MPI:Game of Life

Chirag Majithia

***Abstract*—The aim of the project is to get familiarized to Message Passing Interface (MPI), used in parallel computing. The project uses Open MPI implementation of the MPI standards for solving "Conway's Game of Life" problem.**

I. Game of Life

The Game of Life (an example of a cellular automaton) is played on an infinite two-dimensional rectangular grid of cells. Each cell can be either alive or dead. The status of each cell changes each turn of the game (also called a generation) depending on the statuses of that cell's 8 neighbors. Neighbors of a cell are cells that touch that cell, either horizontal, vertical, or diagonal from that cell.

The initial pattern is the first generation. The second generation evolves from applying the rules simultaneously to every cell on the game board, i.e. births and deaths happen simultaneously. Afterwards, the rules are iteratively applied to create future generations. For each generation of the game, a cell's status in the next generation is determined by a set of rules. These simple rules are as follows:

- If the cell is alive, then it stays alive if it has either 2 or 3 live neighbors
- If the cell is dead, then it springs to life only in the case that it has 3 live neighbors

It is clear from the rules of the game that the parallel implementation of the simulation will have great advantages in terms of speed-up, when compared to a sequential implementation. This is because, the state of a cell in the game is only dependent on its neighbors. Thus, if we have access to multiple processors, we can decompose the 'arena' (X $\otimes$ Y dimensional cell-space) to provide each processes, the data required to compute the cell-state locally. Thereby, we can speed-up the computations by using multiple processes, each working on smaller chunk of data.

The project first implements a sequential algorithm. Then, the same algorithm is parallelized using OpenMPI. Finally, a comparison of the sequential algorithm is done with it's parallel implementation using 1, 2, 4, 8, 16 and 32 parallel processes. Moreover, we use two different implementations of the parallel algorithm - By decomposing the 'arena' along single a dimension (say, along Y) and By decomposing the arena in Blocks (i.e. along both X and Y dimensions)

II. Algorithm

A. Sequential

The sequential implementation of the algorithm is straight forward. We first pad the arena with additional rows on Top and Bottom, and pad additional columns on left and right. This is done to have a seamless calculation of computing neighbors. The pseudo code for the sequential implementation is shown in 1

Algorithm 1 Sequential Implementation

```
1:  Initialize the hyper-parameters of the game. X and Y
    dimensions of arena, number of iterations (generations)
    the simulation should run G.
2:  Initialize the arena by reading an input-file having the co-
    ordinates of live population (live cells). This will represent
    the current generation. [curr_gen]
3:  Initialize another empty arena with same dimensions. This
    will represent the next generation. [next_gen]
4:  Initialize two pointers
5:  for each generation g = 1 : G in generations do
6:      for each cell c = 1 : C(x,y) in [curr_gen] do
7:          Count the neighbors of C(x,y) - cnt
8:          if the cell c is alive then
9:              if No. of it's neighbors cnt < 2 then
10:                 The cell in next generation n ε N(x,y) is
11:                 dead due to underpopulation or
12:             else if cnt > 3 and < 4 then
13:                 The cell in next generation n ε N(x,y)
14:                 stays alive
15:             else if cnt > 4 then
16:                 The cell in next generation n ε N(x,y) is
17:                 dead due to overpopulation
18:             end if
19:         else if the cell c ε C(x,y) is dead then
20:             if No. of neighbors cnt == 3 then
21:                 the cell in next generation is made alive.
22:             else
23:                 the cell in next generation remains dead.
24:                 the above cell state is forced as we keep
25:                 switching current and next arena, to avoid
26:                 false alive cells.
27:             end if
28:         end if
29:         Swap the arena curr_gen with next_gen, for next
30:         iteration.
31:     end for
32: end for
33: The final state of the arena will be stored in next_gen
    when the loop terminates.
```

This algorithm becomes the reference for the parallel implementation of the code. The current implementation of the algorithm uses **for-loop** to compute the neighbors of a cell c(x,y). It is observed that a major **speed-up** can be obtained by un-rolling the for-loop with **If-Else** statement.

(The optimized code is implemented but not submitted with this submission as the tests were pending. The next draft of the submission will have the optimized code as benchmark for other evaluations)*.

B. Partitioning in 1D

The idea is to divide the computations (counting neighbors and setting states), into number of processes **(P)** available. For this to work, each process should have relevant information of the arena to perform computations. Moreover, it is important to equally partition the arena to all process, to have maximum computations to performed simultaneously. That is, the overall computation time should not be bottlenecked by one processor which is very heavily loaded while others complete their task quickly.

Thus, we divide the array into **P**, $X \times Y'$ partitions such that $Y' = Y/P$. Again, it is possible that the Y/P might not be a whole number. Thus, first we assign $\hat{Y} = round(Y/P)$, to each of the **P** processes, and then distribute the remaining $(Y\%P)$ process evenly to the individual processes, one by one. For example, if $Y = 8$ is to be divided by $P = 3$ processes, we first distribute $\hat{Y} = 2$ all the three processes and the remaining two is split between 1^{st} and 2^{nd} process. Thus the processes will have two $X \times 3$ process with final process to be $X \times 2$.

C. Partitioning in 2D

As discussed, the most efficient use of the parallel computing will be when the data is equally distributed. In order to achieve high speed-up, it would be better if we can decompose the arena in small blocks of equal dimensions (as computing neighbor is block operation). We thus extend our idea of 1D partitioning to 2D. We first compute the efficient factors of the 'area' of the arena using MPI_Dims_create and use MPI_Type_vector for efficient extraction of boundary information. (More details in the following submission)

III. OpenMPI and Message Passing

We need MPI, to transfer the data between "master" - rank 0, process and also to communicate the dependent data amongst the "slave" - rank > 0 processes. Communication strategy is as follows:

- Load the hyper-parameters and initial alive cell coordinates in a 'master' process - say with rank 0.
- Broadcast all this information to all processes such that each process has enough 'global' information i.e. it can compute the dimensions and locations of the data it wants to use locally, using global information - i.e. it's own rank. The our implementation, all the process individually computes the dimension of all the partition using the strategy explained in previous section, and chooses the dimension that maps to it's own rank. This way, each process can compute global information, such as which process would be working on what part global data.
- Again, computation of cell state depends on the neighbors. However, partitioning the data in uniform dimension will result in the loss of sufficient information for the boundary cells. Thus, it is required to create 'ghost' cells, whose sole function is to allow state computation for the boundary cells. The values of these ghost cells will change after each simulation of generation. Thus, after each generation, these ghost cell values are required to be updated. In order to efficiently use communication, it is required to communicate only this information to the partitions which require it. Thus, it is important for each process to know about the processes which are working on the neighboring partitions. The 'global' information - explained in the paragraph above, provides a way for each process to compute it's neighboring process.
- The implementation uses MPI's non-blocking send and receive. It is efficient to use non-blocking send and receive because we avoid the process to wait for the data of the boundary cells, while it has sufficient information to compute cell states which are in interior of the partition. Thus, we ask process to request boundary information , followed by sending the edge information, and start computing the cell states. It is highly probable, that the process will receive this information, by the time it computes the remaining cell states. Thus, to ensure correctness of the algorithm, we use MPI_Wait(&req,&stat) after computing the inner cell states, to guarantee the information transfer before computing the cell states on the edge.

** The implementation uses vector<int> type, instead of vector<bool>, as extracting the data to be transfered - boundary row and columns is efficient in vector<int> because c++ compiler uses bits instead of entire byte to represent vector<bool>, losing easily accessible memory addresses for each element. A more tedious implementation can be to use bool vector, which will improve time and space efficiency of the algorithm.

IV. Performance Analysis

It is shown in Tab. I, that the performance of parallel code is slightly worse than the pure sequential code due to computational overhead of partitioning and communication of data. However, there is exponential speed up as we increase the number of processes. Finally, the speedup saturates after

16 processes, as the time to compute overhead balances the speed up due to parallelization.

TABLE I
NO. OF PROCESSES VS TIME (IN SECS)

Processes	Time (in secs)
Sequential	11.24
(1D Partition) 1	12.52
(1D Partition) 2	7.38
(1D Partition) 4	4.38
(1D Partition) 8	2.99
(1D Partition) 16	2.84
(1D Partition) 32	2.78

V. CONCLUSION

A 3 different implementations where done to simulate Game of Life. 1. Sequential 2. Partitioning data in 1D and communication 3. Partitioning data in 2D and communication. The aim of the project was to understand message passing between processes using OpenMPI, hence, the optimality of the code was not worked upon - however few improvements are suggested in the report. The speed-up is very drastic as the number of processes are increased but it soon saturates to a constant value - around 2.8 seconds, as the overhead of parallelizing code and communication balances the speed-up achieved. If we further partition the data, there are chances of that the simulation will take more time, as the communication will become the bottle neck. The code for the above mentioned implementations can be found at Github Repository