15-418 Final Project Report

Game of Civilization

https://yatinghan.github.io/game_of_civilization

SUMMARY

We implemented an enhanced version of Conway's Game of Life -- Game of Civilization in CUDA and OpenMP. Our parallel version achieved 3~10x speedup on reasonably-sized inputs. Game of Civilization follows the same set of rules of life as Conway's Game of Life. The major additional feature is that cells living close to each other would cluster together to form tribes or nations and those civilizations occasionally go to war when they are bored like human societies do.

MOTIVATION & BACKGROUND

The goal of the project is to 1) explore parallelism in graph algorithms; and 2) parallelize the Game of Civilization to achieve optimal speedup.

One major challenge in parallelizing many kinds of computations on graph is dependency issues: results on one part of the graph depends on the results of other parts of the graph (e.g. union find). Our game of civilization serves well as a small playground for this type of graph algorithms, because the cluster building process of each civilization affects the others. As the cells cluster together to form civilizations, they can't all start searching for nearby lives at the same time, but have to wait for a few leader cells to perform a breadth first search to collect everybody in the neighborhood, otherwise there will be a large number of civilizations with almost the same set of members.

We seek to tackle this problem by breaking the graph into many disjoint smaller graphs, so that computations can happen in parallel in each local section of the graph. As we know about the Game of Life, the density of lives are never too high in any of the local areas, otherwise the cells will suffocate to death. The living cells usually live in small groups on different areas of the map. This nature of the game

provides us the opportunity to dissect the graph (map) into several local regions with relatively higher population density, where civilizations are most likely to appear.

Ideally for each civilization or cluster, we want to call the searching algorithm once and find all members at one go. The optimal location to start searching would be the cluster center. So, our major task boils down to locating all cluster centers.

There are three phases of the simulation: update lives (original Game of Life), forming civilizations, and update the nations' population due to war. To recap, the revised game rules are as follows:

- Any live cell with fewer than two live neighbors dies.
- 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.
- Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.
- Lives that are close to each other form a nation.
- At each generation, the power of the nation is technology index * population. Technology index increases by 1 during each generation.
- When two nations are close together, there is a 0.1 chance they will go to war. The engaging nations will not fight for the 5 generations after. Winning nation inherits the population and the higher technology index among the two.

The reason to introduce the rules of war is to inject some randomness in the game, so that we can better show that our algorithm is very responsive to the changing state of the world and they perform consistently well across all situations.

APPROACHES

The following workflow diagram is a rough sketch of all the logic sections of our simulation of how lives and civilizations evolve. We will be explaining the details of each section and our rationale behind it below.

Simulation Workflow

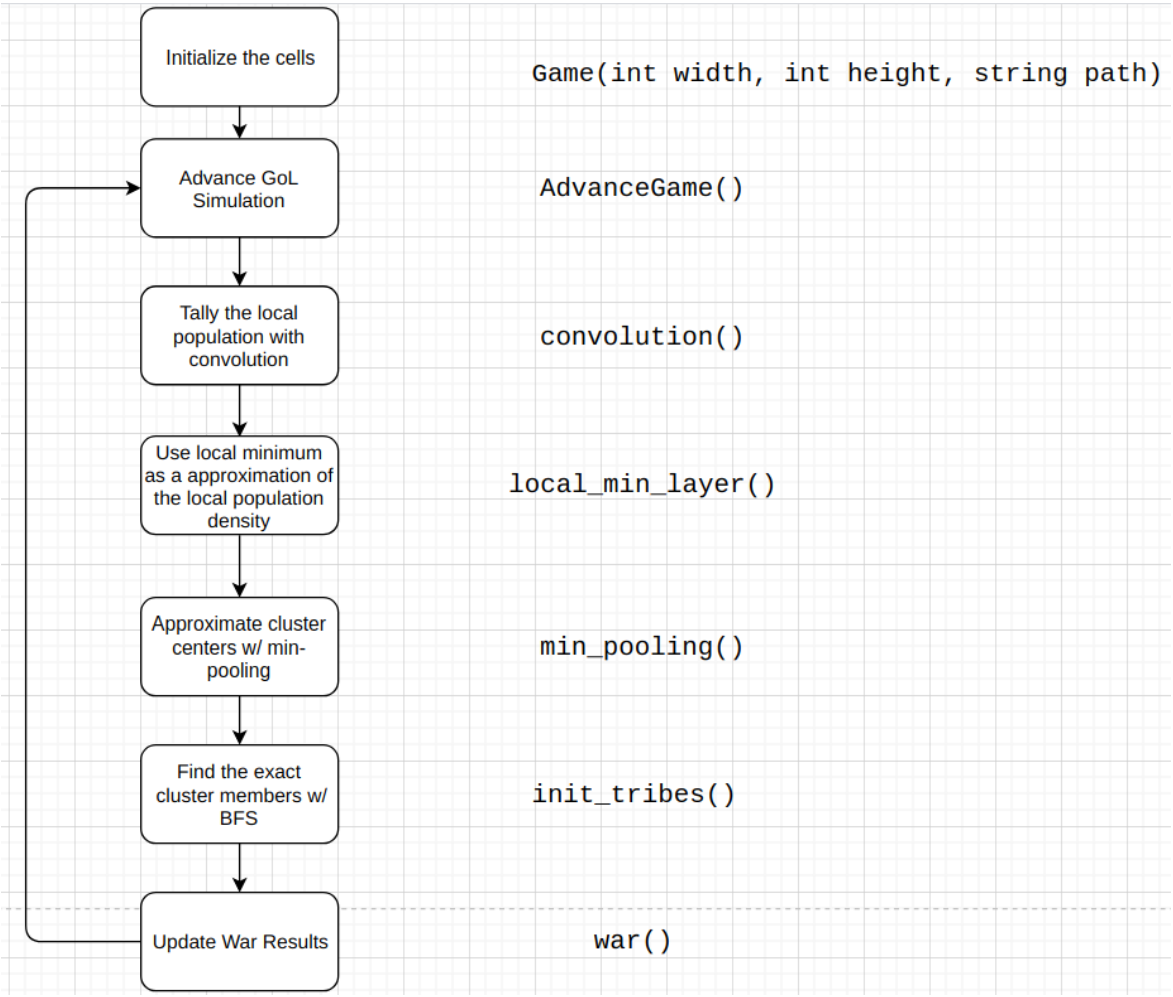

Figure 1: Simulation workflow

Phase 1: Update Lives

The 2-D grid is represented as an 1-D vector. Each index represents a live cell. Since each live cell only depends on the previous state of its neighbors, it can be trivially parallelized on GPUs. The work

is data parallel. Spatial locality can be achieved by launching consecutive live cells at the same time. The algorithm is not suitable for SIMD due to a large number of conditional statements.

We parallelized the original Game of Life with CUDA. The update step was trivially parallelizable due to its independent nature. However, we observed < 2x speedup with naive parallelism. We propose that the work assigned to each thread is too little, making the overhead for creating thread significant. We measure the performance with each thread responsible for 1, 2, 4, 8, 16 live cells. Four cells per thread gave us the best result, with a 3-10x speedup on various number of iterations.

Another problem we encountered was how we should copy the result vector back and forth between device and host. We realized that we do not need the array unless we want to print out the results, so we only copy back when printing and have each thread responsible for copying its corresponding index from the "future" array.

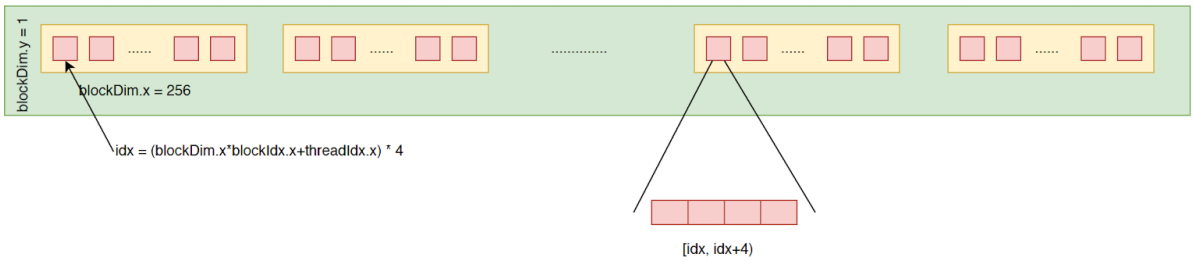

Figure 2: mapping CUDA cells to vector indices

Phase 2: Approximate Cluster Centers

BFS is very difficult to parallelize because of its dependency across loop iterations. While we can consider multiple neighbors at the same time, pushing to the work queue must be done sequentially. There is very little room for parallelism beyond this. However, if we can launch multiple unconnected instances of BFS at the same time, this would save a lot of time, but we need to estimate where the centers of those disjoint clusters are first. In addition, if we have an approximation for the graph's center, we can reduce the average distance between BFS starting point and its connected notes.

To approximate the cluster centers, we need to have a rough idea of the population density distribution across the map. We use three different layers of convolution to do this: local summing, local min, and min pooling. Convolution is perfect for parallelism, especially with tools such as OpenMP, because computation of each local region is independent from each other.

As less portion of work is inherently sequential, and by leveraging on the fact that convolution is highly parallelizable, we should be able to achieve a quite good speedup according to Amdahl's Law.

To better demonstrate the effect of each layer, let's use the world state at the 27th generation as an example, where different color encodes different tribes.

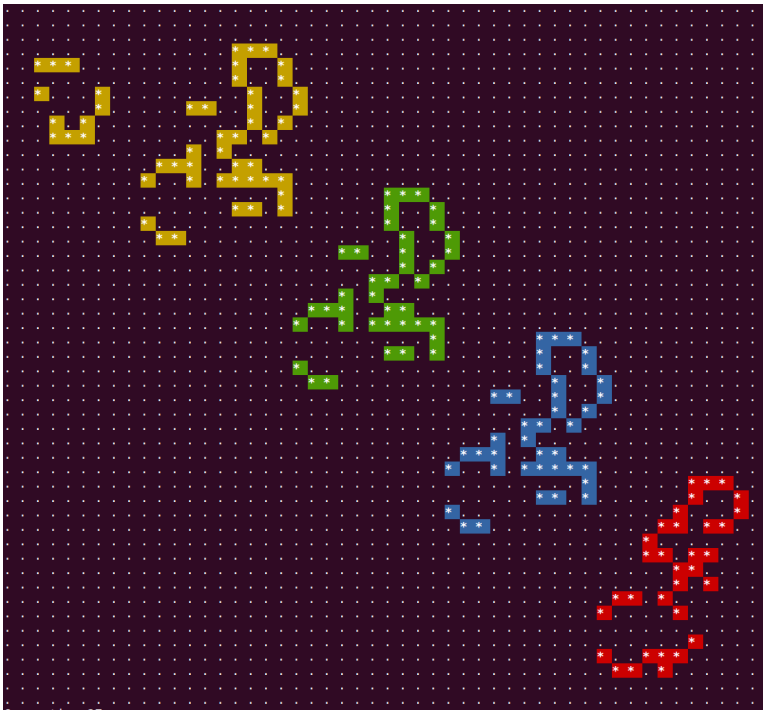

Figure 3: world state at the 27th generation

First layer: Local Summing Convolution

If we use a convolution matrix of size 10, the local summing convolution basically counts the total number of lives within every local 10 by 10 region. We use OpenMP for this part of the task and from repeated experiments, we find this layer of convolution achieves the optimal computation with 4

threads on static scheduling. All the following convolution layers use the same parallelizing method due to their similar computation logic. Although in hindsight, we realize CUDA might be more suitable for this task. More analysis on the performance is discussed in the Results section.

The result in the 27th generation is pasted below. The double digits head count made the central region along the diagonal looks more denser than the rest of the map, which matches the fact that most living cells concentrate around the diagonal. It's very much like a heat diagram of the population density. At this point, we already have a not-so-good approximation of where the clusters are. However, this approximated central region is still quite fat and we still don't see obvious cluster boundaries.

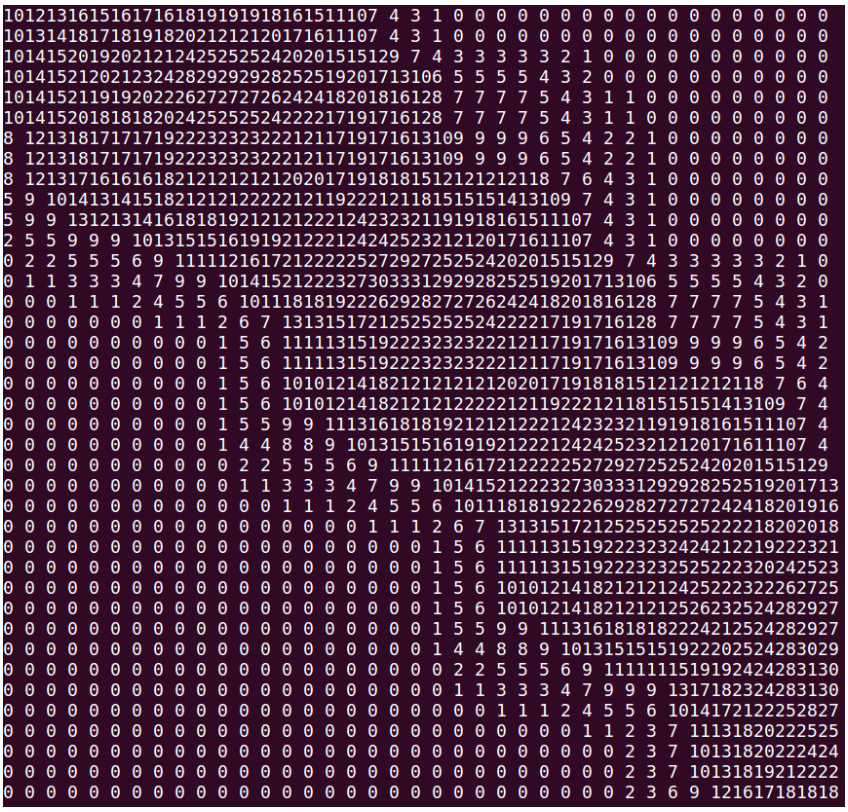

figure 3: Local summing convolution result of the 27th Generation

Second layer: Local min layer

The purpose of this layer is self-evident. If the convolution matrix has size 3x3, we iterate through all cells and marks down the minimum number in the 3x3 area with this cell at the center. This is

to make the central strip along the diagonal slimmer, so that the cells bordered on a lot of zeros (i.e. they live near the cluster boundaries rather than the center) would be also zeroed out. We don't care about these cells because we know that they are not likely to be the cluster centers we are looking for. The resultant matrix is pasted below, from which we can observe a much narrower band of double digit numbers.

figure 4: Local min convolution result of the 27th Generation

Third layer: Min pooling

In the min pooling layer, with a 10x10 convolution matrix for example, we mark down the minimum number of each 10x10 area. We use the minimum population as the approximation of the local population density, because the cluster centers would usually have the highest local min. By looking at the maximin, we can successfully locate the cluster centers. Again, we have the resultant min pooled matrix from the 27th generation below. The central band of double digit numbers along the diagonal has shrunk to a few

discrete groups of numbers cut off by zeros. We pick out a few cells with highest numbers from each group to be our candidates of the cluster centers and pass them to the next phase to do graph searching.

```
8 8 121315151511100 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 9 1013131411100 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 9 9 12121311100 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 9 9 9 1013129 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 9 1111100 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 9 9 108 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 8 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 9 9 9 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 9 9 9 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1111118 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 101012109 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 10101211100 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 9 9 1111100 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 8 8 9 1013129 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 9 1111100 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 9 9 108 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 8 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 9 9 9 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 9 9 9 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1111118 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 101012109 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 10101211100 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 9 9 1111100 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 8 8 9 1013129 9
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 9 111111
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 9 9 9
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

figure 5: Min pooling result of the 27th Generation

Phase 3: Build Tribes with BFS

Now that we have a rough estimation of where the cluster centers are, the next step is to search the neighborhood using BFS to find all members of each tribe. Arguably, it can be easily done with K-means clustering, which is also highly parallelizable. However, note that the motivation of this project is to experiment with how to dissect graphs into smaller parts to enable independent and therefore parallelizable computation in each part. Parallelism in point cloud or cellular automaton is obvious, while parallel graph processing can be tricky because there are edges representing the relationship between the vertices, i.e. vertices are not independent in graphs. So, we use BFS, an inherently not parallelizable algorithm, to test if our idea of dissecting the graph could work.

We still use OpenMP to parallelize across multiple BFS calls. With or without OpenMP turned on, the map outputs are highly identical, meaning the parallel BFS calls rarely interfere with each other. This proves the feasibility of our approach in dissecting the graph into independent subsections.

RESULTS

Phase 1: Update Lives

We measured the program runtime by adding timers just before and after `advanceGame()` on the GHC cluster machines. Initializations such as CudaMemcpy are excluded from the results. We compared the parallel version against sequential version across both input size and number of simulation steps. We used a "repeated pattern" for GoL so that the output never converges.

In the first experiment, we tweaked the grid size and ran the simulation for 1000 iterations. We believe that the speedup will increase dramatically until a certain number (max threads CUDA can launch), where it becomes linear. The experiment results confirmed our hypothesis, as the speedup remains constant after hitting an array size of 50000.

We conducted the same experiment on a purely empty grid. Our initial hypothesis is that empty grids will have a higher speedup because there will be less divergence in the wrap. Interestingly, the speedup is approximately the same with divergence and without. We then counted the number of times we entered each if-else case via the sequential code. It turns out that all our test files either repeats or diverges, so the number of empty cells always overwhelms number of lives. Divergence is less a problem as the simulation progresses.

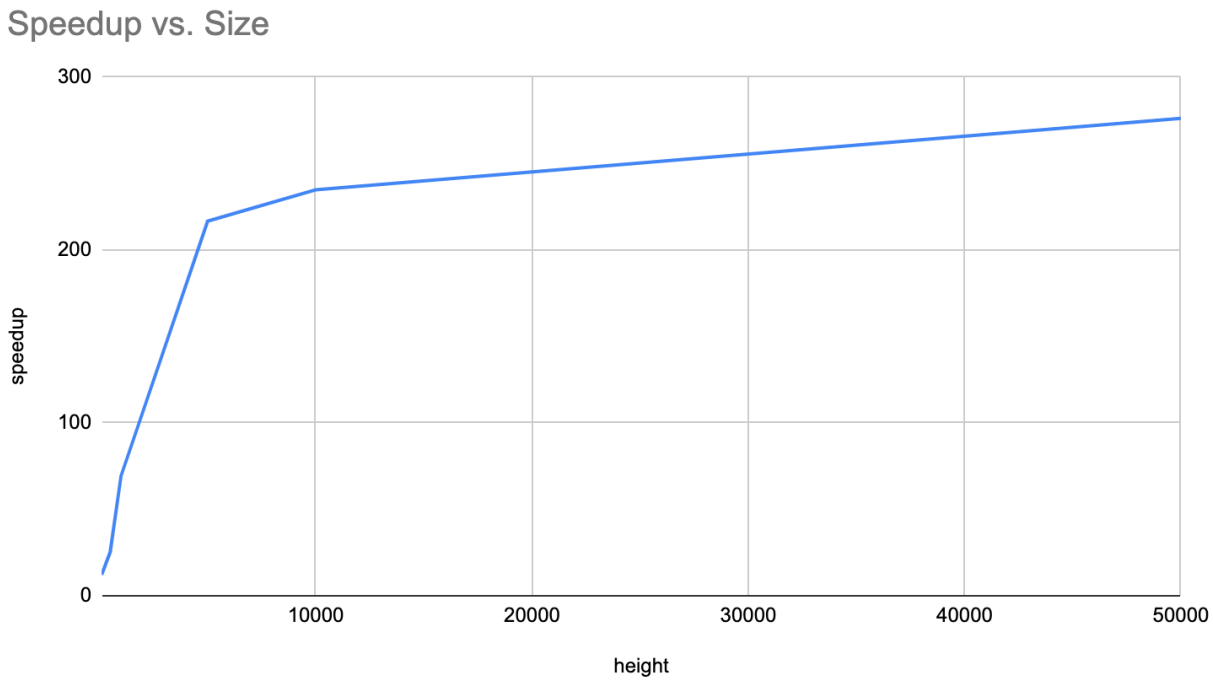

figure 6: [Experiment 1] CUDA speedup across different grid sizes (speedup against size)

The second experiment measures the speedup across different workloads (simulation steps). We used a setup that never converges, with grid size 50 * 50. The speedup increases as the number of steps grows but it starts to decrease after hitting 100, as shown in the figure below. We expected a constant speedup as the output image simply oscillates. Each step should take approximately the same time. After doing some research, we believe that the decrease is due to a long CUDA work queue. This is confirmed by adding `sleep(0.1)` after each iteration of the CUDA simulator. Sleep time was included in the results.

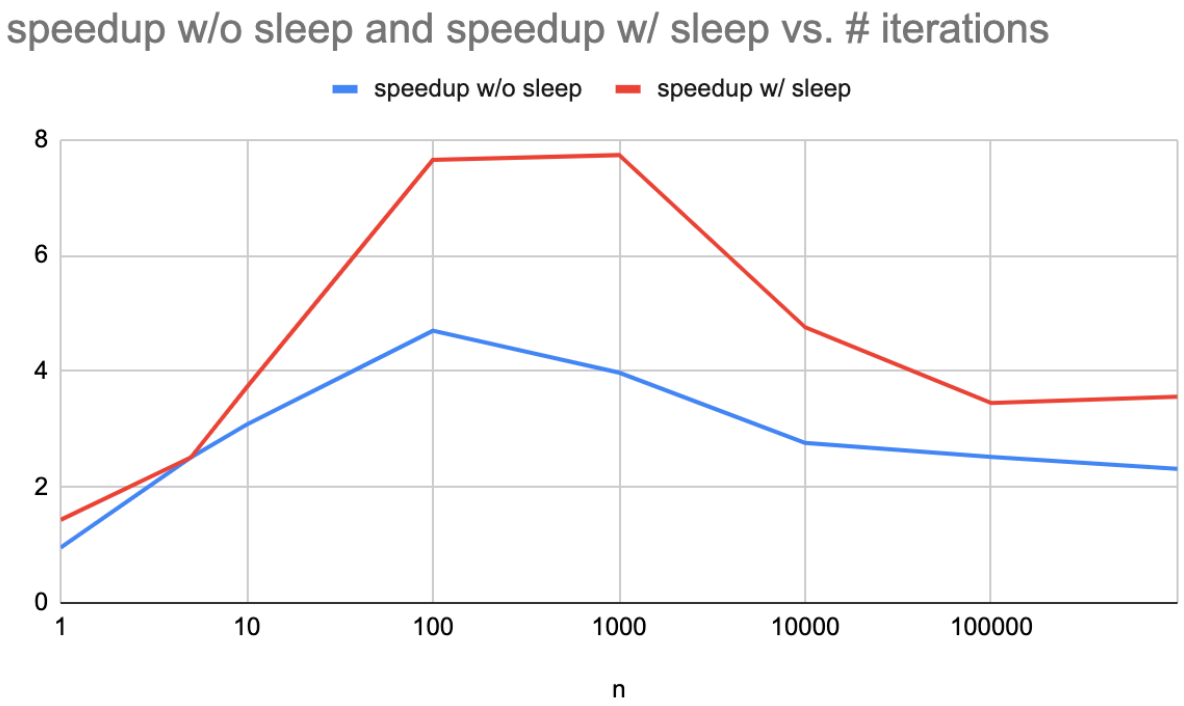

figure 7 [Experiment 2] CUDA speedup across different workload (speedup against number of iterations)

Phase 2: Approximate Cluster Centers

We compared the results across two spectrums: different game patterns and different grid sizes. We timed the function `init_tribes()` which is a glue function for the computation layers mentioned in the previous section. We measured the speedup in comparison to the sequential BFS version without convolution or pooling. Computation time for the original Game of Life was excluded. All the experiments were done on GHC machines, which 4 threads designated.

Performance Across Input Size

We ran a simulation that does not converge within 40 steps. It initially consists of 1*5 dashes, expands to circles, collides with other circles, and finally diminishes. We measured the performance for 40 simulation steps on square grids with different sizes. The parallel version is significantly faster than

the baseline, but the speedup does not meet our expectations. We timed the stages in order to find out the problem:

	parallel	baseline
Convolution	8.928119ms	-
Max pooling time	5.298823ms	-
BFS time	290.626180ms	-
Search nearby tribes	4.622474ms	-
Total time	347.094311ms	459.690008ms

We expected high speedup in convolution and max pooling step but we only get 3x. We believe that our code suffer from false sharing because each iteration updates a corresponding index in a vector.

In addition, we found that we made more calls to BFS than the baseline version. This might be due to the fact that our approximated centers are much more than the actual number of clusters (if we have 4 clusters, we might give 20-30 cluster center candidates), therefore inducing higher number of BFS calls. While each BFS call in the baseline version could make the cluster boundary much less ambiguous for the next BFS call, so that the next call would only start somewhere far. Across repeated experiments, we observe that the parallel version makes less BFS calls than the baseline when the lives are very concentrated and the clusters are very disjointed, but it makes much more calls if otherwise. Therefore, the parallel version does not consistently perform better than the baseline across all iterations. The solution to this is to better appromites tribe centers (adding more layers) so that a giant tribe would not have multiple centers to perform BFS from and then stitch the neighborhoods together.

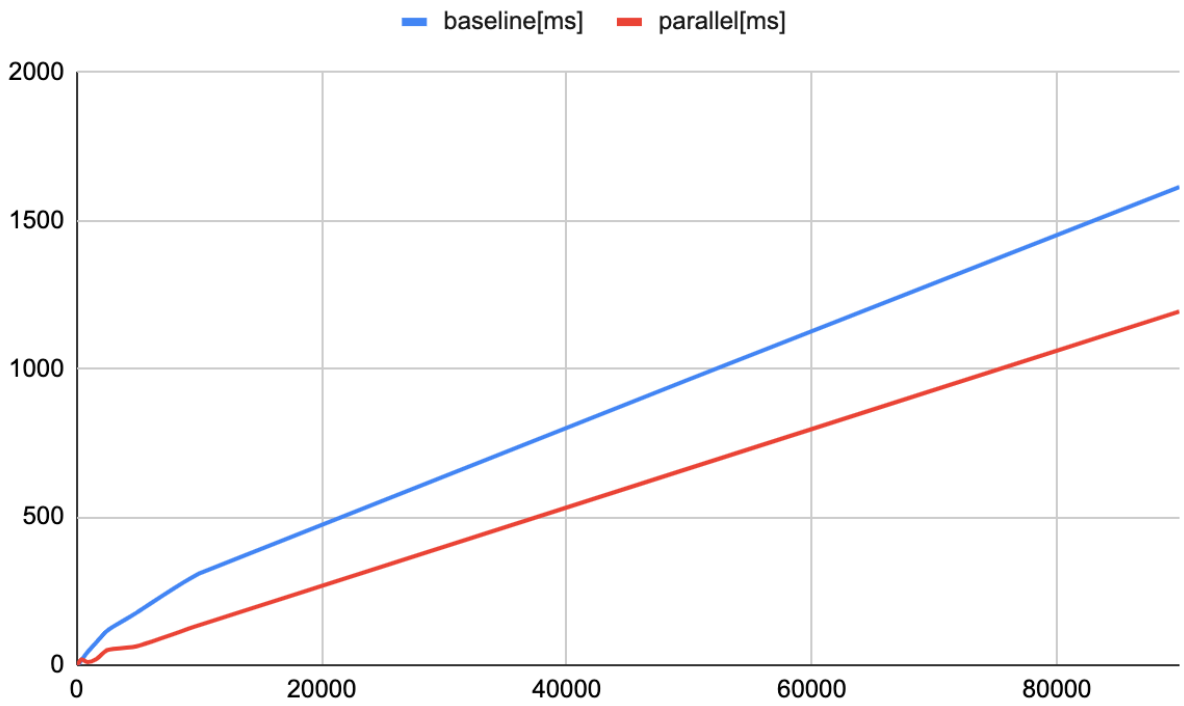

figure 8: simulation time across different input sizes (simulation time in milliseconds vs. grid size)

Performance Across Input Patterns

Our parallel algorithm is best suited for clustered inputs. It reduces to mere BFS on extremely scattered clusters. To test the exact performance, we created four different patterns: oscillating strips, scattered squares, progressing strips, and random. Neither of them extinct in 50 iterations. We measured their performance at 50 iterations, with grid size 100 * 100.

	baseline [ms]	parallel [ms]	speedup
oscillating strips	268.493	275.44	0.974
scattered squares	459.69	347.09	1.324
progressing strips	141.93	62.07	2.286
random	177.12	107.00	1.655

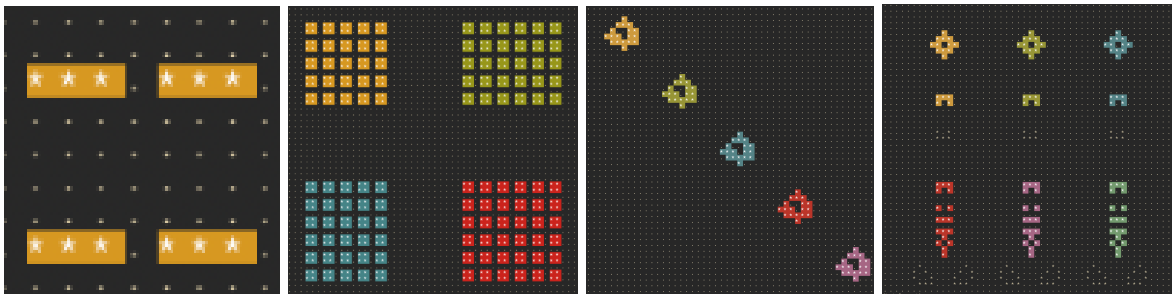

figure 9: oscillating strips, scattered squares, progressing strips, and random

For “oscillating strips”, since the 1*3 strips are evenly separated from each other, they were treated as a giant nation. Our parallel version reduces to pure BFS. The parallel version performs worse because it needs to do the useless convolution work in addition to BFS. “Scattered Squares” also consists of repeated patterns but they are scattered in small nations. We can therefore launch multiple BFS at the same time and there is a tiny speedup. We observe the highest speedup on “progressing strips”, a pattern that won’t converge until the simulation step specified. The number of lives remains approximately the same but they are constantly moving. We can see the benefit of clustering as the lives move closer together.

Conclusion

We successfully improved the performance for the Game of Civilization. We reached our targeted speedup on the cell updating phase but did not reach the targeted speedup for tribe detection. This proves our approach of dissecting the graph into smaller independent subsections could work, which we consider it as a nice attempt at parallel graph processing.

REFERENCES

StackOverflow https://stackoverflow.com/

NVIDIA DevTalk https://devtalk.nvidia.com/

OpenMP Reference 5.0 https://www.openmp.org/resources/refguides/

Previous Homeworks

CONTRIBUTION

Eryn - 50%

Yating - 50%