Daniel Inoa
CSC 415 - Parallel Computing
May 6, 2014
University Of Rhode Island

Game of Life

Final Project

The purpose of this project is to portray Conway's Game Of Life with different levels of optimization through both a serial and parallel implementation. Conway's Game Of Life is a cellular automata that simulates the birth and the deaths of cells/organisms on a grid based on a certain number of rules that dictate what happens throughout the generations.

The rules are:

- Any live cell with fewer than two live neighbors dies, as if caused by under-population.
- Any live cell with two or three live neighbors lives on to the next generation.
- Any live cell with more than three live neighbors dies, as if by overcrowding.
- Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

The basic and most logical way to compute the state of the cells for the next generations is to traverse the grid (represented as a 2-dimensional array), visiting all cells one by one, and checking for the neighbors' state of the current cell we are looking at a certain (x, y) coordinate. This cell would keep a counter that is to be increased based on the number of actual living neighboring cells this current cell has. Once this cell has the final counter value, it determines what its state will be based on the rules mentioned above. The idea of defining each cell's state based on its neighbors is the most logical "recipe" to represent and calculate the grid's next generation, but it is also the slowest and the most computing-intensive one.

Based on this basic implementation I have derived two more optimized implementations. One is a serial optimization, the other is a parallel optimization using POSIX Threads.

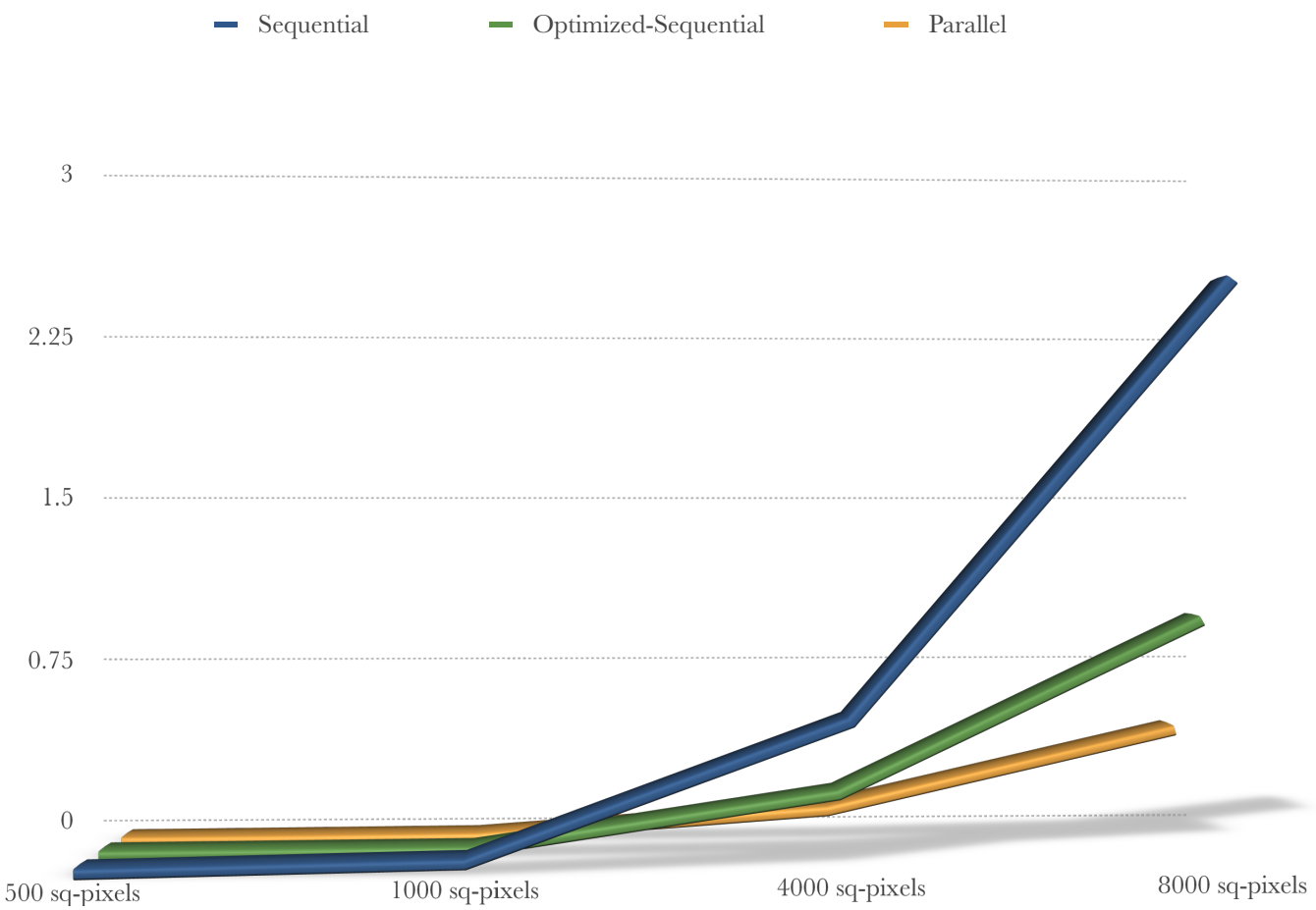
The serial-optimized implementation, produces the same results but in a far more clever and efficient manner. One important fact to understand about the simulation is that only the living cells determine the fate of the grid. Based on this fact it is clear that we would only need to work with the living cells while ignoring those that are dead. In this implementation a cell is not concerned with calculating its state but rather contributing to calculate the state of its neighboring cells. So this works by traversing the whole grid in search of living cells only. Once all living cells have been spotted they increment the counter of their immediate neighbors by one. At this point all cells can determine their state based on the value of their counter and the rules mentioned. Most of the performance gain in this computation comes from ignoring any cluster of dead cells while saving resources. This implementation also promises increasingly speedier computations throughout the generations due to the nature of the simulation which produces more dead cell than living cells.

The parallel implementation is done using POSIX Threads with the intent of having even better performance gain than the last two implementations. This is based on the optimized serial implementation and it further its optimization by diving the work among a very specific number of threads which then perform of fraction of the sequential computation. Initially the plan was to map a thread for each row on the grid, however, the overhead created is heavy enough to slow down the program to the point where it would take longer than the serial implementation to compute the results. The solution is to dispatch an optimal number of threads, where the total thread-creation overhead is minimal. This number would 1% or 2% of the total number of rows/columns in the grid. Naturally performs the best.

One point to talk about is exceptions, conditions, and computational cost regarding how rules after the behaviors of the cell in the grid. It's crucial to understand that in a grid (closed universe) some cells, namely those in the edges and the corners, have a different number of neighbors. The cells in the edges have 5 neighbors and the ones in the corners have 3 neighbors. The algorithm should then be written in a way such that it identifies which out-of-bound cells are not to interact with. However, I found that interacting with these out-of-bound cells would not affect the grid's final result and that the cost of doing work on out-of-bounds cells is cheaper than running condition statements to keep the indices within the grid.

These are the results for the different implementations showing the least amount of time it takes to compute the generation next with a certain of pixels.

Square Pixels	Sequential	Optimized Sequential	Parallel (with Pthreads)
500 sq-pixels	0.01 seconds	0.01 seconds	0.00 seconds
1, 000 sq-pixels	0.04 seconds	0.02 seconds	0.01 seconds
4, 000 sq-pixels	0.60 seconds	0.25 seconds	0.12 seconds
8, 000 sq-pixels	2.38 seconds	0.98 seconds	0.47 seconds



All three implementations were written in C++ along with OpenGL, and Pthreads for the parallel version of the program. All these were ran in a Late 13" MacBook Pro, with a 2.4 GHz Intel Core i5 processor and a 8 GB (2 x 4GB) 1600 MHz DDR3 memory.