Comparisons Between Parallel Octree Implementations

Katherine Abreu, Alexandra Brown, Ryan Bugge, and Ayiana Mallory
College of Engineering and Computer Science
University of Central Florida, Orlando, Florida 32816

***Abstract*—Octrees are tree data structures where each node has eight children nodes, or octants. They are commonly used for three-dimensional modeling and real-world problem solving where binary trees and quadtrees may fall short. In this paper, we have implemented a pointer-based octree using standard representation. We tested four different methods of synchronization, including coarse-grained, fine-grained, optimistic, and non-blocking synchronization. Our targeted problem involved finding an efficient parallel technique when it came to adding, searching, and removing nodes in octrees with the goal of having a lock-free octree implementation. Of the parallelization methods tested, our optimistic approach outperformed all of the other approaches, leading to speedups of about 2.6 times for removal, 4.6 times for insertion, and 2.2 times for mixed operations.**

***Index Terms*—parallel computing, octrees, locking**

I. Introduction

Octrees are tree data structures where each node has either eight children nodes or no children nodes at all. If they don't have any children, they are known as leaf nodes. In this case, each of the children nodes are known as octants. Octrees have many applications in three-dimensional computer graphics and can be used to represent the three-dimensional world. They can be defined as a "recursive, axis-aligned, spatial partitioning data structure" [1, para. 1] which helps optimize collision detection and nearest neighbor search. In other words, they allow us to locate the closest points from a given point as well as focus on a certain area in an optimal way. They are tree data structures with the purpose of subdividing into octants to split a three-dimensional space.

In this paper, we aim to find the most efficient way to implement and utilize octrees with parallel computing. We have implemented four methods of synchronization including coarse-grained, fine-grained, optimistic, and non-blocking synchronization.

The rest of this paper is organized as follows: Section II is about previous works involving multi-threading with octrees. Section III discusses the ways to implement octrees and the algorithms utilized for our research. Section IV describes the different methods of parallelization used and how we implemented them. Section V discusses our results of testing each parallelization method. Section VI compares our results with expected results and considers areas of further investigation.

II. Related Works

In 1992, Chaudhary et al. [2] presented methods for constructing and manipulating octrees in parallel machines. To construct the octree, they continually split a binary image into octants until each node was either all white (object) or black (non-object). To parallelize the octree, it was treated as three quadtrees, representing the top, front, and side views of a 3D object, traversed in parallel. They achieved a slight speedup in comparison to the sequential version. However, they did experience a bottleneck when it came to task creation and shared resources.

In 2005, Hariharan and Aluru [3] discussed software for parallelizing compressed octrees. The octrees were stored in a postorder traversal order and load balancing was not required. Using their method, they saw good speedup regardless of whether or not the distribution was uniform and it scaled well with the number of processors being used.

In 2010, Zhou et al. [4] presented the first parallel surface reconstruction algorithm that ran on the GPU. It used an octree to reconstruct a real-world 3D scan. Their octree implementation consisted of four arrays: vertex array, edge array, face array, and node array. The GPU implementation performed over two orders of magnitude better than a similar CPU implementation. Due to memory constraints, however, the octree could only have a depth of nine.

In 2012, Tero Karras [5] implemented an in-place algorithm for radix-binary trees used to construct octrees on the GPU. It outputted nodes in a strict depth-first order and scaled well, as well as performed better than other generation methods tested by avoiding sequential bottlenecks.

This is by no means a comprehensive review of the work done on octrees, but it provided insight on the types of results we expected to see.

III. Octree Implementation

There are generally two different types of octree data structures. They can be pointer-based octrees or linear-based octrees with certain advantages and disadvantages in each case. Both pointer-based and linear-based have a node structure that contains a pointer of all the objects a node encloses. They also have an Axis Aligned Bounding Box which is stored as two vectors called **Center** and **HalfSize** to represent the volume. They store the center and edge of the Axis Aligned Bounding Box, respectively [6]. We decided to focus on pointer-based octrees because it allowed us to traverse and update the octree in a quick and efficient way, allowing us to implement the parallel techniques we chose to incorporate. Having threads that continuously add, remove, and search concurrently will cause an octree to be updated frequently. Therefore, we decided pointer-based was the best choice.

There are three different ways to represent pointer-based octrees. One way is called *standard representation*. It involves having a pointer for each of the eight child nodes with the option of having a pointer for the parent node. This method also uses a Boolean flag, which can be called **isLeaf**. Its purpose is to allow us to know if the nodes are either inner nodes or leaf nodes. It also allows for on-demand allocation, meaning they don't need to allocate memory for all eight children at the same time, allowing us to save memory. Generally, it also takes up to 105 bytes of storage space [6].

Another type of representation is *block representation*, which is similar to standard, except it involves storing a pointer to all eight children rather than having one pointer for each of them. "That way the storage size of an inner node can be reduced from 105 bytes down to 49 bytes" [6, para. 7]. This ends up being only 47 percent of the original storage size. It also uses a Boolean flag and has the option of having a pointer for the parent node. The disadvantage is that it does not support on-demand allocation, since it would have to allocate all eight children every time a node is subdivided. In a way, it's like all eight children are one block, hence the name [6].

The last type of representation is *sibling-child representation*. Compared to standard representation, each octant only uses two pointers instead of eight. For example, given a created node, according to its node structure, the first pointer, **NextSibling**, points to the next child node that shares the same parent. The second pointer, **FirstChild**, points to the first child node of the node we have just created. This only ends up using one-fourth of the memory used for pointers, since it uses two pointers instead of eight. A consequence of using this is that it ends up "dereferencing on average four times more pointers" [6] when nodes are randomly accessed instead of sequentially. It also uses a Boolean flag and allows for on-demand allocation [6].

Out of all the three choices, we ended up using the standard representation. By using eight pointers for all eight child nodes, we were able to use on-demand allocation, meaning a child node only needed to be allocated when we found an object that needed to be located there. This was better than having to allocate all eight children every time, which would have led to memory waste. We also had the advantage of being able to update and traverse it easily and quickly by using all eight pointers, which was beneficial and crucial when working with concurrent threads. Block representation would have led to too much memory waste since there could have been many empty child nodes. Even though modifying and traversing the list would have been just as easy as the standard representation. In the case of the sibling-child representation, it would have ended up saving a lot of memory, but because of the way the pointers would have been set up, traversing it would have been complicated and slow, which wouldn't have correlated well with concurrent threads. Therefore, at the end, it depended on whether memory usage or traversal speed was important. In this case for us, it was traversal speed, since we were dealing with many threads that were adding, removing, and searching the octree concurrently. Thus, standard representation was the best option.

A. Algorithms Implemented

There were four algorithms we ended up using. They involved finding the location of the correct node, inserting points into the octree, subdividing a node, and resizing the octree without having predetermined bounds. Throughout all of the algorithms, we also incorporated Morton codes, also known as z-order curves, which involves positioning nodes inside of other nodes. Morton codes first denote the location of a node inside its parent node by using three bits to represent the three dimensions. The first bit is used for the x-axis, the second is for the y-axis, and the third is for the z-axis. Therefore, there are eight possible different values which go from zero to seven. For our pointer-based octree, these values were used to represent the index in the array that stored pointers to the parent node's children.

For the find algorithm, which can be observed in Algorithm 1, we take a point as input and output the correct node. In this case, instead of using the Morton codes directly, we translate it into an integer between zero and seven, which allows us to use it as an index to the children's array and store it in **next**. Every time it starts back at the beginning of the while loop, **next** is set to zero to get the new index. The way the algorithm works is that in the beginning, we set the current node to the root node and traverse the list until we encounter the leaf node. We know where to traverse to by comparing the x, y, and z values of the point to the values of the current node in order to know which location in the children's array the current node should traverse to. Once we encounter the leaf node, we return that current node. We know whether the current node is a leaf node by checking **curr.isLeaf**, which is the Boolean flag in the node structure.

Algorithm 1 Find

```
    Input: Point
    Output: Node
 1: curr ← root
 2: while !curr.isLeaf do
 3:     next ← 0
 4:     if Point.x ≥ curr.center.x then
 5:         next ← next + 4
 6:     if Point.y ≥ curr.center.y then
 7:         next ← next + 2
 8:     if Point.z ≥ curr.center.z then
 9:         next ← next + 1
10:     curr ← curr.children[next]
11: return curr
```

In the insertion algorithm, depicted in Algorithm 2, we also take a point as input and output a Boolean result. In this algorithm, we call `find()`, and as stated before, `find()` returns the node with the proper location. Afterwards, we check if the node contains the point already. If it does, we return false. Otherwise, we insert the point into the node. Finally, we check if we need to subdivide the node by checking if the size of the children exceeds the limit. In this case, we don't use a depth limit and instead we use a limit to the number of points an octree can have. The contains and remove algorithms are also quite similar to the insertion algorithm and behave almost the same way. The only difference is that we remove the point or check if the point is located at the node rather than insert it.

Algorithm 2 Insert

```
   Input: Point
   Output: Boolean
1: node ← find(Point)
2: if node.contains(Point) then
3:     return False
4: node.points.add(Point)
5: if node.points.size $\geq$ limit then
6:     node.subdivide
7: return True
```

The subdivision algorithm doesn't take any input or produce any output. In this case, we first mark the Boolean flag, **isLeaf**, to false, so that the current node is no longer a leaf node, since high principle leaf nodes do not contain children. Next, we calculate the new half size of the node's children, which is just the current **halfSize** divided by two, and store it into **newHalfSize**. We can then calculate the center of each child node by using the variables in the three nested for loops as shown in Algorithm 3. Each dimension has a specific formula where all three follow the same structure. The only difference is the value they obtain from **center** and whether they are using the **i** value, the **j** value, or the **k** value from the nested loops. It's these three nested for-loops that allow us to obtain the Morton code. From there, the children's array is populated with new nodes that are initialized using the new center positions, which were stored in **X**, **Y**, and **Z**, as well as **newHalfSize**. Next, every point in the current node needs to be removed and re-inserted back into the child node it belongs in. This section follows a similar structure to the find algorithm where it first needs to locate the index of the children's array by using the **x**, **y**, and **z** values.

Lastly, we also have a resize algorithm, which is used when the bounds are not precomputed beforehand. This is depicted in Algorithm 4. Because this algorithm was difficult to implement for concurrency, for each synchronization method we focused on, we had two versions. Version one involved the resizing algorithm and version two involved using precomputed bounds. Therefore, this resizing algorithm was only used for the synchronization methods we coded in version one. In this algorithm, we take a point as input and produce no output. The first thing we do is check if the point is out of bounds. If it is, the octree must continually resize until it can fit the point. Once we know the point is out of bounds, we need to determine the direction the point has in relation to the center of the octree. In this case, instead of using the

Algorithm 3 Node Subdivision

```
 1: isLeaf ← False
 2: newHalfSize ← halfSize $\div$ 2
 3: for i $\in \{0, 1\}$ do
 4:     for j $\in \{0, 1\}$ do
 5:         for k $\in \{0, 1\}$ do
 6:             X ← center.x - newHalfSize + i $\times$ halfSize
 7:             Y ← center.y - newHalfSize + j $\times$ halfSize
 8:             Z ← center.z - newHalfSize + k $\times$ halfSize
 9:             children[i $\times$ 4 + j $\times$ 2 + k] ← Node({X, Y, Z}, newHalfSize)
10: for all Points in Node do
11:     remove(Point)
12:     next ← 0
13:     if Point.x $\geq$ center.x then
14:         next ← next + 4
15:     if Point.y $\geq$ center.y then
16:         next ← next + 2
17:     if Point.z $\geq$ center.z then
18:         next ← next + 1
19:     children[next].points.add(Point)
```

Morton code for the direction of the point, we initially set the values in the variable, **direction**, with ones. From there, based on the comparisons of the values in **Point** with the values in **root.center**, those in **direction** get set to negative one. Within that same step, we also calculate the Morton code so that we designate a location that the current root will have inside of the new root we are creating. Since we always expand the octree towards the point, the Morton code of the current root is always in the opposite direction of the expansion. The new root is then subdivided in order to initialize its children. It follows a similar approach to the subdivision algorithm. The only difference is that in their formulas they focus on the values of the current root and the values of the direction when subdividing. We then use the **rootCode** of the current root and replace the corresponding node in the child array with the current root. Finally, at the end of resizing, we replace the octree's root pointer with the newly created root. Since the point may still reside outside of the resized octree, the process is repeated until the point is finally within bounds of the root node.

IV. Methods of Parallelization

As stated before, we tested the concurrency of adding, removing, and searching in octrees by testing it with four synchronization methods: coarse-grained, fine-grained, optimistic, and non-blocking.

A. Coarse-Grained Synchronization

The coarse-grained synchronization algorithm is essentially a sequential algorithm and it involves the use of only one lock. Each thread must acquire the lock before it is able to access the list of items. After the thread locks the object, it can begin execution. Afterward, it releases the lock, ready for another

Algorithm 4 Resize

```
    Input: Point
 1: while Point is out of bounds do
 2:     if Point.x $<$ root.center.x then
 3:         direction.x $\leftarrow$ -1
 4:     if Point.y $<$ root.center.x then
 5:         direction.y $\leftarrow$ -1
 6:     if Point.z $<$ root.center.x then
 7:         direction.z $\leftarrow$ -1
 8:     newHalfSize $\leftarrow$ root.halfSize $\times$ 2;
 9:     newCenter.x $\leftarrow$ root.center.x + direction.x $\times$ root.halfSize
10:     newCenter.y $\leftarrow$ root.center.y + direction.y $\times$ root.halfSize
11:     newCenter.z $\leftarrow$ root.center.z + direction.z $\times$ root.halfSize
12:     newRoot $\leftarrow$ Node(NULL, newCenter, newHalfSize)
13:     newRoot.subdivide
14:     rootCode $\leftarrow$ (direction.x + 1) $\times$ 2
15:     rootCode $\leftarrow$ rootCode + (direction.y + 1)
16:     rootCode $\leftarrow$ rootCode + (direction.z + 1) $\div$ 2
17:     newRoot.children[rootCode] $\leftarrow$ root
18:     root $\leftarrow$ newRoot
```

thread to access the list. Thus, the linearization point for any method call is the moment the thread obtains the lock. This is essentially used for all types of method calls, such as `add()`, `remove()`, and `contains()` [7].

1) Our Implementation:

We decided to follow a similar approach when implementing the coarse-grained synchronization method into our program. In our method, every time a thread calls `insert()`, `remove()`, or `contains()`, it attempts to acquire the lock. Once it has the lock, it acts as a signal that they have locked the entire octree. Once the thread is done adding, removing, or searching for an object, it releases the lock.

As mentioned before, we implemented two versions of this. Version one uses `resize()` whereas version two uses precomputed bounds. In this case, both versions utilized the same approach above. The only difference is that version one calls the `resize()` function inside of `insert()` after acquiring the lock and before inserting the point to check if it needs to be resized before inserting. It also allows the thread to resize before any other thread locks the octree.

To make this more optimal, for both versions, we added a finally block at the end to guarantee that the thread will release the lock no matter what happens after the try block exits. This ensures that even in the case of an exception being thrown, the thread will still release the lock. This was an easy implementation and is considered to be correct with an average runtime of 2.20 seconds for version one and an average runtime of 2 seconds for version two when doing insertion and removal with twelve threads. The problem is that it doesn't bode well with contention. A bottleneck can occur with the issue that if one thread gets delayed, the rest of the threads will be delayed as well. We expected this runtime to be worse compared to the other parallel techniques we implemented.

B. Fine-Grained Synchronization

Fine-grained synchronization is similar to coarse-grained synchronization with the exception that, instead of locking the entire list, each thread traverses through the list using two locks: one for the predecessor and one for the current node. All the threads are able to go down the list together and only lock individual nodes in the list. It involves a few more rules with the way they are locked. Once the predecessor is locked, the current node can be locked as well, but only while the predecessor remains locked. If the predecessor is unlocked and then the current node is locked, there's a chance another thread could delete that current node or insert another node in between before the current node is locked. Another name for this is lock coupling. All of the methods must also lock in the same order to avoid deadlock [7].

This type of synchronization is deadlock-free as well as starvation-free. As stated before, deadlock is avoided because all the methods lock the predecessor and current node in the same order, which in this case, is predecessor and then current node. Let's say there are two nodes *a* and *b* where *a* points to *b* and let's say the `delete()` function is set to have the thread lock the current node and then lock the predecessor while the `add()` function is set to have the thread lock the predecessor and then the current node. If a thread calls `delete()` and locks node *b* while another thread calls `add()` and locks node *a*, the first thread will forever wait for node *a* to unlock while the second thread will forever wait for node *b* to unlock, resulting in a deadlock. Since fine-grained synchronization makes sure that each method locks in the same order, deadlock-freedom is achieved. Starvation-freedom is also achieved since the threads go down the line with no deadlock. One thread will never starve from locking a node because no matter what happens, the threads must unlock at some point and continue down the line. This is especially true because of the finally block in the implementation that guarantees the threads will unlock even if an exception occurs [7].

1) Our Implementation:

We also followed a similar approach in both of our versions of fine-grained synchronization. Both versions involve using a predecessor and a current node. The current node gets set to root and once it's locked, we find the next location in the octree. Once we do, we set the predecessor to current, so that the predecessor will now be locked. Then, once we set current to the next location, we lock the current node and then unlock the predecessor, starting the cycle again. This happens for `contains()` and `remove()` for both versions. This also happens for `insert()` in version two.

For `insert()` in version one, things are a little different, since it involves `resize()`. In this case, for version one, instead of locking the root right away, both `insert()` and `resize()` start in a while loop if it hasn't been able to lock the root in order to make sure that if a thread is in the middle of resizing, the thread trying to insert will have to wait and vice versa.

Similar to coarse-grained synchronization, we also followed a similar pattern to optimize it. We added a finally block for both versions in order to make sure that no matter what happens, the thread will release the lock. With this implementation, we had an average runtime of about 5.0 seconds in version one and an average runtime of about 3.20 seconds in version two when doing insertion and removal for twelve threads. Unfortunately, it produces a long chain of acquiring and releasing locks, making the algorithm inefficient. It also has the problem of allowing some threads to block others that need to lock nodes later down the octree. Therefore, we focused on trying to limit the number of times we had to lock, which led us to creating an optimistic synchronization method.

C. Optimistic Synchronization

As stated before, optimistic synchronization further reduces synchronization costs by reducing the amount of locks needed. Optimistic approaches do not lock while traversing the data structure. Instead, once the thread finds the target node, it will lock the target and its predecessor and proceed to validate the data structure. If everything is as it should be, the thread continues with its operation. Otherwise, the thread will restart its traversal. This occurs when there is a synchronization conflict where the wrong nodes are locked. Fortunately, this type of situation can be rare. When validating it, the threads need to make sure they also check if the predecessor still points to the current node. This is because the references leading to the predecessor or the reference between the predecessor and the current node could have changed between the moment they were read and the moment the thread locked them. Unfortunately, the method is not starvation-free because if a thread is trying to insert or remove, they could get delayed forever from other threads continuously adding and removing. This is especially true because of the validation method. They would have to keep re-traversing the list, since the reference would keep changing between the predecessor and the current node [7].

1) Our Implementation:

Just like the optimistic synchronization method mentioned above, in `insert()` and `remove()` we lock the node after we have found its location. In this case, we don't need a predecessor anymore, which means to validate, we check and make sure the node is a leaf node before removing or inserting the vertex. The way the octree is built, we only need to use a predecessor when having to lock while going through the traversal, which is why we don't have one for the optimistic approach. When inserting, it also subdivides, if needed, after the vertex has been added.

This general structure is followed for both versions; except there's an added component in `insert()` for version one. Before finding the node, we first check if the octree needs to be resized. If it needs to be resized, we use an AtomicBoolean **resizing** to make sure only one thread is resizing at a time. When it can resize, it validates one more time to make sure the vertex is still not within the bounds of the octree. It stays in this while loop until the vertex is in bounds and doesn't need to be resized anymore.

Because we are using locks, we use a finally block that ensures they will unlock even if an exception is thrown. This is the same optimal approach we used for coarse-grained and fine-grained synchronization. The average runtime for version one was about 0.45 seconds and for version two it was about 0.49 seconds when doing insertion and removal with twelve threads. This is a far better implementation than the last two because it only locks when it finds the location rather than locking as it traverses. This allows other threads to move forward if they need to access other nodes down the octree. With the way the octree is created, we don't need to work with predecessors and current nodes anymore, allowing `contains()` to be lock-free and wait-free. The reason why is because it only needs to traverse and find the location using `find()`, which is established to also be lock-free.

D. Non-Blocking Synchronization

Non-blocking synchronization is a parallelization method that doesn't make use of any locks. `contains()` is wait-free while `add()` and `remove()` are lock-free. The primary way to achieve the lock-free nature of this method is by using the atomic method `compareAndSet()`. Every time a method tries to remove, add, or check if an element is in a list, it validates actions with `compareAndSet()`. If it fails, it retries again, going back to the beginning. It does this if the reference between the predecessor and the current node has changed, which is usually when `compareAndSet()` returns **false** [7].

1) Our Implementation:

There are more components involved in the method, but the ones mentioned above about non-blocking synchronization are the ones we implemented in our lock-free approach. In our approach, the functions `insert()`, `remove()`, and `contains()` all have a while loop that allows them to retry if they fail to do their objective.

For both versions, `insert()` is made lock-free by checking certain instances before inserting the node. After finding the location in the octree, the thread makes sure the node is a leaf node and that the vertex hasn't already been added into the octree by another thread. When it attempts to insert, it first checks if it needs to subdivide. If it does, it uses `compareAndSet()` to then set the AtomicBoolean **subdividing** to **true** so that no other threads can subdivide. Then, it adds the vertex into the octree. On the other hand, if it doesn't need to subdivide, it checks to make sure no other thread is subdividing. If they are, the insertion failed. Failing any of these cases means the thread needs to start all over again.

In terms of `remove()`, the thread first checks if the vertex is in the octree. Then, it enters a while loop and finds the node that contains the vertex. Once it finds it, it makes sure that the node is a leaf node and then attempts to remove it. If it's not a leaf node or if it fails, it tries again by finding the new location.

Every time both `insert()` and `remove()` check if the vertex has already been added to the octree, it does so by calling on `contains()`. Although `contains()` is wait-free in the optimistic approach, `contains()`, on the other hand, is not able to be wait-free. Part of the reason is because of subdivision. It has a while loop that will continually retry if the node it found is not a leaf node or if the node is in the middle of being subdivided. The reason why it needs to loop again if there's subdivision is because the points in the octant could have been modified during subdivision before it was being read by `contains()`.

Since `resize()` is already lock-free in the optimistic approach we did, we were able to incorporate it into `insert()` in version one. It's called before we check if we can insert the node, which means, if needed, it would resize before entering the while loop.

In this case, the method does not use any locks, therefore, it does not need to implement a finally block unlike the other methods. Similar to the optimistic approach, we were also able to make `contains()` be lock-free even though it's not wait-free. Because we avoid using locks, we guarantee that some thread will eventually return from a method call. This definitely helped it improve its runtime compared to the coarse-grained and fine-grained approach. Here, the average runtime for version one was about 0.75 seconds and for version two it was about 0.5 seconds when doing insertion and removal with twelve threads. A possible reason why optimistic outperforms this approach by a small margin could be because `contains()` was not able to be wait-free like the optimistic approach.

V. Results

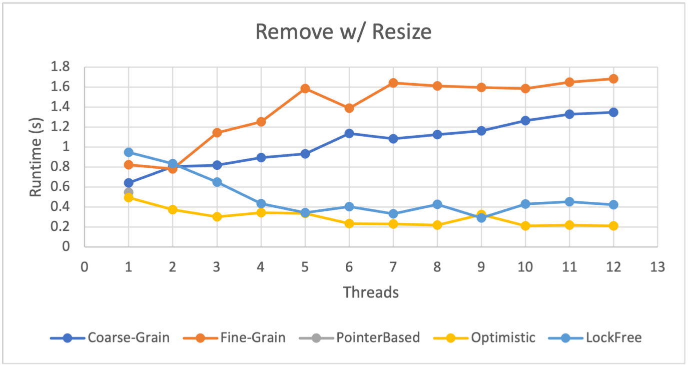
To test the different parallelization methods we ran a stresstest on a 12-core processor with 32gb of RAM, with up to twelve concurrent threads running. For both the resizable and bounded octrees, we tested the concurrency of only inserting nodes, of only removing nodes, and a mixture of inserting and removing them. We inserted or removed a total of 1,009,026 points in each test.

A. Resizable Octree

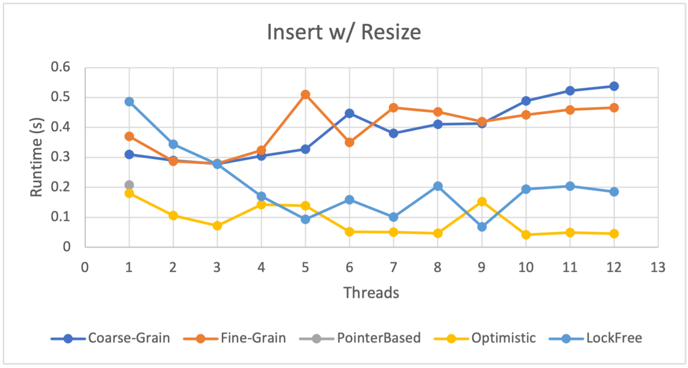
These first set of results involved testing the concurrency of octrees that needed to be resized.

1) Removal:

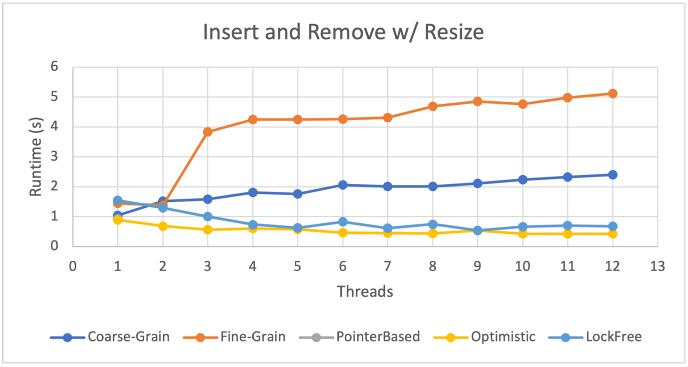
As depicted in Figure 1a, for most of the tests we ran involving removal, optimistic performed better than lock-free, while the coarse-grained and fine-grained algorithms increased runtime when the number of threads increased as well. Compared to a pointer-based sequential approach, we reported a speed-up of about 2.6 times using the optimistic approach with twelve threads.



(a) Removals Only



(b) Insertions Only



(c) Removal and Insertions Mixed

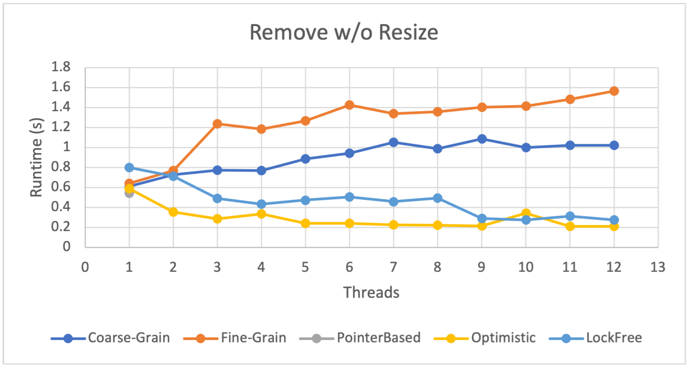
Fig. 1: Removal, insertion, and mixed operations test for the resizable octree version.

2) Insertion:

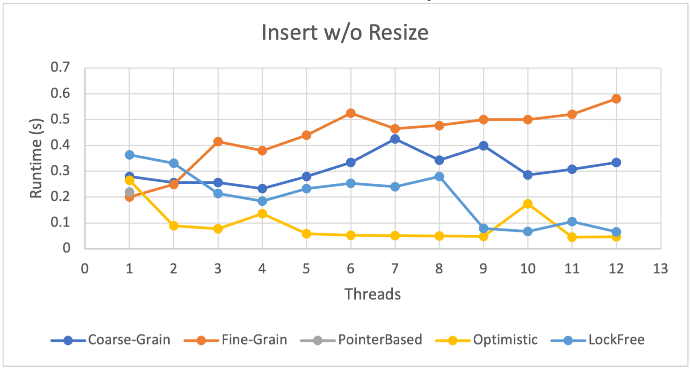
As can be seen in Figure 1b, for insertion, the best performing algorithm varied between lock-free and optimistic depending on the number of threads used. Ultimately, optimistic outperformed lock-free while both saw decreasing runtime with the number of threads increasing. Compared to a pointer-based sequential approach, we reported a speed-up of roughly 4.5 times using the optimistic approach with twelve threads.

3) Mixed Operations:

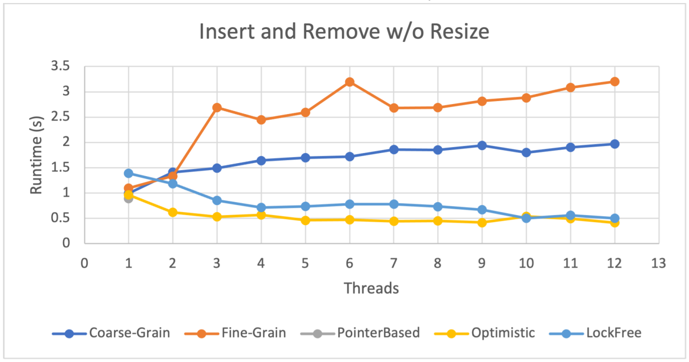
As shown in Figure 1c, both the optimistic and lock-free approaches performed similarly when testing them with mixed operations, but in the end, the optimistic synchronization performed better than lock-free. Once again, we see an increase in runtime for the coarse-grained and fine-grained approaches when the number of threads being used increase. Compared to a pointer-based sequential approach, we reported a speed-up of just over 2.2 times using the optimistic approach with twelve threads.



(a) Removals Only



(b) Insertions Only



(c) Removal and Insertions Mixed

Fig. 2: Removal, insertion, and mixed operations test for the non-resizable octree version.

B. Bounded Octree

The following results involved testing the concurrency of octrees that had precomputed bounds.

1) Removal:

In terms of removal, the optimistic approach consistently provided the greatest speedup, regardless of the number of threads. This can be seen in Figure 2a. Overall, the coarse-grained and fine-grained methods slowed down with an increase in the number of threads, but the optimistic and lock-free techniques did see an overall decrease in runtime. Compared to a pointer-based sequential approach, we reported a speed-up of roughly 2.6 times using the optimistic approach with twelve threads.

2) Insertion:

As shown in Figure 2b, for insertion, the lock-free and optimistic approaches varied in best performance when the number of threads increased, but ultimately, optimistic performed better in a test with twelve threads. Similar to the removal method, the coarse-grained and fine-grained approaches slowed down with an increase in the number of threads whereas optimistic and lock-free sped up. Compared to a pointer-based sequential approach, we reported a speed-up of about 4.75 times using the optimistic approach with twelve threads.

3) Mixed Operations:

When it came to mixing both insertion and removal operations in the octree, our optimistic approach outperformed the lock-free method. This is shown in Figure 2c. However, both optimistic and lock-free performed up to seven times better than the coarse-grained and fine-grained synchronization techniques. Compared to a pointer-based sequential approach, we reported a speed-up of nearly 2.2 times using the optimistic approach with twelve threads.

VI. Conclusion

We expected the lock-free implementation to provide the best results with twelve threads. Overall, however, we saw that optimistic had the best performance in both versions of our implementations and in all of the tests performed. There was no significant difference in runtime when testing both resizable and bounded octrees. We also saw similar speedups using the optimistic approach with twelve threads among both versions.

A. Challenges

In working through the different algorithm implementations, we faced many roadblocks that we did not initially anticipate. First, as we increased the complexity of the synchronization method, moving from coarse-grained to lock-free, the location of the linearization point for functions became much stricter. When we started the lock-free implementation, we realized that, due to the subdivision algorithm, it was necessary to rework the insert, remove, and contains methods. The methods needed to be able to retry if the node they were operating in was being subdivided while they were doing work. The reason why is because our subdivision algorithm may not see the newly inserted point before reinserting and it may also clear the points list before the remove or contains methods are able to operate on it.

B. Future Research

For the future, we would like to see comparisons of pointer-based octrees with linear-based octrees and how each implementation performs with the different synchronization techniques we tested. We were also planning on implementing a lazy synchronization. However, we realized we would have to change our removal method entirely, so that it would be able to remove nodes in the octree as well. In the future, we'd like to be able to change our removal method to allow for lazy removal and see how the runtime compares with the others.

REFERENCES

[1] D. Geier, "Advanced octrees 1: Preliminaries, insertion strategies and maximum tree depth," Nov 2014. [Online]. Available: https://geidav.wordpress.com/2014/07/18/advanced-octrees-1-preliminaries-insertion-strategies-and-max-tree-depth/

[2] V. Chaudhary, K. R. K. Kamath, P. Arunachalam, and J. K. Aggarwal, "Parallel manipulations of octrees and quadtrees," in *ICPIA*, 1992.

[3] B. Hariharan and S. Aluru, "Efficient parallel algorithms and software for compressed octrees with applications to hierarchical methods," *Parallel Computing*, vol. 31, no. 3, pp. 311–331, 2005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016781910500027X

[4] K. Zhou, M. Gong, X. Huang, and B. Guo, "Data-parallel octrees for surface reconstruction," *IEEE transactions on visualization and computer graphics*, vol. 17, 05 2010.

[5] T. Karras, "Maximizing parallelism in the construction of bvhs, octrees, and k-d trees," in *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics*, ser. EGGH-HPG'12. Goslar, DEU: Eurographics Association, 2012, p. 33–37.

[6] D. Geier, "Advanced octrees 2: Node representations," Aug 2014. [Online]. Available: https://geidav.wordpress.com/2014/08/18/advanced-octrees-2-node-representations/

[7] M. Herlihy, N. Shavit, V. Luchangco, and M. Spear, "Chapter 9 - linked lists: The role of locking," in *The Art of Multiprocessor Programming (Second Edition)*, second edition ed., M. Herlihy, N. Shavit, V. Luchangco, and M. Spear, Eds. Boston: Morgan Kaufmann, 2021, pp. 201–228. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780124159501000197