Report for Assignment 2

20191165 Sim Eunyeong

1. How to execute parallelization

Algorithm: Parallelized Game of Life

```
0: /* Set MPI environment */
1:          MPI_Init(&argc, &argv);
2:          MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
3:          MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
4:          int dim[1] = {comm_size}, int period[1] = {0};
5:          MPI_Cart_create(MPI_COMM_WORLD, 1, dim, period, 0, &RING_COMM);
6: Read_file(argv, board_size, gens, ghosts, s_board);
7: Parse string type input board into board[board_size][board_size] 2D array. whose type is bool
8: /* Decompose initial board */
9:          int my_row, my_col, idx_in_board;
10:         bool my_board[my_row][my_col];
11:         Copy board[idx_in_board:idx_in_board+my_row][0:my_col] into my_board[0:my_row][0:my_col] (element-wise copy)
12: /* Define boundary buffers */
13:         bool upper_boundary[my_col],
            bool lower_boundary[my_col];
14:         int n_upper_ghost, int n_lower_ghost;
15:         int n_upper_comm, int n_lower_comm;
16:         Copy board[idx_in_board-1][0:n_upper_ghost] into upper_boundary[0:n_upper_ghost];
17:         Copy board[idx_in_board + my_row][0:n_lower_ghost] into lower_boundary[0:n_lower_ghost]
18: Deallocate board[board_size][board_size];
19: /* Communicate metadata for size of comm section(n_upper_comm, n_lower_comm) in each node's boundary buffer */
20:         int metadata[2];
21:         MPI_Cart_shift(RING_COMM, 0, -1, &source, &dest);
22:         if(my_rank == 0)
23:             MPI_Sendrecv(&n_lower_comm, 1, MPI_INT, 1, 130, &metadata[1], 1, MPI_INT, 1, 110, RING_COMM, &status);
25:         else if(my_rank == comm_size -1)
26:             MPI_Sendrecv(&n_upper_comm, 1, MPI_INT, my_rank -1, 110, &metadata[0], 1, MPI_INT, my_rank-1, 130, …);
27:         else
28:             MPI_Cart_shift(RING_COMM, 0, -1, &source, &dest);
29:             MPI_Sendrecv(&n_upper_comm, 1, MPI_INT, dest, 110, &metadata[1], 1, MPI_INT, source, 110, …);
30:             MPI_Sendrecv(&n_lower_comm, 1, MPI_INT, source, 130, &metadata[0], 1, MPI_INT, dest, 130, …);
31: Do game
32: for g == 0 to gens-1
33:   /* Communicate comm section in boundary array*/
34:   if(ghost*ghost != my_col*2)
35:         if(my_rank == 0)
36:             MPI_Sendrecv(&my_board[my_row-1][my_col-metadata[1]], metadata[1], MPI_BYTE, my_rank+1, 290, &lower_boundary[my_col-
                    n_lower_comm],n_lower_comm,MPI_BYTE,my_rank+1,230, RING_COMM, &status);
37:         else if (my_rank == comm_size -1)
38:             MPI_Sendrecv(&my_board[0][my_col-metadata[0]], metadata[0], MPI_BYTE, my_rank -1, 230, &upper_boundary[my_col-
                    n_upper_comm],n_upper_comm,MPI_BYTE, my_rank -1, 290,RING_COMM, &status);
39:         else
40:             MPI_Cart_shift(RING_COMM, 0, -1, &source, &dest);
41:             MPI_Sendrecv(&my_board[0][my_col-metadata[0]], metadata[0], MPI_BYTE, dest, 230, &lower_boundary[my_col-n_lower_comm],
                n_lower_comm, MPI_BYTE, source, 230, RING_COMM, &status);
42:             MPI_Sendrecv(&my_board[my_row-1][my_col-metadata[1]], metadata[1], MPI_BYTE, source, 290, &upper_boundary[my_col-
                n_upper_comm], n_upper_comm, MPI_BYTE, dest, 290, RING_COMM, &status);
43:   /* update my_board */
44:   bool updated_my_board[my_row][my_col];
45:             for r = 0 to my_row -1
46:                 for c = 0 to my_col -1
47:                     updated_my_board[r][c] = get_next_state(my_board[r][c], alive_neighbors);
48:   Copy updated_my_board[0:my_row-1][0:my_col-1] into my_board[0:my_row-1][0:my_col-1] (element-wise copy)
49:   /* Communicate ghost section in boundary array */
50:             if(my_rank == 0)
51:                 MPI_Sendrecv(&my_board[my_row-1][0], my_col-metadata[1], MPI_BYTE, my_rank + 1, 370, &lower_boundary[0], n_lower_ghost,
                    MPI_BYTE, my_rank + 1, 310, RING_COMM, &status);
52:             else if(my_rank == comm_size -1)
53:                 MPI_Sendrecv(&my_board[0][0], my_col-metadata[0], MPI_BYTE, my_rank -1, 310, &upper_boundary[0], n_upper_ghost,
                    MPI_BYTE, my_rank -1, 370, RING_COMM, &status);
54:             else
55:                 MPI_Cart_shift(RING_COMM, 0, -1, &source, &dest);
```

```
56:             MPI_Sendrecv(&my_board[0][0], my_col - metadata[0], MPI_BYTE, dest, 310, &lower_boundary[0], n_lower_ghost, MPI_BYTE,
                source, 310, RING_COMM, &status);
57:             MPI_Sendrecv(&my_board[my_row-1][0], my_col - metadata[1], MPI_BYTE, source, 370, &upper_boundary[0], n_upper_ghost,
                MPI_BYTE, dest, 370, RING_COMM, &status);
58: end for
59: /* Write my_board to output file */
60:   if(my_rank =0)
61:       Write my_board to output file and then MPI_Send(&can_go, 1, MPI_INT, 410, RING_COMM);
62:   else if(my_rank == comm_size -1)
63:       MPI_Recv(&can_go, 1, MPI_INT, my_rank -1, 410, RING_COMM, &status)
64:       Write my_board to output file
65:   else
66:       MPI_Recv(&can_go, 1, MPI_INT, my_rank -1, 410, RING_COMM, &status)
67:       Write my_board to output file
68:      MPI_Send(&can_go, 1, MPI_INT, my_rank +1, 410, RING_COMM);
69: MPI_Finalize();
```

2. Description of parallelization strategy

a. 1D cartesian topology (line 0-5)

This Algorithm uses 1-Dimension cartesian topology.
To minimize communication volume of boundary cells in each node, 1D cartesian tolopogy is selected.
This is because communication occurs only in up to two outermost rows.
And This 1D cartesian topology is not cyclic. Because first and last node don't need to communicate each other.

b. All nodes are peer.

This algorithm is not master-slave based parallelization. All the nodes are peer. So to set up initial environment, all should read file, query it, partition initial board and set ghost cells independently. Line 6 to 17 implements those tasks.
Although there will be memory redundancy about initial board between all nodes, but it is deallocated right after finishing its usage (line18). It means that it is temporal memory.
If master-slave based parallelization is selected, slave nodes are idle until master node finishes initialization step.
And that parallelization generates communication between master and slave during initialization step.
However, peer based parallelization doesn't. there's no idle time and no communication during initialization.

c. All node owns its partition

Each node owns partition from initial board. **My_board[my_row][my_col]** is the partition. Type of each element is **bool** (1byte). Because it only stores 1 or 0.
Because this system is 1D cartesian, initial board is split by rows. (line 10 - 11)
My_row is row size of partition. All nodes have equal rows (board_size/n). If there is a remainder, distribute it one by one from the first node. For example, suppose that board_size is 15 and n is 4. Then there occurs remainder 3. Therefore node 0,1,2 gets 4 rows and node 3 gets 3 row.
My_col is same with board_size. Because this is 1D cartesian topology.

d. Ghost cells

Ghost cells are duplication of boundary cells of adjacent nodes. It is used to reduce communication volume.
In this algorithm, ghost cells are stored in upper_boundary[my_col] and lower_boundary[my_col]. If ghost cells are from next node, then it is stored in lower_boundary[my_col] (vice versa).
That boundary consists of two section. First is ghost section which is duplicated boundary of adjacent node and second is comm section which needs communication during generation. Detailed computation method about deciding each section's size is not written pseudo code. Logic is as follows.
About ghost section, n_upper_ghost and n_lower_ghost indicates that size of ghost cell in both upper and lower boundary. If ghost cell's size is K*K, n_upper_ghost and n_lower_ghost become K*K/2 respectively. (if K*K/2 occurs remainder, n_upper_ghost gets 1 more cell) Maximum size of those is my_loc. Because it is for just boundary cells. However, those size are not always same in all nodes. In case of first and last node, there's no upper and lower boundary respectively. In this case, one of them becomes 0 and the other becomes K*K. because the content of ghost cell is duplication of adjacent nodes, copy from adjacent nodes is implemented in line 16 to 17. First n_upper_ghost or n_lower_ghost elements in upper_ or lower_boundary is filled.
About comm section, n_upper_comm and n_lower_comm indicates rest of my_loc – ghost cell size. Because size of ghost cell can be different among nodes and among boundaries in one node and comm section size

depends on ghost section, comm section size can also be different. Because that section will be filled through communication during generation, there's no initialization in those sections.

e. Communication of metadata for send count

Because size of comm section can be different among nodes and among boundaries in one node, adjacent sender node should know size of comm section of receiver. Line 19 to 30 implements this.

Metadata[0] holds send count for previous node (n_lower_comm of previous node). Metadata[1] holds send count for next node (n_upper_comm of next node). Tag for communication of negative direction is 110 and tag for communicaton of positive direction is 130. In general case, one node becomes sender to previous node and sender to next node, all elements in metadata should be filled. However, in first and last node, they only have 1 adjacent node. So only one element of metadata will be filled.

This communication is synchronous because it needs to be prepared before starting game and this task is last section before starting game.

f. Communication for boundary of adjacent nodes

Before starting every generation, each node should have copy of boundary from adjacent nodes. Line 35 to 42 implement it. If size of ghost section is same with my_col which means that same with boundary, then all the boundary cells are already copied in ghost cells. So no communication is needed. Line 34 checks it. If not, each node should communicate with adjacent nodes. First and last node have only 1 adjacent node, There are only 1 set of send and receive respectively. In general case, 2 sets of send and receive are needed. Tag for negative direction of communication is 230 and tag for positive direction of communication is 290.

All the communication is synchronous. Asynchronous communication can complicate program code because needed boundary is different with each cell's location. To guarantee the correctness of computation, MPI_Wait will be needed but it can complicate logic. So I used synchronous version.

4 types of communication exist.

First is sending subset of last row to next node. **MPI_Sendrecv(&my_board[my_row-1][my_col-metadata[1]], metadata[1], MPI_BYTE, ..., 290, ...)** indicates this. Metadata[1] is size of comm section of receiver. Receiver already holds replication of sender's last row in its upper_boundary array. Moreover ghost cells are filled from 0 index of upper_boundary, sender only needs to send remainder of ghost cells which starts from index of my_col-metadata[1]. And because data type of my_board is bool(1byte), send_data_type should be MPI_BYTE.

Second is sending subset of first row to previous node. **MPI_Sendrecv(&my_board[0][my_col-metadata[0]], metadata[0], MPI_BYTE, ..., 230, ...)** indicated this.

Third is receiving subset of last row of previous node. **MPI_Sendrecv(..., &upper_boundary[my_col-n_upper_comm], n_upper_comm, MPI_BYTE, ..., 290, RING_COMM, &status)** indicates this. Because first my_col-n_upper_comm are filled with ghost cells, remainder of upper_boundary can be filled thorugh communication. Its size is n_upper_comm and type if MPI_BYTE because data type of element of my_board is 1 byte.

Last is receiving subset of first row of next node. **MPI_Sendrecv(..., &lower_boundary[my_col-n_lower_comm], n_lower_comm, MPI_BYTE, ..., 230, RING_COMM, &status)** indicates this.

g. Do calculation

After step f, each node holds all needed data locally. So one can progress computation easily. Line 43 to 47 implements this. Detailed implementation is ommitted because it is not that related to parallelization. Logic is simple. There are total 9 types of cells in my_board. cells in 4 corners (4) and cells in first and last row (2) and cells in first and last column (2) and general case (1). Depending on those types, neighbors are different. Code in project3.cpp implements it case by case. After deciding neighbors and calculating alive neighbors, its next state is stored in **updated_my_board** not my_board. because generation(state) changes all at once. After finish calculation in all cells, updated_my_board is copied into my_board (line 48).

h. Communication for ghost cells

After finishing updating state, ghost cell should also be updated. Line 48 to 57 implements this. Each node sends its updated boundary to ghost cell of adjacent nodes. Tag for negative direction of communication is 310 and tag for positive direction of communication is 370. Communication pattern is same with section f.

i. Write to output file

Because each nodes are peers, each node holds output of final N's generations. To get whold board, To get a whole board, we just need to write to the output file one by one, starting with the node of the small rank. Line 60 to 68 implement it. Except for first and last node, each node waits receiving signal from previous node and then write to output file and then send signal to next node.

3. **Performance of parallelization strategy**

a. **Communication volume per node**

To analyze this, a node who has 2 adjacent nodes is considered.

- Communication for metadata of send_count
  Sending 2 MPI_INT type data + Receiving 2 MPI_INT type data
  ⇨ **Total: 16 bytes**

  *supposed that number of generation is G
- Communication for boundary of adjacent nodes at every generation
  Sending metadata[0] sized data with type of MPI_BYTE+ Sending metadata[1] sized data with type of MPI_BYTE
  Receiving n_lower_comm sized data with type of MPI_BYTE + Receiving n_upper_comm sized data with type of MPI_BYTE
  ⇨ **Total G * (metadata[0] + metadata[1] + n_lower_comm + n_upper_comm) bytes**

- Communication for ghost cells at every generation
  Sending (my_col-metadata[0]) sized data with type of MPI_BYTE + Sending (my_col-metadata[1]) sized data with type of MPI_BYTE
  Receiving n_lower_ghost sized data with type of MPI_BYTE + Receiving n_upper_ghost sized data with type of MPI_BYTE
  ⇨ **Total G * (my_col-metadata[0] + my_col-metadata[1] + n_lower_ghost + n_upper_ghost) bytes**

Total communication volume per node is **(16 + 4 * G * my_col) bytes.**
This is smaller than master-slave based parallelization.
Suppose that during generation process is same. Then there will be communication in initialization step whose volume per node is (**my_row*my_col) bytes**. Then total commication volume of general node will be **(my_row * my_col + 16 + 4 * G * my_col) bytes**. If my_row is larger than 4 * G, then upper bound of communication volume at master-slave based parallelization becomes larger than peer-based parallelization.

b. Performance and Scalability

Below table indicates **average execution time per node** with different computation size and parallel nodes. And graph is efficiency (s/(p*T(p)) of each execution.

	15x15 with 3 gen.	**100x100 with 100 gen.**	**512x512 with 50 gen.**	**2048x2048 with 200 gen.**
serial	0.0005725 sec	0.021555 sec	0.265662 sec	16.274756 sec
2 nodes	0.00035 sec	0.013669 sec	0.150365 sec	9.2483395 sec
4 nodes	0.000682 sec	0.008780 sec	0.078224 sec	4.66716225 sec
8 nodes	0.000699 sec	0.004836 sec	0.042200 sec	2.083982875 sec

Figure a. execution time on different number of parallel nodes and matrix size

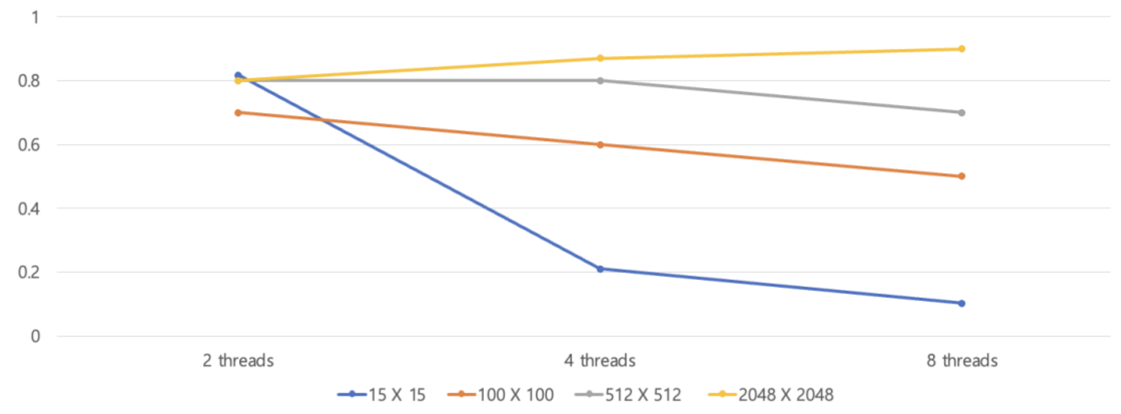

Figure b. parallel efficiency on different number of parallel nodes and matrix size

Overall, as parallel degree increases, execution time per node decreases.

In case of 2048x2048 input board with 200 generations, we can see that average elapsed time per node decreases almost (number of parallel node) times compared to serial version. And parallel efficiency is larger than 0.8 (quite good).

In case of 15x15 input board with 3 generations, computation volume is not that large to hide context switching between nodes. So as the number of parallel node increases, its efficiency decreases.