Rishabhy-yiannig

Project 2 Beta-WriteUp

After reading through the project prompt we decided to implement a quadtree. We designed the quadtree so that it can be reconstructed for each time step, predicting that we would not have enough time to implement a quadtree that did not need to be reconstructed for each time step.

For our quadtree design, we decided to construct it in a recursive fashion, where the child of each parent was given a subset of the lines of the parent that fit within the child. Lines that did not fit in any of the child quadrants were designated to the parent. Each node held a pointer to an array of the lines that were contained within that node. Once we were able to successfully construct the quadtree, we then designed a search algorithm to detect all of the collisions. Our original design for the search algorithm consisted of starting at the root node, comparing the lines contained within the root node to check for collisions at that level, then passing each line to every descendent to check for additional collisions. We repeated this process for each node in the tree until we reached the leaves. This implementation was significantly faster (approximately 400% to 3000% times faster depending on the input file) than the original O(n2) implementation.

Our next step was to implement more optimizations before parallelization. For the quadtree construction, we experimented with using a “quad” linked list to store all the lines. So rather than storing nodes at each level that pointed to the lines contained within that node, every time we partitioned the space, lines would get added to a linked list that started at the leaves and pointed to the root. To clarify, after we partition the space for the first time, lines that did not fit within any of the child nodes would be added to the linked list. Lines that did fit in each child would be passed to each child in the tree. Each child would also receive a pointer to the last element in the linked list. Each line that was not “contained” at a child node would now be added to the linked list. This would potentially branch off into four branches for each level in the tree. We hoped that we could then store the leaves of the linked list in a separate array and loop through the leaves and compare all the lines within a given path from the leaf to the root for possible collisions. We thought this would improve performance because the collision search operation could be easily parallelized for each path from the leaf to the root. It turns out that this implementation was slower than our original. We hypothesized that this decrease in performance was due to the fact that we needed to call malloc() for each node in the linked list during the quadtree construction. This was significantly more malloc() calls than needed for allocating arrays at each node to store the lines contained within each node. After attempting this we decided to revert back to our original quad tree implementation.

We were able to improve performance in our collision search. Rather than passing each line down to the children to check for collisions, we stored a pointer at each node to the parent and would compare all the lines in a current node with all the lines in the parents by looping through all the parents. We then refactored parts of the line collision detection code, such as rearranging the locations of if statements, storing the length of a line, and storing the velocity of a line, to limit the number of repetitive calculations we needed to make.

The next step was to parallelize the process. We parallelized the construction of the quadtree by spawning the recursive calls for each child. We then parallelized the collision search algorithm by replacing IntersectionEventList with a reducer, looping through the lines at each node in parallel, and spawning recursive calls to the children of each node. We finally tuned our algorithm by adjusting the size of R and setting a max depth for quadtree in case there were more than R lines that collide with each other at a given time step.