CAUT: A Cellular Automata Simulation Framework for Parallel Programming Practice

Zacree Carroll, Troy Clendenen, Raul Patel

December 7, 2021

1 Abstract

Our project aimed to find new ways to parallelize cellular automata. Our group wanted to create a framework in which we can test various approaches to this, on a variety of different simulations. We first set out to parallelize John Conway's game of life. More specifically, we wanted to parallelize the process of updating each cell on the grid and their state for the next generation, based on the current orientation of neighboring cells on the grid. We came up with two main approaches, and both will be discussed in this report, along with the benefits and downfalls for each. To further test our project's capabilities as a general-purpose system for parallel programming practice, we designed two more simulations after we finished implementing the game of life. These other simulations were a rudimentary forest fire simulation, and a flocking algorithm simulation. Both of these are better real-world examples of how cellular automata can be useful in scientific simulations. We discuss the benefits and shortcomings of the two different parallel approaches in each of these simulations, and the results are fairly interesting. Note that there were some issues with testing on the Talapas supercomputing system, so our report includes findings from that, along with our personal machines. Despite this not being an ideal scenario, there is merit in what we find, which will help improve this project for the future, and how parallel algorithms can be observed on multiple machines.

2 Introduction to Cellular Automata

Cellular Automata (C.A.) is a very popular field of study and important framework when it comes to the realm of computational simulations. Its use spans the spectrum from simulating bio-remediation of contaminated soils to fluid flow simulations. CAMEL for example is a framework which abstracts away some of the hardware specifics of performing such computational simulations and allows the user to focus more time and energy on fine tuning their C.A. implementation. [CDGR+95] A cellular automaton can be thought of as a collection of cells from which complex behaviors can arise from a relatively simple set of rules. As can be gathered from this view of C.A. it clearly is a very useful method for framing certain phenomena whose characteristics match the specification.

One of the most widely known CA simulations out there is John Conway's Game of Life. The Game of Life is a game where there exists an m bhy n grid of cells, and the cells can be either alive or dead. The player instantiates the grid with a certain amount of live cells, and begins the simulation. Each subsequent generation of cells

on the grid will be either live or dead based on 3 rules. 1: Any live cell with fewer than two live neighbors dies, as if by underpopulation. 2: Any live cell with two or three neighbors lives on to the next generation. 3: Any live cell with more than three neighbors dies, as if by overpopulation.

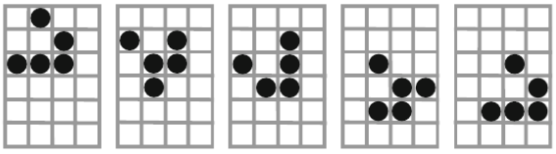

Figure 1: An example of a 'glider', a complex pattern that arises in the game of life.

This game may be seemingly pointless, although there are a lot of interesting behaviors that arise when cells are instantiated in a certain way. As is for other CA simulations, we can define a relatively simple set of rules in which complex behavior can arise. Our group first tried to implement the game of life from scratch, and parallelizing the process in which the grid updates each cell's state for the next generation. We will discuss the main ways we thought about this in the next section.

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 by 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 end up updating the same neighboring cell, we want to make sure that the neighbor 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

First, let's show the results of the redundant algorithm on talapas. For this, we instantiated a grid of 4,000 x 4,000 where we randomly generated live cells at about a 1/10 chance. This meant that we initialized a sparse grid of live cells, where about 90% of the cells were dead at the start.

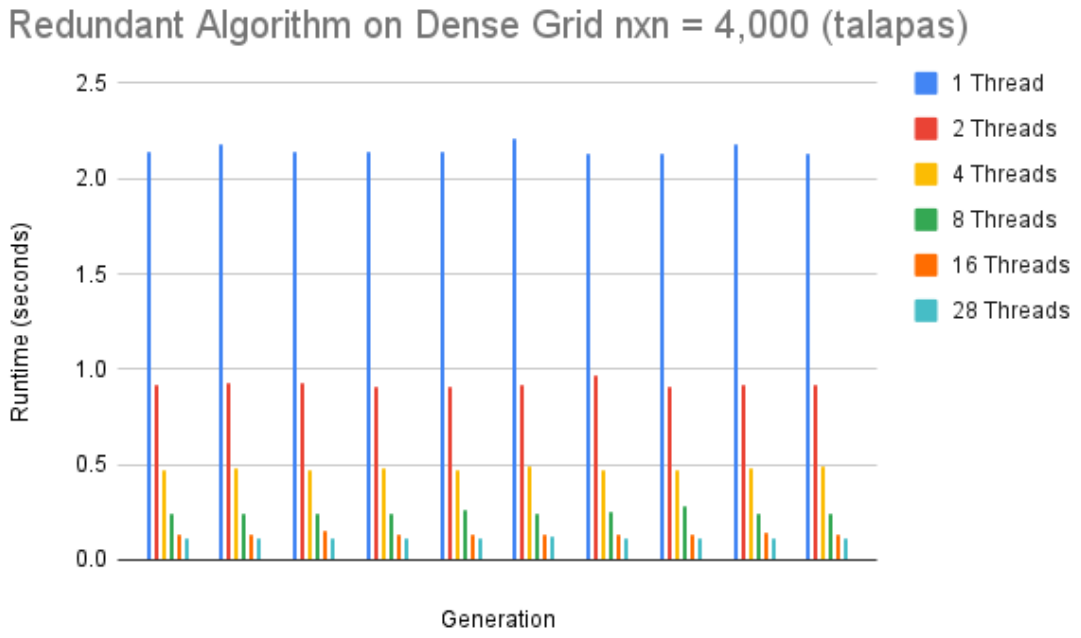

Figure 2: Redundant algorithm results on Talapas.

As you can see from Figure 1 above, the time it takes to update each generation is

halved when we double the number of OpenMP threads. This makes sense, because each OpenMP thread is independently updating cells in the grid (an example of an embarassingly parallel approach). We also verified that the results for each generation in the simulation itself was correct by printing out the grid after each generation. This is an impressive speedup, however we would like to see if we can avoid the redundancy and still see a good amount of performance improvement.

Before we get into the LCR algorithm, there is a much different trend in how performance differs based on the number of thread counts. Unfortunately, we were not able to test accurately on Talapas to compare the 2 algorithms. At some point in implementing the LCR algorithm, the runtime behavior for the redundant algorithm changed somehow, and we couldn't get consistent results. On our local machines, however, there were more readable results that still have merit, since they highlight the difference between the 2 algorithms. Listed below are both charts run on a Macbook Pro 2013 running Ubuntu natively, with an intel i5 chip.

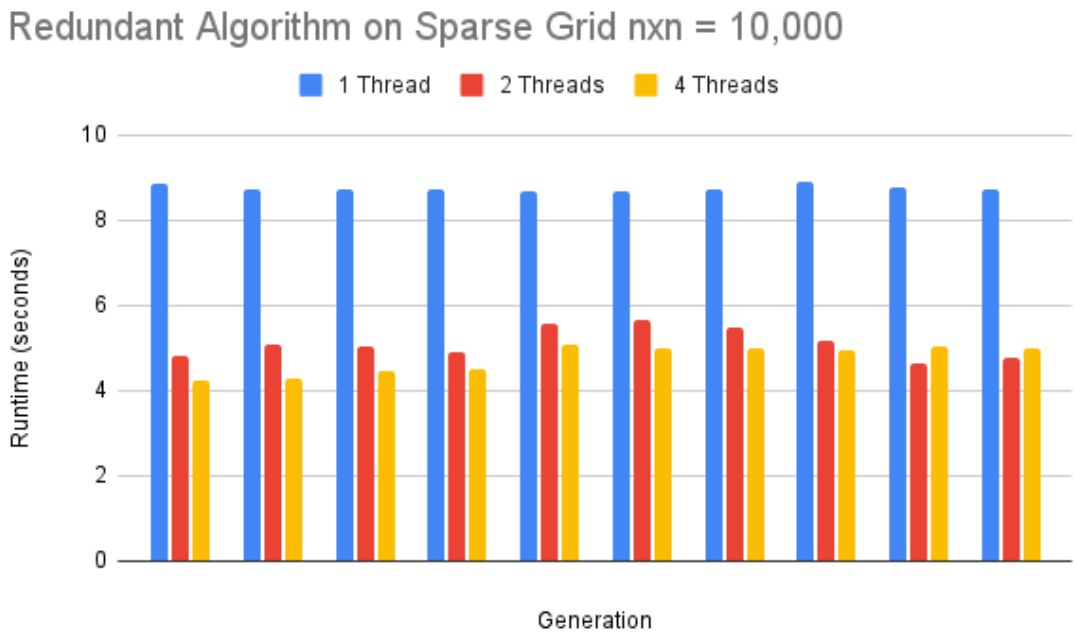

Figure 3: Redundant algorithm results.

The above chart is the redundant algorithm as before, but on a grid of 10,000 x 10,000 cells. Note that we do not get a near-perfect speed improvement as we double the cores, but we have this here as a baseline comparison for the LCR results in figure 3 on the next page.

Now there are quite a few things to unpack with the LCR results. Let's discuss some things before we talk about how increasing thread count actually made the run time of each generation even worse. First, the time it takes to go through each generation and update the cell's next state improves over the course of the program itself. This is due to 2 main reasons. First, given the fact that live cells have stricter rules regarding their 'liveness' means that most of the time, there are less live cells over the course of a simulation. The exceptions would be for large pulsars, many gliders, etc.

The next reason has to do with data that exists in cache. Because the LCR algorithm deals with cells that are nearby live cells, on the first iteration of the simulation, the vector for the live cells are in cache, but the location of these cells

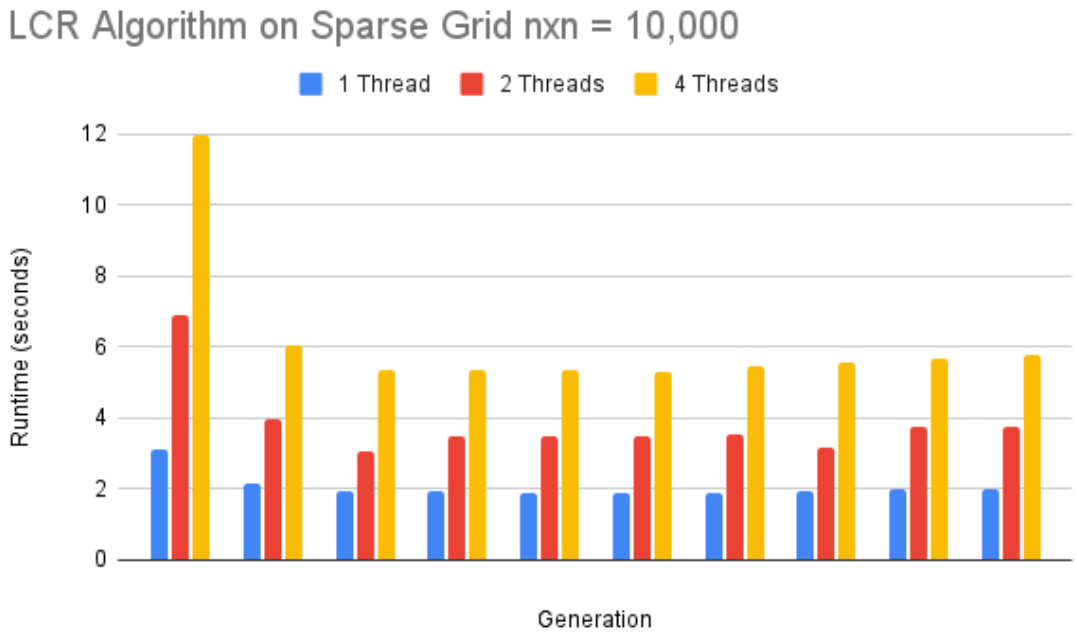

Figure 4: Live-Cell Relative algorithm results.

in the grid are not necessarily there. Because the Game of Life is predictable in where cells will appear for the next generation, we get to utilize spacial locality where cell activity goes on. When the 2nd generation of live cells are produced, the locations of those nearby cells are still in cache because we used them for the previous generation. Thus, as the program continues running, predicting where important cells are become easier, since the behavior of those cells become more clear as time goes on.

On the bright side, we do notice that the time it takes to update each generation in the LCR approach is less than the time it takes to update each generation in the redundant approach at 1 thread. So, the idea of the LCR approach is there, what's needed is to go in and look at how OpenMP tasks are being assigned and generated, and rewrite the code so that we actually get a performance improvement.

4.2 Wildfire Simulations

4.2.1 Implementation

For our wildfire simulation we wanted to keep it as simple as possible while also producing something that behaved in a way similar to how a wildfire would spread in real life. With a few simple rules and extra attributes in the cells we were able to accomplish something we felt looked good.

The attributes we added were cell health/burnable material (0, 50] and a percentage of dry brush (0, 1] each initialized to a random value for every cell. The grid is initialized in a specific way so as to produce interesting output because a fire simulation where the fire does not spread is just not interesting. If a cell is not currently alive and still has hp and also has two neighbors that are alive then its chance to come alive/ignite is based on its percentage of dry brush it contains. If a cell is alive/on fire then it will lose a certain amount of its health every turn as would happen during a real fire. Both of these values were tinkered with a fair amount in

an attempt to fine tune the fire spread to make it appear to be more realistic in the way that it moved through the cells.

Our sample runs for the fire simulation consisted of 20 runs at each thread count and each run calculated the simulation for roughly 50 iterations. We tried various grid sizes from 1,000 x 1,000 to 10,000 x 10,000 and eventually settled on a 5,000 x 5,000 size grid. The reason behind this is that we wanted a large enough grid that the amount of work to do was not trivial but also small enough that we did not spending days running our simulation to gather and compile our results.

4.2.2 Wildfire Results

Unfortunately we ran into some very troubling issues when attempting to run our simulations on talapas. There were certain attempts that would never finish and others that would finish in a suspiciously short amount of time. However running locally we were not able to reproduce these problems on either of the two machines that we ran our simulations on. Below you will see a graph depicting run times via a Ryzen 5 2600x 6 core machine.

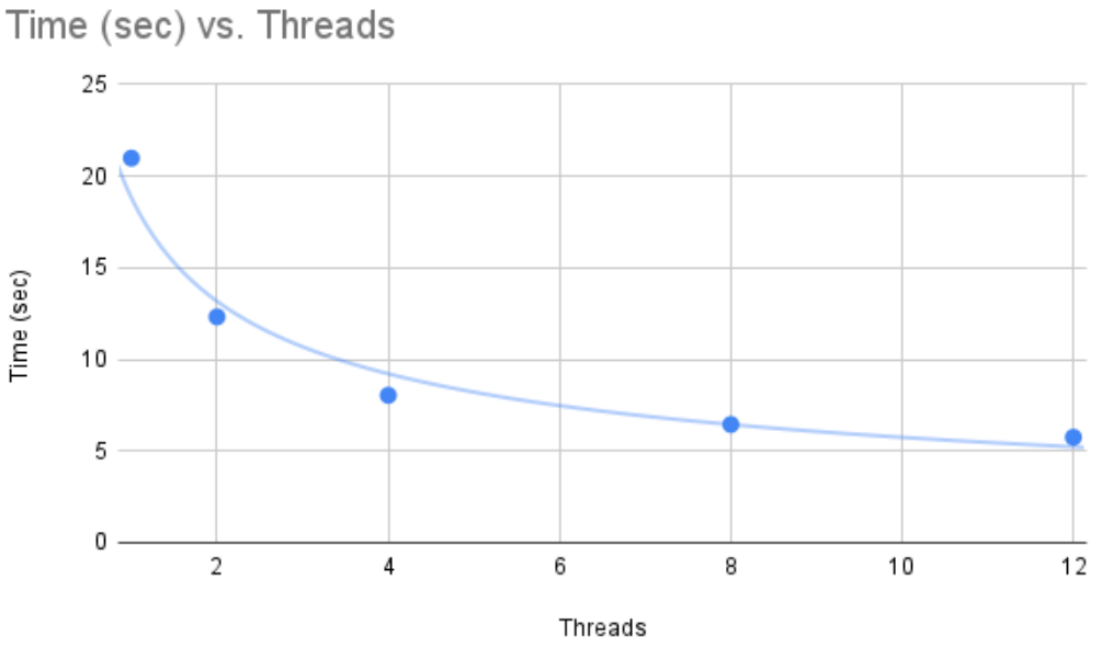

Figure 5: Fire Simulation Performance Results

As can be seen from the data we were able to achieve speed up with increasing the thread count for this algorithm which came as little surprise to us. We had planned to apply the Live Cell Relative approach to this algorithm given that it is not rule set specific however as mentioned previously we encountered some troubling issues there as well. However we speculate that the Live Cell Relative algorithm may not actually produce much speed up for the wildfire simulation because of how the cells tend to behave. Typically on what we would consider to be a useful/interesting trial of the simulation (i.e. the fire doesn't immediately burn out) large clusters of cells are alive simultaneously. This means that even if we only considered certain cells who have the potential to ignite at some point the Live Cell Relative algorithm would degenerate back into the naive approach where we are simply iterating

through the entire grid.

4.3 Flocking Algorithms

4.3.1 Implementation

This flocking algorithm is an oversimplification of the way that birds or fish tend to move in groups. To implement this simulation there are three overall rules: the "cells" must try to stay close to one another, they may not collide, and they must attempt to match velocity with one another.

Because this simulation involves "moving" cells, we had to find another way to implement this while maintaining our API. The simplest way to do this was to add a separate state of cells specific to Flocking that kept an int array of size 2 holding an X and Y direction (1, 0, -1) that determines the direction of travel. We can add the vector to the current cell location to find the location that the "cell" is moving towards. Given more time, we could have implemented some polymorphism by implemented separate Flocking subclasses of the cells and grid.

By checking the status of the space to which the cell is moving, we can determine whether or not a collision will happen. To avoid this, we can have the cell match its direction with whatever cell is already moving there. To add some randomness, we have birds peel off from the group if they have the right amount of neighbors.

By nature of the algorithm, Flocking necessitates having a more sparse grid so that the cells are not over crowded and have room to move without colliding. Because of the implementation of the LCR, we have an updated list of the live cells each iteration which we can use to update the status of the simulation without checking every single cell each iteration. Unfortunately, because only one neighbor of each live cell will be affected each iteration, there would be a lot of redundant work involved in getting the LCR algorithm to work for Flocking. However if we didn't want to account for birds branching off, we could eliminate any need for counting live neighbors in this algorithm.

4.3.2 Flocking Algorithm Results

We tested the flocking algorithm on a sparse 10,000 by 10,000 grid to see how long it would take to update the grid and to pull off the whole simulation with different numbers of threads. For the first set of tests, we used 1, 2, 4, 8, 16, and finally, 28 threads to run the simulation with 10 iterations and compare results. This was a single run for each thread count just to compare immediate results.

As shown in Figure 5, there was quite a noticeable speed up here from 1 thread all the way to 28 threads for the time it took to update the grid each iteration. As with GOL, the first iteration took longest on all threads which we believe is due to loading cache. On all threads, the runtime decreases as we go on.

When we compare the actual runtime for the full simulation across the thread counts, as seen in Figure 6, we notice this trend again. We see a log scale decrease in runtime as thread count increases. From around 356 seconds to around 49 seconds makes for more than a seven times speed up from the parallelism added.

Then we did repeated tests with single-thread and 28-thread execution to get an average and compare those results for the speedup. For these tests, we did 50 iterations with the same 10,000 by 10,000 grid. What we found matched the previous results. Given an hour to run on the compute nodes of Talapas, the single-threaded

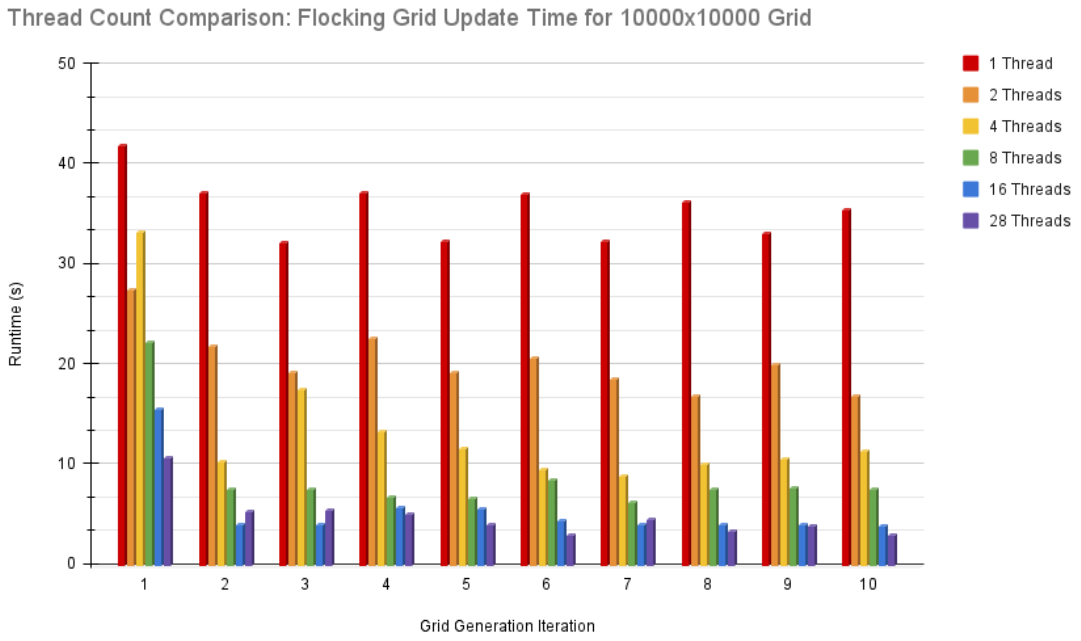

Figure 6: Single Test, 10 Iterations, Time to Update Grid at Different Thread Counts

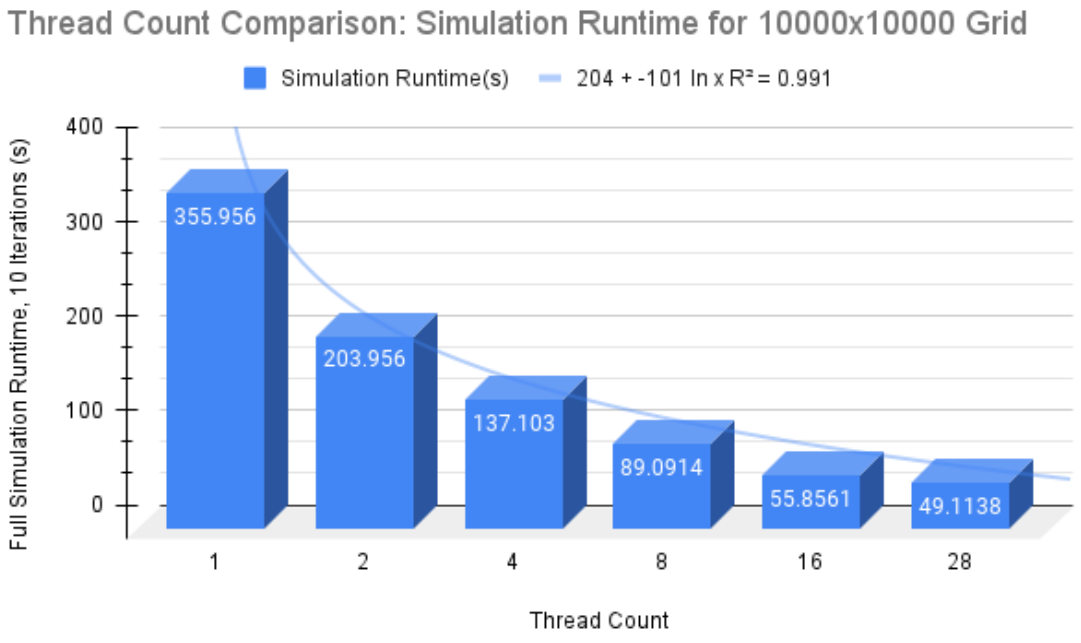

Figure 7: Single Test, 10 Iterations, Full Simulation Time at Different Thread Counts

job completed one full simulation and almost completed a second before timing out. In the meantime, the 28-threaded job completed all 15 simulations in just over 38.6 minutes.

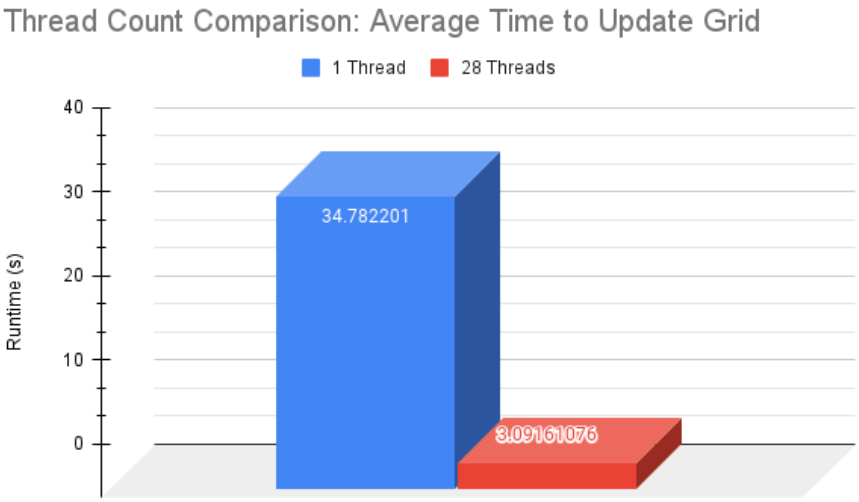

Figure 8: 15 Test Average, 50 Iterations, Time to Update Grid at 1 and 28 Threads

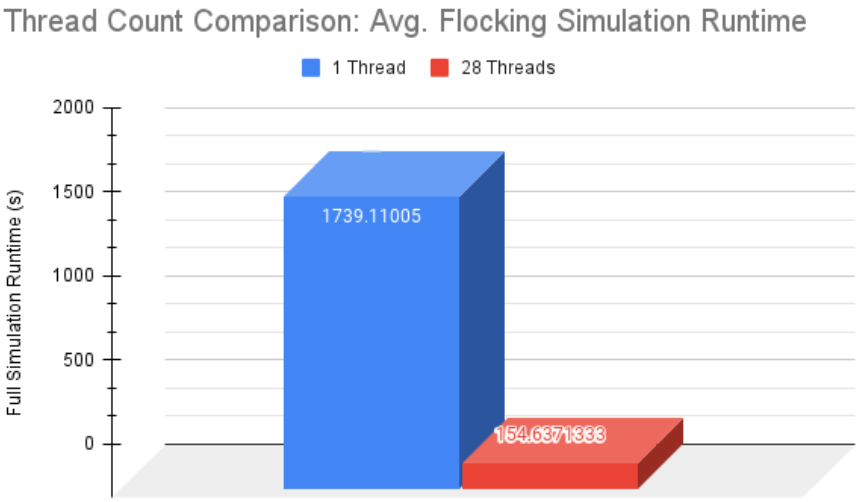

Figure 9: 15 Test Average, 50 Iterations, Full Simulation Time at 1 and 28 Threads

The average for updating the grid, as seen in Figure 7, is consistent with the single test results for 1 and 28 threads. Because we had more iterations, however, the average time increased for both for the full simulation runtime, as seen in Figure .

Both the average simulation runtime, as well as the average runtime for each iteration showed about an 11.25x speedup between the single-threaded sequential version and the 28-threaded parallel version.

5 Conclusion

For this project we set out to see how we, as university students, could implement a well researched topic from scratch and attempt to parallelize it on our own to see a speedup. In those regards, we definitely met our goal. Using our redundant algorithm, we saw measurable speedup in all three simulations that were implemented. When we tried to optimize our working algorithm is where we started seeing mixed results. Although logically it makes more sense and we saw a large decrease overall in sequential execution by checking less cells each iteration, the race conditions created by the algorithm resulted in the opposite of the intended effect. In sparse grids this could help a lot but it could do more damage than good in crowded grids.

We did, however, implement an interface to more easily create simulations using a cellular automaton framework, implement three simulation, and parallelize their execution. This work will set us up to be able to expand the scope of this project past what we covered for this course.

References

[CDGR+95] Mario Cannataro, Salvatore Di Gregorio, Rocco Rongo, William Spataro, Giandomenico Spezzano, and Domenico Talia. A parallel cellular automata environment on multicomputers for computational science. *Parallel Computing*, 21:803–823, 01 1995.