Carnegie Mellon University

Large-scale Crowd Collision Simulations with Parallel Quadtree

Rose Liu (roseliu), Elly Zheng (ellyz)
Parallel Computer Architecture and Programming (15-418)
Professor Mowry, Professor Railing
28 April 2025

Website: https://github.com/ellyzh/crowd-simulation

Table of Contents

SUMMARY

We implemented a C++ crowd collision simulator with a parallel quadtree in OpenMP. The simulator consists of a number of agents, independent moving objects, on a grid that are given a starting cardinal direction (north, east, south, or west), and each agent moves in their specified direction until a collision with either another agent or the grid edge. If two agents are predicted to move onto the same grid space, they bounce off each other and change directions instead of overlapping. The quadtree is leveraged to reduce the number of collision checks per agent by only returning relevant colliders. We ran our algorithm on the GHC machines and the PSC machines to check scalability. During the poster presentation, we plan to show graphs to compare the speedup achieved by our parallel quadtree algorithm to a sequential algorithm and a parallel algorithm without a quadtree for the crowd control simulation. Additionally, we also created a visualization tool using the Simple DirectMedia Layer (SDL2) library to illustrate the agents' positions as they move and interact with other agents on the grid, visually showcasing how the crowd collision simulation works.

BACKGROUND

General Algorithm

The overall algorithm used for both our sequential and parallel crowd collision detector is such that for every timestep, each agent calculates their next step (based on their direction and grid boundaries), and before the agent actually makes that move, the algorithm checks that none of the agents' next steps are the same as another agent. If no next steps overlap, which indicates no collisions, the agents will move to their originally calculated next steps. If two agents are trying to move to the same coordinates, the collision is resolved by assigning the agents a new direction and next step in that new direction. In our testing, this is repeated for 500 iterations.

Collision Definitions

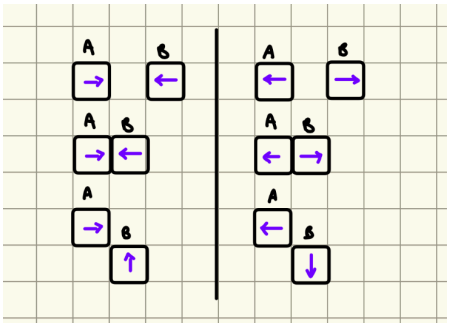


Figure 1: Types of Agent-Agent Collisions

Three possible collisions can occur, which are illustrated above in *Figure 1*. In the first scenario, two agents who share either the same x or y coordinate are attempting to move to the same location (they have the same *next_x* and *next_y*). This collision is resolved by setting the direction of the agents to be the opposite of their original direction. In the second scenario, two agents are adjacent but trying to move past each other (their *next_x* and *next_y*

equal the other's *x_pos* and *y_pos*). This marks a collision between the two agents, and their directions will be reversed to avoid the collision. Finally, two agents that are attempting to move onto the same square but are not travelling on the same axis may collide (they have the same *next_x* and *next_y*).

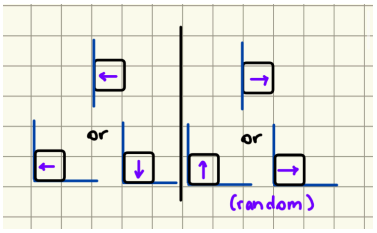


Figure 2: Types of Agent-Grid Collisions

In addition to agents colliding with each other, they may also bump into the edges of the grid since each simulation has a constrained grid size, as seen in *Figure 2*. If the agent collides with the top, right, left, or bottom edge of the grid, it bounces back and moves in the opposite direction. If the agent collides with a corner, the new direction of the agent is randomized between the two possible options.

Inputs

The crowd collision simulation takes in a text file as an input. The first row of the input file indicates the grid dimensions (*dim_x* and *dim_y*), the second row represents the number of agents, and the remaining rows describe each agent. The first column contains the x coordinate, the second column contains the y coordinate, and the third column contains the starting direction of the agent, with 0 representing north, 1 representing east, 2 representing south, and 3 representing west. We created multiple input files to test differing grid sizes and numbers of agents. Our largest input file tests 50,000 agents on a grid size of 8192 x 8192. The simulation outputs the computation time of the crowd collision simulator – this is measured with timing code that encapsulates the process of calculating agents' next steps, checking for collisions, resolving collisions, and updating agent positions.

Sequential Implementation

In our sequential implementation, a for loop iterates through all the agents in the input file and calculates their next location with a call to our *move_agent* function, which determines whether the agent takes a step in their specified direction or bounces off a boundary edge. Afterward, we use a nested for loop in our *check_collisions* function to check every agent's next step against every other agents' to see if their next steps overlap (*agent_a.next_x == agent_b.next_x && agent_a.next_y == agent_b.next_y*) or if their next steps would cause them to move onto another's current position *(agents_a.x_pos == agent_b.next_x && agent_a.y_pos == agent_b.next_y* and vice versa*)*. If there is a collision, they are resolved by sending the agents in the opposite direction (as described in *Figure 1*). While complete, this algorithm is extremely expensive, as to check for potential collisions, each agent must be compared to all the other agents on the grid (performing in $O(N^2)$).

Project Focus

Thus, **our project focuses on parallelizing this collision checking functionality**. To reduce the number of collision checks, we implemented a quadtree that returns an agent's nearby agents (only agents in positions it could potentially collide with, *collidable_agents*). A quadtree is a tree data structure that partitions a 2-dimensional space into four quadrants, and recursively subdivides quadrants into fourths until a condition, such as max tree depth (*max_depth)* or max agents per quadrant *(max_agents)*, is met.

A key strength of a quadtree is that it partitions the grid space into regions, allowing each agent to check collisions with nearby agents within its node instead of every pair of agents. They also dynamically adapt to agent density, as they will *split* when an area becomes crowded, while also being particularly effective and scalable for large grids, as only active regions are subdivided. Lastly, insertions into the quadtree are performed in O(log(N)) time, ensuring that querying the tree for *collidable_agents* and collision detection are fast even with increasing numbers of agents.

Workload

With this in mind, the workload of our implementation using a quadtree can be broken down into the following phases:

1. Initialization (initial quadtree insertions, setting up agents)
2. Agent movement
3. Collision detection
4. Collision resolution
5. Quadtree updating (inserting or removing agents from nodes based on movement)
6. Position updating

Initialization, agent movement, and position updating are independent, thus, they can be easily and very effectively parallelized with OpenMP directives. Similarly, collision detection can also be implemented to work independently, as it merely performs read-only queries on the quadtree for *collidable_agents*. Meanwhile, collision resolution and quadtree updating have dependencies, as errors and undefined behaviour may occur when collider positions are modified in the former and multiple threads attempt to update (inserting or removing) from the same node in the latter. Caution must be exercised in those phases, especially when parallelizing, and critical sections or locking may be necessary to ensure correctness and expected performance.

With our algorithm, one of our key challenges is that the quadtree itself is not very parallelizable. Agents frequently move across quadrants, requiring either rebuilding the quadtree or carefully removing and re-inserting agents into the quadtree. The former is computationally expensive, as it would need to be performed every iteration (of which there are 500). The latter would require careful implementation to avoid conflicts (as mentioned previously). Additionally, since quadtrees divide the grid into hierarchical quadrants, some subtrees may contain far more agents than others, leading to significant load imbalance across

threads. The hierarchical structure of the parent-child nodes also complicates parallelization, as nodes cannot be independently built without risking stale or conflicting data. Lastly, there can be substantial synchronization overhead during recursive operations to maintain correctness and prevent race conditions, which further limits parallel efficiency.

The simulation is primarily data-parallel, as agent movement and updates can be performed independently (except for collision resolution). Regarding locality, the quadtree naturally provides good spatial locality by clustering nearby agents into the same or adjacent nodes. Temporal locality is also present, as agents typically move only short distances between frames, allowing queries and updates to benefit from recently accessed memory regions.

APPROACH

Parallel with No Quadtree

Our parallel implementation of the algorithm had many iterations. To start, we extended our sequential implementation and introduced the use of OpenMP directives for our loops. A *#pragma omp for* is added above the for loop with our *move_agent* function that iterates through each agent and calculates its next move. Within our *check_collisions* function, two OpenMP locks are added to make sure that the *next_x* and *next_y* coordinates for an agent are not being edited by another thread. We enforced the order in which these locks are obtained to prevent deadlocking. However, each agent is still being compared to every other agent to check for collisions brute-force style. This was our parallel with no quadtree implementation, a baseline to compare our later quadtree implementations against and to inform our future design choices. We saw very little speedup and, with timing code, recognized that a majority of the computation time was spent on collision checking and synchronization (critical sections, locking).

Thus, our next step was to create our preliminary version of a quadtree class with functions that would aid in constructing the tree, including *split* (subdivides the given quadtree into four quadrants, creating a new quadtree for each), *insert* (recursively inserting an input agent to the quadtree), *getQuadrant* (returns the quadrant id that the agent is located in), *insert* (recursively traverses the quadtree to find the quadrant that the agent should be placed, splitting and redistributing agents if necessary), *get_leaf_node* (returns the leaf node that an input agent is located in), *collidable_agents* (returns all agents contained within the given leaf node, serving as the array of agents an agent in this code could possibly collide with), and *reset* (resets the entire quadtree). In *getQuadrant*, quadrants are assigned indexes such that 0 represents the top left, 1 represents the top right, 2 represents the bottom left, and 3 represents the bottom right. A quadtree leaf will store a vector of agents that could collide with another agent within its spatial quadrant.

First Quadtree Implementation

In our first quadtree implementation, we initialize a quadtree, and for each iteration, we reset and rebuild the quadtree by inserting each agent into the tree. Then, the quadtree is queried (calling *collidable_agents* on the result of *get_leaf_node* with the given agent) to find

potential colliders of a given agent, and then the agent is only compared against these potential colliders. Although this approach gave us the most updated values, as the quadtree was consistently being updated, we soon realized that rebuilding the tree for every timestep was greatly slowing down our algorithm, since agents do not move across quadrant boundaries in every iteration.

Another issue we encountered was that agents located near or on the edges of quadrant boundaries could potentially collide with agents in other quadrants, which would not be accounted for if we only checked neighbouring agents within an agent's quadrant. This required us to find ways to have an agent check for possible colliders in other quadrants. We devised that it would be most effective to expand the possible collider space to ensure completeness, such that an agent can be considered for multiple nodes.

Second Quadtree Implementation

Our approach was to extend our *getQuadrant* function, which originally only returned the index of the one quadrant that the agent resides in, to create a *getMultiQuadrant* function where all quadrants where there exists a position at which the agent could possibly collide are returned. Our strategy is illustrated in *Figure 3* below:

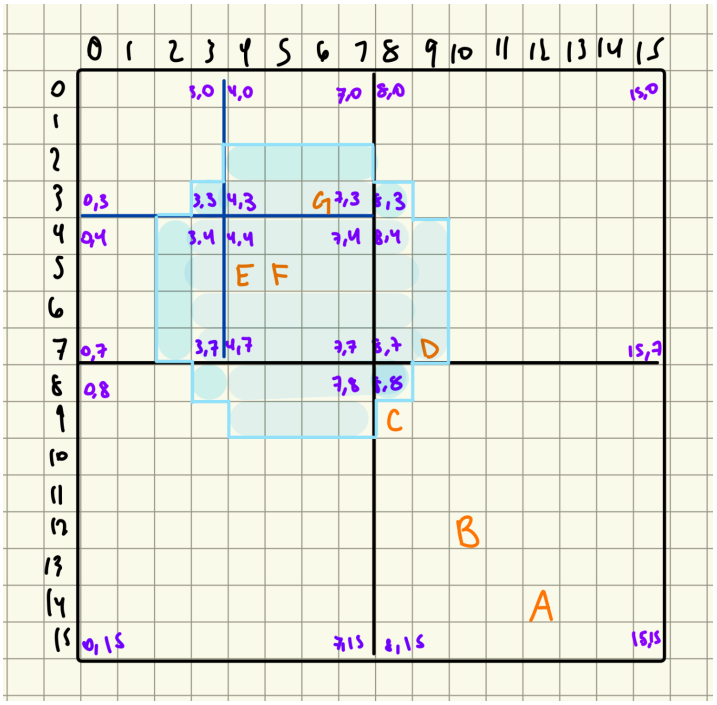


Figure 3: Quadrant Bounding Box

This figure represents a 16 x 16 grid with agents lettered A through G and the quadtree boundaries that divide the agents into their respective quadrants. Looking at agent F, which belongs to the bottom right quadrant of the top left quadrant, the blue highlighted area represents all potential areas where a neighbouring agent could potentially collide with an agent in F's quadrant.

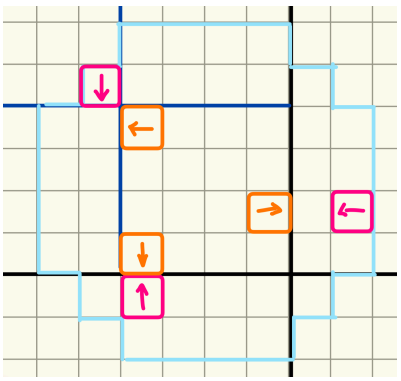


Figure 4: Collisions within Quadrant Bounding Box

To clarify, *Figure 4* focuses on the highlighted collision bounding box from *Figure 3*. The bounding box is formed by the positions of all possible colliders of an agent within the quadrant. Given that an agent belongs to this quadrant (orange), it could not only collide with other agents moving within the quadrant itself but also with agents in other quadrants, as previously explained (pink). This figure shows three possible collisions, each with the other agent belonging to a different quadrant, thus highlighting the necessity of this extended bounding box.

Thus, we introduce the concept of adding an agent to multiple nodes. In this same bounding box of the specified quadrant (based on *Figure 3*), we expect that a query for *collidable_agents* to the leaf node will not only return agents E and F, but also D and G. D and G are within this bounding box, and thus are capable of colliding with an agent within the quadrant.

The quadtree can be visually represented like the following:

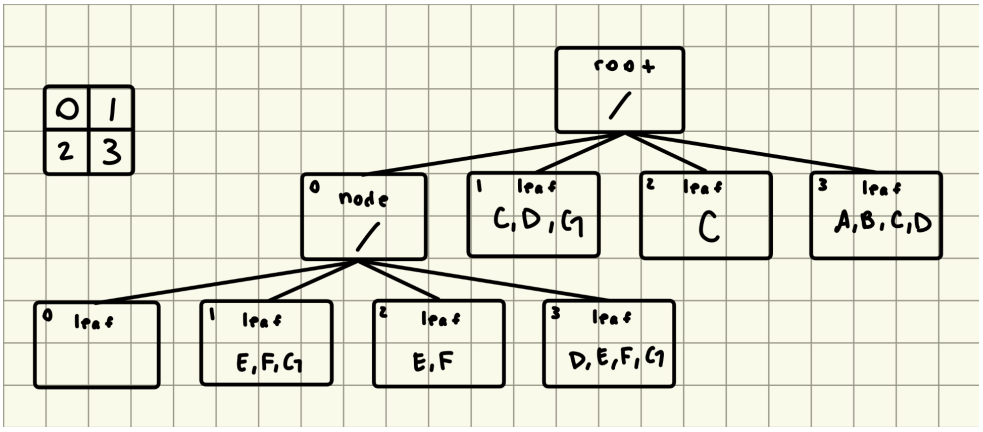


Figure 5: Quadtree Partitioning of Grid

The letters in each of the nodes represent the agents that an agent in that quadrant can collide with. By including nearby agents in neighbouring quadrants, we guarantee completeness and ensure that we are accounting for all possible collidable agents.

With this new *getMultiQuadrant* function, we extended our *insert* function to become *multiInsert*, such that the agent can be inserted into multiple nodes. After these changes were made, we noticed through our visualization that all types of collisions, including those on boundaries of quadrants, were accounted for.

Third Quadtree Implementation

Our next optimization was parallelizing the building of the quadtree, as it was originally implemented by making iterative calls to our *insert* function sequentially. After profiling, we observed that quadtree building was a significant bottleneck. However, simply parallelizing *insert* is unsafe, as multiple threads may simultaneously edit the same quadtree node, leading to data races or an invalid tree. Thus, we introduced locks into our new *multiInsert* function to ensure that two threads could not edit the same node concurrently. With these thread-safety measures in place, we were able to parallelize the insertion of agents into the quadtree using OpenMP directives.

On the topic of locks, we also removed the OpenMP locks on *check_collisions*, as we noticed that that was serving as a bottleneck. Instead, we let the agent with the smaller ID handle the storage of collider IDs. There was a tradeoff made between load balancing (the threads on the agent with the smaller ID will be doing more work) and reducing synchronization, but ultimately we saw that there was a speedup benefit – the reduced synchronization costs outweighed the potentially less balanced workload.

These changes and parallelization strategies greatly improved our performance and speedup across various numbers of threads; however, when we tested on the PSC machines with higher thread counts, we found that it was not scaling well.

Final Quadtree Implementation

We observed that the resetting and rebuilding of the tree for every timestep was greatly slowing down our algorithm. Although we were able to parallelize the process, the locks still enforced critical regions that introduced significant synchronization overhead, especially when there were many agents on the grid, and in any case, the constant rebuilding was excessive, as agents do not move across quadrant boundaries at every iteration. Thus, we brainstormed a strategy where we individually remove and reinsert agents into the quadtree, as necessary.

Initially, we planned to implement thread-local quadtrees, where each thread would build its own local tree before merging to a shared global quadtree. However, we realized that this approach would not be efficient. The recombination process would introduce significant logical complexity and likely offset any gains from parallelism, resulting in performance similar to or worse than our current implementation.

Instead, we devised a quadtree-id-based strategy. At the start, each quadtree node is assigned a unique ID upon creation, tracked with a shared counter. We build the quadtree once at the beginning of the simulation – before any iterations are run – using an iterative *multiInsert*

process. In this loop, we also record the ID values of the quadrants each agent is inserted into by storing them in a vector of vectors (*vector<vector<int>>*), *agent_leaves*, indexed by the agent's ID. This way, we can maintain a record of which quadtree leaves each agent occupies.

After each collision detection and resolution phase, we use a new function, *get_leaf_nodes*, which returns the set of leaf node IDs an agent should belong to based on its new position. We then compare this new set of IDs to the original set stored in *agent_leaves*. This comparison will detect whether the agent has moved into different quadrants or left any existing ones. If a difference is found, we remove the agent from its old locations using *multiRemove* (a recursive function that removes all instances of an agent from the relevant leaf nodes) and then re-insert the agent using *multiInsert*.

With this addition, we can maintain the quadtree's correctness without needing to rebuild it from scratch each iteration, significantly improving performance while preserving parallelism.

Visualization

To aid in our implementation, we created a visualization tool to better understand the movements of the agents and identify whether collisions were being resolved properly. We used the SDL (Simple DirectMedia Layer) library to set up the grid environment and render the agents as they moved for each timestep. When we noticed odd behaviour between agents through the visualization tool, we created specific input files to simulate these errors and identify how to fix the problem.

Computational Model and Problem Mapping

In general, we mapped our problem to a data-parallel computation model, which was the most intuitive choice given the nature of our simulation. Agents are independent objects with their own positions, movement logic, and collision interactions (aside from agent-agent collisions). Position updates were highly parallel across agents, and all of our for loops were parallelized using OpenMP's *#pragma omp parallel for schedule(dynamic)*, which enabled threads to dynamically steal work and achieve better load balancing when agent workloads varied. We leveraged OpenMP threads to fully utilize available CPU cores, with each thread responsible for processing a subset of agents (based on *schedule(dynamic)*) during the movement, collision detection, and update phases.

However, our quadtree data structure introduced additional complexity. Inserting agents into the quadtree naively was not parallel-safe, as multiple threads could attempt to edit the same tree nodes simultaneously, leading to race conditions, invalid trees, and segmentation faults. To address this, we introduced locks at the node level, allowing concurrent *multiInsert* and *multiRemove* operations without invalidating the quadtree structure. Thus, while agent arrays mapped cleanly to thread-level parallelism, the quadtree required careful synchronization to safely handle concurrent modifications (which was detailed thoroughly in this section).

We also carefully considered the structure of the quadtree itself, including its tree depth and maximum agents per node, to balance search efficiency and update costs. One of the main strengths of a quadtree is that it preserves spatial locality by organizing agents based on their positions in the grid. Instead of searching through the entire agent list, the quadtree narrows down the searches to the most relevant nearby agents. In smaller grid sizes with only a few agents, the quadtree is not as effective. However, when the grid size and number of agents are increased, the advantages of a quadtree can be clearly seen since the number of searches dramatically reduces the number of comparisons needed to find nearby agents. Thus, we created large input files to showcase the full benefits of the quadtree structure. Additionally, we targeted the PSC machines in addition to GHC, as their large thread counts allowed us to further scale performance: as the number of agents and the quadtree's size grew, more threads could operate on different parts of the quadtree, increasing overall parallelism.

Conclusion

Our final solution was not achieved in a single step. Instead, we went through several iterations of testing, profiling, and redesigning our algorithm. Throughout the project, we explored multiple strategies, identified their bottlenecks, and incrementally improved our approach based on experimental results. Ultimately, through a combination of data-parallel thinking, careful synchronization, and improvements to the quadtree functionality, we achieved substantial speedups across a wide range of agent counts and grid sizes.

RESULTS

Experimental Setup

Overall, our results were successful and show increasing speedup as we increase the number of threads. We measured our performance by calculating the speedup, the execution time on 1 processor divided by the execution time on P processors. We timed our code for two different portions, initialization and computation. The initialization time consists of the time the program takes to read the agent input files, create the vector of agents with their positions and directions, and initialize the quadtree. The computation time includes the time to build the quadtree by inserting the agents and running through 500 iterations of agent movements by calculating the agents' next step and detecting and resolving any potential collision. The execution time is gathered from our code, where we insert the timing code from the C++ chrono library.

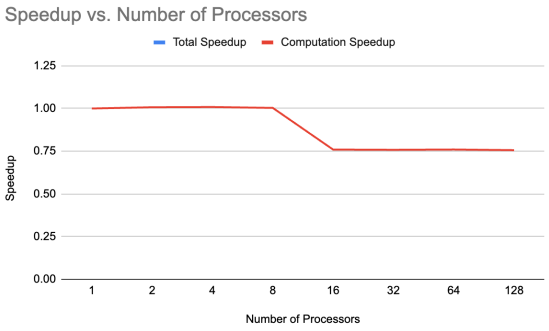
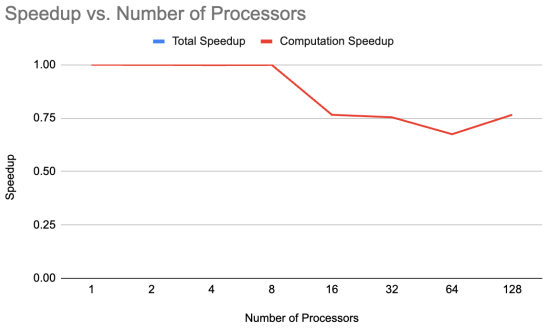
Our experimental setup includes various input files of differing grid sizes and agent numbers. We ended up testing our algorithm on four different input files: medium, large, dense, and sparse. These input files allowed us to determine how useful the quadtree was with collision detection in varying situations.

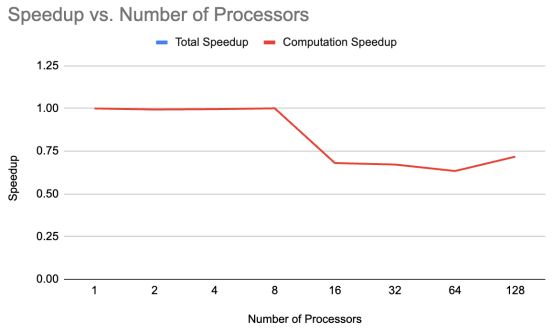
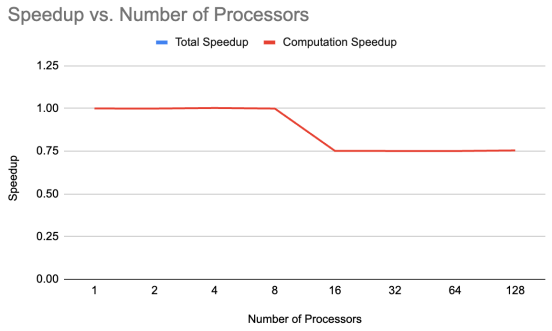
Serial

medium.txt (4096 x 4096 grid, 3,000 agents)	Initialization time (sec): 0.0009036250 Computation time (sec): 18.9723937740
large.txt (8192 x 8192 grid, 20,000 agents)	Initialization time (sec): 0.0031934910 Computation time (sec): 416.38128315
dense.txt (8192 x 8192 grid, 50,000 agents)	Initialization time (sec): 0.0067212590 Computation time (sec): 2327.28094055
sparse.txt (8192 x 8192 grid, 5,000 agents)	Initialization time (sec): 0.0007079990 Computation time (sec): 40.1533476300

For our baseline, we timed the serial algorithm and observed that it took a considerable amount of time, as it required looping through every agent to perform the necessary collision checks. As the number of agents increased in the input files, both the initialization and computation times grew significantly, with our largest input file taking 2,327 seconds to complete.

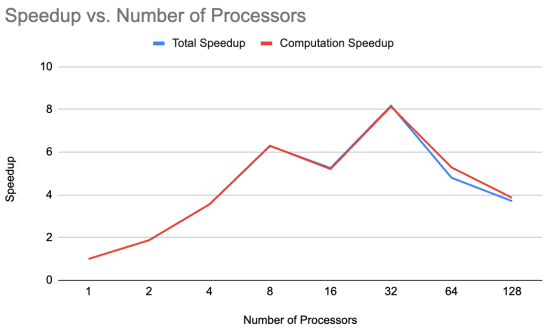
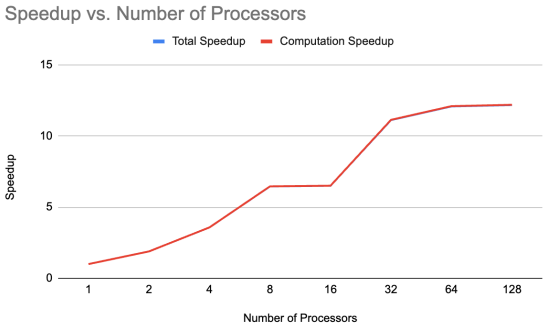
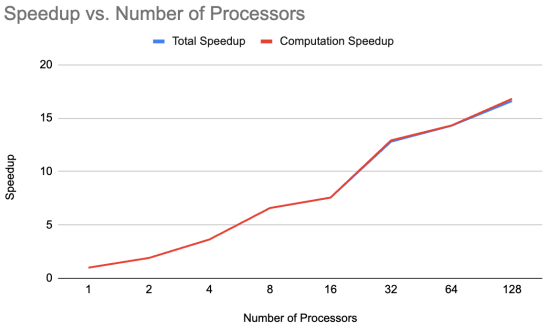
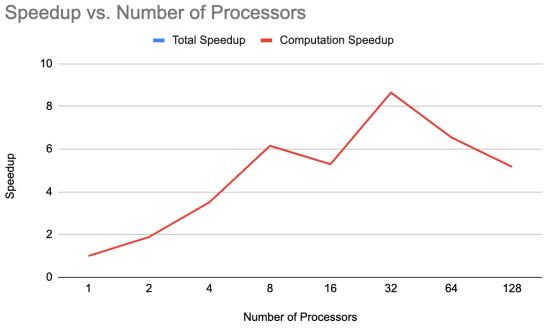
Parallel with No Quadtree

medium.txt (4096 x 4096 grid, 3,000 agents)	
large.txt (8192 x 8192 grid, 20,000 agents)	

dense.txt (8192 x 8192 grid, 50,000 agents)	
sparse.txt (8192 x 8192 grid, 5,000 agents)	

Our parallel implementation with no quadtree showed improved times from our serial algorithm, but in terms of scaling with the number of threads, we observed no significant speedup. Although work can be done in parallel with OpenMP, the time per thread is still very large. Without the quadtree to help optimize collision detection, each agent still needs to check for collisions against all other agents on the grid. As the number of agents increases, this results in a growing number of checks for each thread, which quickly becomes computationally expensive. Moreover, there is poor scaling because as the grid size and the number of agents increased, the total amount of work required grew much faster than the performance benefits of adding more threads. The overhead associated with managing multiple threads and synchronizing their work became a limiting factor, reducing the potential speedup.

Parallel Quadtree

medium.txt (4096 x 4096 grid, 3,000 agents)	
large.txt (8192 x 8192 grid, 20,000 agents)	
dense.txt (8192 x 8192 grid, 50,000 agents)	
sparse.txt (8192 x 8192 grid, 5,000 agents)	

Once the quadtree was implemented, we observed noticeable speedup as the number of threads increased. While the large and dense input files showed consistently improving speedup with increasing thread count, the medium and sparse input files revealed a drop-off in performance past 64 threads. The benefits of a quadtree are seen clearly in our large and

dense cases: the quadtree partitions the agents spatially, allowing the algorithm to focus collision checks on a much smaller subset of nearby agents rather than comparing each agent against tens of thousands of others. As the number of agents increases, the difference in the number of checks becomes even more pronounced. Furthermore, by spatially partitioning the grid, each thread can work on different parts of the grid independently, reducing overall computational load.

However, for the medium and sparse input files, performance began to decline once the thread count exceeded 64. This drop-off is most likely because there is not enough parallel work to fully utilize the additional threads, as the simulation reaches a point of saturation. Additionally, operations such as quadtree updates, which involve locking shared data structures, introduce synchronization bottlenecks that become more pronounced at higher thread counts. Furthermore, when the amount of work per thread becomes too small, excessive thread creation and management overhead, cache contention, and potential memory bandwidth limitations begin to outweigh the benefits of parallelism and even degrade the performance. In cases where the problem size is insufficient to fully occupy all threads, the overhead of coordinating and synchronizing parallel work ultimately outweighs the benefits of additional parallelism.

Breakdown of Execution Time on large.txt (8 Threads)

Component	Time (sec)	Percentage
Quadtree Building	0.0324702060	0.501%
Calculate Agent's Next Step	0.2856667290	4.413%
Quadtree Readjustment	3.6284403280	56.055%
Collision Detection	1.6743250540	25.867%
Collision Resolution	0.3805861450	5.880%
Move Agent	0.3917671800	6.052%
Total Computation Time	6.4729444590	98.768%

The table above shows a breakdown of each step within our crowd collision algorithm. More than half of the computation time is spent on the readjustment of the quadtree, where we determine if an agent needs to be reinserted into the quadtree if they move out of their original quadrant. The second most time-consuming step is checking for collisions, which is almost 26% of the total computation time. These two steps being the most time-consuming is expected because the agents are constantly moving, requiring updates to the quadtree, and if the number of agents on the grid is on the denser side, there would still be a large number of agents to check in a node.

Collision detection can be improved by more precisely determining the neighbouring agents with which an agent could potentially collide, thus reducing unnecessary checks. The readjustment of the quadtree can also be improved by rearranging how the nodes are structured and making trade-offs between having the most updated position and frequently rebuilding the tree, or having potentially stale data but adjusting the tree less. In addition, a lock-free implementation of a quadtree would also improve the efficiency of readjusting the quadtree as threads would no longer have to wait for locks, enhancing concurrency. The remaining components of the algorithm each account for less than 10% of the total computation time. This aligns with expectations, as these steps are largely independent and highly parallelizable.

Machine Target

Our machine target was a CPU to run it on GHC and PSC. We believe that the CPU was the right choice because the CPU allows for more control over the structure of the quadtree and dynamic updates. Since we did not plan to create massive inputs with extreme scalability, a CPU was sufficient for our problem statement. If we had targeted a GPU, we would have needed to flatten the quadtree and significantly restructure the algorithm to fit the GPU's parallel processing model.

REFERENCES

Quadtree and Collision Detection | pvigier's blog

SDL Renderer Example

numactl(8) - Linux manual page

Final Project Metrics - Spreadsheet of all the data collected for our graphs

LIST OF WORK BY EACH STUDENT AND DISTRIBUTION OF TOTAL CREDIT

Both teammates worked closely together on all aspects of the project, including the creation of our project idea, gathering resources, brainstorming test cases, and writing the project proposal, milestone report, and final report. We collaborated on the design, implementation, optimization, debugging, and testing of the parallel algorithms, as well as the development of the visualization tool. Overall, we feel that we contributed equally across all areas of the project and would like to elect for a 50/50 credit distribution.