CPSC 5600 Pongpichit Poonpiriyasup
Shaista Usman

The Game of Life

The Game of Life, also known simply as Life, is a cellular automaton devised by the British mathematician John Horton Conway in 1970. The universe of the Game of Life is an infinite, two-dimensional orthogonal grid of square cells, each of which is in one of two possible states, alive or dead, (or populated and unpopulated, respectively). Every cell interacts with its eight neighbors, which are the cells that are horizontally, vertically, or diagonally adjacent. At each step-in time, the following transitions occur:

1. Any live cell with fewer than two live neighbors dies, as if by underpopulation.
2. Any live cell with two or three live neighbors lives on to the next generation.
3. Any live cell with more than three live neighbors dies, as if by overpopulation.
4. Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

These rules, which compare the behavior of the automaton to real life, can be condensed into the following:

1. Any live cell with two or three neighbors survives.
2. Any dead cell with three live neighbors becomes a live cell.
3. All other live cells die in the next generation. Similarly, all other dead cells stay dead.

The initial pattern constitutes the seed of the system. The first generation is created by applying the above rules simultaneously to every cell in the seed; births and deaths occur simultaneously, and the discrete moment at which this happens is sometimes called a tick. Each generation is a pure function of the preceding one. The rules continue to be applied repeatedly to create further generations.

For the sequential approach, the life pattern or the grid is defined as a two-dimensional array; data member of a class where 0's represents dead cells and 1's represents alive cells. The grid row and column sizes are kept constant. Two arrays are used one to hold the current generation, and one to calculate its successor. The grid representing the current generation is initialized with random number generator, such that each run of the program gets different and dynamic patterns. The grid representing the next generations are initialized with 0's. The start() method loops through each cell and counts it's live neighbors, using the previously mentioned rules which are implemented by the aliveNeigbourCount() method to decide whether the corresponding element of the successor array should be 0 or 1. For the next iteration, the arrays swap roles so that the successor array in the last iteration becomes the current array in the next iteration.

Therefore, new generation cells are generated incrementally based on the count of neighboring alive cells and new grids for latter generations are created by simply assigning the next generation grid to the current generation grid. This allows switching to new generations and helps in displaying the final states of each generation. The following implementation ignores the edge cells as it supposed to be played on an infinite plane. We can observe the performance of this implementation by calculating the number of generations spawned in five seconds.

This is the breakdown of the tasks performed by the start() method:

- Iterate over all the cells and update them - updateNextGen(row)
- To update a cell, decide whether it will live or die based on the count of its neighbors - aliveNeighbourCount(row, col)
- Update the original grid with the new generation grid and transition into the next generation - swapNextGen(row)

This process takes O(n2), and involves a major memory copy and for small sizes that will not be a problem.

We use a similar base structure for parallelization of this algorithm. An array of threads is defined within the class and a new thread is launched for each row of the grid, which in turn loops through each cell in the row and calculates the number of it's alive neighbors and spawns new generations of cells and populates the grid based on the rules of the game. The join() function is used to wait for all of these threads to complete. In a similar manner new threads are launched for each row of the grid, looping through each cell in the row and updating the current generation grid with the next generation grid, thus enabling the transition to new grids in successive generations. Then we wait for all the threads to complete.

The approach to use a class with the grid as a data member and the various functions which implement the Game of Life rules as methods arises from the need to avoid passing the grid as a parameter to each of the various functions.

Animation delays add to the overhead for the implementation of the algorithm making both the sequential approach and the parallel approach performance appears very identical. However, when we implement the same approach without animating, the number of generations populated in the same time frame of five seconds increases significantly. Due to little computation needed to compute the algorithm, the program is better off with the sequential approach when the data size is small (1,000 or less) and better off with the parallel approach when the data size is large (10,000 or more).

Work Analysis

C = number of columns -> n
R = number of rows -> n
compute state = swap state = 1

Total work = C x R -> n x n = n^2

Time Analysis

Sequential Approach:

Total time = C x R -> n x n = n^2

Parallel Approach

t = thread overhead

Total time = (R x t) + C -> nt + n -> nt

Possible Improvements

A variety of minor enhancements to this basic scheme are possible, and there are many ways to save unnecessary computation. A cell that did not change at the last time step, and none of whose neighbors changed, is guaranteed not to change at the current time step as well. So, a program that keeps track of which areas are active can save time by not updating inactive zones.

If it is desired to save memory, the storage can be reduced to one array plus two-line buffers. One-line buffer is used to calculate the successor state for a line, then the second line buffer is used to calculate the successor state for the next line. The first buffer is then written to its line and freed to hold the successor state for the third line.

Alternatively, we can consider representing the Life field with a different data structure other than a 2-dimensional array, such as a vector of coordinate pairs representing live cells. This approach allows the pattern to move about the field unhindered, as long as the population does not exceed the size of the live-coordinate array.

References

https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life

100-point proposed rubric

Criteria	Ratings
Explanation of the approach and comparisons of the serial and parallel implementation based on work and latency	35
Successful serial and a parallelized implementation of the problem with speedup in the parallelized version	45
Style: Good code. Beautiful to read and suitable to show off to your friends. No memory leaks, etc.	20