15-418/618, Spring 2024 Final Project Report
Parallel Fast Multipole Method (FMM) On Shared Memory

Name: Nicole Feng (nvfeng), Marian Qian (marianq)

URL

https://github.com/marianqian/parallel-fmm

1 Summary

We analyzed and improved the parallelization of the Fast Multipole Method (to solve the N-body problem) using OpenMP for shared memory systems, making it comparable to parallelization on distributed memory systems. Our implementation achieves up to 7x speedup for 8-core GHC machines, and up to 83x speedup for 64-core PSC Bridges-2 machines. Our deliverables include graphs comparing speedup across different core counts, problem sizes, and particle distributions.

2 Background

The N-body problem is foundational in understanding the movement of celestial bodies, and is widely used in physics and mathematics for a diverse set of problems. In its purest mathematical form, the N-body problem comes down to an O(N^2) computation (see Figure 1) of all pairwise interactions between N particles, a complexity that becomes impossibly expensive when working on problem sizes that are large enough to be practical.

$$f(x_i) = \sum_{j=1}^{N} K(x_i, u_j) s(y_j), \; i = 1, \ldots, N,$$

Figure 2.1: N-body problem calculation.[1]

The Fast Multipole Method (FMM) brings the complexity down to a linear time O(Nlog(1/ε)^3) in 3 dimensions, where ε is the tolerance. It does this by:

1. Building an octree so that any leaf node (or octant) has approximately a certain number of particles.
2. Evaluating the sum using tree traversals:
    a. Performing a post-order (bottom-up) traversal involving the octant and its children — this upward pass computes multipole expansions.
    b. Performing a pre-order (top-down) traversal involving several octants in a neighborhood around the octant being visited — this downward pass converts the multipole expansions to inner expansions.
3. Output force and energy potential calculations of each particle interaction

The tree traversals that build the tree can benefit from parallelization since for each level, we can take advantage of data parallelism across the octants in that level. In the worst case, we have a O(log N) chain of dependencies that can't be parallelized. However, this is only the case when the octree is evenly distributed. For nonuniform particle distributions, where the octree is highly skewed with varying leaf levels, the parallel scalability and complexity analysis becomes more difficult.

Algorithm 1. $\{f_i\}_{i=1}^{N} = \text{FMM}\left(\{x_i, y_i, s_i\}_{i=1}^{N}, \text{octree}\right)$

```
// APPROXIMATE INTERACTIONS
// (1) S2U: source-to-up step
  $\forall \beta \in L$: source to up densities interaction
// (2) U2U: up-to-up step (upward)
  Postorder traversal of $T$
  $\forall \beta \in T$: interact $(\beta, P(\beta))$
// (3a) VLI : V-list step
  $\forall \beta \in T : \forall \alpha \in \mathcal{V}(\beta)$ interact $(\beta, \alpha)$ ;
// (3b) XLI : X-list step
  $\forall \beta \in T : \forall \alpha \in \mathcal{X}(\beta)$ : interact $(\beta, \alpha)$ ;
// (4) D2D: down-to-down step (downward)
  Preorder traversal of $T$
  $\forall \beta \in T$: interact $(\beta, \alpha)$ ;
// (5a) WLI: W-list step
  $\forall \beta \in L : \forall \alpha \in \mathcal{W}(\beta)$ : interact $(\beta, \alpha)$ ;
// (5b) D2T : down-to-targets step
  $\forall \beta \in L$: evaluate potential on $x_i \in \beta$;
//DIRECT INTERACTIONS
// ULI: U-list step (direct sum)
  $\forall \beta \in L : \forall \alpha \in \mathcal{U}_\beta$: interact $(\beta, \alpha)$ ;
```

Figure 2.2: Breakdown of FMM tree traversals[1]

Specifically, the M2L phase of the downward pass takes up the majority of the computation time as the algorithm traverses through every level and every node of that level, then sums up all of the multipole to local expansions of the current node's neighbors (or relevant other nodes that "interact" with the current node, whether that means they share a same parent or in the same level of the octree). The calculation of this expansion then has an additional two loops over the order of the expansion to get the final value. The M2L phase[2] has 94.5 times more computations than the M2M phase and L2L phase combined. The P2P phase also takes up a decent amount of time; however this can be run in parallel with the other stages (as shown by the other direction of the arrow). Therefore, the M2L phase in the downward traversal of the tree holds the best opportunity to be parallelized.

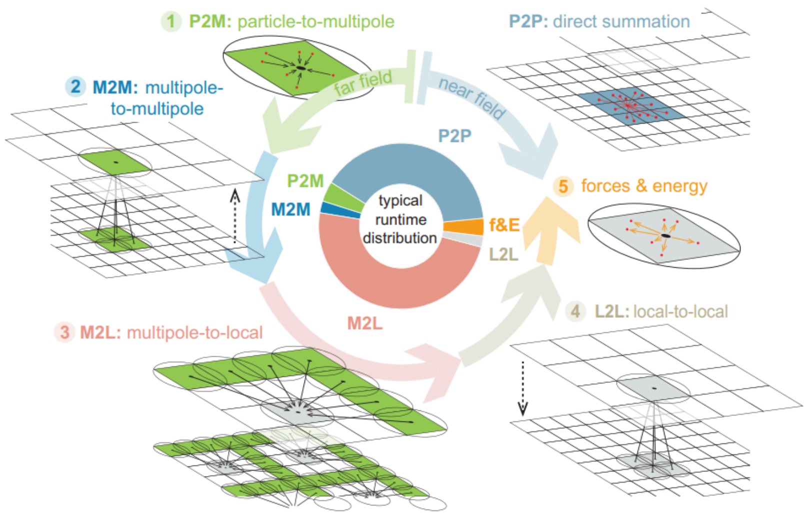
1 P2M: particle-to-multipole
P2P: direct summation
2 M2M: multipole-to-multipole
far field
near field
P2P
P2M
M2M
typical runtime distribution
f&E
L2L
M2L
5 forces & energy
4 L2L: local-to-local
3 M2L: multipole-to-local

Figure 2.3: Breakdown of time spent in the FMM algorithm.[2]

Data dependencies that exist within this algorithm mainly consist between stages of the algorithm that are listed in Figure 1. Each step numbered 1 to 5 must be executed one step after the other (approximate interactions: S2U → U2U → VLI + XLI → D2D → WLI + D2T) while steps that are labeled with a and b can be executed concurrently. This makes it harder to parallelize because each stage depends on the expansions that were computed in the previous stage, so there is no way to parallelize across stages. Another dependency exists between tree levels during the upward and downward traversals (computing multipole to multipole, multipole to local, and local to local expansions). In these individual stages, computation depends on the tree level before it or after it, which also makes parallelization difficult in this case. However, outside of the tree level dependencies, it is easier to find a way to parallelize the algorithm whether that is across all nodes within a level or parallelize computation of an expansion for a single node.

The purpose of the FMM is to construct a tree such that locality is improved in the calculation of the N-body problem. Because the octree exists, evaluating potentials and forces are much quicker as spatial locality is taken advantage of by using information that is within the octants near the given point we want to calculate the potential of.

The algorithm itself has poor temporal locality; every stage (multipole to multipole, multipole to local etc.) iterates through the same levels of the tree and often the same nodes within the tree. However, they are all spaced out across in time (as they are in different stages of the algorithm). The spatial locality within the stages is already pretty good because of how calculations are done node by node, and these calculations often use other nodes that are related to the given node (like being in the same level of having the same parents). Therefore, there is spatial locality in which nodes are accessed for these calculations.

Data-parallel exists when we want to use the already constructed tree to evaluate potentials and forces of many different points. This would simply involve probing the tree in parallel, which does not have any dependencies.

3 Approach

Our final approach involved using an existing FMM implementation that was parallelized with OpenMP (https://github.com/jrotheneder/FMM) and re-parallelizing the codebase with OpenMP to improve the speedup. The algorithm used parallelization with the for-loops during the upward and downward passes that were discussed in the previous section and also parallelized across all points in the tree when calculating the final potentials and forces at the very end of the algorithm.

More specifically, for the upward and downward passes, there was parallelization across nodes (per level). We specified a static work schedule so that each thread would have a set of nodes that were contiguous in memory together. Therefore, every level in the tree could be separated into chunks that were delegated to individual threads, and every level in the tree would invoke another set of threads to continue the upward/downward pass. The nodes themselves at the same depth were independent of each other when conducting these expansions which allowed the parallelization to be flexible when using OpenMP and a simple static scheduler.

Our original goal was to implement a parallel FMM for distributed memory using MPI due to its performance benefits over a shared memory model such as OpenMP, as noted in various papers[1]. We planned to implement a sequential FMM, make it suitable for distributed memory using MPI, then optimize intra-node parallelism using OpenMP and CUDA. However, we ran into some issues along the way, which is why we eventually ended up on this project of optimizing FMM with OpenMP. We discuss our previous iterations below.

Originally, we had planned to implement a sequential version of the FMM ourselves from scratch by rewriting existing Python code for the algorithm in C++. When we finally started to write our sequential code, we soon realized that it would take a lot of time and effort to write clean C++ code for implementing the FMM, when this was not even the main portion or intent of our project. We wanted to be able to focus on the real challenge, which was focusing on how to parallelize the FMM for distributed memory using MPI. Of course, parallelizing the FMM using OpenMP and CUDA would also be a significant portion of the project, but this parallelization is more straightforward than distributed memory parallelization due to the fairly obvious parallelizable portions of the algorithm. As such, we decided to pivot and use an existing C++ codebase that already implemented the FMM with OpenMP optimizations.

Then, we had planned to implement CUDA on this FMM repository but ran into complications with how to get CUDA to compile with the program and complications with the modularity of the repository (the FMM repository consisted of many classes with inheritance playing a large role in how the program ran). This made it difficult to implement in CUDA because we would have to either copy a large amount of data from the host CPU onto the device GPU and then transfer that information from the device back to the host. In addition, the modularity of the code made it harder to use the strategy of using an index of a CUDA thread to determine work scheduling (using thread index to determine which nodes and which parts of work to give to the current thread); this was harder because we could not access the specific node we wanted to due to the modular nature of the repository without copying the entire tree to GPU memory (which would have a significant amount of overhead). Because of these difficulties, we decided to find an alternative sequential code base that was more bare bones sequential code, so that it was easy to use OpenMP and CUDA to parallelize it.

Below is a detailed explanation of our plan, which consisted of changing the OpenMP program into CUDA by replacing the for-loops parallelized with OpenMP with a CUDA kernel. We referenced Kohnke et al.'s paper which discussed how to write the M2L (a stage in the downward pass) using CUDA. Below is the pseudo code for the two implementations in OpenMP and in CUDA.

OpenMP Pseudocode:

```
● Loop over levels of the tree (depth)
    ○ Loop over ith nodes at that depth in the tree
        ■ Loop over nodes in ith node's interaction list
            ● Loop over the order of expansion for this operation (aka number of
              coefficients that exist for this expansion)
                ○ Loop over order of expansion for this operation again
                    ■ Some multiplication and addition operations
```

CUDA Pseudocode:

```
● Loop over levels of the tree (depth) – loop over on CPU
    ○ Within the kernel:
        ■ Number of threads = order of the expansion
        ■ Number of blocks = number of nodes in level * number of max. nodes in
          interaction lists
        ■ Example CUDA code:
            ● int node = blockIdx.x;
            ● int interaction_node = blockDim.x;
            ● int coefficient = threadIdx.x;
        ■ Given coefficient of a node in an interaction for a node in specified level –
          calculate the coefficient (using a for-loop over the order of the expansion)
        ■ Sync – wait for all threads in block to finish (since all threads in block
          correspond to other coefficients for an interaction node)
        ■ One thread in block will add the interaction node coefficients to the node
          in the specified level
```

The way we structured the CUDA code was based on the fact that the nodes per level were independent of each other during this downward pass (and this was the same way the OpenMP parallelization did it, by parallelizing across nodes). Each node would then have an interaction list of nodes that it needs to compute the multipole to local expansion of, which would be summed up together to get the final value for that node. The way this code base is structured would require each node to add one interaction node at a time. Within these interaction nodes, there is a for-loop to calculate the coefficients for the expansion that is based on the order of the expansion. The data dependency mainly lies between the summation of the interaction list and the node, and also finishing calculating all of the coefficients of one node within the interaction list. We decided to dedicate one thread to calculate one coefficient as according to the paper because the coefficients are independent from other coefficients. One block would contain all the threads for all the coefficients of one interaction node, so we would add a synchronization for that specific block to see if the coefficients of the expansion of that specific interaction node was finished. Then, we would have one thread to add this interaction node to the general node at the specified level. The number of blocks we would have would be the number of nodes * number of nodes within each interaction list. This way, we would be able to maximize the parallelization of computing all of these expansions at the same time, with the main dependency of synchronization being with blocks.

Moving onto parallelizing the bare bones sequential code. We found the code written by Nakano, which was a short file (around 300 lines of code) that had 3 separate functions for the upward/downward passes. We parallelized the code with OpenMP by parallelizing across the nodes per level (same as the original FMM repository), but we did not get great results because the speedup was less than 1.

We spent a decent amount of time debugging and analyzing why our performance was not improving. We first looked at the cache misses of each program to see if we were slowing down the program by not utilizing the cache well (either poor temporal or poor spatial locality when distributing work). We found that the cache misses were about the same for 1 thread vs. 8 threads, with 1 thread having around 100 million cache misses and 8 threads having 101 million cache misses. Therefore, we eliminated the idea of false sharing and workload distribution with poor locality from the reasons why our program was slower. We then looked at the breakdown of the program to see where our time was going by using perf; we found that the majority of the time was spent on the downward pass which was expected (around 25%, which was the largest percentage out of functions that were relevant to the construction of the tree). We then manually timed this downward pass to see the amount of time computing each level would take. The code ran each level sequentially, and we parallelized across the nodes at each level, so OpenMP gave each thread some chunk of nodes to calculate per level. Figure 3.1 shows the timings at each level of the tree – we took the log of each time to help with visualization, therefore more negative values (longer bars) mean a shorter time. You can see that the 1 thread timings are consistently faster than the timings for 8 threads across all levels, and by a magnitude of 10 faster as well. Looking at the timings, we also realized that the runtime of the entire algorithm is very fast (for 64000 particles of a depth of 7 of the tree, it took around 1 second to run for both 1 thread and 8 threads). Because of these findings, we concluded that the additional overhead of the OpenMP pragmas were too long, and that the speedup we gain from parallelizing the code does not outweigh this overhead.

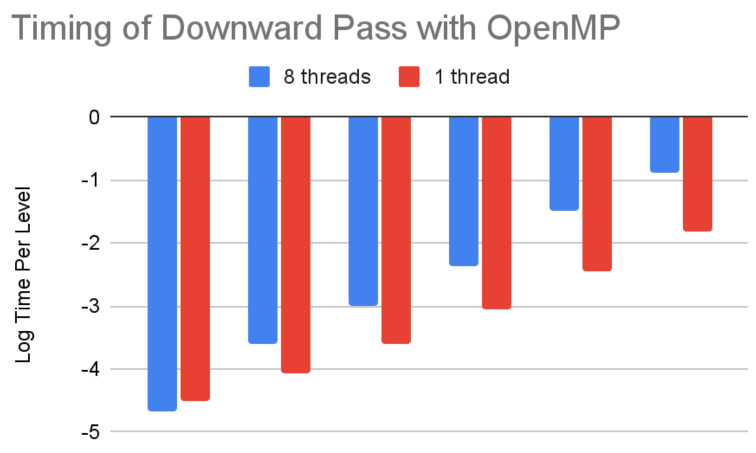

Figure 3.1: Timings of each level in the tree during downward pass when using OpenMP with 8 threads versus sequentially with 1 thread.

Because of the above complications we mentioned, and because we were limited on time, we decided to stick with the original FMM repository and improve the existing OpenMP implementation. We also conducted some experiments on this repository to see the effect of the number of points and whether the point distribution was uniform or non-uniform would impact the speedup, which we will discuss in the next section.

4 Results

Our project aim was to optimize the FMM algorithm using OpenMP. We measured performance by calculating the speedup achieved through our OpenMP implementation, using the total wall-clock time it took to build the FMM tree as the initialization time, and how long it took to calculate the potentials and forces for the particle interactions as the computation time. Additionally, we were particularly interested in the FMM tree building time and the speedup achieved through parallelization regarding this step since this was the portion of the N-body problem specifically related to the FMM that we were trying to optimize.

We defined 3 problem sizes: small (15,000 particles), medium (150,000 particles), large (1,500,000 particles); and 2 particle distributions: uniform and non-uniform. Thus, our experiments consisted of the following 6 inputs: [1] small non-uniform, [2] medium non-uniform, [3] large non-uniform, [4] small uniform, [5] medium uniform, [6] large uniform. These inputs were generated by editing the "fmm_tree.cpp" file in the tests directory.

```
size_t N = {15000,150000,1500000}      // # of particles
const size_t items_per_leaf = 2000     // max # of sources per leaf
const size_t d = 3                     // dimension of the particles
const double eps = 1E-3                // desired accuracy
#define adaptive false                 // adaptive for non-uniform,
                                             //     balanced for uniform
const bool uniform = true              // uniform or non-uniform
                                             //     particle distribution
const bool field_type = false          // true for gravitational,
                                             //     false for electric
```

Figure 4.1: Input parameters used for each test case. `N` was chosen to be one of `{15000,150000,1500000}` depending on the problem size (small, medium, large). `adaptive` was set to be `false` for uniform distributions and `true` for nonuniform distributions. `uniform` was set to be `true` for uniform distributions and `false` for nonuniform distributions.

For uniform particle distributions, we used a balanced FMM tree because the structure of the tree (i.e. octree) ensures that workload distribution will be more balanced due to each cell of the tree having approximately the same number of particles, and is especially conducive and beneficial to scenarios where the particle distribution remains relatively uniform. For non-uniform particle distributions, we used an adaptive FMM tree that adjusts its structure according to the particular distribution of the particles, which is hard to know and predict due to the non-uniform nature of the distribution.

The baseline we were using to measure speedup was the execution time for the same parallel OpenMP code but only running with 1 thread on the CPU (i.e. `OMP_NUM_THREADS=1`). All subsequent thread counts were still run on CPUs.

We defined 3 types of speedups as follows:

$$Total\ Speedup = \frac{Initialization\ Time\ for\ 1\ Processor + Computation\ Time\ for\ 1\ Processor}{Initialization\ Time\ for\ N\ Processors + Computation\ Time\ for\ N\ Processors}$$

$$Computational\ Speedup = \frac{Computation\ Time\ for\ 1\ Processor}{Computation\ Time\ for\ N\ Processors}$$

$$FMM\ Speedup = \frac{Initialization\ Time\ for\ 1\ Processor}{Initialization\ Time\ for\ N\ Processors}$$

Using the following definitions:
`Initialization Time` = time (s) for FMM tree to be built including performing upward and downard passes.
`Computation Time` = time (s) for the potential energies and forces of all the sources to be evaluated using the FMM.

From the graphs at the end of this section, we see that we were able to achieve significant, almost linear speedup on the GHC machines with our parallel FMM using OpenMP, which was one of our original goals (7x speedup for 8 cores is what Lashuk et. al. achieved in their paper). Additionally, we saw that the speedup we achieved for non-uniformly distributed particles was actually greater than that for uniformly distributed particles, which was also one of our original goals we set out to achieve by using MPI. However, we were able to achieve this same level of parallel efficiency and speedup using OpenMP only instead of MPI. This was surprising because we had read in Kurzak et. al's paper that the benefit of using distributed memory for FMMs was to be able to better deal with non-uniform particle distributions, which was a limiting factor to the gains that could be achieved from a straightforward OpenMP implementation. However, after reading Pan et. al's paper about optimizing OpenMP FMM implementations, we discovered that by restructuring loops, using guided schedules, and specifying static schedules for certain loops that calculated multipole expansions for nodes on the same level, we were able to get our OpenMP implementation to get the same performance gains as the MPI heterogeneous implementation of Lashuk et. al's paper.

Problem size is important for the N-body problem because this is where the complexity comes from. The more particles there are, the more interactions need to be evaluated. The purpose of algorithms like FMM was to make the N-body problem more efficient so that it could be run for problem sizes large enough to be of practical use without being impossibly expensive. This is why we chose 3 problem sizes {15000,150000,1500000} in order to see how well our parallel FMM scaled with the number of particles in the problem. Lashuk et. al. noted in their paper that they were able to achieve significant speedups for problem sizes of 22 million particles, but they had access to core counts greater than 24000, whereas our highest core count was 64 on the PSC machines, 128 when hyperthreading was enabled. Thus, we believe the problem sizes we chose are appropriate for the number of cores that were available to us, and we were able to perform sensitivity studies to problem size successfully.

We observed that our parallel FMM generally exhibited less speedup gains the greater the problem size was. For our GHC experiments, we see that we go from ~7x total speedup for the small and medium inputs run on 8 cores, to around ~6x total speedup for the large input run on 8 cores. Similarly, for the PSC experiments, we see that we go from 83x speedup for 128 threads on the [medium, non-uniform] input, to a speedup of 80x for 128 threads on the [large, non-uniform] input. The reason for a decrease in speedup for larger problem sizes lies in the fact that more particles means our FMM tree will have more levels (greater depth). The levels of the tree correspond to the sequential portion of our implementation, since there are dependencies between levels during the upward and downward passes of the tree traversals. The parallel portion of the code comes into play when we parallelize across nodes in the same level, not between levels. Thus, by Amdahl's law, even though we are parallelizing and optimizing the parallel (intra-level) portion of our code, we are limited in speedup gains by the sequential part (inter-levels), which is why we see performance plateaus and dips as the problem size increases.

For the PSC experiments, we noticed that we no longer saw linear speedup gains once we hit 128 threads. Up until that point, speedup was pretty linear, although slightly less than ideal at 64 threads. This is most likely due to the fact that the PSC machines only have 64 cores, so having 128 threads means that hyperthreading is enabled on the cores, meaning there are extra synchronization and communication costs due to multithreading that inhibit our speedup. Each core now has to multithread and manage 2 OpenMP threads, resulting in more context switching and overhead involved in managing the threads.

We used `perf` to get the execution breakdown of our program for 1 thread and 8 threads on the medium non-uniform input, and only for the portion of the program when the FMM tree is being built (including downward and upward passes). For 1 thread, we saw that 96.37% of our execution time was spent in the function `LocalExpansion`. This makes total sense because this function calls `constructLocalExpansion` and `multipoleToLocal`, which are pretty much the bulk of the FMM algorithm. As noted in the background section, the M2L (multipole to local) phase during the downward pass is a major time sink in terms of execution time, and the construction of this local expansion is what builds the tree. For 8 threads, we saw that 93.5% of our execution time was spent in the function `LocalExpansion`. However, 11.7% of that time was spent in `__start`, which indicates that there was a greater fraction of time setting up environments for threads to be able to enter the function. This makes sense because having more threads means there are more entry points to the function and more contexts that need to be set up, thereby increasing synchronization/communication overhead of having OpenMP threads. As noted by these observations, the performance bottleneck comes from the very expensive M2L phase. Improvement can be made on the existing OpenMP implementation by incorporating CUDA in order to accelerate this phase. While working on this project, this was something we attempted to achieve but ultimately were not able to, so it is something that can be improved about the current implementation we have.

Finally, our choice of machine target was sound since we were trying to optimize a parallel FMM for shared memory using OpenMP, which uses CPUs. Originally, we also aimed to use GPUs to accelerate the M2L step, making for a heterogeneous architecture solution. However, after shifting the focus of our project to be optimizing a shared address space solution using OpenMP, the CPU was the correct choice.

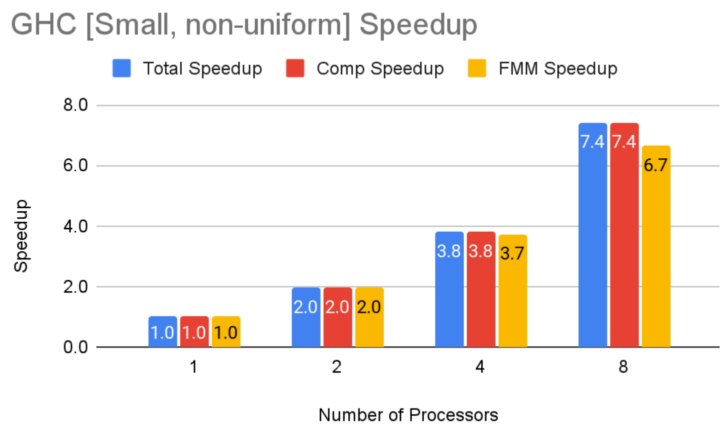

Figure 4.2: Speedup for [small, non-uniform] input, run on GHC machine 33. Parameters: N = 15000, eps = 0.001, Order is 23, non-uniform charge distribution, Orthtree is adaptive, height is 1, centered at 0.00147706, -0.000334211, 6.01815e-05, parent box size = 20.

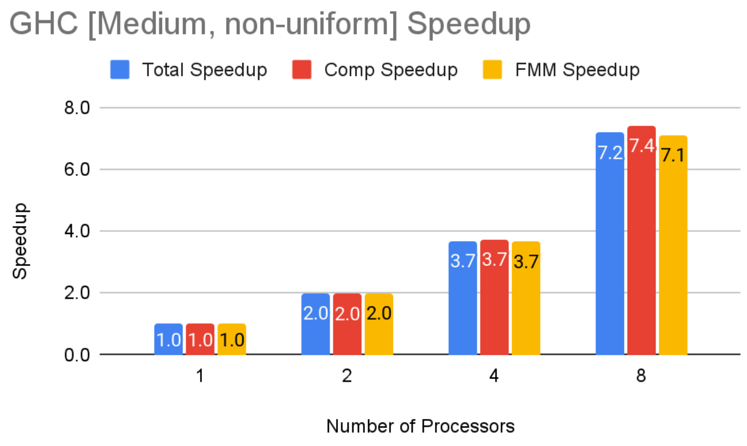

Figure 4.3: Speedup for [medium, non-uniform] input, run on GHC machine 33. Parameters: N = 150000, eps = 0.001, Order is 27, non-uniform charge distribution, Orthtree is adaptive, height is 4, centered at 5.67181e-05, -0.000315009, 3.62812e-05, parent box size = 20.0001.

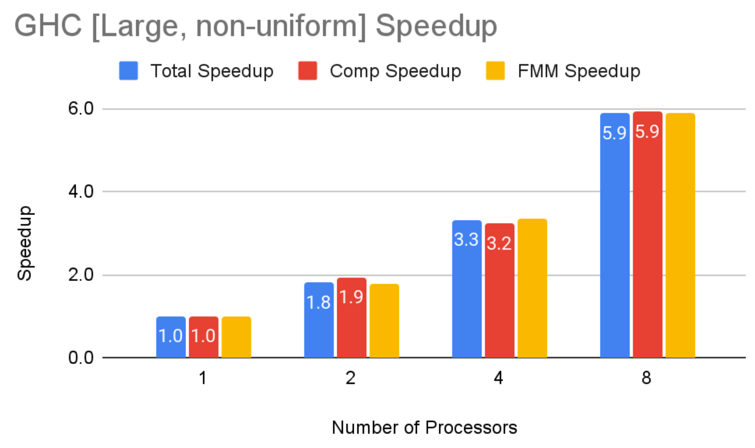

Figure 4.4: Speedup for [large, non-uniform] input, run on GHC machine 33. Parameters: N = 1500000, eps = 0.001, Order is 30, non-uniform charge distribution, Orthtree is adaptive, height is 5, centered at -1.47965e-06, -1.51272e-05, -1.68804e-06, parent box size = 20.0002.

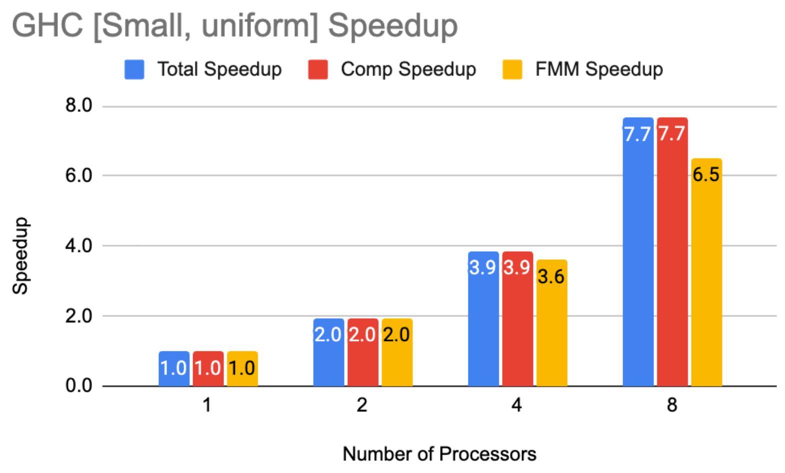

Figure 4.5: Speedup for [small, uniform] input, run on GHC machine 33. Parameters: N = 15000 eps = 0.001, Order is 23, uniform charge distribution, Orthtree is non-adaptive, height is 1, centered at -5.5519e-06, -1.97081e-05, -0.000105477, parent box size = 4.99999.

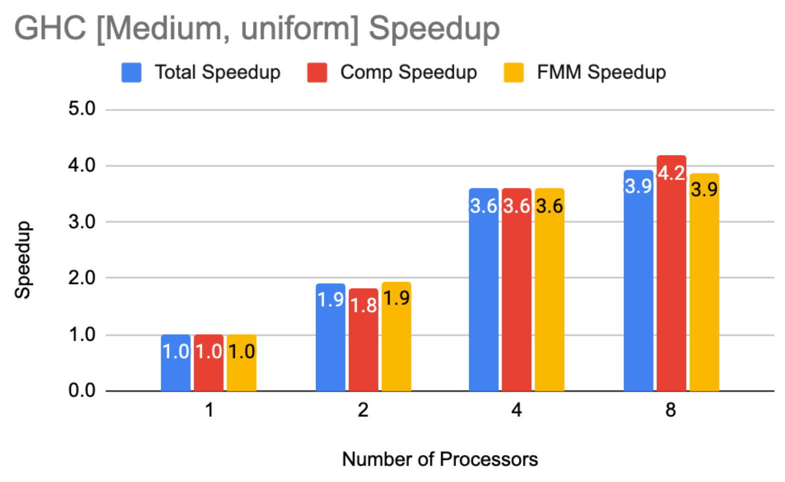

Figure 4.6: Speedup for [medium, uniform] input, run on GHC machine 33. Parameters: N = 150000, eps = 0.001, Order is 27, uniform charge distribution, Orthtree is non-adaptive, height is 3, centered at 3.91322e-05, -1.46894e-05, -6.51932e-08, parent box size = 5.00001.

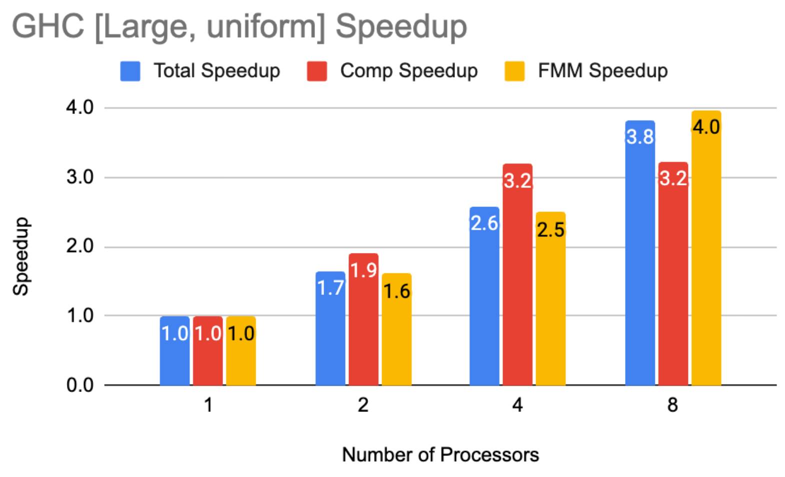

Figure 4.7: Speedup for [large, uniform] input, run on GHC machine 33. Parameters: N = 1500000, eps = 0.001, Order is 30, uniform charge distribution, Orthtree is non-adaptive height is 4, centered at -1.1793e-06, 2.60074e-06, 1.85568e-06, parent box size = 5.00004.

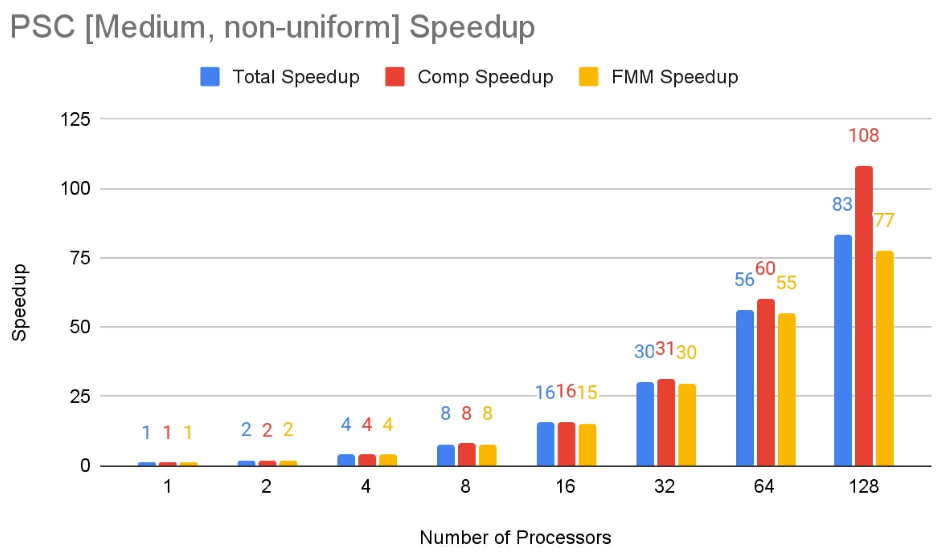

Figure 4.8: Speedup for [medium, non-uniform] input, run on PSC Bridges-2 Machine. Parameters: N = 150000, eps = 0.001, Order is 27, non-uniform charge distribution, Orthtree is adaptive, height is 4, centered at 5.67181e-05, -0.000315009, 3.62812e-05, parent box size = 20.0001.

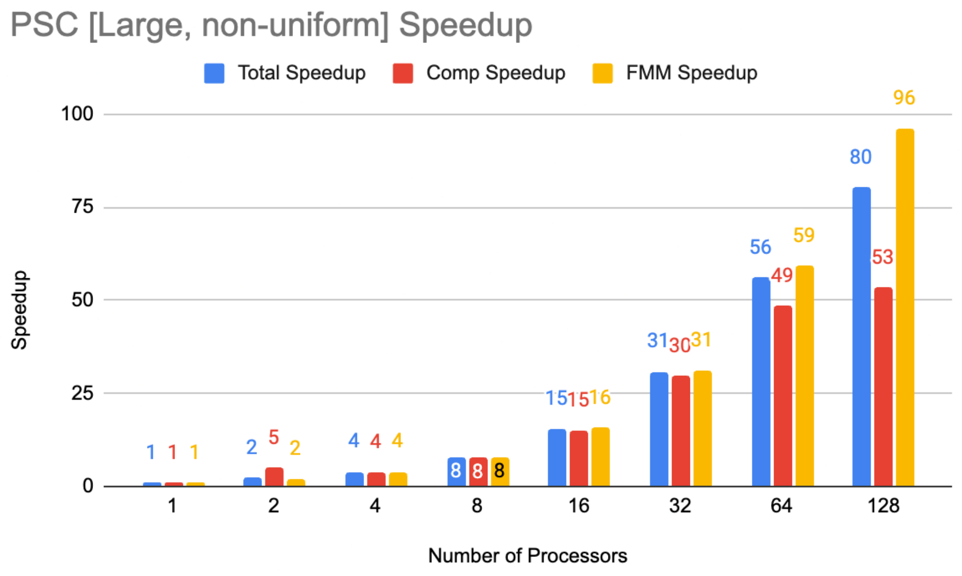

Figure 4.9: Speedup for [large, non-uniform] input, run on PSC Bridges-2 Machine. Parameters: N = 1500000, eps = 0.001, Order is 30, non-uniform charge distribution, Orthtree is adaptive, height is 5, centered at -1.47965e-06, -1.51272e-05, -1.68804e-06, parent box size = 20.0002.

After optimizing the OpenMP implementation further by specifying `static` schedules for previously non-specified `#pragma omp parallel for` constructs, we observed significant improvement in speedup for the uniform test cases. Prior to our optimizations, we noticed that non-uniform distributions actually had greater speedups than uniform distributions for our GHC experiments. For example, at 8 cores, the uniform distribution only achieved about 4x speedup for medium and large input sizes, while the non-uniform distribution was able to achieve about 7x speedup. However, after specifying a static schedule for the for loops that parallelized the multipole expansions, we saw comparable speedup (almost identical) for our GHC experiments between uniform and non-uniform. This performance increase is likely due to decreased synchronization overheads involved in dynamic or other non-static scheduling. For uniform particle distributions, we can statically assign these for loop iterations because we know that the workload resulting from it will likely be balanced due to the uniform distribution of the particles. However, on the PSC machines, we noticed that the speedup for 128 threads for the explicit static schedule implementation was actually worse than before. This is not ideal and is most likely due to the fact that hyperthreading is enabled on the PSC machines with 128 threads since there is a maximum of 64 cores. When hyperthreading is enabled, every time there is a context switch, there are cache misses that occur when the thread is accessing the nodes in the tree that were assigned to it through static scheduling (because previously the cache contained memory related to the other thread and other nodes assigned to that thread). However, with a dynamic schedule, when there is a context switch, there is a possibility that the memory in the cache is relevant to this new thread that is running, reducing the number of cache misses and therefore hyperthreading has less of a negative impact on the runtime of the algorithm. This is one possible reason for the lower speedup we saw for 128 threads on the PSC machines when we changed the scheduling to static from dynamic.

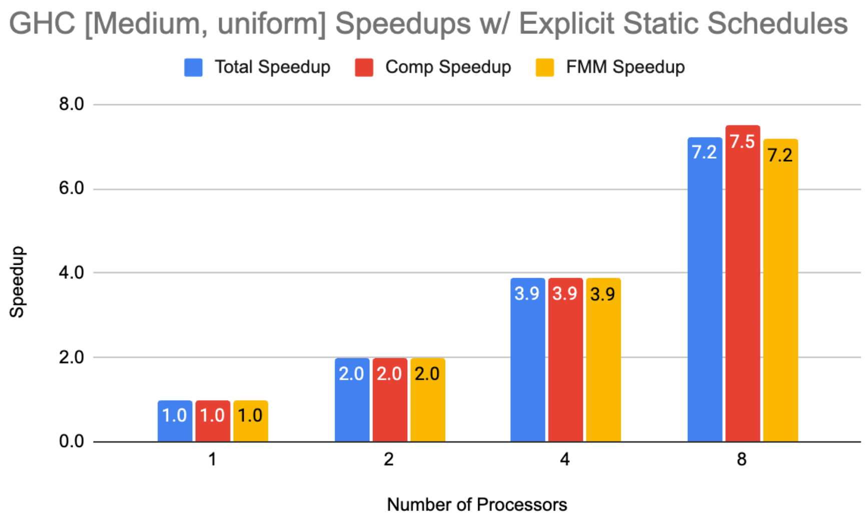

Figure 4.10: Speedup for [medium, uniform] input, run on GHC machine 33, making sure all previously unspecified `pragma for` schedules were made to be `static`.

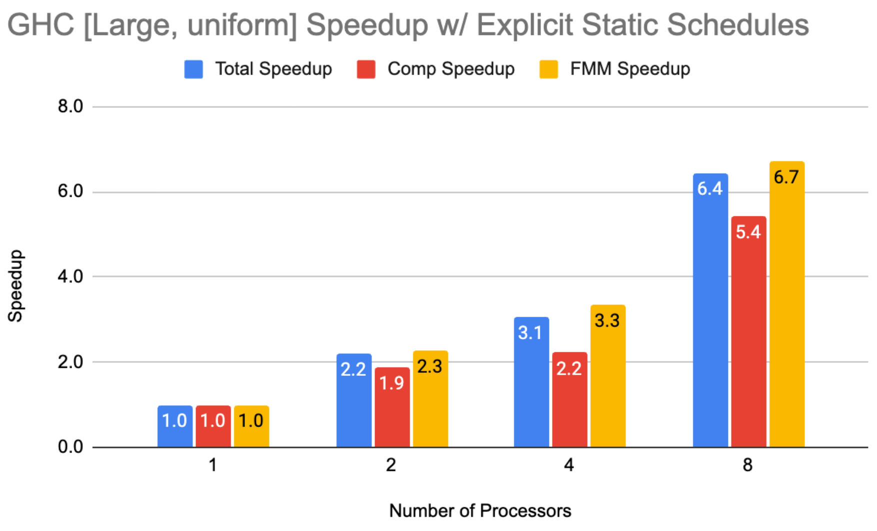

Figure 4.11: Speedup for [large, uniform] input, run on GHC machine 33, making sure all previously unspecified `pragma for` schedules were made to be `static`.

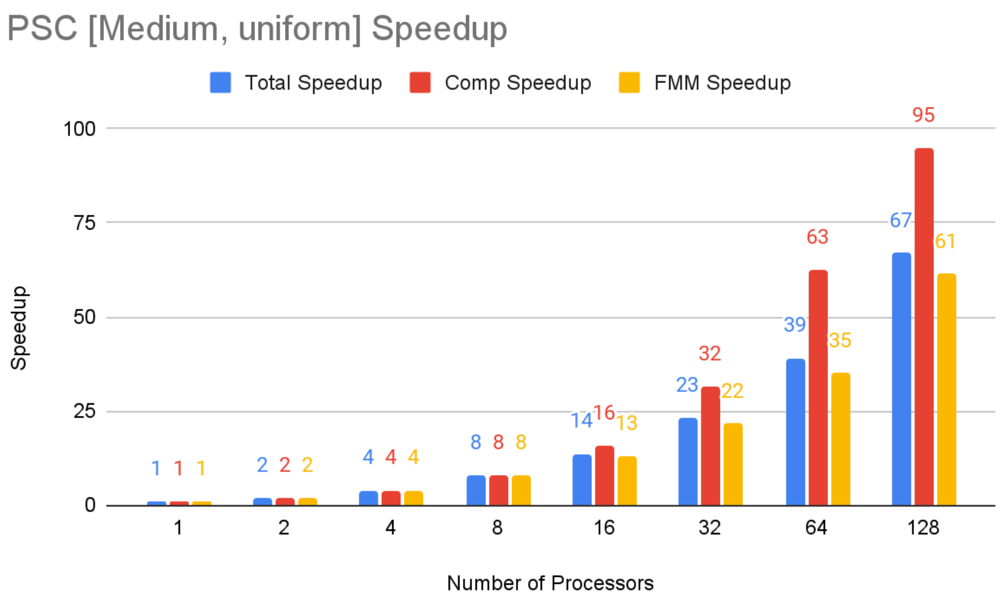

Figure 4.12: Speedup for [medium, uniform] input, run on PSC Bridges-2 Machine.

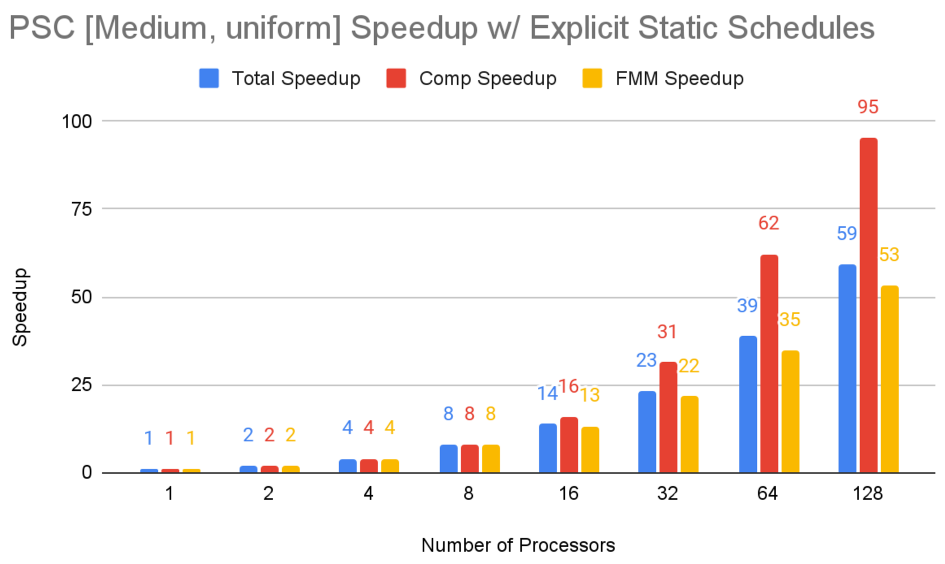

Figure 4.12: Speedup for [medium, uniform] input, run on PSC Bridges-2 machine, making sure all previously unspecified `pragma for` schedules were made to be `static`.

When trying to identify performance limiters on speedup, we used `perf` to analyze the cache misses of our program. We observed that in general, the cache misses varied wildly between core counts, with no concrete pattern or trend, especially on the GHC machines which came as a surprise to us. For the PSC machines, the cache misses increased slightly for higher thread counts which makes sense since there are more threads that are accessing memory and can create cache traffic. However, upon closer inspection, we saw that in any case, the per-thread cache misses went down significantly with higher core counts, especially when scaling up to high thread counts, which led us to believe that cache conflicts and false-sharing was most likely not the bottleneck on performance or speedup improvement.

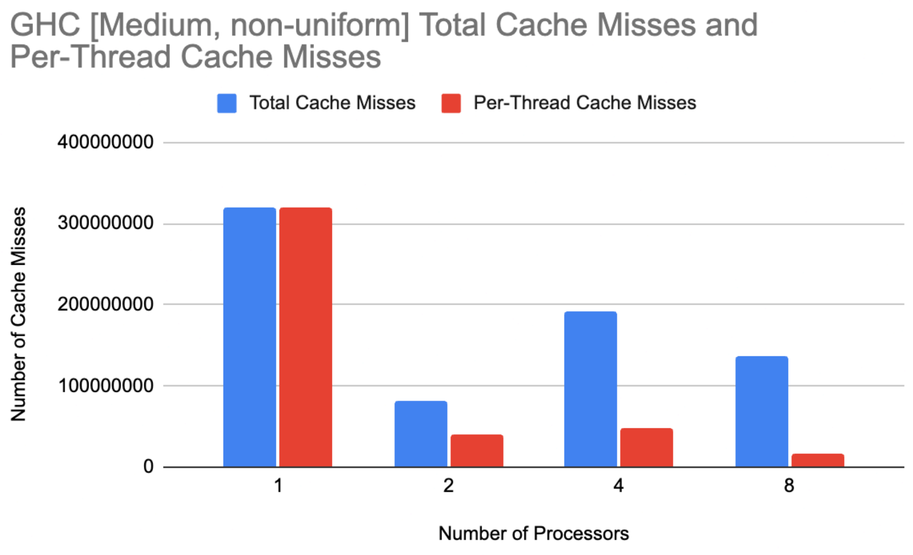

Figure 4.13: Total and per-thread cache misses for [medium, non-uniform], run on GHC machine 33.

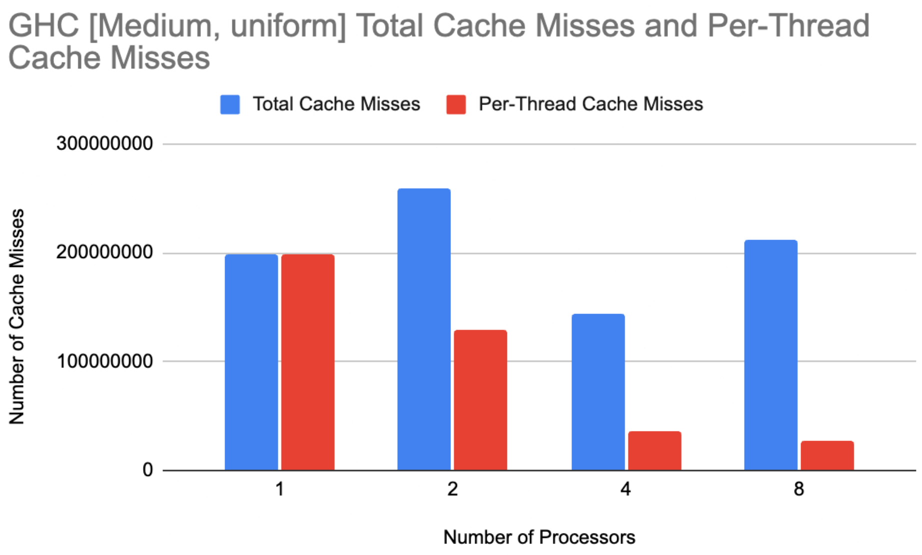

Figure 4.14: Total and per-thread cache misses for [medium, uniform], run on GHC machine 33.

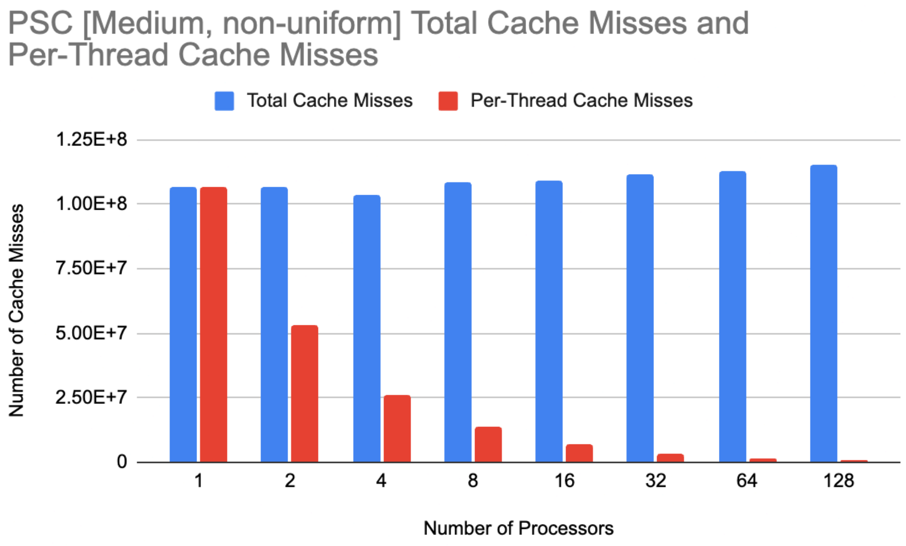

Figure 4.15: Total and per-thread cache misses for [medium, non-uniform], run on PSC Bridges-2 machine.

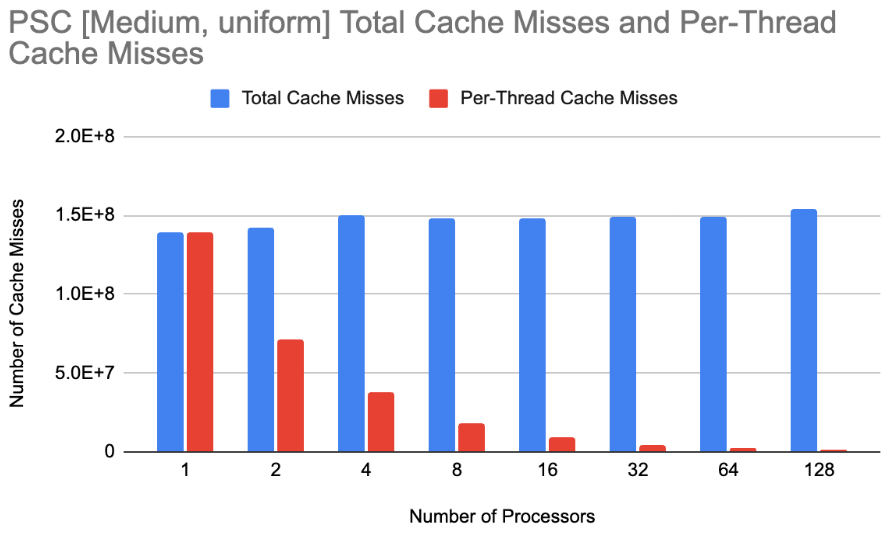

Figure 4.16: Total and per-thread cache misses for [medium, uniform], run on PSC Bridges-2 machine.

References

1. Ilya Lashuk, Aparna Chandramowlishwaran, Harper Langston, Tuan-Anh Nguyen, Rahul Sampath, Aashay Shringarpure, Richard Vuduc, Lexing Ying, Denis Zorin, and George Biros. 2012. A massively parallel adaptive fast multipole method on heterogeneous architectures. Commun. ACM 55, 5 (May 2012), 101–109. https://doi.org/10.1145/2160718.2160740
2. Kohnke B, Kutzner C, Beckmann A, et al. A CUDA fast multipole method with highly efficient M2L far field evaluation. The International Journal of High Performance Computing Applications. 2021;35(1):97-117. https://doi.org/10.1177/1094342020964857
3. Pan, Xiao-Min & Pi, Wei-Chao. (2011). On OpenMP parallelization of the multilevel fast multipole algorithm. Progress In Electromagnetics Research, 112, 199-213. https://doi.org/10.2528/PIER10120802
4. Kurzak, J., & Pettitt, B. M. (2005). Massively parallel implementation of a fast multipole method for distributed memory machines. Journal of Parallel and Distributed Computing, 65(7), 870-881. https://doi.org/10.1016/j.jpdc.2005.02.001
5. Kurzak, J., & Pettitt, B. M. (2006). Fast multipole methods for particle dynamics. Molecular Simulation, 32(10–11), 775–790. https://doi.org/10.1080/08927020600991161
6. Nakano, Aiichiro. Handwritten Notes on FMM Algorithm, aiichironakano.github.io/cs653/01-1FMM.pdf.
7. Nakano, Aiichiro. Slides on Fast Multipole Method, https://aiichironakano.github.io/cs653/01-1FMM.pdf.
8. Jrotheneder. (2024). Implementation of the fast multipole method (FMM) for the evaluation of potentials and force fields in gravitation/electrostatics problems. GitHub. https://github.com/jrotheneder/FMM.git

List of Work & Credit Distribution

Nicole Feng	Marian Qian
● Rewrote and optimized existing OpenMP implementation. ● Rewrote/reworked and debugged an existing sequential FMM implementation to be compatible with parallelization via OpenMP and CUDA. ● Parallelized sequential FMM using OpenMP; used profiling tools to find bottlenecks. ● Gathered data on GHC and PSC Machines for speedup graphs, cache miss study, and sensitivity studies. ● Performed analysis on best points to parallelize in FMM codebase.	● Attempted to optimize M2L step using CUDA (following a research paper) on existing sequential FMM implementations. ● Debugged sequential FMM parallelized with OpenMP, ran diagnostics using profiling tools to find bottlenecks. ● Attempted to parallelize existing sequential FMM using MPI. ● Gathered data on GHC and PSC Machines for speedup graphs, cache miss study, and sensitivity studies. ● Performed analysis on best points to parallelize in FMM codebase.

We feel that we have equally contributed to this project, with each of us reading copious amounts of material in order to attempt different parallelization strategies, even if we were not always successful, and eventually successfully optimizing our final implementation together. Thus, we want to split the credit 50% 50%.