Benchmarking the Concurrent Skip Quadtree

Luo Qian, Alexander Matveev, Nir Shavit
{wqian94, amatveev, shanir}@mit.edu

April 30, 2015

Abstract

We can imagine a neural cell as a single point in 3-dimensional space. Though a single point requires few resources to store and process, 70 million such points can present a far greater challenge for efficient storage and processing. At the time of writing this paper, Harvard University is conducting research in mapping mouse neural cells [2]. Since a mouse brain has over 70 million neurons, studying those neurons, their properties, and their interactions can be both space- and time-consuming.

Fortunately, theoretical computer science has given us the fundamental ideas for applicable data structures, such as the skip quadtree [1]. In this paper, we explore an implementation of this theoretical data structure in serial and in parallel using locks. The parallel implementation represents the beginning of a suite of tools to improve analysis performance on multidimensional datasets in the scientific community. Furthermore, we also present a suite of benchmarking tools to better understand the performances of data structures that belong in this suite.

1. Introduction

Consider the problem of mapping the complete neural network for a mouse. This includes the locations of the neurons, the locations of the synapses, physical orientations, and more. Although modeling and analyzing a single neuron costs little runtime and memory, the same cannot be said of 70 million neurons. Currently, Harvard's Connectome project [2] aims to fully map and analyze a mouse's neural network. As a mouse has over 70 million neurons, this project can incur exorbitant time and space costs.

Fortunately, the Connectome data has the advantage that analyses on these neurons do not affect one another. Therefore, though running each analysis serially can be incredibly time consuming, there exists the possibility of concurrently analyzing the neural data to reduce overall runtime. Unfortunately, this can incur very high space costs, as modeling this data can be very memory-intensive, and would require a replica of the data for each analysis that runs.

Furthermore, as data is collected and discarded, the dataset would need to grow and shrink with the data. As a result, the analyses must be able to see live updates to the dataset as changes occur, thus adding additional constraints the problem.

Clearly, this project needs a data structure to assist in managing the project's data. Such a data structure would need to be capable of providing fast live access to data points, while still accommodating for the time and memory constraints of the project. Furthermore, this data structure must be able to store data points not just in one dimension, but in many dimensions.

There exist many more projects with similar requirements as the Connectome project. These projects often use *multi-dimensional data*, which span several dimensions in coordinate values. Examples of such projects include higher-dimensional feature vector operations in facial recognition software, analyzing distances between discrete words on the bases of meaning and other characteristics, modeling weather patterns as functions of several physical phenomena, and other similar projects with potentially-profound impacts on our lives as technology becomes more and more integrated with human society.

Undeniably, the lack of such a data structure represents a void in the current state of software. In this paper, we address this void by presenting a data structure capable cheaply representing multi-

dimensional data while providing fast operations to access, manipulate, and organize such data. Careful design and clever implementation come together to produce a data structure that delivers upon theoretical claims. Furthermore, we propose the initialization of a library of tools that can be used to analyze similar datasets, and implement a benchmarking suite to aid in comparing runtimes of different data structures that follow the interface set forth in this library.

2. Background

This paper primarily focuses on elaborating the design and implementation details of the *concurrent skip quadtree*, so we will take this time to briefly summarize the theoretical concepts underlying the data structure. The concurrent skip quadtree promises logarithmic runtime, while utilizing only linear amounts of space, all while providing thread-safe concurrent access, modification, and organization. Such a data structure can be seen as a potential solution for the problems faced by the projects in section 1.

2.1. Previous Work

The concept of the concurrent skip quadtree builds upon the theoretical data structure described in the 2005 ACM paper by Eppstein, Goodrich, and Sun [1], the skip quadtree. In the paper, the authors present the skip quadtree as a data structure capable of providing access, management, and organization of 2-dimensional data points while providing $O(\log N)$ runtimes for each operation and consuming only $O(N)$ memory. The paper than continues to prove the theoretical logarithmic runtimes for each operation: insertion, deletion, and querying.

Theoretically, the data structure is a combination of the compressed quadtree and the skip list data structures. As seen in Figure 1, the skip quadtree contains several hierarchical levels of compressed quadtrees, labeled Q_0 at the *lowest* level, and Q_k for the k^{th}-level quadtree. Furthermore, for $k > 0$, we determine the nodes that exist in Q_k by randomly selecting nodes from Q_{k-1}. Each node has a 0.5 probability of being *promoted*, so the expected number of nodes in Q_k is roughly half the number of nodes in Q_{k-1}. Therefore, if there are N nodes in Q_0, the total number of nodes in the data structure is given by

$$N_{total} = \sum_{k=0}^{K} N_k = N + \frac{1}{2}N + \frac{1}{2^2}N + \cdots + \frac{1}{2^K}N.$$

Recognizing this as a geometric series with the initial term being N and the ratio between terms being $\frac{1}{2}$, as K tends to ∞, we see that

$$N_{total} = \frac{N}{1 - \frac{1}{2}} = 2N = O(N) \text{ as } K \to \infty.$$

Therefore, we conclude that the amount of space required to represent this data structure is linear with respect to the number of nodes.

Additionally, as shown by the giant, curved double-arrows in Figure 1, each node contains references to the node in the level above, as well as the node in the level below, assuming that those levels exist. These arrows represent the edges between sparser levels and denser levels of the skip quadtree. Thus, following the example of the skip list, we can expect to traverse the tree to any node in logarithmic time, giving us the runtime of $O(\log N)$ for each operation. As an example, we can follow the bolded gray arrows in the tree diagram in Figure 1 to consider the traversal of the tree to find the node v.

2.2. Implementation

At the time of writing this paper, the authors cannot find a complete, publicly-accessible, and well-generalized implementation of the skip quadtree in serial, though we have found domain-specific partial implementations of the data structure. As a result, we first set off to implement the data structure naïvely in serial to determine the feasibility of the runtime claims in the theoretical paper. Thereafter, we worked on parallelizing the data structure using various methods, and we describe one in the body of this paper.

2.3. Benchmarking

As we implemented the data structure, we also developed a suite of basic tools to analyze and compare the performances of the different variants of the skip quadtree. We soon realized that we can further develop these tools into a benchmarking suite that can accommodate any data structure

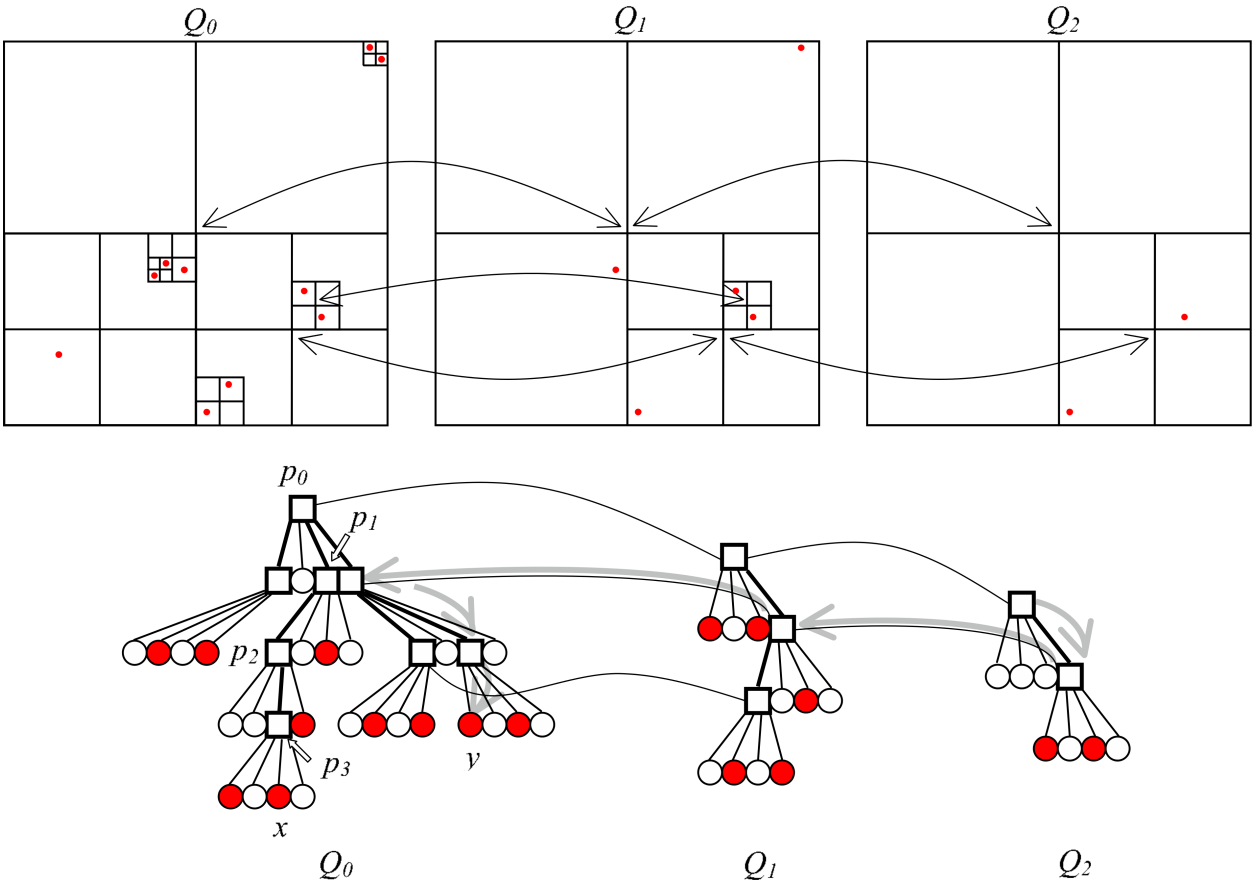

Figure 1: A visual of a skip quadtree (Figure 2 from [1]). The shadowy gray lines show a traversal pattern through the structure.

that operates on multi-dimensional data using only the insertion, deletion, and querying functions. As a result, we developed these tools into a benchmarking suite that works on the data structure interface for insertion/deletion/querying operations.

3. Contributions

In this paper, we describe our initial approach to implementing the skip quadtree in serial. We then follow this up with modifications to the initial approach to produce our final serial skip quadtree. Afterwards, we describe how we modified the serial implementation to produce a parallel implementation using fine-grained locks.

We then discuss our design and implementation of the insertion/deletion/querying interface for multi-dimensional data structure and the benchmarking suite. In particular, we will demonstrate the fairness and repeatability of the results obtained in the benchmarking suite. We then describe the results we obtained when running this suite on our two skip quadtree implementations: serial and fine-grained locks.

The final products of this paper can be summarized into two categories: the data structure library, and the tools for that library, both implemented in the C programming language. The library itself contains the serial and concurrent implementations of the skip quadtree, while the tools contain the data structure interface and benchmarking suite. Additionally, as a product of the tools, we present performance benchmark data on each of the three variants.

4. Serial Implementation

We initially approached the serial implementation as a proof-of-concept to show the skip quadtree's feasibility. Once we established feasibility, we then re-implemented the serial skip quadtree with a redesign to improve time and space performance.

4.1. Proof-of-Concept

We began the proof-of-concept with a simple pseudocode design. This design highlighted the key components of the data structure: what the interface should the tree have, what information the nodes in the tree should contain, and how to ensure that we traverse the tree in the method described in [1].

This proof-of-concept turned out to have a profound impact on our implementation of the skip quadtree. In particular, we learned from this implementation that the data structure requires far more careful planning, especially with regard to the interplay between the insertion and deletion functions. Although we initially considered patching this proof-of-concept code into a production-level data structure, we found that our original design had far too many flaws in its design, and that salvaging the code may take more effort than carefully redesigning and rewriting the implementation.

4.2. Naïve Serial

One of the greatest challenges in our proof-of-concept design correlated to the lack of organization in the implementation details. As an example, the insertion function created new nodes at different points in the code, as was needed. While this made sense at the time of implementation, it made for incredibly messy debugging, since the resulting code had a lot of different components entangled into one code block. Furthermore, this entanglement actually caused bugs in very rare cases.

Additionally, as with any initial implementation, the proof-of-concept code possessed many bugs. During our refactoring, we noticed three major bugs, though we have since found and fixed more.

The first bug related to how we traversed the tree. During traversal, the proof-of-concept code accidentally reset the search pointer to the root of each level as it traverses down through the levels, thereby mitigating any possible gains from the skip list-inspired design of the tree. Furthermore, this incurred up to a factor of two overhead in the worst case!

Our second bug also had to do with traversal. During insertion, the correct traversal requires starting at the highest level and traversing down, inserting the point only on levels that were determined through random coin tosses at the beginning of the function. In our proof-of-concept implementation, however, we neglected to start at the root of the highest level of the tree, and instead started at the root of the highest level *that the point will insert into*. This makes for a dramatic difference, since around half of all nodes inserted will exist only on the lowest-level tree, thereby mitigating most of the benefits of using a skip list-inspired design.

The third bug that we discovered at this point had to do with deletion. As it turned out, the symptoms of the bug proved to be easy to understand: nodes retained pointers to invalid addresses; however, the cause still remains largely a mystery to us, partly as a result of us having already committed to rewriting the data structure when we discovered this bug. Our suspicions, though, mainly involve improperly removing references when we collapse internal nodes, which happens when an internal node has fewer than 2 children.

Due to these observations, before we started implementing the naïve serial implementation, we redesigned each function as a composition of many smaller code blocks, where each code block performed a fixed task. Since this rewrite greatly motived the breakdown of these tasks, we will discuss the breakdown now. This will also provide the reader with some insight into how this data structure performs its tasks.

4.3. Query

We begin with the query function, as this presented the least challenge in the redesign. Since the design of this function proved to be quite simple, we retained much of the code from the proof-of-concept design.

We naïvely implemented the query function recursively using a helper function, with the actual query function acting as a driver function that de-

termined the first step in the recursion. This provided us with the ability to think about each level in the tree as a single recursion depth, thereby simplifying the problem to only worrying about one level at a time.

The query function starts off with a *horizontal traversal*, where we start at the node passed in to the function, and keep traversing through that level of the tree until we find either a leaf node, or an internal node that cannot contain the query point. Once we have reached this part of the traversal, we can now check for some basic conditions.

If we have stopped at a leaf node, we can check to see if that leaf node represents the query point. If not, we attempt to branch down a level through the parent. If the current level has no lower level, we then report that the query point cannot be found in the tree. If we have stopped at a internal node, we directly proceed to branching down a level if possible, and returning false if not.

4.4. Insertion

We now turn to the insertion function, which contains significantly more complexity than the query function, though the basic recursive structures remain similar.

We first consider the driver function. Since we determine how many levels we promote the point through at insertion time, we first make several coin tosses in the driver function and count the number of levels that we will promote the point. During this time, we also create any new levels that need to be created to accommodate the point.

Then, we keep traversing up the levels until we reach the root of the topmost level. During this time, we count how many levels exist between the topmost level and the highest level we will be inserting at. We then pass the root node, the point to insert, and the level gap to the recursive helper function.

The recursive helper function then does a horizontal traversal to find the location to insert the point. Once we find this location, we branch down a level if possible; we use this as our recursive step. At the bottommost level, we skip this recursive step, and simply proceed to insert the node at the appopriate location if no node currently occupies that location. Otherwise, we must construct a new square to encapsulate our new node and this sibling node, before replacing that part of the tree.

Of course, right before insertion, all pointers in the new node, or new nodes, must be assigned, so that once inserted, the tree can still be traversed. This presented an interesting challenge in the proof-of-concept code, which we believe we have solved in this iteration of the code.

Finally, once the insertion has completed, we return the node we have just created. This allows the parent call to easily determine the node corresponding to the point, one level down, and thereby assign the pointer.

4.5. Deletion

Deletion presented a very tricky problem: collapsing internal nodes that have insufficiently many children. This turned out to require additional logic, and thus the deletion process also included a third function to properly collapse and delete nodes.

Otherwise, the overall structure relates easily to the query function. Both have roughly the same driver and recursive functions. The single greatest difference manifests in how the function behaves once we find the target point. In the query function, we simply return `true`, to indicate that we have found our target point. In deletion, however, we must actually delete the node, and to do this, we use the third function.

This third function takes care of updating pointers and references for nodes that we delete, as well as recursing to other nodes in the tree to fully delete all nodes corresponding to a point once we have found that point's representation in the tree. This function also checks the parent to see if the parent node needs collapsing, since every internal node must maintain the invariant of having at least two children.

4.6. Other Considerations

Since we have implemented this data structure in C, we required the use of dynamic memory allocation. This can present a challenge, as without proper deallocation of memory, the data structure can easily contribute to memory blowup due to

memory leaks. While the deletion function certain deallocates memory, there exists the distinct possibility of wanting to fully deallocate the data structure while not all points have been deleted. To do this normally, one would need to keep track of all points and query the tree to delete each point one by one, incurring up to $O(N \log N)$ time simply to free this memory. Since this seems to incur an unnecessary runtime cost, we have also implemented an $O(N)$ deallocation function that will free all the memory in the tree.

5. Parallel Implementation

Once we completed the serial implementation, we began working on parallelizing the skip quadtree. While the serial implementation required two iterations over a long duration of time, we went through several iterations of parallelizing the skip quadtree in a very short time, each time aiming to improve performance and eliminate bugs. Currently, we have focused on only a fine-grained lock implementation, with locks on every node in the tree, but aim to also produce other impementations in the near future.

In this paper, we will describe some design decisions and learnings in the parallelization process of the skip quadtree, as well as some performance results relative to the serial implementation.

5.1. Optimistic Reading

One of our most critical design decisions involved when and where to lock. Since we expect the general use case to be around 90% reads and 10% writes, we decided to apply the optimistic reading paradigm so that reads do not have the overhead of lock contention. This allowed us to have incredibly fast reads, but at a cost to writes, since we now have to be more careful about how we write our data, even after we have acquired the locks.

5.2. Lock Ordering

Another critical design decision had to do with the lock ordering. In order to prevent deadlocks, we defined the following total lock ordering rules:

1. Nodes on a higher level must be locked before nodes on a lower level, and
2. Nodes closer to the roots of their levels must be locked before nodes farther from the roots of their levels.

The first rule effectively prevents deadlock due to level differences, and the second rule prevents deadlock due to parent-child interactions. Together, these two rules enforce a traversal pattern of horizontal-then-down, with no chance to backtrack. This becomes important when we discuss insertion and deletion, as these turned out to present significant design problems for those functions.

5.3. Query

As a direct result of choosing the optimistic reading paradigm, we did not have to make any changes to the query function code. This also means that, under the assumption of 90% reads and 10% writes, our parallel code runs the same code as the serial code 90% of the time, except in parallel, so the parallel code has a performance advantage here.

5.4. Insertion

The insertion function, however, became far more complex as a result of optimistic locking. One major consideration that came into play here has to do with invariants in the tree. Specifically, if a point exists on level Q_k, it must also exist on all levels $Q_{k'}$ for $k' < k$. This means that, in the implementation, nodes must be inserted at the bottommost level first, then backtracking up to the higher levels.

At the same time, we must maintain an total lock ordering to prevent deadlock. As we explained in section 5.2, however, this total lock ordering prevents backtracking up the tree. Thus, we have arrived at a situation where, in order to insert, we must first traverse to the bottom of the tree and insert while backtracking, but on the other hand, we can't take locks during the backtrack.

The only solution, then, must be to obtain the locks on the way down to the bottommost level, and then do the backtracking to insert the actual nodes. As the astute reader may note, however, this increases lock contention and may result in a huge performance hit. Even so, we decided to start with this implementation, and if such performance hit does occur, we will analyze the locks to decide

how to elide locks during this process.

Unfortunately, we also encountered other problems along the way. The original recursive implementation may have been fine for the serial implementation, but it proved to be far too costly to keep setting up stack frames for recursive calls, so we had to design a non-recursive implementation instead.

Fortunately, due to the nature of the problem, we can consider the process of taking locks to also serve as the process for identifying the locations where we will insert nodes. Therefore, if we simply kept track of which nodes we lock on, we can determine where we will be inserting, and already have the locks held for us when we make the modifications.

Of course, to use this fact to our advantage, we need a way to keep track of what nodes we have taken locks on. This manifested itself in the code as the LockSet struct. The LockSet struct simply keeps track of which nodes we have taken locks on, and stores pointers to those nodes so that we can refer to them later. The problem now becomes a three-step process:

1. Acquire all locks that we might need,
2. Validate the locked nodes for invariants, and
3. Make the modifications in the data structure.

We can see that step 1 represents the initial traversal down the street, but it may not be immediately clear why step 2 must also take place. Finally, step 3 makes the modifications once we have acquired the necessary locks by iterating through the locked nodes, and then unlocks all the nodes we took out locks on.

To clarify the rationale behind step 2, consider the following scenario. As we acquire locks down the tree, another concurrent operation ahead of us makes some modifications to the tree. For simplicity, let us assume that this modification added a new square around the sibling node that we must take a lock on. Now, once we have acquired that lock, the other operation, clearly an insertion, must have completed, so its effects can now be considered visible to all.

Unfortunately, the fact that there's now a square that bounds the previously-calculated sibling node can be troublesome. In fact, we can imagine two possible cases, and we will need to do something different for each case.

In the first case, the bounding square does not contain the point we wish to insert, so we would like to acquire a lock on that bounding square and make that our new sibling. This is problematic, though, because our total lock ordering says that we cannot lock parents after children, so we would need to release the lock on our current predicted sibling in order to acquire the lock on the new bounding square. This bounding square may, in turn, also end up having a new parent between the time that we see this change and the time that we acquire the lock. So, this can proceed for a very long time, which means all the locks we have been holding has been preventing other threads from getting work done, thus lowering our throughput. This presents a problem.

In the second case, the bounding square does contain the point we wish to insert, so we would like to make this bounding box our new parent. Thus, we will need to acquire the lock on this square and release the lock on the parent. Again, we run into the problem that we cannot lock on this new bounding square after we have already locked on the sibling, which represents a child of the bounding square. Thus, we would need to unlock the sibling, lock the bounding square, unlock the parent, and re-lock the sibling, during which time the bounding square could have been further divided, at which point we may need to release and acquire new locks. This also presents a throughput problem.

We believe that we have found a rather clever solution to this problem, by means of step 2. Once we have acquired the locks, we validate those locked nodes to ensure that our insertion locations have not changed, so that we still have all the locks that we need to insert our point. If validation fails, this means that something changed, and we only need to unlock all the locks we have acquired, and try again by starting at the top and acquiring new locks. While this may seem to be a very time-intensive method, it does prevent throughput disasters due to lock contention on a lock that one thread holds while continually trying to find the next lock.

In the end, we ended up almost entirely rewriting the insertion function in the conversion from

serial to parallel.

5.5. Deletion

The story behind deletion mirrors much of the story behind insertion. One additional key thing that changed in the transition from serial to parallel code has to do with deallocating memory. While previous, we were able to allocate memory once the node has been removed, we now have to worry about the possibility that another thread will look at a removed node because the updates have not been flushed to that processor yet. To prevent such scenarios, we have currently disabled the memory deallocation part of the deletion process. Of course, this incurs a large memory leak overhead, but we aim to fix that in later work.

Since many of the constraints that we found in the insertion function also carried over to the deletion function, we ended up rewriting the deletion function as well. By using the same LockSet structure that the insertion function used, we came up with a deletion function that also followed the lock-validate-modify paradigm. Again, the motivations for switching to this paradigm strongly mirror those for switching the insertion function to this paradigm, so we will not repeat the same arguments here.

We will clarify, however, the process for actually deleting a node. Since we cannot do physical deletes, we opted for logical deletes. Each node has a dirty bit, so upon deletion, we simply toggle this dirty bit. Once toggled, all functions will treat that node the same as they would a `NULL` node – as if that node did not exist. In this way, we can artificially "delete" all pointers to a node with only one instruction.

6. Benchmarking Suite

We will also briefly describe the benchmarking suite that we developed as part of this project. Part of our goal in developing this suite revolves around the idea that this marks the beginning of an entire library of tools, so we need some method to reliably test their performances. To do this, we first developed a simple interface, then we used deterministic seeding to provide consistent, reproducible, pseudorandom data sets. While the suite still has a few modifications left, in its current state, it can reliably benchmark these any data structure that fits the interface and uses the 2-D Point interface that we have developed as well.

6.1. Interface

Our first task began with designing the interface for this library of tools. While initially we intended on working with just trees containing point-like data, we eventually expanded this to any data structure that support insertion, deletion, and querying operations on point-like data. Since this library consists of data structures written in C, we also provided the constructor and destructor functions as part of the interface as well.

6.2. Benchmarking Tool

The benchmarking tool itself consists of a single C program that acts as the driver for a variety of insertion, deletion, and querying operations on the data structure. During compile time, compiler flags encode various parameters for the benchmarking tool, such as the header file to include, the type of data structure, and the names of the constructor, destructor, insertion, deletion, and query functions. Additional configuration factors include the read-to-write ratio, the insertion-to-deletion ratio, and how many total operations to prepare. The programmer can then compile this benchmarking program into an object file, and link everything together with the data structure to produce the final executable.

6.3. Benchmarking Report

The benchmarking tool reports the total time required to execeute all operations, measured in seconds with a granularity of microseconds. In addition, the tool also outputs the total number of operations, as well as a breakdown for how many insertions, deletions, and queries the benchmark included. By default, the tool outputs this information in the Comma-Separated Values format, but can be configured to output human-readable statistics using a verbosity flag at the tool's compile time.

Figure 2: Plots of runtimes for different variations of the data structure, given varying numbers of operations, with 90% reads and 10% writes. The top pair indicate times when compiled with the gcc -O0 flag, while the bottom pair indicates times compiled with the gcc -O3 flag. The left two plots contain data regarding all variants for all operation counts, while the right half plots only selected data to emphasize the runtime differences.

7. Results

We wished to compared the performance of the serial and parallel variants of our data structure. To do this, we ran the benchmark suite for varying operation counts for each variant of the data structure, on a machine belonging to the MIT Computer Science and Artificial Intelligence Laboratory, nicknamed the "BEAST," which has four Intel(R) Xeon(R) CPU E7-4870 cores, each running at 2.4 GHz, with 20 hyperthreads per core. One major disappointment for us involves the fact that the parallel execution times tend to trail the serial executions, with the best only performing on-par with the serial executions, as we can see in Figure 2. We have several theories about this, which we detail below.

From these plots, we can see that the parallel executions tend to perform sub-par compared to the serial executions, with the best parallel execution, the 8-threaded parallel execution, performing

approximately on-par with the serial executions. We believe that the performance at 8 threads may be due to the effects of hyperthreading, and running more than two hyperthreads on a core may incur performance hits, while running fewer than two hyperthreads does not fully saturate all the cores' computational capacities.

8. Analysis

We believe that the parallel code incurs a high performance hit from its use of the `malloc()` command. The `malloc()` command has thread-safety built in, so calls to it may result in high contention. We intend on testing this hypothesis by using Google's TCMalloc, which should perform better in parallel environments.

Another cause of the performance hit may be system calls. In the driver insertion function, we make use of system calls to determine a random seed for that function, which we use for our own concurrency-supporting pseudorandom number generator. This may, however, incur a slight performance cost, since system calls tend to have high overhead.

Finally, we will need to consider how the compiler may be optimizing our code. Based on the charts in Figure 2, we can see that the O3-level optimization produces a huge performance improvement for the 1-thread parallel execution, while the 8-thread parallel execution does not seem to have benefited all that much.

9. Future Work

We intend on exploring other methods of parallelization, such as transactional memory, and comparing those results with our existing results. We would also like to optimize our code some more, to see how much faster we can make this code. Additionally, we will continue to work on the benchmark suite, and intend on publishing it as part of the library, along with the interface and the data structure. Finally, we also look forward to working on other data structures that follow this interface, and seeing what kinds of parallel performance results we can achieve in those areas.

10. Acknowledgements

I would personally like to thank Alex Matveev, my mentor for this project, for his patience with me as I stumbled around in implementing this code, as well as meeting with me to discuss solutions to problems that I encountered. Alex has been pivotal in explaining to me why I have noticed some of the strange phenomena surrounding the implementation. I would also like to thank Nir Shavit for piquing my interest and supporting my endeavors in multicore algorithms. In addition, I thank Denny Freeman, Anantha Chandrakasan, and all the MIT SuperUROP staff for providing me with the opportunities and tools to have arrived at this moment. I also thank Actifio for their support of my project through the SuperUROP program.

Finally, I would like to thank all my friends, colleagues, mentors, and professors for their unyielding support of my work and their unending kindness and sympathy when I encountered pitfalls in this project.

References

[1] David Eppstein, Michael T. Goodrich, and Jonathan Z. Sun. The skip quadtree: A simple dynamic data structure for multidimensional data. In *Proceedings of the Twenty-first Annual Symposium on Computational Geometry*, SCG '05, pages 296–305, New York, NY, USA, 2005. ACM.

[2] Harvard Center for Brain Science. Connectome project. In *Harvard Center for Brain Science*, 2010-2014.