Group Partners: Amy Lee (alee3) & Jan Orlowski (jorlowsk)

Final Project Checkpoint

Make sure your project schedule on your main project page is up to date with work completed so far, and well as with a revised plan of work for the coming weeks. As by this time you should have a good understanding of what is required to complete your project,I want to see a very detailed schedule for the coming weeks. I suggest breaking time down into half-week increments. Each increment should have at least one task, and for each task put a person's name on it.

- One to two paragraphs, summarize the work that you have completed so far. (This should be easy if you have been maintaining this information on your project page.)
- Describe how you are doing with respect to the goals and deliverables stated in your proposal. Do you still believe you will be able to produce all your deliverables? If not,why? What about the "nice to haves"? In your checkpoint writeup we want a new list of goals that you plan to hit for the poster session.
- What do you plan to show at the poster session? Will it be a demo? Will it be a graph?
- Do you have preliminary results at this time? If so, it would be great to include them in your checkpoint write-up.
- List the issues that concern you the most. Are there any remaining unknowns (things you simply don't know how to solve, or resource you don't know how to get) or is it just a matter of coding and doing the work? If you do not wish to put this information on a public web site you are welcome to email the staff directly.
- Meet with the course staff to discuss your progress

Summary

The first step was to gain an understanding of the implementation of the algorithm to analyze the performance and identify bottlenecks upon which to focus our efforts. After reading through the C++ code we identified the following functions that would take up most of the execution time and have potential for parallelization:

- `observe`: The purpose of this function is to determine the current state and entropy of the grid. It makes a call to `find_lowest_entropy(..)`, which iterates over every cell in the grid to find the cell with the smallest entropy. It also sums up the entropies from all of the cells in order to check whether the total entropy is 0, indicating an "impossible" state. The `observe` operation is also performed during every iteration of the algorithm, which means it is called quite often.
- `OverlappingModel::propagate(..)`: The purpose of this function is to propagate changes throughout the grid. It currently loops over every cell and every pattern to propagate changes from changed cells to neighboring unchanged cells. This function is also

repeated as long as there are any unpropagated changes. There are 5 nested for loops here, which has potential for parallelization.

We timed both of these functions using a high resolution clock from the `<chrono>` library. We found that these two functions take up most of the execution time, with propagation taking the longest to execute. We decided that our first goal would be to parallelize the propagation step.

After reading through the propagation step, we agreed on 3 possible avenues of parallelizing propagation:

- GPU with CUDA: If we propagate over the entire grid every time, we can try propagating by assigning each thread in the kernel in the GPU a single cell in the grid, or by dividing the grid into smaller chunks assigned to each thread.
- CPU multi-core V1: If we divide the cells into batches of rows, we can assign each set of rows to a thread and therefore do each set of rows in parallel.
- CPU multi-core V2: We can rely on the fact that when a cell changes to have less possibilities, the only cells that are affected are its neighbors. Therefore, we could start at the changed cell and then keep track of cells that could be propagated to with a lock-free queue that each thread grabs from and adds new potential cells to.

Our second step was to establish test cases, we looked at the existing examples (i.e. tile sets with constraints) and picked the ones that a varied number of constraints (e.g. ones with no constraints, some constraints or a very large number of constraints) and then for each tileset had it create images of varying size. Unfortunately, some tilesets with very strong constraints fail very often and cannot create large images reliably.

Preliminary Results

- We were able to double the execution speed of the algorithm by using simple OpenMP loop parallelism in the propagation step.

Issues

- Some sets of constraints that we consider interesting test cases are unable to reliably generate images past a certain size, as the algorithm keeps running into contradictions too many times (and the program fails after a number of tries).
- We identified several more functions with parallelization potential, but some of them may be difficult to parallelize because they update data structures (such as vectors) that do not support parallel threads operating on it at the same time.
- We may also need to clean up and re-read the base code again, as there are some variables and processes that remain a little unclear.

Updated Goals & Deliverables

We still plan to produce results for each of the three approaches for parallelization listed above. For each approach, we will have a graph that shows the speedup of the algorithm and how it changes with problem size. We will also include the large images generated by the parallel algorithm as examples. We will do all of these in 2D, as I do not think we have time to do anything related to a 3D grid, unless we end up having a lot of spare time.

On a more detailed level, some other parts of the code we may try to parallelize include:

- `find_lowest_entropy(..)`: instead of sequentially iterating over / summing over all the cells in the array, we may want to parallelize this step using reduction.
- `OverlappingModel::graphics(..)`: this is another function that sequentially iterates over the grid to compute a sum of entropies(?) for each pixel in a cell. We may also want to parallelize this step using reduction. However, it also updates a vector data structure during each iteration, so it may not be possible to parallelize this step.
- `image_from_graphics(..)`: this function sequentially iterates over pixels in the grid to set the pixel colors. Each pixel is independently updated, so we may want to parallelize this step.

Updated Schedule

Date	Goal	Description
Nov 8	Research + Literature Review	Read the code and look into descriptions of the algorithms to make sure we fully understand what the algorithm is doing on an implementation level
Nov 11	Code Analysis	Time code to find where to focus parallelization efforts
Nov 16	Initial simple propagation parallelization attempts (OpenMP)	Use basic OpenMP pragmas to see if we can speed up propagation
Nov 18	Checkpoint	Formal progress report
Nov 21	Develop grid correctness checker (Jan & Amy)	Write code that will ensure our parallel solution did not finish with an invalid solution that violates constraints (currently we check results by eye which is not ideal)
Nov 25	Finish Multi-Core CPU propagation parallelization	We will look to see if we can get any speedup past using OpenMP primitives. If

	approach V1 and record results (Jan)	not, We'll see how far we can get with using OpenMP primitives to their full potential.
Nov 29	Finish GPU propagation parallelization approach and record results (Amy)	Using CUDA, we should be able to dramatically boost the propagation step by propagating form each cell at the same time.
Dec 2	Finish Multi-Core CPU propagation parallelization approach V2 and record results (Jan)	We're hoping that a lock-free queue will be comparably fast to accessing cells and we will be able to use the queue to distribute work between threads in a manner that does not require us to scan the entire grid of cells and gives each thread the same amount of work.
Dec 5	Attempt parallelization of cell observation + any other approaches that come to mind after (Jan & Amy)	We will try to use the techniques that speed up propagation to improve the observation step as well as any other small parts of the code. If we get any new ideas on how to parallelize the algorithm, we will do them up to this point.
Dec 8	Finish timing all of the different attempts and compile the final results (Jan & Amy)	After all versions are complete, we will time them all and create all of the graphs that show our speedu as well as compile example generated images.
Dec 9	Final Report	
Dec 10	Presentation	