MPI Programming Assignment Report

In this project, the task is to implement the life of game program in both sequential and parallel versions. Results and performance of sequential code are the baseline and reference outputs. In the parallel code, program will be executed on a designated number of processes. It is SPMD style of parallel computing. We will compare the results' correctness and performance between sequential code and parallel code. Since "board" will be distributed to multiple processors. We need to consider how to decompose the board and solve the load-balance and communication issue. In this report, ways to do decomposition and how to balance load will be discussed. Additionally, performance of parallel code would be discussed further.

• Decomposition

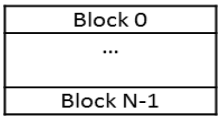

Figure 1: 1D-decompostion

- 1D-decomposition

  Either cutting the horizontal axis or vertical axis works for 1D-decomposition. If we cut the board at column side, then the addresses of cells in the new small blocks are no longer contiguous. Therefore, cutting the row side is selected shown in figure 1.

  If number of rows are Ylim, then average number of rows in block i is Ylim/N where N is number of processors. Cases are that Ylim cannot be divided by N. Therefore, we need to decide how many rows every block has. Due to load balance issue, I tried to make sure that every block has the same number of rows. If there is non-zero residue(R) for Ylim/N, the first R processes have blocks with [Ylim/N]+1 rows and the rest has [Ylim/N]. This guarantees the total number of rows for all N processes equals Ylim.

Figure 2: 2D-decompostion

- 2D-decomposition

  In this situation, both row and column will be cut like figure 2. N blocks are decomposed into n*m. Size of block i will be y_i *x_i. Summation of y_i is Ylim and summation of x_i is Xlim where Xlim and Ylim is the row and column length of the board. Shape of every block should be as close to square as possible due to load balance and performance issues. Generally, a n*m decomposition will generate about (n-1)*2*Xlim+(m-1)*2*Ylim number of data transimission during one iteration. To reduce the time and data for transmission, m+n should be as small as possible if Xlim is equal to Ylim. As we all know that only when m and n are closer, m+n is smaller given N=m*n fixed. So program will find decomposition where m and n are close enough. One straightforward way is to do factorization. In a while loop, try: n = n+1; m=N/n. If residue = 0, then record this decomposition. Iterations will continue until n is equal or larger than m. In my code, the program will firstly figure out a best decompisition given any number of processors not limit to power of 2.

  Way to figure out size of every sub block is similar to that in 1D-decomposition. Make sure that row/column of each block is similar or almost the same to each other. The difference should be less or equal to one.

• Load Balance

To get a better performance, we need to ensure that work load for every processor should be close to each other. I focused on how to do decomposition and communication evenly. About decomposition, make sure that number of cells in every block s partitioned evenly (see Decomposition section). Especially, in 1D-decomposition, when column size cannot be divided by number of whole processes, residue(R) should be partitioned evenly on the first R processes. In 2D-decomposition, whether size of board would be divided by the designated n and m should be

paid attention to as well. If not, deal with residue the same as that in 1D-decomposition. About communication, way to update data discussed in class guarantees that almost all processors' data transmission times and data quantity are controlled. Therefore when one rank reading data from file sends data to other processors, Bcast functions are used to reduce number of Send and Recv functions being used. In 2D-decomposition, when updating boundaries, first buffer all columns' data in an array before sending to neighouring blocks so that only one Send is needed.

- Performance

Table 1 and 2 show the performance of running final.data with different code versions and configurations. Since execution time is small, every configuration is run for 5 times and average of performance are computed as final results. It is expected that when resources (number of processes) doubles, the execution time could be half. That is linear speedup expectation. In 1D-decomposition, when resource doubles, the execution time becomes almost half. However, in 2D-decomposition, when number of cores increases from 1 to 16, the speedup is close to linear. When there are 32 cores, execution time is decreased a little but not to half. In addition, performance for serial code is better than that for parallel code with 1 core when running in login node.

About 1D-decomposition, reasons why it is not speedup is not exactly linear are that (1). there are overhead such as communications and initializations which could not be fastened by doubling resource. According to Amdahl's Law, the performance would be limited by overhead; (2). as cores increases, number of data transimission increases as well; since overhead for preparing sending/receiving is longer than real data transmission, the performance would be interfered too. About 2D-decomposition, reasons of non-linear speedup has the addtionally following reasons: (1). boundary updates between iterations are more complicated; all corners and 4 edges need to be sent to its 8 neighbours while in 1D version, only 2 edges need to be updated; (2). two of the edges' data are not address-contiguous which needs to buffered first and 4 corners only have one data to be updated but needs an independent Send/Recv. Observing fixed processes' number and different configurations, we could find that the more the nodes are configured, the worse the performance is. And additionally, when number of nodes is fixed as well, the performance is better if cores are assigned more evenly. These might result from data transimission and communication takes more between different nodes. Possible reasons why serial code is faster than parallel code with 1 core are that (1). there are some if statements in parallel code for data transmission which means there are more costs if compiler cannot predict branch taken cases; (2). There are extra more initialization and memory allocation for decomposition and data transmission.

Table 1: Sequential code execution time

Sequential code execution time (s)	1.95 (running at login node)	2.196 (sbatch shell file to submit job)

Table 2: Parallel code execution time

Configuration			1D-decomposition Execution Time (s)	2D-decomposition Execution Time (s)
Number of processes	Number of nodes	Number of processes per node		
32	2	16	0.0903585	0.1403795
32	2	20	0.1062185	0.146016
32	3	11	0.0920095	0.1405235
32	4	8	0.1120905	0.159999
16	1	16	0.173713	0.161745
8	1	8	0.293066	0.3014875
4	1	4	0.5812945	0.5817305
2	1	2	1.120049	1.12463
1	1	1	2.1990475	2.209973667

Note: when there are 32 cores,number of processes assigned per node is 20, and there are 2 nodes, that means one node is 20 cores and the other is 12.