15-418 Project Report

Alex Patel and William Liu
Andrew IDs: acpatel and wxl

May 6, 2019

Parallel Galaxy Simulation with the Barnes-Hut Algorithm

Summary

We implemented multiple optimized parallel implementations of a galaxy evolution simulator for use on multi-core CPU platforms using the OpenMP framework. Given the success of our implementations, we have demonstrated that galaxy simulation is highly-parallelizable on the CPU, even when computed using more involved methods such as the Barnes-Hut Algorithm.

Background

A Simple Galaxy Simulator

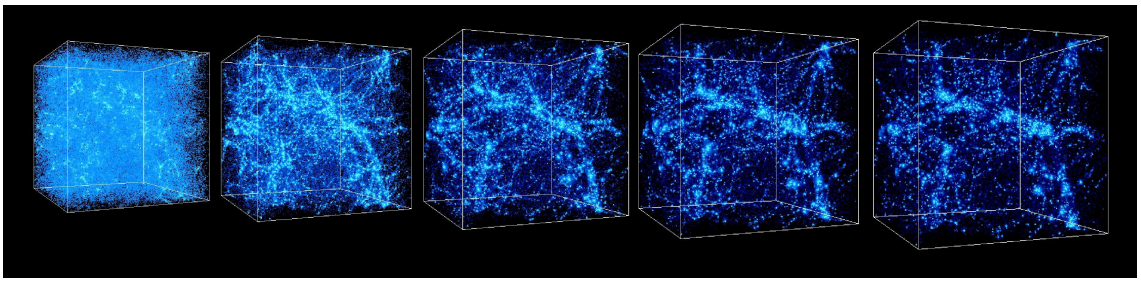


Figure 1: Galaxy evolution from uniform body distribution.

Galaxies are complex natural phenomena with such intricate continuous behavior that it is seemingly impossible to simulate them on a discrete platform, such as the modern day computer. However, if one could be made, an accurate galaxy simulator would provide tremendous insight into many questions humanity has pondered over the nature of the universe, and our place in it:

- What does the arrangement of a galaxy's bodies tell us about the history of the galaxy?
- Can we form a model of the life cycle of galaxies based on how the simulation evolves? Are there multiple common patterns of galaxy life cycles?
- How frequent are collisions of galaxy bodies or clusters?

- How significantly do initial conditions, such as the initial space, mass, and size distribution of bodies, affect the future evolution of the galaxy. In a similar vein, what initial conditions lead to the birth, death, and split of galaxies?
- Given the observed arrangement of bodies in the Milky Way, can we make estimations with high probability of our galaxy's future? What is the probability Earth will collide with another body in our galaxy? How would a likely future galaxy event impact life on Earth?

A suitable starting point for this inquiry is the most easily observable force on galaxy bodies from afar: **gravity**. The Moon around the Earth, the Earth around the Sun, and clusters of stars around the Milky Way's center. We have observed this force for centuries, and the force on any body seems to independently follow the following formulation:

$$\mathbf{F}_i = \sum_{\substack{1 \le j \le N \\ j \neq i}} \mathbf{f}_{ij} = G m_i \cdot \sum_{\substack{1 \le j \le N \\ j \neq i}} \frac{m_j \mathbf{r}_{ij}}{\|\mathbf{r}_{ij}\|^3}.$$

Figure 2: Gravitational force on a single body considering N total bodies.

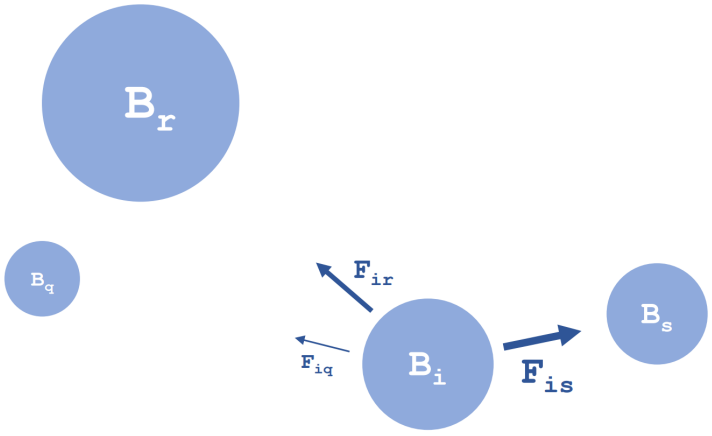


Figure 3: Gravitational force on B_i considering surrounding bodies.

Where **G** is the Universal Gravitational Constant $(6.67408(31) \times 10^{11} \quad m^3 \cdot kg^{-1} \cdot s^{-2})$. In the above formulation we see that the force on a body from another body is inversely proportional to the square of the distance between the bodies. We also see that the force is directly proportional to the product of the masses of the bodies. We can simplify the expression above by deriving an expression for the acceleration on any given body. This can be done by simply using $F = ma$, and dividing out the mass of the body the force is acting on. Finally, we notice that as the distance between two bodies becomes arbitrarily small, the acceleration approaches infinity. To resolve this issue, we introduce a small softening factor ε to set the acceleration between bodies to zero in the event of an arbitrarily small distance. We arrive at the following formulation to independently compute the acceleration of each body given a configuration of the N bodies:

$$\mathbf{a}_i \approx G \cdot \sum_{1 \leq j \leq N} \frac{m_j \mathbf{r}_{ij}}{\left(\left\| \mathbf{r}_{ij} \right\|^2 + \varepsilon^2 \right)^{3/2}}.$$

Figure 4: Acceleration of a single body considering N total bodies.

Our goal in this project is to demonstrate that the simulation of galaxy evolution is highly parallelizable on CPU platforms. Here we have reduced the immense task of constructing an accurate galaxy simulator to one that approximates the effect gravity has on the evolution of a galaxy's bodies. This is a sub-problem that must be considered during the construction of any accurate galaxy simulator, and in particular we will demonstrate that this sub-problem is highly parallelizable on CPU platforms, even with more involved sequential methods of approximation.

We now note that the computation of each body's acceleration is independent from the computation for each other body. In this way, a naive approach of computing every body's acceleration by considering all pairs of bodies is embarrassingly parallelizable, since we can evenly balance load by partitioning the bodies into equal buckets. However, this naive algorithm is sub-optimal for larger scale simulations, since the computational cost grows with $O(n^2)$, where n is the total number of bodies we are considering, which becomes ridiculously expensive for large n.

Next, we consider that this problem is a classic example of an N-body problem, in which we have a configuration of bodies and their positions in space, and we aim to update the position of each body by considering the positions of each other body. This problem domain has historically been the subject of extensive inquiry, and as such there is a wide array of potential algorithms to choose from. A notable sequential algorithm is the Barnes-Hut Algorithm, in which we build a spatial tree to form a hierarchical clustering of bodies, so that during the acceleration computation phase, each body can treat far away clusters as a single larger body to reduce total computation. This results in an average $O(n \log n)$ algorithm instead of the all-pairs naive $O(n^2)$ algorithm, where n is the number of bodies we are considering.

The Barnes-Hut Algorithm

The Barnes-Hut Algorithm operates by first constructing a spatial tree to hierarchically distribute bodies between tree nodes based on closeness in space. A common type of tree used in this scenario is the quadtree in 2 dimensions, due to the relative simplicity of its construction. Quadtrees subdivide the 2D square they represent into 4 equally sized squares corresponding to 4 subtrees. When considering bodies in space, the quadtree continues to subdivide until each leaf node only contains a single body. A fully constructed quadtree given a configuration of bodies can be seen below in Figure 5:

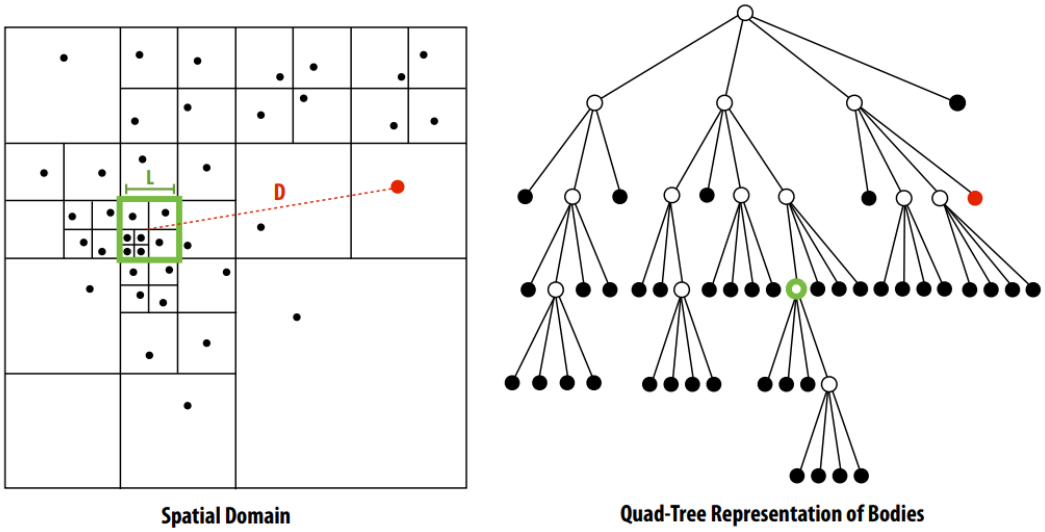


Figure 5: Quadtree constructed from spatial configuration of bodies.

After the quadtree is constructed, we aggregate forces for each body. To do this, we first consider the root, then recurse into each of the 4 subtrees until one of the following conditions are met:

1. If the node we are looking at is a leaf, then add the force contribution from the body at the leaf if it exists.
2. If the side length of the node's region divided by the distance from the body to the center of mass of the node is less than some defined θ, treat the node as a single mass and add its force contribution.

The second condition is a little more complicated, but it is what makes the Barnes-Hut Algorithm more efficient than a naive implementation. Looking at Figure 5 above, we test if

$$\frac{L}{D} < \theta$$

to decide when we should approximate a cluster of bodies as a single body. In this way, by choosing a larger θ, we can improve performance at the cost of increased approximation. We note that the

expected number of nodes touched during force aggregation for a single body is

$$\approx \frac{\log(N)}{\theta^2},$$

resulting in an $O(n \log n)$ algorithm as long as $\theta > 0$.

The above description is how we can compute forces and thus accelerations for every body given the current configuration of the bodies in space. However, to perform a simulation, we need to evolve the galaxy over time. To do this, we will iterate over a number of simulation steps, and at each step we will compute the acceleration for each body, then integrate over a short timestep to get the new position for each body.

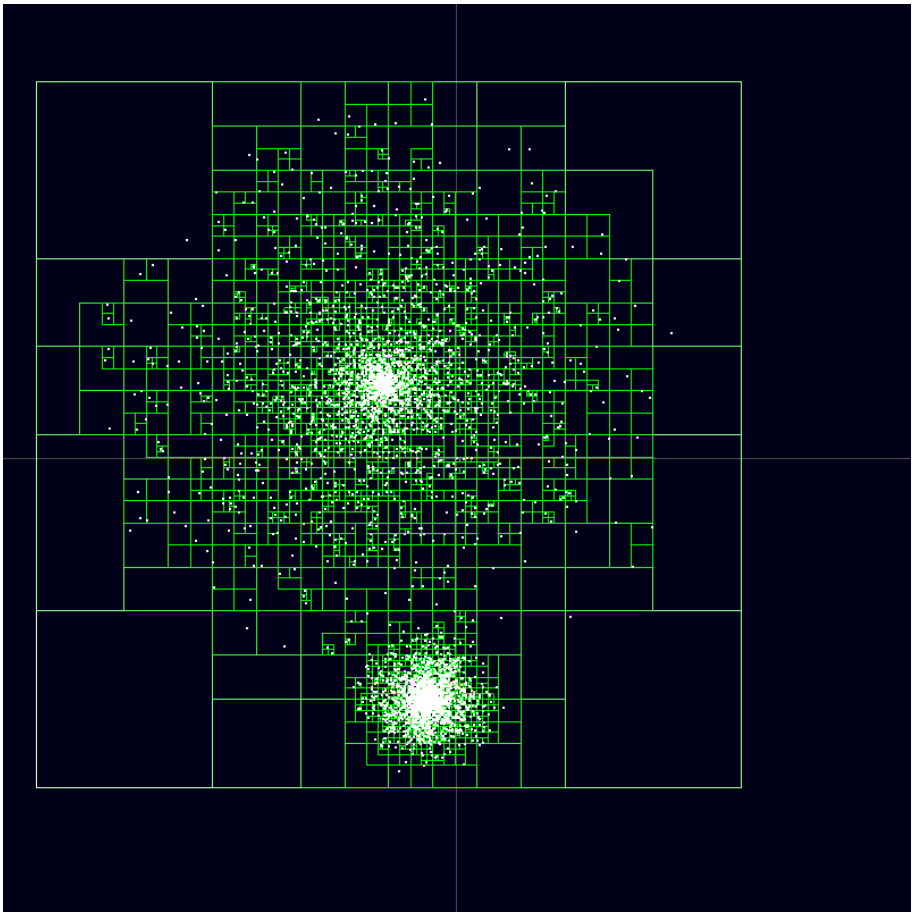


Figure 6: Example of quadtree hierarchical clustering.

Incorrect Correctness

As noted previously, the Barnes-Hut Algorithm is an approximation of the discrete N-body problem. But the N-body problem itself is an approximation of the evolution of a galaxy. Since we are unable to setup a galaxy and watch it evolve to compare with our simulation, we must work under the context of the approximations we are making. In this case, since we are not using the simulation for detailed scientific analysis, we will judge correctness based on how reasonable body movements appear in a visualizer.

Now there is a further point of consideration for our algorithm in terms of correctness. How can we integrate acceleration and velocity to compute updated positions for each body? We could simply multiply acceleration by the timestep to compute the change in velocity, and multiply the new velocity by the timestep to compute the change in position. This method is known as Forward Euler, or the Explicit Euler method. However, one consideration we need to make when designing our numerical integrator is the change in energy of the system an integration will incur. This is important, because, for example, if the energy of the system is constantly increasing, we will notice on the visualizer that bodies are getting farther and farther apart from one another as the simulation continues. This is not reasonable behavior, and as such does not satisfy our definition of correctness above.

The Forward Euler method described in the paragraph above will change the energy of the system quite quickly given any reasonable timestep (one that is not extremely close to zero). An alternative numerical integration technique which is popular in this domain is Verlet Integration. The technique lowers change in energy by using velocity a half timestep in the future to integrate position, instead of velocity an entire timestep in the future. The practical implementation of the technique for our application is as follows:

1. $\vec{v}(t + \frac{1}{2}\Delta t) = \vec{v}(t) + \frac{1}{2}\vec{a}(t)\Delta t$
2. $\vec{x}(t + \Delta t) = \vec{x}(t) + \vec{v}(t + \frac{1}{2}\Delta t)\Delta t$
3. Compute $\vec{a}(t + \Delta t)$ from $\vec{x}(t + \Delta t)$ using the Barnes-Hut Algorithm
4. $\vec{v}(t + \Delta t) = \vec{v}(t + \frac{1}{2}\Delta t) + \frac{1}{2}\vec{a}(t + \Delta t)\Delta t$

Below in Figure 7 we can see both numerical techniques used to estimate a simple harmonic function:

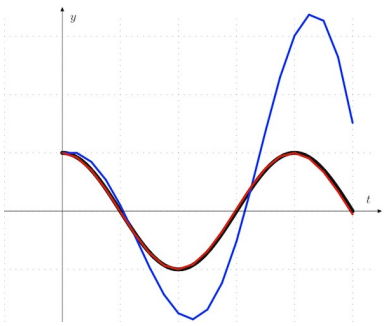


Figure 7: Black is actual, Red is Verlet, Blue is Euler

Overview

We have explored how we can construct an optimized sequential implementation that satisfies our conditions of correctness for a gravity-based galaxy simulation approximation. Our application structure is presented in the pseudo-code below:

Algorithm 1 Sequential Galaxy Simulation with the Barnes-Hut Algorithm

```
1:  procedure SIMULATE
2:      for step in simulation steps do
3:          for body in galaxy bodies do
4:              update body position and half timestep velocity
5:          initialize quadtree
6:          for body in galaxy bodies do
7:              insert body into quadtree
8:          traverse quadtree to assign masses and center of masses to nodes
9:          for body in galaxy bodies do
10:             update body acceleration by accumulating accelerations from the quadtree
11:         for body in galaxy bodies do
12:             update body velocity
13:         write simulation state to file for visualizer
```

The above pseudo-code has many independent for loops around the bodies of the simulated galaxy. This may seem to result in an embarassingly parallelizable application, but we notice two challenges:

1. Inserting bodies in parallel to the quadtree requires the data structure to handle concurrent operations.
2. Different bodies require a different amount of work to accumulate accelerations from the quadtree. This would result in an imbalanced work load if we were to arbitrarily assign bodies during the acceleration update phase.

Next we discuss our approach to tackling these challenges as well as our different implementations of the formulation.

Approach

Tools and Target Architecture

Our application targets multi-core CPU platforms, and specifically machines with homogeneous compute resources such as the GHC machines we have used for other assignments in this course. Processors on the GHC machines have 8 cores supporting simultaneous multithreading (Intel hyperthreading), allowing for efficient use of a maximum of 16 threads in our application. We have performed in-depth benchmarking and cross-implementation analysis on the GHC machines, and have performed a higher-level cross-system analysis between the GHC machines and the Latedays clusters. All code was written from scratch in the `C` programming language, using the OpenMP parallel framework. OpenMP is preferable in our case to a lower-level API since it allows us to simply identify parallel blocks of code for the compiler and machine to map to compute resources and execute. This both eases development time as well as allowing our implementations to be portable between machines with varying system characteristics.

One piece of starter code we did use is the `cycletimer.c` from the Graph Rats starter code https://github.com/cmu15418/asst3-s19. This allows us to perform fine-grained timings of the various sub-routines of our implementations. We implemented a module `monitor.{h/c}` to provide macros to keep track of the timings of each sub-routine in the algorithm.

```
-bash-4.2$ ./gsim-lock-free 16 100 100000 10 0.5 123
Running with 16 threads. Max possible is 16.

        total time: 1330.69 ms
        ------------------------------------------------
        14.21 ms        1.07 %          setup
        0.00 ms         0.00 %          write_file
        4.55 ms         0.34 %          update_positions
        119.30 ms       8.97 %          build_quadtree
        184.50 ms       13.87 %         traverse_quadtree
        16.50 ms        1.24 %          partition_quadtree
        781.33 ms       58.72 %         compute_forces
        2.29 ms         0.17 %          update_velocities
        207.49 ms       15.59 %         free_quadtree
        0.51 ms         0.04 %          other
```

Figure 8: Example output of our monitoring module

Another note on our implementation is that we used the `gcc` compiler without any flags (except `-Wall` to catch warnings). This means that we excluded all optimization flags, i.e. `-O`, `-O2`, `-O3`, `-Ofast`, etc. We did this to avoid strange floating point behavior, since running our program with differing thread counts must preserve the exact output. In addition to this, removing optimization allows our speedup to be more observable, and preserves the relative timings of various subroutines in our implementations with varying thread counts. The lack of compiler optimization is for clarity's sake; compiling our code with `-Ofast` preserves our standard of correctness introduced in the background section above (visualizer results are reasonable).

Visualization

Before implementing any version of our application, we wrote a visualization program `gviz` to be able to verify the correctness of our implementations. This program was written in `C++` compiled with the `g++` compiler using the flags `-m64 -std=c++11`, and uses the OpenGL graphics framework with the library `glfw3` to quickly render bodies as they evolve through simulation steps. `gviz` reads an output file from our application implementations to render the visualization.

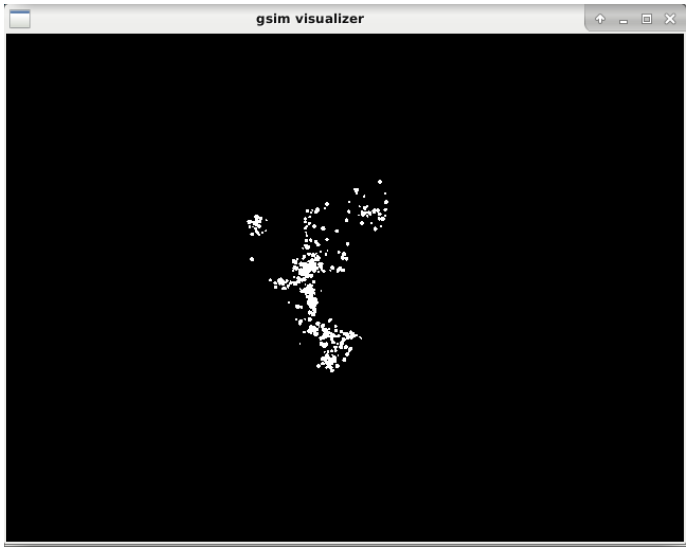


Figure 9: Example visualized galaxy configuration (snapshot of visualizer in motion).

Survey of Implementations

In total, we implemented 3 parallel implementations of the galaxy simulator:

1. Parallel naive all-pairs $O(n^2)$ algorithm.
2. Parallel Barnes-Hut Algorithm with a fine-grained locking quadtree.
3. Parallel Barnes-Hut Algorithm with a lock-free quadtree.

We measure the performance of each implementation on 3 benchmarks:

A 1-to-1: Equal number of clusters and bodies.

B sqrt: The number of clusters is the square root of the number of bodies.

C single: There is a single cluster of all the bodies.

For each benchmark we vary θ between the values 0.1, 0.3, and 0.5, and the number of threads between all values in the range [1, 16].

gsim-bad

Our first implementation is named `gsim-bad`, since it is the naive all-pairs $O(n^2)$ algorithm introduced above in the background section. Here our goals were to set up the infrastructure from which we could develop the more sophisticated implementations. We decided to not use pre-developed starter code in the aim of having as much control over the details of the implementation for ease of parallelization.

```
Usage: ./gsim-bad <thread count> <number of clusters> <number of bodies>
                  <number of simulation steps> <random seed for initialization>
                  [output file]
```

Figure 10: Usage of gsim-bad.

First, we introduce constants and optimizations used in this implementation that are shared between all 3 of our implementations. It is essential that we define simulation constants in such a way that our visualizer is interpretable. Our main header file `gsim.h` contains definitions of the mass, distance, and other constants used throughout our implementations:

```
/* general constants for gsim */
#define GRAV_CONSTANT      6.67408e-11  // m^3 * kg^-1 * s^-2
#define DIST_SCALE         20.0e15      // m
#define INIT_MASS          4.0e30       // kg
#define MASS_RANGE         2.0e30       // kg
#define SOFTENING_FACTOR   1.0          // arbitrary small value
#define TIME_STEP          1.0e2        // s
#define MAX_ACCELERATION   7.0e6        // m * s^-2
```

Figure 11: Simulation constants used in all of our implementations.

Some things to note:

- `DIST_SCALE` is the side length of the square of space we are simulating.
- `MASS_RANGE` is a range of possible body masses; each body is uniformly assigned a mass centered at `INIT_MASS`.
- `SOFTENING_FACTOR` is the softening factor ε described in the background section.
- `TIME_STEP` is the Δt between simulation steps, and was chosen so that the visualizer would evolve nicely in real time.
- `MAX_ACCELERATION` limits the maximum magnitude of a body's acceleration so we do not get the "sling shot" effect as bodies get arbitrarily close to one another. This is required because we do not consider body collisions.

In the `gsim-bad` implementation we introduce the idea of body clustering, and its possible effects on parallel performance. To initially distribute bodies in space, we first iterate over the number of input clusters. These clusters are uniformally distributed in space, and each of the cluster's arbitrarily assigned bodies are positioned near the center of the cluster. In our Barnes-Hut implementations, clustering affects quadtree performance.

A notable optimization we made is to ensure that the memory pertaining to each body is on an independent cache line during execution. By doing this we ensure that we do not get significant false sharing between threads, which may significantly bottleneck memory performance given a snoop-based cache coherence system or similar. If two bodies had memory on the same cache line, and two threads tried to access and modify that data at the same time, one of the threads would have their cache line invalidated by the other writing thread. Thus, we would get more bus traffic, as well as increased cache misses, which drastically reduces memory performance. Our body structure `body_t` is 64 bytes with padding, perfectly enough to fit on a single x86 cache line.

Next we discuss our parallel approach to the sequential pseudo-code illustrated in the background section above. For the naive algorithm, it is not necessary to even build the quadtree; we simply need to consider each other body as we accumulate accelerations for a body. Therefore, it requires the same work to update the acceleration of any body as any other body. From this we see that it is very simple to load balance between threads effectively: we statically assign an equal number of bodies to each thread. Updating positions, accelerations, and velocities of bodies can then be done with `#pragma omp parallel for schedule(static)`, and all threads have roughly equal work.

Below are the benchmark results for the parallel naive implementation, measured in NPM (Nanoseconds Per Move), where each move is considered to be updating the position of a single body.

We notice that there is a jump in NPM as we increase the thread count to 9 from 8. This is due to the fact that processors on the GHC machines only have 8 cores, so by using more than 8 threads we start taking advantage of Intel's hyperthreading. Hyperthreading is a type of simultaneous multithreading, which allows the processor core to pipeline more independent instructions at once. However, each thread is still limited by the functional units of the core they are running on. Since our application requires a significant amount of floating point operations, it is unlikely that hyperthreading could improve performance dramatically as we are throughput bound. This is why we see insignificant speedup past 8 threads.

In addition to this, all three of the benchmark graphs below seem identical. This makes sense, since by using the naive all-pairs algorithm, clustering does not affect the amount of computation we must do, since each body must consider each other body.

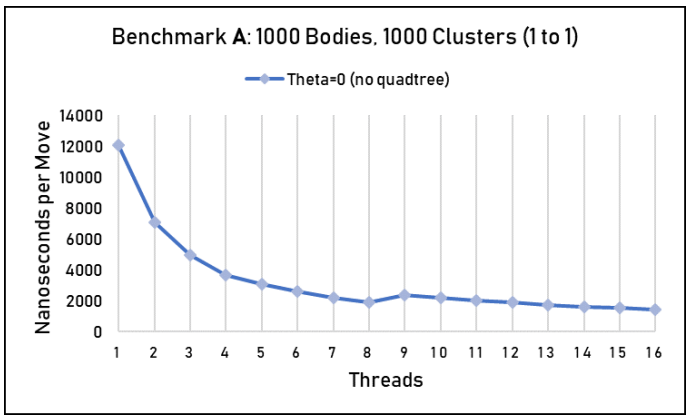


Figure 12: Benchmark A on Naive Implementation.

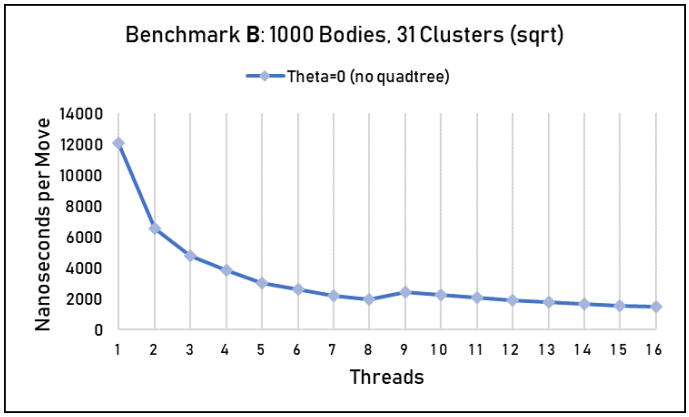


Figure 13: Benchmark B on Naive Implementation.

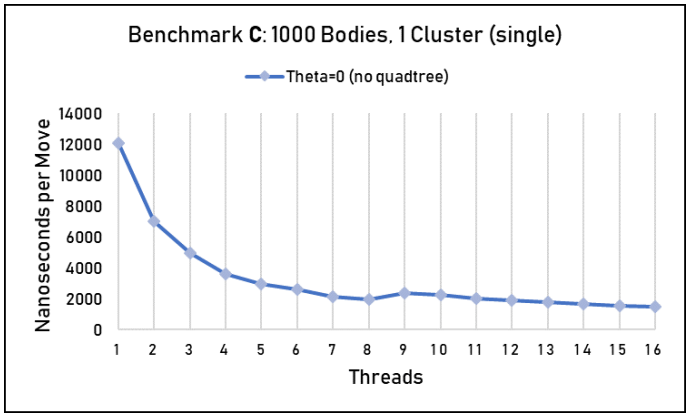
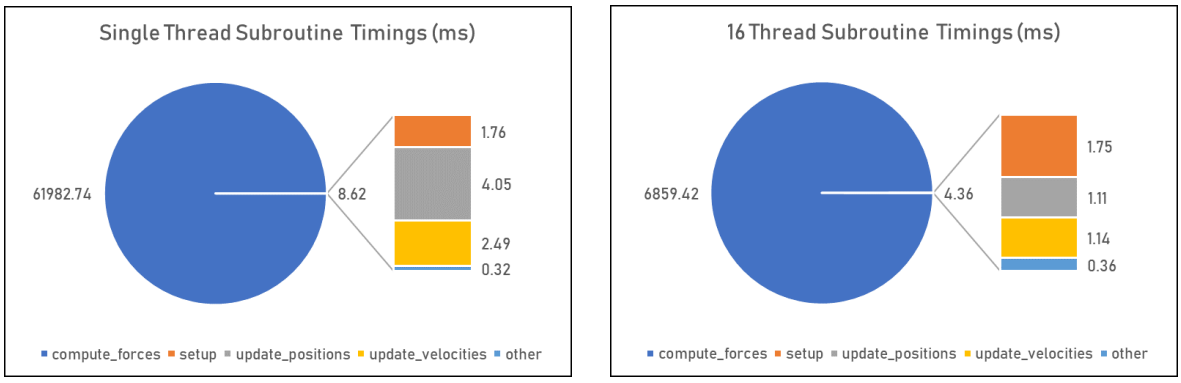


Figure 14: Benchmark C on Naive Implementation.

Optimistically, we expected to see linear speedup here, because the work each thread is doing is completely independent in terms of both memory and local data. However, we need to consider that all threads are running on the same machine, which opens the door to memory bandwidth issues, cache coherency limitations, and other system-specific bottlenecks. The speedup we see is evident that our implementation is efficiently balancing load between the threads, even if speedup is not directly linear. In addition to this, each simulation step comes with some amount of overhead, such as spawning and joining threads, which costs a constant amount per step and reduces speedup.

Naive Implementation Speedup	
Threads	Speedup
1	1.0×
2	1.71×
4	3.27×
8	6.26×

Now we consider the relative cost of each subroutine of the naive implementation. As evident in the figure below, almost all of the work is dedicated to updating forces for bodies. This makes sense because other costs are linear in the number of bodies, while updating forces is quadratic in the number of bodies. Below are the results from running the naive implementation on 10000 bodies for 50 steps, with 1 and 16 threads:



(a) Single Thread

(b) 16 Threads

Figure 15: Naive implementation subroutine timings with 10000 bodies and 50 steps

`gsim-barneshut`

Our next implementation is a parallel implementation of the Barnes-Hut Algorithm formulation introduced in the background section above. We adapt the sequential algorithm as follows:

Algorithm 2 Parallel Galaxy Simulation with the Barnes-Hut Algorithm

```
1: procedure SIMULATE
2:     for step in simulation steps do
3:         #pragma omp parallel for schedule(static)
4:         for body in galaxy bodies do
5:             update body position and half timestep velocity
6:         initialize quadtree
7:         #pragma omp parallel for schedule(static)
8:         for body in galaxy bodies do
9:             insert body into quadtree
10:        traverse quadtree to assign costs, masses, and center of masses to nodes
11:        partition bodies between threads using the cost-zones method
12:        #pragma omp parallel for schedule(static)
13:        for thread in thread count do
14:            for body in thread's bodies do
15:                update body acceleration by accumulating accelerations from the quadtree
16:        #pragma omp parallel for schedule(static)
17:        for body in galaxy bodies do
18:            update body velocity
19:        write simulation state to file for visualizer
```

There are two key changes when adapting the algorithm from sequential to parallel:

1. Outer **for** loops during the simulation step are identified with OpenMP pragmas to run in parallel with static scheduling.
2. Instead of arbitrarily assigning bodies to threads, we partition bodies between threads based on their expected work while traversing the quadtree. This is known as the cost-zone partitioning scheme.

Before considering our method of partitioning to promote even work-load balancing, we will tackle an essential challenge: how can we handle concurrent operations into the quadtree data structure?

This problem is actually simpler than we anticipated, because the only concurrent modifying operation that is done to the quadtree is body inserts; once the tree is fully built no thread ever modifies its data. Our basic approach to this problem is to have threads acquire locks for nodes they are currently considering, then releasing them when they have completed the insert or moved on to another node. This is a form of fine-grained locking since we are not locking the entire tree structure, only the nodes that threads are considering for the insert.

When a thread is inserting into a tree node there are 3 possible options:

1. The node is not a leaf node: The thread continues traversing down the tree until it reaches a leaf.
2. The node is a leaf node and there is no body at the node: The thread simply inserts the body to the node.
3. The node is a leaf node and there is a body at the node: This signifies a collision, and the thread needs to allocate subtrees for the node and insert both bodies of the collision into the correct subtree.

Initially, it seems that a thread only needs to lock the node after it recognizes that the node is a leaf, since that is the only case in which any tree modification will occur. However, if we do not lock the node before checking that it is a leaf, multiple threads could be trying to update the same node's body after recognizing the node as a leaf, which results in a race condition and undefined behavior. Since we lock every node of the tree we consider during an insert, the root node receives a significant amount of lock activity, which slows down concurrent tree inserts. However, we still observe speedup because multiple bodies are pipelined through the tree nodes as threads attempt to insert them, which is much faster than sequentially inserting bodies one by one. Even so, we noticed that our tree building algorithm could take better advantage of parallelism, so it is the topic of our next implementation.

Now we reach the meat of the parallelization challenge: how can we partition bodies between threads such that each thread does roughly equal work during the force accumulation phase? The high-level idea of the cost-zone method is that we use the work done for each body in the previous iteration as an estimate for how much work each body requires in the current iteration. We keep track of how much work a body requires by counting the number of nodes it traverses during the force aggregation phase of the previous iteration.

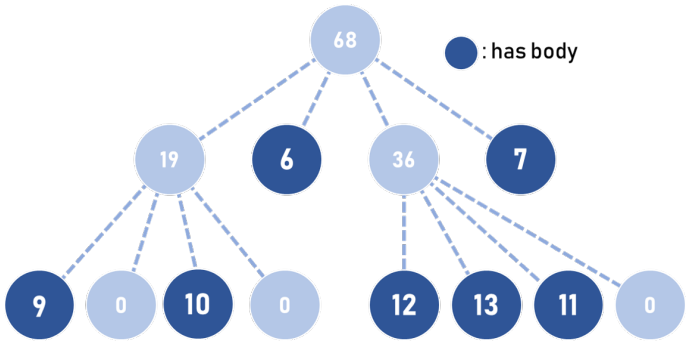


Figure 16: Example of quadtree nodes with cumulative work.

To perform the partition, we first assign every node of the quadtree to have the work of the sum of its subtree. This allows us to perform an in-order traversal of the tree without needing to recurse down to each leaf nodes. We do this during the tree traversal stage highlighted in the pseudo-code above.

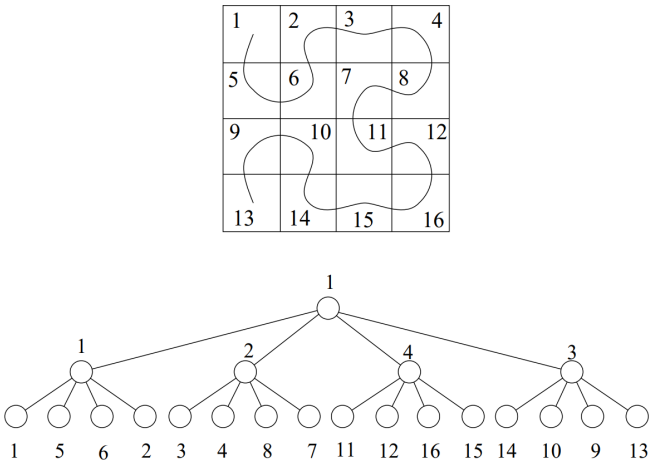


Figure 17: Ordering of quadtree nodes to perform in-order traversal.

Next we perform the partition by dividing the total work W into T roughly equal chunks, where T is the number of threads. Each thread is assigned a disjoint range in $[0, W]$ with length $\approx \frac{W}{T}$, and we perform an in-order traversal of the tree to pick out bodies to add to each thread's partition. These traversals can easily be done in parallel, because no modification is being done to the tree.

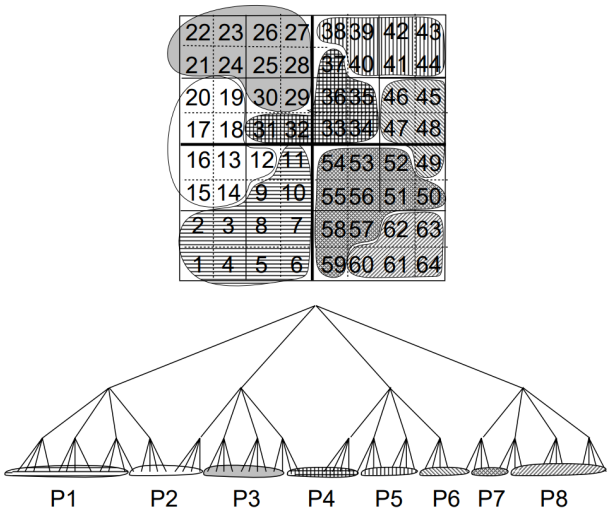


Figure 18: Example of resulting body partitions using the cost-zone method.

Below are the benchmark results for our first fine-grained locking implementation of the Barnes-Hut Algorithm:

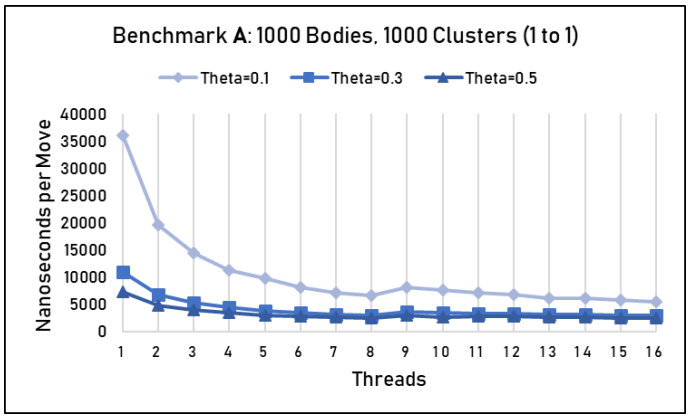


Figure 19: Benchmark A on Fine-Grained Locking Barnes-Hut Implementation.

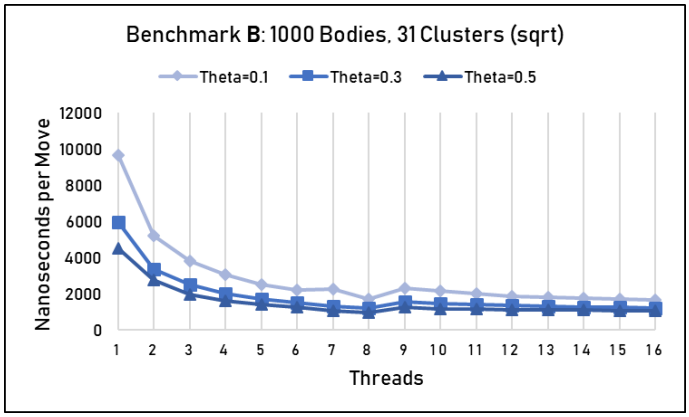


Figure 20: Benchmark B on Fine-Grained Locking Barnes-Hut Implementation.

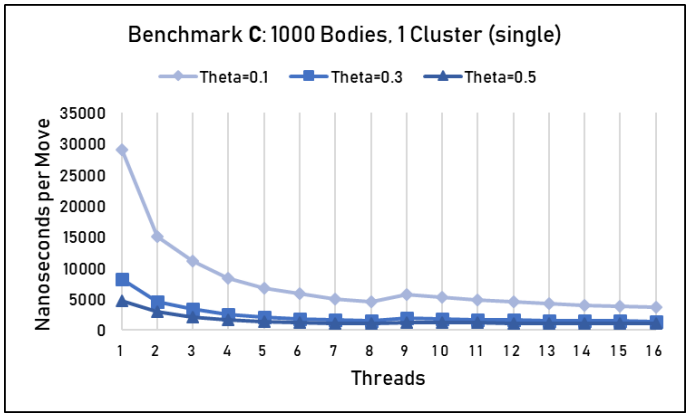


Figure 21: Benchmark C on Fine-Grained Locking Barnes-Hut Implementation.

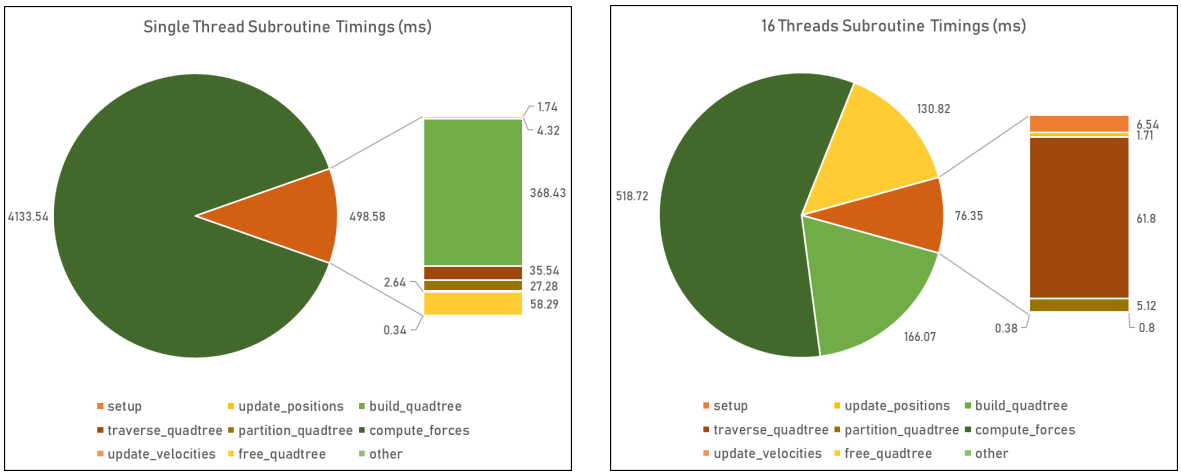
From these graphs we notice that performance is significantly better in benchmark **B** than benchmark **A** or **C**. Consider the quadtree structure implied by a uniform distribution of bodies: since strong clustering is rare, it becomes difficult to approximate a cluster as a single body. Because of this, we utilize the quadtree much less in benchmark **A**, especially with lower θ values such as $\theta = 0.1$. In the same vein, we can think of the single cluster in benchmark **C** as being a uniform distribution in a smaller space around the center of the cluster, resulting in similar behavior to benchmark **A**. We focus on the speedup of benchmark **B**, which we consider to be the representative benchmark, since it effectively utilizes the quadtree structure.

$\theta = 0.1$	
Threads	Speedup
1	1.0×
2	1.84×
4	3.17×
8	5.69×

$\theta = 0.3$	
Threads	Speedup
1	1.0×
2	1.77×
4	2.93×
8	4.94×

$\theta = 0.5$	
Threads	Speedup
1	1.0×
2	1.64×
4	2.75×
8	4.61×

Looking at how the speedup of our implementation on benchmark **B** varies with θ, we see that we get less speedup with larger values of θ. This is because as θ increases, we utilize the quadtree more to make broader force estimates. Because of the increased use of the quadtree to make approximations, it is more likely that body costs will vary between iterations, since the amount we will need to traverse through the quadtree can more easily very as bodies move through space. This causes the cost-zones partition scheme to be less accurate, and thus introduces more imbalanced work between threads, meaning one thread will become a bottleneck and slow execution for the simulation step.



(a) Single Thread

(b) 16 Threads

Figure 22: Subroutine timings with 10000 bodies, 50 steps, and $\theta = 0.3$

In the naive implementation, over 99% of the time spent during execution is on force aggregation. Here we see significant portions of time spent on quadtree operations, such as building, traversing, partitioning, and freeing the quadtree. As we increase the number of threads, the relative portion of work done to the quadtree increases, since the work of building and traversing the tree is less parallelizable than aggregating forces using the tree. Specifically, we see a significant portion of time spent on freeing and building the tree in the 16 threadded version, which we attempt to reduce in our next implementation.

gsim-lockfree

In the `gsim-barneshut` implementation, we utilized fine-grained locking to only lock the portions of the quadtree which threads are currently considering during inserts. These locks were implemented utilizing the `omp_lock_t` structure provided by the OpenMP API. However, as we increase the number of threads which perform concurrent inserts into the quadtree, the fact that root nodes are constantly being acquired and released poses a significant bottleneck.

To resolve this issue, we made a single major change from our `gsim-barneshut` implementation: we implemented a lock-free quadtree. In order to preserve the atomicity of tree inserts, we need to be able to perform an atomic compare-and-swap on two values simultaneously:

1. A flag determining whether the node is a leaf
2. A pointer to the body attached to the node (or `NULL` if no body is attached to the node)

We initially considered performing the double compare-and-swap operation discussed in our lecture on lock-free data structures. However, this operation is not natively supported by the x86 architecture, and thus we were forced to explore other alternatives. Next we considered treating the combination of the leaf flag and the body pointer as a single double word value (pointers are 64 bits, so this would require 128 bits), but the x86 64 bit architecture also does not support 128 bit compare-and-swap operations. Our solution to this problem is to treat the body pointer as an integer of type `uintptr_t`, and use the last bit of the pointer as the leaf flag. We are able to do this since body structures are 64 byte aligned, since we ensure that each body is on an independent cache line, resulting in the least significant bit of any valid body pointer to be zero.

Figure 23: Value stored at quadtree nodes to represent body and leaf flag.

Now this value stored in quadtree nodes can be compare-and-swapped by threads atomically to determine the state of the node and what action should be performed during an insert. For example, if we successfully test that the value is a leaf and the body is `NULL` and replace it with a new body in a single CAS, then we have completely atomically inserted a body at a leaf node. The other case is a little more complicated: if we successfully test that the value is a leaf and the body is not `NULL` and replace it completely with `0x0`, then we have successfully initiated a collision reinsertion. However, before we perform this test, if we detect the condition at all, we need to ensure that subtrees are allocated before the test succeeds. This is because as soon as the test succeeds, another thread can attempt to insert into a subtree which must be initialized by this point. To do this we perform a series of loops and compare-and-swaps for each of the 4 subtrees to ensure they are allocated and assigned to the quadtree node before updating the leaf and body status. We present our benchmark results in the following section.

Results

Representative Implementation and Parameters

We have chosen the lock-free implementation `gsim-lockfree` as our representative implementation, since it is our most well-performing Barnes-Hut Algorithm variant. Below are the implementation's results on all benchmarks:

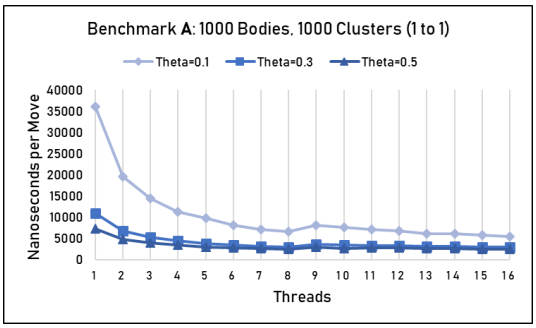


Figure 24: Benchmark A on Lock-Free Barnes-Hut Implementation.

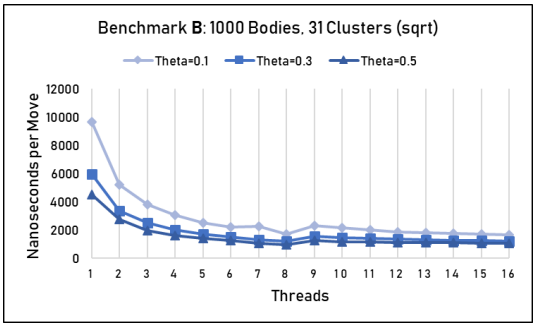


Figure 25: Benchmark B on Lock-Free Barnes-Hut Implementation.

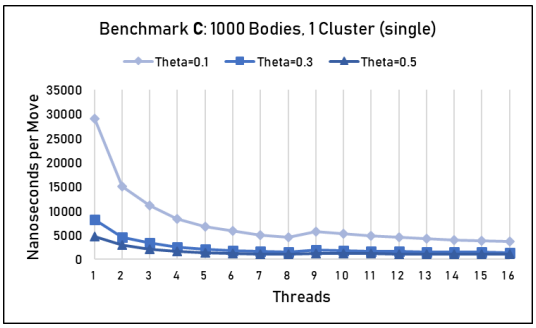


Figure 26: Benchmark C on Lock-Free Barnes-Hut Implementation.

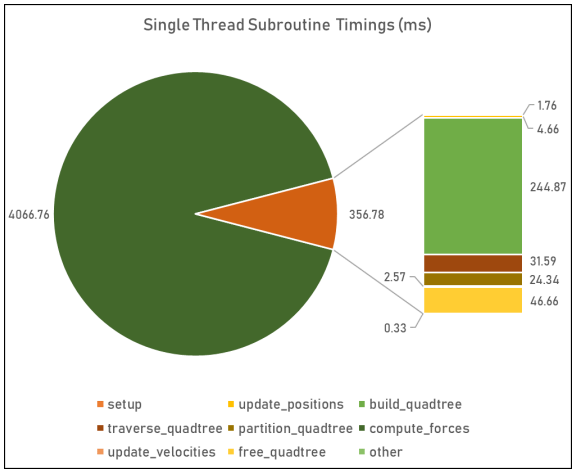
Here we will not go into any analysis or discussion that we have previously explored for other implementations (this can be found in the `gsim-bad` and `gsim-barneshut` subsections of the Approach section above). The following table illustrates our lock-free implementation's speedup with varying θ on our representative benchmark **B**:

$\theta = 0.1$	
Threads	Speedup
1	1.0×
2	1.83×
4	3.17×
8	5.71×

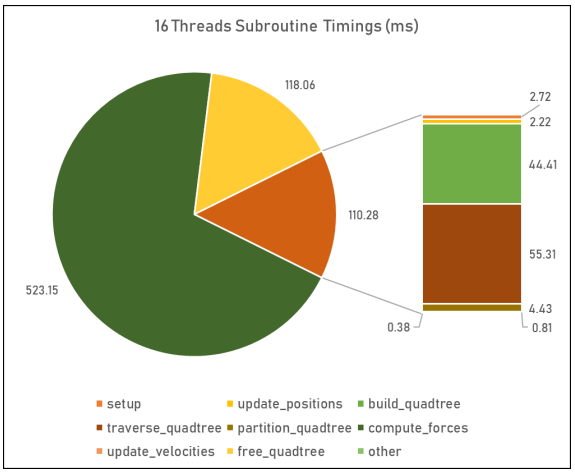
$\theta = 0.3$	
Threads	Speedup
1	1.0×
2	1.79×
4	3.00×
8	5.18×

$\theta = 0.5$	
Threads	Speedup
1	1.0×
2	1.75×
4	2.81×
8	4.73×

Our speedup illustrated in the tables above is more linear with number of threads than our fine-grained lock implementation. This is because we have significantly reduced the bottleneck of building the tree concurrently by introducing atomic inserts without locks. By parallelizing the tree construction phase of the algorithm, we have lowered the constant overhead which must be done at the start of each simulation step, and thus we have improved parallel performance as a whole.



(a) Single Thread



(b) 16 Threads

Figure 27: Subroutine timings with 10000 bodies, 50 steps, and $\theta = 0.3$

As we can see in the subroutine timing breakdown above, a more significant majority of the work is dedicated to updating body forces than our fine-grained lock implementation, especially as the number of threads increases. This is further evidence of the success of our lock-free implementation, and how it reduces tree construction overhead. We note that freeing the quadtree remains a significant bottleneck, as it must be done at the end of each simulation step and it is not easily parallelizable, since it is completely reliant on memory performance. It is possible to arrange the memory of our tree structure in such a way to promote spatial locality and reduce the need for freeing the entire tree each step, but this is an extremely complex task that we do not expect to see much performance gain from. Our current implementation already has quite good temporal locality, since each thread is essentially assigned a cluster of bodies, and that cluster will access the same memory over and over during the force computation phase of the algorithm (bodies consider closer bodies more often than farther away ones).

We notice here that on our representative benchmark, benchmark **B**, with our representative choice of $\theta = 0.3$, we complete the benchmark simulation with 993.93129 NPM, whereas our fine-grained lock implementation requires 1231.57978 NPM. This is a 1.24× speedup and indicates the success of our lock-free implementation. In all following benchmarks, including our cross-system analysis, we will be using benchmark **B** with $\theta = 0.3$ on `gsim-lockfree` as the best example our of our parallel implementations.

Problem Size

In our application, problem size has a significant impact on parallel performance. That is to say, different workloads do in fact exhibit different execution behavior, since as we increase the number of bodies we are simulating on, the influence of constant overhead costs are reduced, and speedup becomes more obvious.

8 thread speedup	
Number of Bodies	Speedup
10	0.52×
100	2.49×
1000	3.63×
10000	5.82×
100000	7.12×

With few bodies, we see that additional threads actually reduce performance, since we must deal with the overhead of spawning and joining threads when there is not much work to do over all. As we increase the number of simulated bodies, much of the overhead becomes insignificant. The tests above were performed using `gsim-lockfree` with benchmark **B** and $\theta = 0.3$ for 50 steps.

Our benchmarks in previous sections do not use large numbers of bodies for ease of benchmarking; 100000 bodies takes minutes to simulate on single-threaded operation with our poorer-performing implementations.

Profiling

By using `perf` we can get the following performance counters from our lock-free implementation:

```
perf stat -B -e cache-references,cache-misses,cycles,instructions,branches,
faults,migrations ./gsim-lockfree 16 316 100000 50 0.3 1234

Running with 16 threads. Max possible is 16.

        total time: 9422.69 ms
        -------------------------------------------------
        14.33 ms        0.15 %          setup
        0.00 ms         0.00 %          write_file
        22.69 ms        0.24 %          update_positions
        465.71 ms       4.94 %          build_quadtree
        624.23 ms       6.62 %          traverse_quadtree
        69.07 ms        0.73 %          partition_quadtree
        7328.51 ms      77.78 %         compute_forces
        11.08 ms        0.12 %          update_velocities
        886.55 ms       9.41 %          free_quadtree
        0.52 ms         0.01 %          other

 Performance counter stats for './gsim-lockfree 16 316 100000 50 0.3 1234':

     5,861,759,489      cache-references:u
       158,042,755      cache-misses:u            #    2.696 % of all cache refs
   413,083,906,167      cycles:u
   324,504,728,015      instructions:u            #    0.79  insn per cycle
    37,619,676,991      branches:u
           538,956      faults:u
                 0      migrations:u

       9.448114445 seconds time elapsed
```

Figure 28: Profiling results for our lock-free implementation with 16 threads.

Only 2.696% of all cache references are cache misses, which means that our algorithm does in fact take advantage of temporal locality to increase cache performance. This could also be a result of our avoidance of false sharing between threads by padding structures to cache lines.

We also notice that our implementation runs with 0.79 instructions per cycle, even though the Intel Xeon processors on the GHC machines advertise 8 dual-precision floating-point operations per cycle for each thread. This is clear evidence that we are not in fact taking full advantage of the processor's functional units. Even though we have great cache performance, our program is still fundamentally memory bound due to the cost of traversing the tree and the large amounts of memory accesses that requires.

Overall, these results indicate that our implementation is in fact efficient with the way it accesses memory, and that we could actually introduce more accurate math libraries to take advantage of the CPU resources without significant loss to performance. However, our main performance bottleneck is still the large amount of memory accesses introduced by quadtree operations. This includes costly dynamic memory allocation and freeing function calls.

If we were to continue this project and optimize one aspect of our implementation, it would be to statically allocate a large space for the quadtree initially, and utilize those resources until we are forced to reallocate. This prevents the need to allocate and free the entire quadtree each simulation step and would drastically improve memory performance. However, we would need to be very careful with how we keep track of allocated nodes and how they connect. We could take advantage of the in-order tree traversal method presented in the Approach section above to index each quadtree node in an allocated array, so we know which ones to assign as subtrees in the event of an insert collision. This method would also increase spatial locality and decrease cache misses further.

Cross-System Analysis

Speedup against 1 thread on GHC (100000 bodies)		
Threads	GHC	Latedays
1	13043.22 NPM $\rightarrow$ 1.00$\times$	15601.17 NPM $\rightarrow$ 0.84$\times$
2	6877.98 NPM $\rightarrow$ 1.90$\times$	8301.92 NPM $\rightarrow$ 1.57$\times$
3	5008.33 NPM $\rightarrow$ 2.60$\times$	10084.55 NPM $\rightarrow$ 1.29$\times$
6	3082.10 NPM $\rightarrow$ 4.23$\times$	3512.18 NPM $\rightarrow$ 3.71$\times$
12	2404.02 NPM $\rightarrow$ 5.43$\times$	2260.04 NPM $\rightarrow$ 5.77$\times$

Nodes on the Latedays cluster have inconsistent speedup results as we increase the number of threads. For example, we get a super-linear speedup of 2.87$\times$ when increasing the number of threads from 2 to 3. This is likely because of characteristics of the memory system that Latedays utilizes. This is purely speculation: it is possible that cache misses are extremely costly on the Latedays nodes. This means that as we increase the number of threads and subdivide the bodies into smaller chunks, there are fewer conflict cache misses, and thus memory performance can vary drastically between thread counts, assuming each thread has an independent cache. Apparently there is a Latedays node with 60 cores (Intel Xeon Phi), however none of the queues for `qsub` result in using this node, so we were not able to perform any tests with over 12 threads.

Other Considerations

Our project specifically targets multi-core CPU platforms by utilizing multithreading across cores (and simultaneous multithreading if it is supported by a single core). We also have the loose requirement that the CPUs compute resources are homogeneous, since our cost-zone partition scheme assumes that each core of the CPU has the same computing power. However, when performing large-scale galaxy simulations, we would ideally be able to simulate orders of magnitude larger than 100000 bodies. This is not really possible to do on a single CPU, since by Moore's law, there is only so much performance we can get out of area on a chip, and we cannot have unbounded cores on a CPU. A more practical target architecture would be a distributed scientific supercomputer, whose compute nodes each have multicore CPUs and communicate with one another to provide a broader level of parallelism. This would result is a much greater degree of scaling, which would be suitable for the scientific application.

We decided to use the cost-zones method of work load partitioning between threads. This is a reasonable method of estimating the work each body requires to compute forces, and easily partitioning bodies between threads. However, if we wanted this application to scale to scientific uses, we would likely utilize the WS partition scheme with MPI as a framework to communicate betweeen processes running on nodes of a supercomputer. The WS method assigns a spatially relevant key to each body, and assigns bodies to processes by sorting the bodies based on that key.

A final approach we could have taken is to implement a similar algorithm on the GPU to take advantage of its massive parallelization. This, however, would have required a completely different approach from our algorithm, since it is difficult to have complex data structures such as a quadtree accessed in a kernel. Popular GPU methods include the fast multipole method, and variants of the Barnes-Hut Algorithm which utilize sparse tree structures, heaps, or similar.

Conclusion

We have implemented 3 versions of a simple galaxy simulation focused on only body-to-body gravitational forces. One of the implementations is an optimized parallel naive all-pairs $O(n^2)$ algorithm to serve as a baseline. The other two implement the Barnes-Hut Algorithm with variants of a concurrent quadtree: fine-grained locking and lock-free. We have shown that galaxy simulation in terms of purely gravitational forces is highly parallelizable on multi-core CPU platforms with homogeneous compute resources by utilizing the cost-zones partition scheme. Our implementation is significantly memory-bound, although our cache performance is quite good with a $\approx 3\%$ cache miss rate. This indicates that an ideal target architecture utilizes an efficient memory system, including quick L1 cache-hit accesses and an optimized cache-coherence system for multi-core. Our lock-free implementation can perform a simulation of over one million bodies in seconds with compiler optimization flags enabled (`gcc -Ofast`), and our visualizations are intuitively reasonable and preserve our definition of correctness.

As a whole the project was an exploratory dive into many parallel architecture and programming concepts: problem subdivision, work-load assignment, concurrent data structures, artifactual communication, cache-coherence considerations, profiling, benchmarking, and scaling analysis.

Figure 29: NGC 4414, approximately 60 million light-years from Earth.

References

[1] *CMU 15-418/618 - Lecture 9: Parallel Programming Case Studies*,
http://www.cs.cmu.edu/~418/lectures/09_casestudies.pdf

[2] Ananth Y. Grama, Vipin Kumar, and Ahmed Sameh. *Scalable Parallel Formulations of the Barnes-Hut Method for n-Body Simulations*,
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.49.7107&rep=rep1&type=pdf

[3] John K. Salmon. *PARALLEL HIERARCHICAL N-BODY METHODS*,
https://thesis.library.caltech.edu/6291/1/Salmon_jk_1991.pdf

[4] Lars Nyland, Mark Harris, and Jan Prins. *Fast N-Body Simulation with CUDA*,
https://developer.download.nvidia.com/compute/cuda/1.1-Beta
/x86_website/projects/nbody/doc/nbody_gems3_ch31.pdf

[5] Guy Blelloch and Girija Narlikar. *A Practical Comparison of N-Body Algorithms*,
https://www.cs.cmu.edu/afs/cs.cmu.edu/project/scandal/public/papers/dimacs-nbody.pdf

[6] Benedict Steinbush, Marvin-Lucas Henkel, Mathias Winkel, and Paul Gibbon. *A Massively Parallel Barnes-Hut Tree Code with Dual Tree Traversal*,
http://juser.fz-juelich.de/record/808800/files/ParCo2015-paper.pdf

[7] David Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware / Software Approach*,
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.101.4418&rep=rep1&type=pdf

[8] Martin Burtscher, Keshav Pingali. *An Efficient CUDA Implementation of the Tree-Based Barnes Hut n-Body Algorithm*,
http://iss.ices.utexas.edu/Publications/Papers/burtscher11.pdf

[9] *Verlet Integration*,
https://en.wikipedia.org/wiki/Verlet_integration

[10] *Oscilation*,
http://kahrstrom.com/gamephysics/wp-content/uploads/2011/08/oscilation.jpg

[11] *Instructions per Cycle*,
https://en.wikipedia.org/wiki/Instructions_per_cycle

[12] *Galaxy*,
https://en.wikipedia.org/wiki/Galaxy

[13] *Universe Box*,
http://www.lsw.uni-heidelberg.de/users/mcamenzi/images/Universe_Box.gif

Division of Work

Equal work was performed by both project members.