designed to balance the computational load and additional communication process, initialisation process.