Heterogeneous Distributed Barnes-Hut Scheduler:

Final Project Report

Zhuoyi Zou (zhuoyiz), Vanessa Lam (yatheil)

WEBSITE URL

https://cchewies.github.io/15418-project.html

https://github.com/cchewies/418-final

SUMMARY

We implemented a single-node, multi-thread-per-node MPI Barnes-Hut simulation and a multi-node, single-thread-per-node distributed "MiniMPI" Barnes-Hut simulation and compared the two. Running on a distributed network of GHC machines connected by high-latency, low-bandwidth interconnects, our project goal was to achieve speedup comparable to the same number of CPU threads on a single GHC machine. We then extended our project to support dynamic load balancing for clusters of non-uniformly powerful computers.

BACKGROUND

Note: We use terminology specific to galaxy simulations (body = star) throughout the report and code, though our work is applicable to any N-body simulation.

Barnes-Hut is an algorithm for efficient O(NlogN) N-body simulations. The algorithm computes the net force on each star by summing pairwise interactions (e.g., gravitational or electrostatic forces) from all other stars, using their masses, positions, and distances to determine the resulting acceleration and motion in each iteration. The algorithm has two computationally expensive components, which are run at every single simulation timestep:

1. An O(NlogN) quadtree construction step, involving iterating over the stars and building a quadtree to represent their spatial distribution.
2. An O(NlogN) force calculation step, involving iterating over the stars and computing an approximation of the force experienced by each star.

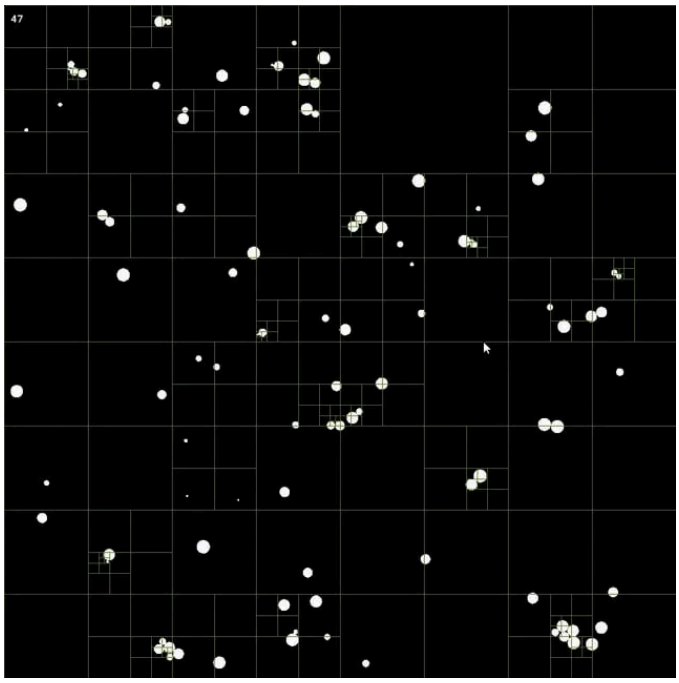


Figure 1: A quadtree for Barnes-Hut simulations (1)

Both the quadtree construction and force calculation have many parallelizable portions, with the most straightforward strategy being to partition the N stars among different computers. However, both steps also potentially involve a substantial amount of communication, which makes it less ideal for running in a distributed computing environment. Especially on a high-latency, low-bandwidth interconnect, distributed computing is often bottlenecked by communication, where factors like wireless connections, network routing, and the speed of light can increase communication time tenfold over computing on a single machine.

Parallelized workloads generally have one of three work assignment techniques: static, semi-static, or dynamic. Semi-static is particularly well-suited for Barnes-Hut, since workloads for each star are cheap to compute and the spatial distribution of the stars changes slowly. A well-balanced work distribution is likely to remain well-balanced for several iterations, reducing the amount of synchronization required for assignment.

Existing dynamic scheduler implementations, such as CUDA Dynamic Parallelism (2) and MPI-based distributed schedulers (3), enable runtime adaptation of work allocation. While CUDA primarily focuses on intra-device parallelism, multi-GPU or multi-node execution requires additional host-side coordination, as device-side kernels cannot directly manage work across nodes; concurrency is limited to the scope of a single device, and memory coherence is guaranteed only within the device's global memory. In contrast, MPI-based dynamic schedulers, such as the one proposed in the paper, explicitly target multi-node environments by assigning tasks with the goal of minimizing overall completion time based on predicted convergence and scaling efficiency. The scheduler dynamically adjusts the number of workers assigned to each job using heuristics driven by marginal performance gains, enabling adaptive resource distribution at runtime. However, this approach assumes homogeneous worker performance, modeling training speed primarily as a function of worker count rather than node-specific characteristics. As a result, it improves utilization and scalability in distributed settings but remains limited in its ability to optimally balance workloads in heterogeneous clusters.

This project aims to explore the possibility of achieving effective scaling beyond a single machine to perform larger simulations and improve throughput, as well as accurately measuring the heterogeneity of machines to improve resource utilization.

MINIMPI

We chose to use the GHC machines (and other campus computers) for our computing environment, which are most easily accessed through SSH. Since campus computers do not allow for passwordless SSH, the built-in distributed configuration for MPI does not work. We instead implemented MiniMPI, our own subset of MPI, to run across different campus computers.

MiniMPI is built on top of TCP, involving an initialization handshake where every node provides its hostname to a coordinator node, which redistributes the hostnames to the rest of the nodes. Each pair of nodes can then set up a TCP connection, for a crossbar interconnect.

To simplify logic, we require the user to supply each node with the hostname of the coordinator, as well as a unique rank number and the total number of nodes.

MiniMPI implements a total of 3 operations used by our Barnes-Hut simulation:

1. Blocking send
2. Blocking receive
3. Sync

Sync is a more memory-efficient version of Allgather that uses the same source and destination buffer. It is particularly useful in situations where each node modifies a disjoint contiguous subset of an array of elements, and only needs to "sync" the remaining elements with other nodes.

Our initial naive sync implementation had every node send its data section to the coordinator, which would then broadcast the synced data back. This approach was prohibitively expensive for any number of threads over 2.

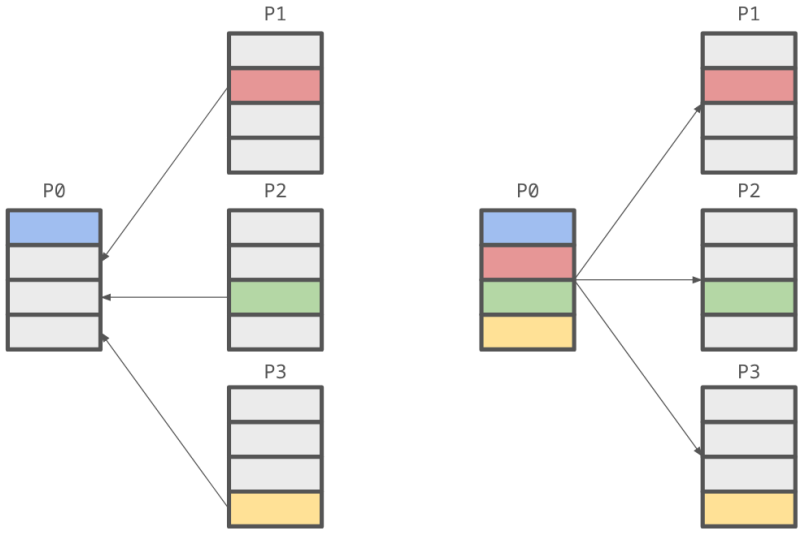


Figure 2: Naive sync implementation

Our final design implements sync using blocking sends and receives in a ring. Our specific implementation does unfortunately require an even number of nodes, but we did not have time to write a more sophisticated implementation that works with any number of nodes. A potential improvement may be pipelined allgather (5).

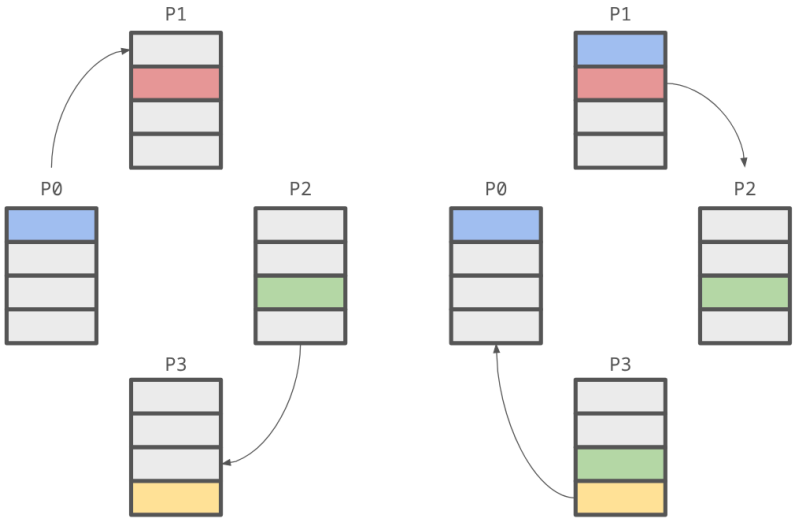
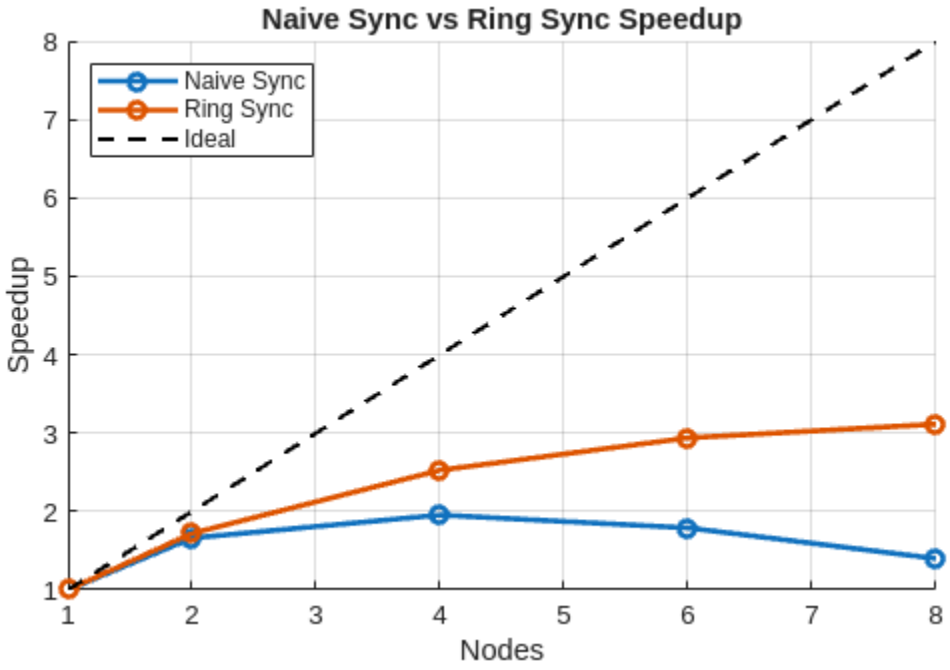


Figure 3: Ring sync implementation

With N stars and M nodes, the total number of communication steps and total amount of data sent in both implementations are O(M) and O(N*M), respectively. However, naive sync passes a maximum of O(N*M) data through a single node (P0) during the broadcast phase, as it needs to send the full updated stars vector to all other nodes. Ring sync, on the other hand, only passes a maximum of O(N) data through a single node, as each node receives each chunk of the buffer once and sends each chunk of the buffer once. For bandwidth-bound communication, this contributes to a large speedup increase with ring sync over naive sync.



We also implemented barriers and broadcast, though these are not used in our final distributed Barnes-Hut design.

DESIGN CONSTRAINTS

Our designs all satisfy the following constraint on the Barnes-Hut simulation:

- The force calculation for each star must use only the state of the simulation at the previous timestep

This constraint eliminates two specific optimizations that would result in better speedup but non-deterministic simulation:

1. Reducing communication by allowing the state of stars not allocated to a specific node to become stale
2. Increasing spatial locality by updating the positions of a star in the same data that the positions of other stars are being read from

APPROACH: SCALING

We started with a naive single-threaded Barnes-Hut implementation to determine the slowest part of the algorithm, which was by far the force calculation. Although the force calculation and quadtree construction are both O(NlogN), the additional math required in force calculation (namely, square roots and divisions) resulted in a much larger constant factor. Force calculation consistently took 10x as long as quadtree construction.

Building an initial parallel implementation using MPI on a single GHC machine, targeting only the force calculation, we were able to achieve a 2x force calculation speedup on 8 threads by evenly distributing the stars based on index. However, the quadtree calculation time also tripled, meaning that the previously fast quadtree construction was now taking up 40% of our total runtime. Additionally, we ran all of our benchmarks on a randomly generated "galaxy", so the initial index-based distribution resulted in horrible spatial locality as well. Running on MPI, we were only able to get a speedup of barely over 2x on 8 threads. Running on MiniMPI, speedup was capped at 1.5x with 2 threads. The naive broadcast-based sync operation likely also contributed to the bad MiniMPI speedup.

To cut down on the quadtree construction time, we attempted a parallel quadtree construction algorithm where each node builds a subset of the quadtree, joining with other nodes' results to form a complete quadtree. Because quadtrees are pointer-based structures, though, the serialization and deserialization step in joining took longer than the full quadtree construction, so we did not continue with this optimization attempt.

To improve spatial locality, we implemented an additional step during each iteration of sorting the stars by their Morton ordering.

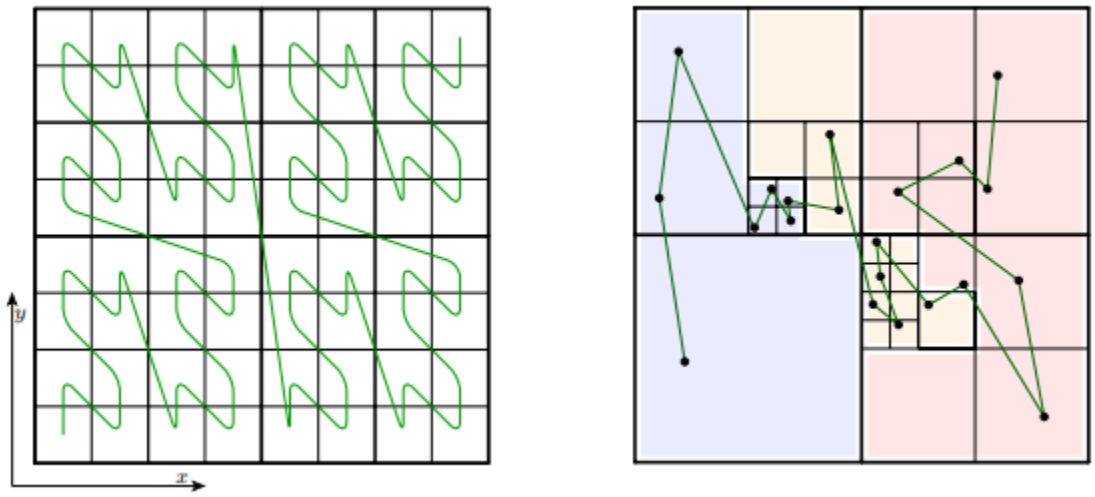
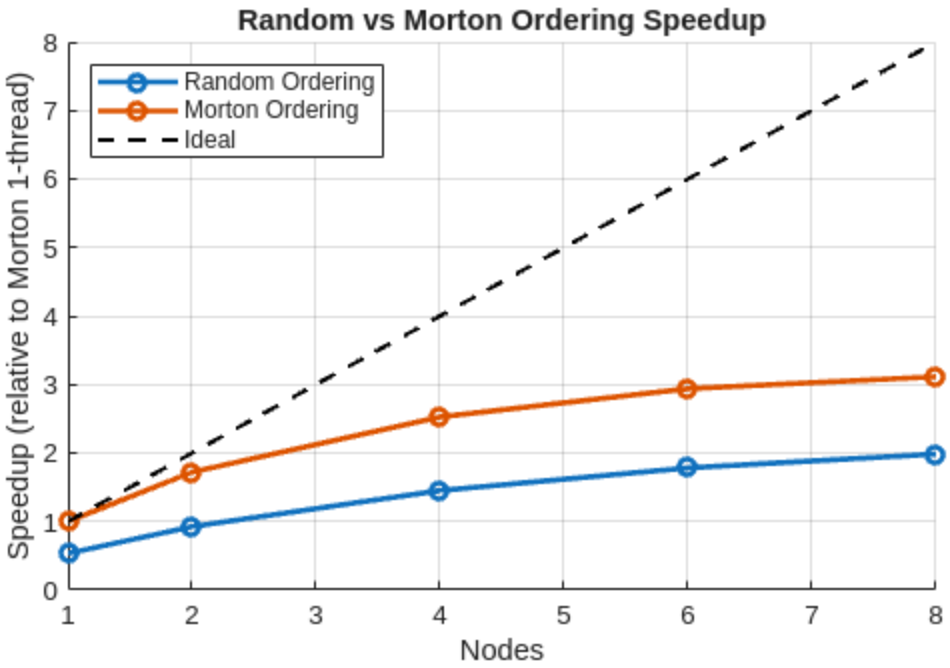


Figure 4: Morton ordering of a quadtree (4)

Now, using index-based work distribution, Morton ordering allows any contiguous region of the stars array to also have high spatial locality. Although this added an extra 5% time spent sorting the stars, it cut the force calculation time in half and sped up quadtree construction by 5x, resulting in a net speedup of around 2x.

Sorting the stars by Morton ordering every cycle is unnecessary, however, and sorting per batch can result in much higher speedup. From our testing, 10 iterations per batch yielded the best speedup of 3.34x with 8 threads on a single machine, and 3.14x with 8 threads on separate machines. Increasing the batch size above 10 resulted in spatial locality decreasing enough to outweigh the benefit from lower sorting work.



With our final design, Morton ordering consistently produces a 1.5x speedup over random ordering with the same parameters.

APPROACH: LOAD BALANCING

Distributed computing differs from single-machine computing in two main ways:

1. Interconnect latency is typically much higher, and bandwidth is much lower
2. Distributed computers are more likely to have larger variances in performance compared to cores on a single machine

We developed a reasonably efficient distributed Barnes-Hut algorithm already, but the load balancing so far has been static, equal-star allocation. Running in a heterogeneous computing environment, the speedup is bound by the performance of the slowest computer in the cluster.

Our first load-balancing strategy was prefix-sum-based. The motivation was to distribute computational work evenly across processing units despite non-uniform force evaluation costs per star. We assign each star a cost corresponding to its force computation time. Using this, we construct a prefix-sum array over the Morton-ordered list of stars, where each entry represents the cumulative workload up to that point. To partition the workload, we divide the total accumulated cost into equal-sized buckets (one per processor). The start

and end indices for each partition are then determined by locating the boundaries in the prefix-sum array that correspond to these equally spaced workload thresholds.

Algorithm 1: Domain Decomposition for Load Balancing

```
// Initialize boundaries
boundaries ← array of size nprocs + 1
boundaries[0] ← 0
boundaries[nprocs] ← stars.size()

// Calculate split points based on cumulative work
for r ← 1 to nprocs − 1 do
    target ← ⌊(r / nprocs) × total⌋
    // Find first index where prefix sum reaches target work
    boundaries[r] ← lower_bound(prefix.begin, prefix.end, target)

// Assign local range to current processor
my_start ← boundaries[pid]
my_end ← boundaries[pid + 1]
```

We extended prefix-sum-based partitioning to support heterogeneous nodes via a dynamic, performance-aware strategy. In the absence of timing information, we use the standard prefix-sum load balancing described above. From subsequent iterations onward, partitions are adjusted based on observed execution times. Each node records its previous force computation time, which is treated as an inverse proxy for performance to compute normalized weights (with a small epsilon for stability). A prefix sum over these weights determines each node's proportional share of the total workload. For node i, its weight interval is mapped into the global prefix-cost space to obtain target cumulative cost bounds, which are then converted into index ranges over the Morton-ordered star list via binary search. This approach ideally preserves spatial locality while adapting workload distribution to heterogeneous node performance.

Algorithm 2: Adaptive Load Balancing via Performance Weights

Input: Previous execution times T, total work W_{total}, prefix sum of work $prefix$

```
// Compute performance weights (inverse of time)
```
$sum\_w \leftarrow 0.0$
for $i \leftarrow 0$ **to** $nprocs - 1$ **do**
  $weights[i] \leftarrow 1.0/(T[i] + \epsilon)$ // $\epsilon = 1e-9$
  $sum\_w \leftarrow sum\_w + weights[i]$

```
// Normalize and compute cumulative weight distribution
```
$W_{prefix}[0] \leftarrow 0.0$
for $i \leftarrow 0$ **to** $nprocs - 1$ **do**
  $weights[i] \leftarrow weights[i]/sum\_w$
  $W_{prefix}[i+1] \leftarrow W_{prefix}[i] + weights[i]$

```
// Map to prefix-cost space and find indices
```
$target\_start \leftarrow \lfloor W_{prefix}[pid] \times W_{total} \rfloor$
$target\_end \leftarrow \lfloor W_{prefix}[pid+1] \times W_{total} \rfloor$
$my\_start \leftarrow \text{lower\_bound}(prefix, target\_start)$
$my\_end \leftarrow \text{lower\_bound}(prefix, target\_end)$

However, this approach didn't achieve effective balancing. While the cost metric is a good measurement of the amount of work a thread has, it adds tracking overhead per star and requires additional calculation to turn it into a useful metric for load balancing. Specifically, this increases the amount of information stored in the star struct, reducing the amount of data per cache line and worsening cache locality.

Our timing code already exposes a much more straightforward measurement of performance. Specifically, since each node already knows the amount of work every other node has for the previous batch, we can introduce code to share the time each node took for computation. This information together allows the code to determine the optimal workload for each node.

Algorithm 3: Timing-Based Adaptive Load Balancing

Input: Average compute times T, current work distribution C, total stars N

Output: Updated work counts C' and displacements D for $nprocs$

`// Compute total relative speed of the system`

$S_{total} \leftarrow 0$

for $i \leftarrow 0$ **to** $nprocs - 1$ **do**

$speed_i \leftarrow \frac{C[i]}{T[i]}$

$S_{total} \leftarrow S_{total} + speed_i$

`// Calculate base work per unit of speed`

$W_{unit} \leftarrow \lfloor N / S_{total} \rfloor$

$N_{assigned} \leftarrow 0$

for $i \leftarrow 0$ **to** $nprocs - 1$ **do**

$C'[i] \leftarrow \lfloor W_{unit} \times speed_i \rfloor$

$N_{assigned} \leftarrow N_{assigned} + C'[i]$

`// Distribute remaining stars`

$N_{rem} \leftarrow N - N_{assigned}$

$base \leftarrow \lfloor N_{rem} / nprocs \rfloor$

$extra \leftarrow N_{rem} \pmod{nprocs}$

for $i \leftarrow 0$ **to** $nprocs - 1$ **do**

$C'[i] \leftarrow C'[i] + base + (i < extra ? 1 : 0)$

`// Update displacements for MPI communication`

$D[0] \leftarrow 0$

for $i \leftarrow 1$ **to** $nprocs - 1$ **do**

$D[i] \leftarrow D[i-1] + C'[i-1]$

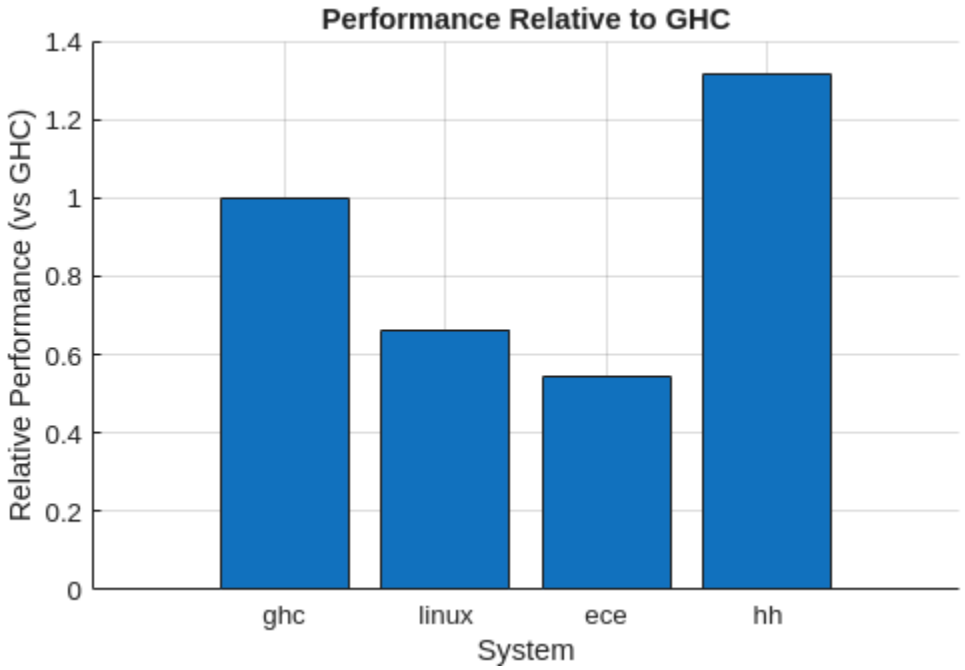
This design does have the limitation of assuming the work per star is roughly evenly distributed, as it only provides a number of stars to assign to each node, not specific stars. In the Barnes-Hut simulation, work per star can be uneven depending on the simulation state. However, our single-galaxy simulation did satisfy the evenly distributed work assumption. For an uneven star-work distribution, an alternative timing-based approach where workloads are gradually increased or decreased depending on calculation time may result in better load balancing.

Since we run the load balancing timing data sharing every batch, the impact on the speedup is minor, with around a 0.03x decrease in speedup for both the single-node MPI and distributed implementations.

RESULTS

As mentioned earlier, we ran all of our benchmarks on a randomly generated "galaxy". We structured the galaxy around a central black hole to keep the simulation visually stable. Through experimentation, we chose to use 100 iterations of a 50,000 star simulation for our benchmarks, striking a good balance between the amount of work that could be sped up and the time needed to get results. We measured performance as the wall-clock time starting right after distributing the initial galaxy state and ending after the last iteration fully completes.

Our benchmarks also revealed the actual performance gap between the different campus computers.



Comparing the 4 different campus machines tested, the Hamerschlag computers have by far the fastest single-threaded performance, followed by GHC. The ECE machines are optimized for GPU-heavy workloads and do not have very good CPUs, and while

linux.andrew runs on a fairly powerful CPU, it is inside of a VM and has additional performance overhead for many common tasks.

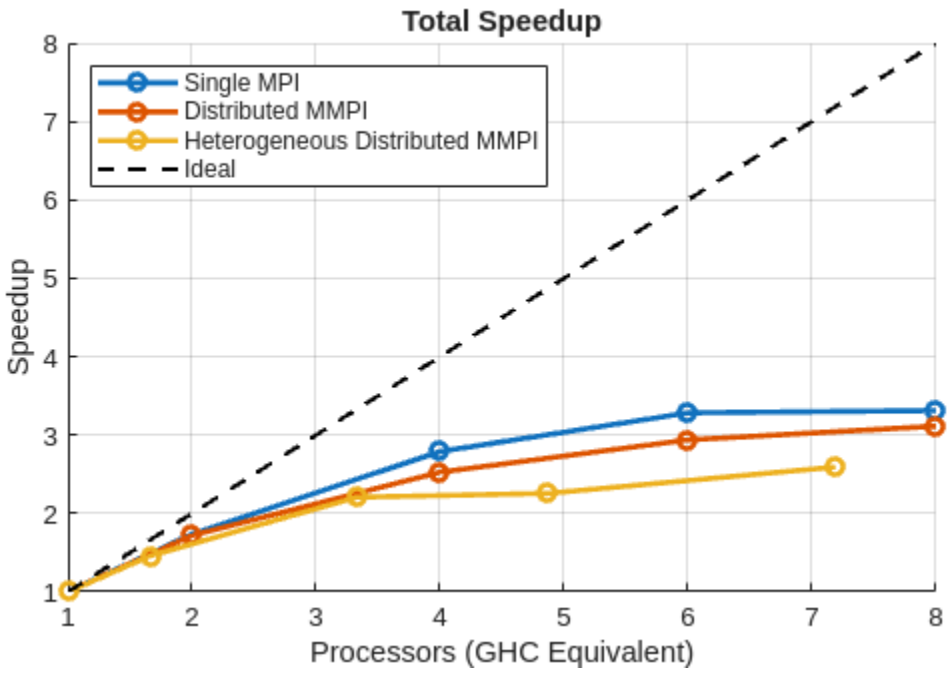
To get consistent data for comparing the heterogeneous environment versus the GHC-only environment, we ran configurations of computers that had similar GHC equivalent performance to their actual computer count:

- 1 GHC + 1 linux: 1.66 GHC equivalent on 2 machines
- 2 GHC + 2 linux: 3.23 GHC equivalent on 4 machines
- 3 GHC + 2 linux + 1 ECE: 4.87 GHC equivalent on 6 machines
- 4 GHC + 2 linux + 1 ECE + 1 HH: 7.19 GHC equivalent on 8 machines

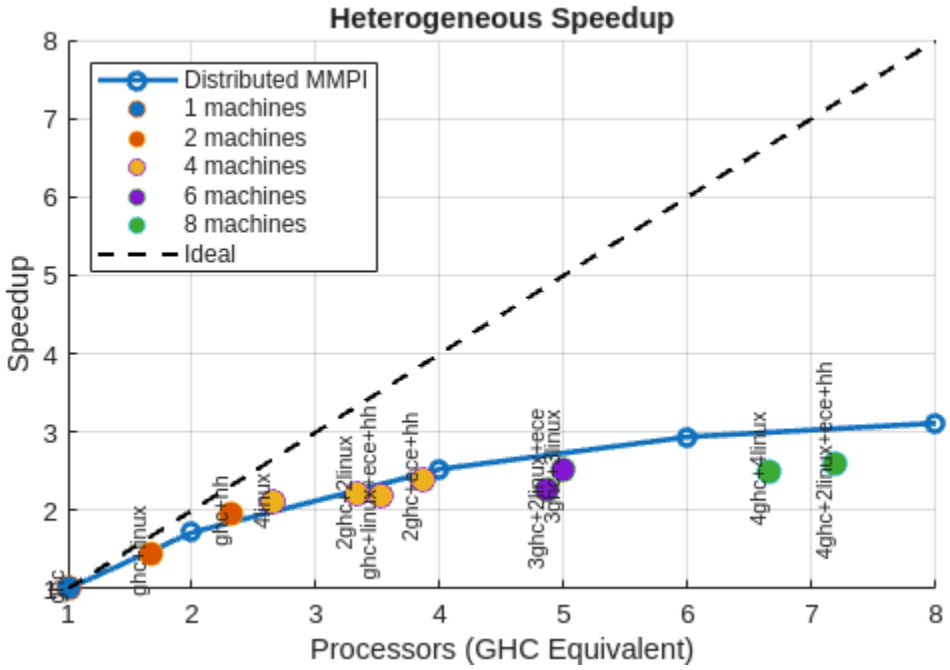
Using these configurations would allow us to best compare the efficiency of our load balancing algorithms between N GHC computers and N assorted computers with a similar equivalent performance.

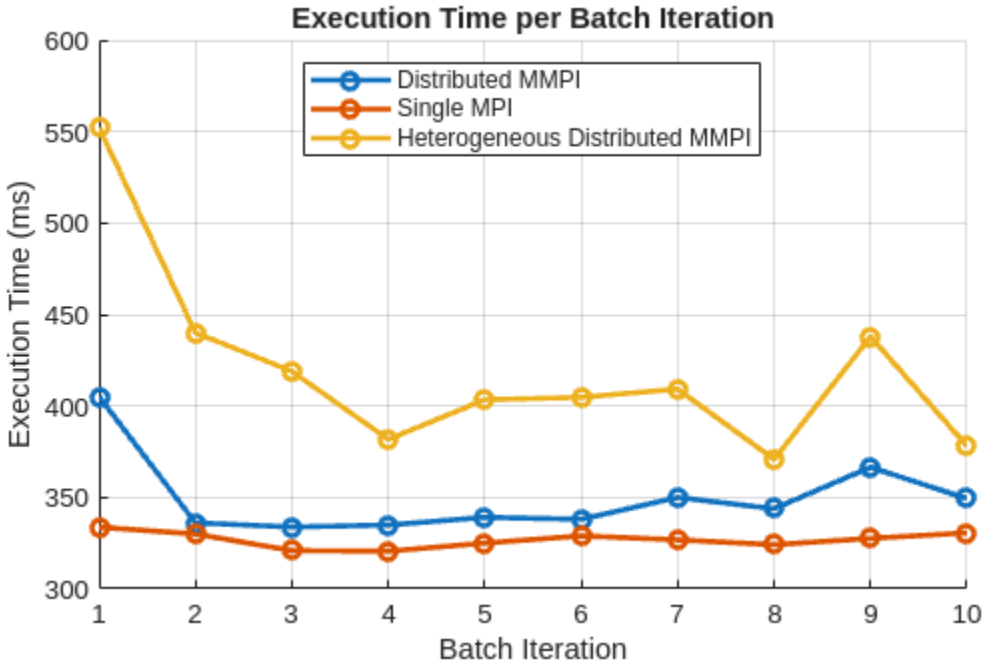
Ideally, we would like to have an even distribution of the assorted machines, but the HH computer network configuration prevented us from using more than one at a time. As linux.andrew and ECE have very slow CPUs, we increased the GHC computer ratio to maintain similar performance to the GHC-only benchmarks.

Measuring performance against single-threaded CPU code (with all of the compute-improving optimizations), and also against the single-machine MPI implementation, we obtained the following speedup results:

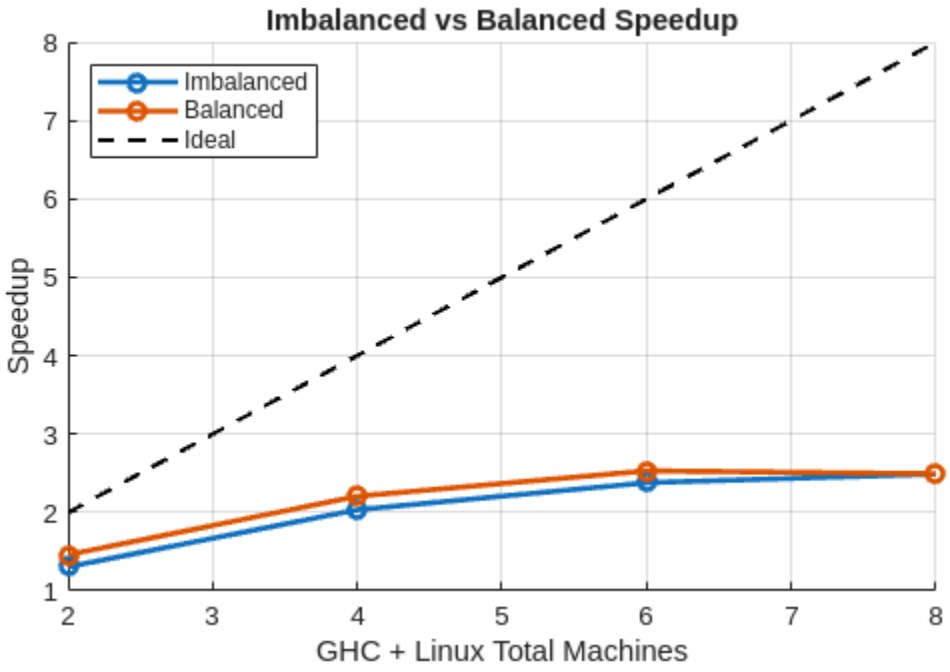


All three benchmark sweeps resulted in similar speedups. As expected, single-node MPI, with much lower communication costs, consistently outperforms distributed MMPI, which outperforms heterogeneous distributed MMPI. Although our load balancing is quite effective, the main contributor is the additional communication latency from the machines being physically farther apart, sometimes with communication times spiking to 4x as long as GHC-only.

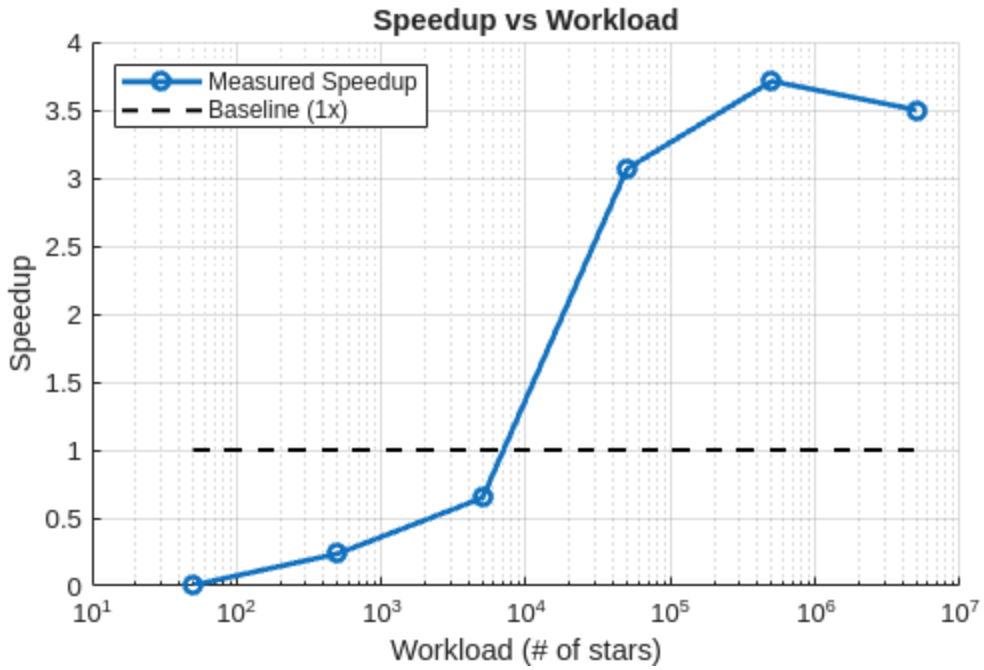




For a single-galaxy workload, our load-balancing algorithm converges fairly quickly, reaching a reasonably balanced work distribution by batch iteration 3 for the heterogeneous benchmark. Surprisingly, the GHC-only distributed benchmark also did not start out with good load balancing. The uneven performance may be from other users on the GHC machines, or from other factors like temperature that affect CPU performance.

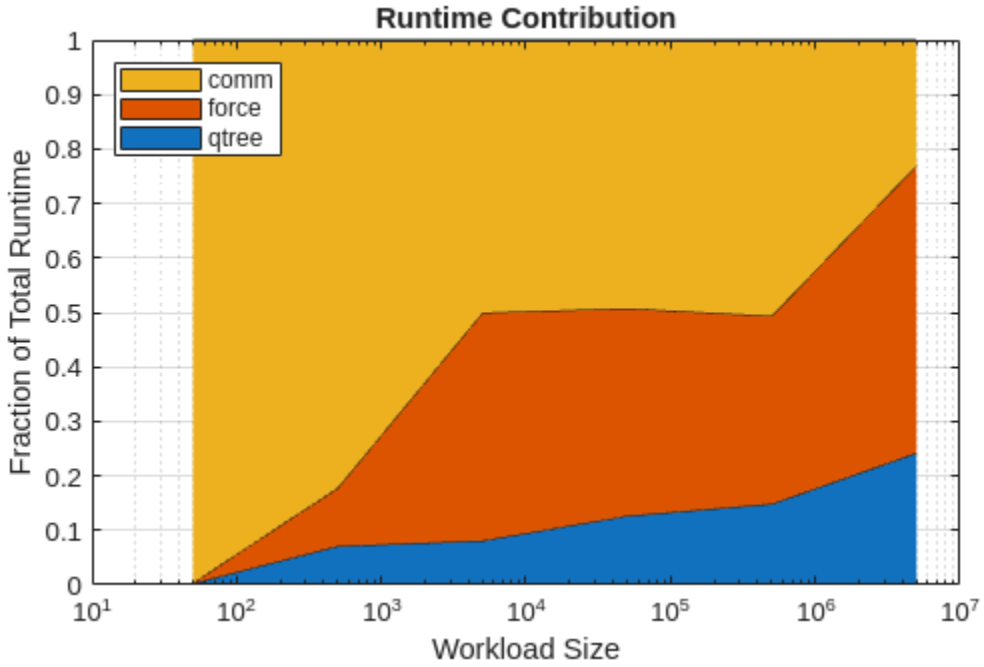


Load-balancing is also a general improvement over not load balancing. Running on various numbers of total GHC + linux.andrew (even numbers of each), the balanced implementation consistently has better performance than the imbalanced implementation, despite the extra communication costs of load balancing.



Measuring performance on different workloads on 8 GHC machines, we see a general upward trend as problem size increases. The most likely explanation is that while computation time scales as O(NlogN), communication time should only scale as O(N), as long as communication is bandwidth-bound.

Varying problem size on the 8-GHC configuration, we also gathered the following data:



In our benchmarks, communication is latency-bound below 5000 stars, and bandwidth-bound above. We are unsure of the exact cause of the sharp relative communication time decrease above 500,000 stars, but we speculate that it is from the computational working set no longer fitting in the cache, causing relative force/qtree times to drastically increase.

SPEEDUP ANALYSIS

Despite nearly matching (in speedup) the single-machine MPI implementation with our distributed MMPI implementation, neither was able to reach linear speedup. True parallel Barnes-Hut is an inherently communication-heavy workload. Since our optimizations targeted communication costs over a high-latency, low-bandwidth interconnect, many parallel computation speedups, such as parallel quadtree construction and parallel workload calculation, were infeasible.

Part of the reason the distributed and single-node speedups were so close was partially because our single-node implementation used the same communication and calculation pattern as the distributed implementation, despite it not being subject to the same high communication costs. Although our project did not target single-node MPI speedup specifically, we believe that much higher speedups would be possible in an environment with much lower communication costs.

Our target machine choice was intentionally suboptimal for Barnes-Hut, as our goal was to achieve meaningful speedup in a computing environment not well-suited for the target workload. Other environments, specifically a shared address space, non-message-passing parallel computing environment, may allow for drastically better speedup due to how they simplify many of our most expensive operations. The sync operation would be trivial to implement, as each thread would be able to just write into the same buffer. Additionally, building a pointer-based shared quadtree would no longer require expensive serialization and deserialization.

FUTURE WORK

Our project primarily targeted problem-constrained speedup scaling by using a distributed network of computers to run the same simulation faster. While we did achieve our goal, one of the primary benefits of distributed computing that we did not explore was the

opportunity for memory-constrained scaling. Distributed computing opens the possibility for much larger simulations than would be possible on a single machine, which may happen if a simulation has so many stars to simulate that a single machine would simply not have enough memory to store the state of the entire simulation.

We already ran into an upper bound on problem size, with memory allocation occasionally failing on the 5 million star workload. If we had designed for problem-constrained scaling, each node could get away with only storing a fraction of the simulation state, allowing for much bigger simulations.

REFERENCES

(1) Barnes-Hut Quadtree Image: https://www.reddit.com/r/IndieDev/comments/1gv3z51/the_barneshut_algorithm_on_a_quadtree/

(2) CUDA Dynamic Parallelism: https://docs.nvidia.com/cuda/cuda-programming-guide/04-special-topics/dynamic-parallelism.html

(3) MPI Dynamic Scheduling: https://arxiv.org/abs/1908.08082

(4) Distributed Barnes-Hut Algorithms: https://arxiv.org/abs/2203.08966

(5) Pipelined Allgather: https://web.cels.anl.gov/~thakur/papers/allgatherv-journal.pdf

WORK DISTRIBUTION

Zhuoyi Zou (50%):

- Boilerplate graphics code
- Initial serial/MPI Barnes-Hut implementations
- MiniMPI interface and implementation
- Timing-based load balancing

Vanessa Lam (50%):

- Parallel quadtree construction
- Morton ordering work distribution
- Prefix-sum and cost-based load balancing