15-418/618, Spring 2024 Final Project Proposal

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

1 Title

Parallel Fast Multipole Method (FMM) on Distributed Memory

2 URL

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

3 Summary

We are going to implement a parallelized Fast Multipole Method on distributed machines, using MPI for distributed memory parallelism and utilizing OpenMP and CUDA acceleration for shared memory parallelism. We will run our implementation on the NVIDIA GPUs in the lab and the PSC GPU Nodes to test the scalability and performance of our solution, and analyze the effectiveness of our algorithm on different problem sizes and particle distributions.

4 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 1: N-body problem calculation.

The Fast Multipole Method 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.

The tree traversals 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. Some techniques to work against the nonuniform distribution of the tree involve using local essential trees and a parallel tree construction algorithm. Previous work also involved building a communication graph to represent the octree; each node represented a tree octant and edges represented the interactions between octants. These communication graphs are bounded by O(N^⅔ (log N)^4/3), and with a good sorting algorithm such as parallel radix sort to distribute the particles, the overall algorithm scales in O(N/p) + O(log p), where p is the number of processors (which is the same overall scalable complexity as if we had uniform particle distributions). These previous works show that it is theoretically possible to parallelize FMM for any distribution (both uniform and nonuniform distributions).

5 The Challenge

FMM is hard to parallelize efficiently on distributed memory computers because of irregular patterns (especially when there is a nonuniform distribution of particles) and the high volume of communications.

What We Hope to Learn:
From this project, we hope to learn how to integrate different types of parallelism (across distributed memory and within shared memory) to speed up FMM calculations across the octree (so during the bottom-up and top-down traversals) and within octants of the octree (such as constructing the interactions within each leaf octant).

Workload:
The tree traversals means that we access the data in a way that's not necessarily optimized for cache locality, and can seem irregular from the particles' partitions.

When particles interact, that computation needs to be passed through different portions of the octree, which can also mean it needs to be passed between different memories since the particles would be partitioned over distributed memory. Thus, there is a lot of communication that needs to happen when executing on distributed memory computers, which can limit gains from parallelization due to this overhead.

In addition, if we are to write FMM that is scalable for nonuniform particle distributions, we also need to find a way to partition the particles so that they are balanced within the octree.

Constraints:
As mentioned previously, the structure of an octree, especially if the octree is uneven, makes the FMM difficult to parallelize. Dependencies within the octree for certain computations means that our implementation would either have to efficiently communicate information between processors or balance work in a way that removes these dependencies between processors.

6 Resources

Computing Resources:

- We will use GHC machines for testing and debugging serial, and parallelized versions of our algorithm (OpenMP and MPI).
- We will use GHC machines (NVIDIA GeForce RTX 2080 B GPUs) for when we incorporate CUDA parallelization into our algorithm.
- We will use PSC machines with GPU nodes for testing the scalability of our parallelized implementation (OpenMP, MPI, CUDA).

Algorithmic Resources:

- 2006 context paper:
  - https://www.tandfonline.com/doi/full/10.1080/08927020600991161
  - J. Kurzak and B. M. Pettitt. “Fast multipole methods for particle dynamics”. Mol Simul. 2006 ; 32(10-11): 775–790. doi:10.1080/08927020600991161.
- 2005 paper talking about how to parallelize in distributed memory:
  - https://www.sciencedirect.com/science/article/pii/S0743731505000249
  - “Massively parallel implementation of a fast multipole method for distributed memory machines”
- 2012 paper with combining parallel models (MPI, OpenMP, GPU):
  - https://cims.nyu.edu/gcl/papers/lashuk2013mpa.pdf
  - “A Massively Parallel Adaptive Fast Multipole Method on Heterogeneous Architectures”

Codebase:

- FMM Python Implementation for 2D Coulomb Particles: https://github.com/lbluque/fmm
- FMM Tutorial for Trees: https://github.com/barbagroup/FMM_tutorial

We plan to use the repos listed under the “Codebase” to help us implement a sequential C++ FMM, and then use the research papers to guide us in our parallelization efforts.

7 Goals and Deliverables

Plan to Achieve:

- Correct implementation of FMM in C++, parallelized using OpenMP and CUDA to exploit data parallelism within a shared memory.
- Correct implementation of FMM in C++, parallelized using MPI to make the implementation adaptable and scalable across distributed machines/memory.
- We aim to have our parallel implementations at least exhibit some sort of speedup compared to the sequential implementation, perhaps almost linear speedup for up to 8 cores (7x speedup for 8 cores) — as this is what they achieved in the paper.
- We aim to have our MPI version perform better (speedup) on non-uniformly distributed particle data compared to the implementation that just has parallelism in the shared memory space (CUDA, OpenMP).

Hope to Achieve:

- We hope to achieve an extremely scalable solution that has good speedup on larger core counts (up to 128), maybe around 64x speedup (just extrapolating from the speedup numbers in the paper).

Poster Session Demo:

- For the poster session demo, we plan to show our speedup graphs — one for a shared memory parallelization implementation (OpenMP and CUDA), and one for a combined shared and distributed memory approach (MPI, OpenMP, and CUDA).
- We will also show speedup graphs for our 2 implementations on different problem sizes and particle distributions, so we can demonstrate the benefits of a heterogeneous architecture (distributed memory) in FMMs with uneven particle distributions.

8 Platform Choice

We plan to implement our algorithm in C++ so that we can easily rewrite it using CUDA for GPU parallelization, as well as implement MPI and OpenMP calls like we've done in previous class assignments.

The GHC machines with the NVIDIA GPUs are a good platform to test our code on because they can handle CUDA, MPI, and OpenMP for small problem sizes that will allow us to debug for correctness and perform basic analysis on synchronization costs and how well our solution scales.

Since FMM is necessary due to the large problem sizes it is used on in practice, using the PSC machines and PSC GPU nodes makes sense since we will be able to see how well our solution scales on larger problem sizes.

9 Schedule

Due Date	Task(s)
Sat, 4/6	● Finish implementing a correct, sequential version of FMM using C++. ● Outline a plan to use CUDA and OpenMP to take advantage of data parallelism within tree levels.
Wed, 4/13	● Finish implementing shared memory parallelism (OpenMP and CUDA acceleration). ● Record performance metrics on different problem sizes and particle distributions. ● Outline a plan to use MPI to implement distributed memory parallelism.
Tues, 4/16	Milestone Report DUE
Sat, 4/20	● Finish incorporating MPI parallelization into implementation. ● Record performance metrics on different problem sizes and particle distributions. ● If speedup is not ideal, do performance debugging to identify bottlenecks.
Sat, 4/27	● Finish iterating on design. ● Collect and record performance metrics on PSC machines to see if the solution is scalable. If not, do performance debugging to identify bottlenecks.
Sat, 5/4	● Finish collecting all finalized performance metrics for the report. ● Finish report.
Sun, 5/5	Final Report DUE
Mon, 5/6	Poster Session DUE