CAUT: A Cellular Automata Simulation Framework for Parallel Programming Practice

Zacree Carroll, Troy Clendenen, Raul Patel

December 6, 2021

1 Abstract

2 Introduction to Cellular Automata

3 Methodology

Our goal with this project was to obviously get practice with parallel programming. So, we figured since many of our CA simulations we wanted to implement had already been done before, we did not want to look up how others implemented it until after we finished. So, we came up with 2 main approaches to how we could parallelize the simulations we wanted to.

3.1 The Redundant Approach

The simplest approach we could think of involved iterating across the entire m x n grid and perform some action on each cell in the grid. This action involves observing cells around the current cell in order to determine what the state of the current cell would be for the next iteration. Since we store the 'current' state and the 'next' state separately in memory, this allowed us to make changes to each cell on the grid completely independently, which turned the problem of parallelizing each generation of cells an embarassingly parallel problem.

On the downside (as you can assume based on the name of this approach), this meant that we were looking up the entire m x n grid, regardless of cell state. Usually in CA simulations, there are only a sparse set of cells that people care to observe. For example, if the user instantiated a grid of size 10,000 by 10,000, but there were only <10 active cells at any given time, that means that this algorithm is performing redundant work on the cells that aren't remotely close to these active cells, about 99.99% of the work done is redundant. Cells in most CA simulations only affect the cells around them, and so we had to think of another approach that would potentially speed up our simulations significantly.

3.2 The 'Live-Cell Relative' Approach

There are a few main differences between the Live-Cell Relative (LCR) approach and the redundant approach. First, we initialized the grid with an array of pointers to all of the live cells that were instantiated by the user in the input file and assigned OpenMP threads to each of the cells in that array. This is not required in the

previous approach. Another key difference is that the threads did not change the cell it was assigned to directly. Because we can obviously assume that the cells in the array are live, we update the next state of every cell around the live cell instead of the live cell itself. This works for most simulations because interesting interactions only ever happen around cells that are 'live'.

Because each OpenMP thread updates cells that are around live cells instead of the cell itself, this opens up a few possible race conditions. If 2 live cells are next to each other, and both OpenMP threads that are updating the cells around those both end up updating the same cell, we want to make sure that the value for that cell gets updated appropriatelly. This is easily solved with locks, and it's what we did in our implementation. However, the performance of this algorithm can vary a lot more, depending on if the grid is sparse in the number of live cells, or if the grid is dense in the number of live cells. If the grid has a higher live cell to dead cell ratio, the runtime of the program would actually be worse, because the race conditions introduced would mean that the cells are not being updated efficiently, and the benefit of only looking around live cells isn't as important, because most of the grid is live.

4 Results

4.1 Game of Life

4.2 Wildfire Simulations

4.3 Flocking Algorithms

5 Conclusion