DISSERTATION: NOVEL PARALLEL ALGORITHMS AND PERFORMANCE OPTIMIZATION TECHNIQUES FOR THE MULTI-LEVEL FAST MULTIPOLE ALGORITHM

By

Michael Lingg

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

Computer Science – Doctor of Philosophy

2020

ABSTRACT

DISSERTATION: NOVEL PARALLEL ALGORITHMS AND PERFORMANCE OPTIMIZATION TECHNIQUES FOR THE MULTI-LEVEL FAST MULTIPOLE ALGORITHM

By

Michael Lingg

Since Sir Issac Newton determined that characterizing orbits of celestial objects required considering the gravitational interactions among all bodies in the system, the N-Body problem has been a very important tool in physics simulations. Expanding on the early use of the classical N-Body problem for gravitational simulations, the method has proven invaluable in fluid dynamics, molecular simulations and data analytics. The extension of the classical N-Body problem to solve the Helmholtz equation for groups of particles with oscillatory interactions has allowed for simulations that assist in antenna design, radar cross section prediction, reduction of engine noise, and medical devices that utilize sound waves, to name a sample of possible applications. While N-Body simulations are extremely valuable, the computational cost of directly evaluating interactions among all pairs grows quadratically with the number of particles, rendering large scale simulations infeasible even on the most powerful supercomputers. The Fast Multipole Method (FMM) and the broader class of tree algorithms that it belongs to have significantly reduced the computational complexity of N-body simulations, while providing controllable accuracy guarantees. While FMM provided a significant boost, N-body problems tackled by scientists and engineers continue to grow larger in size, necessitating the development of efficient parallel algorithms and implementations to run on supercomputers. The Laplace variant of FMM, which is used to treat the classical N-body problem, has been extensively researched and optimized to the extent that Laplace FMM codes can scale to tens of thousands of processors for simulations involving over trillion particles. In contrast, the Multi-Level Fast Multipole Algorithm (MLFMA), which is aimed for the Helmholtz kernel variant of FMM, lags significantly behind in efficiency and scaling. The added complexity of an oscillatory potential results in much more intricate data dependency patterns and load

balancing requirements among parallel processes, making algorithms and optimizations developed for Laplace FMM mostly ineffective for MLFMA. In this thesis, we propose novel parallel algorithms and performance optimization techniques to improve the performance of MLFMA on modern computer architectures. Proposed algorithms and performance optimizations range from efficient leveraging of the memory hierarchy on multi-core processors to an investigation of the benefits of the emerging concept of task parallelism for MLFMA, and to significant reductions of communication overheads and load imbalances in large scale computations. Parallel algorithms for distributed memory parallel MLFMA are also accompanied by detailed complexity analyses and performance models. We describe efficient implementations of all proposed algorithms and optimization techniques, and analyze their impact in detail. In particular, we show that our work yields significant speedups and much improved scalability compared to existing methods for MLFMA in large geometries designed to test the range of the problem space, as well as in real world problems.

ACKNOWLEDGEMENTS

I would like to thank all of the staff at Grand Valley State University for starting me on this journey, and the staff of Michigan State University for guiding me to its present conclusion. Drs Greg Wolffe, Paul Jorgensen, Jonathan Engelsma and Christian Trefftz at GVSU were particularly instrumental in giving me the confidence to learn new skills above and beyond the ones they thought.

At MSU, Dr Metin Aktulga was tireless in his efforts being my primary advisor while creating this PhD thesis. Dr Aktulga was always available to discuss any question regarding any of the work and provided very calm and measured guidance. The entire process felt more like a series of minor steps, rather than major hurdles to overcome. Dr Shanker Balasubramaniam provided an excellent example of what an expert in the field should strive for. His depth of knowledge seemed endless, yet he could make the most technical of topics make sense.

Dr Stephen Hughey laid the groundwork for a majority of my research at MSU. After completing his PhD, Dr Hughey continued to provide his knowledge for continued research. Of particular values was his explanations of not only how the program worked but how the parts of the program and the theory behind the algorithm fit together.

Finally I would not be here without my wife Sara's support. While putting up with the long hours, she made sure I could do what needed to be done, and stood by me over four long years.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

CHAPTER 1

INTRODUCTION

Simulations of many many physics phenomena can be broken down into the interaction between two or more bodies. In the 17th century, Sir Issac Newton discovered that knowing the trajectory of a planet or moon was not sufficient to determine the orbit of the body. He found that the gravitational interaction of each body with all other bodies in the system had to be considered. Newton's discovery is one of the earliest known definitions of the N-Body problem. An N-Body simulation might range from gravitational interaction between two small bodies at the most basic, to simulations of interactions between a trillion separate particles Potter et al. (2017). From the early use of the N-Body problem to compute movement of bodies in the solar system, use of the N-Body problem has extended to design and development of many modern technologies. A small subset of technologies includes sonar and radar distance measuring equipment, medical imaging technologies, and wireless communication.

One of the earliest N-Body simulations was performed by Erik Holmberg using lamps and photo cells to simulate gravity interactions between stars. In 1956, Sebastian von Hoerner began developing computer simulations for gravitational interactions, von Hoerner (2001). Typical gravitational simulations solve Laplace's equation for N-Body interactions, Greengard & Rokhlin (1987). This case computes the pairwise interactions of all particles in the form:

$$\Phi(x_j) = \sum_{\substack{i=1 \\ i \neq j}}^{n} \frac{q_i}{|x_j - x_i|}. \tag{1.1}$$

Beyond gravitational interaction, applications of Eq. 1.1 exist in the field of fluid dynamics (Salmon & Warren (1994)), molecular simulation (Shimada et al. (1994)) and atomic simulations (Ding et al. (1992)), among many others. Another class of N-Body problems uses the Helmholtz equation to explore oscillatory interactions between particles, Greengard (1988). For the Helmholtz N-Body problem, Vikram & Shanker (2009), pairwise interactions of all particles are computed in

the form:

$$\Phi(r) \doteq \sum_{\substack{i=1 \\ i \neq j}}^{n} g(r - r_i) u_i, \tag{1.2}$$

where g is Green's function for the Helmholtz equation:

$$g(r) = \frac{e^{-jk|r|}}{4\pi|r|}. \tag{1.3}$$

In Eq. 1.3, $k = 2\pi/\lambda$ is the wave number in rad/m and λ is the wavelength in meters.

The need to track the waveform of the oscillatory interactions adds additional complexity to the Helmholtz N-Body problem, that is not found in Laplace kernels, where only the magnitude needs to be considered. The impact of this added complexity will be discussed in detail at the end of section 1.2. Some applications of electromagnetic wave modelling include microwave medical devices, radar wave prediction, and studying how antennas interact with their environment, for example a small antenna on a large naval vessel made of metal. Uses of Helmholtz kernels are not limited to electromagnetic fields, but are applicable to any N-Body problem dealing with oscillatory interactions. Acoustic applications can range from medical imaging, Huttunen et al. (2005), to modelling sound waves generated in automotive, Chaigne et al. (2007), or aviation, LóPez-PortugéS et al. (2012), and other applications, Ergul (2011). Potential applications can range as far as predicting how light interacts with certain materials as investigated by Burresi et al. (2014).

Within the N-Body simulation, more bodies allows for a larger and more complex scenario, or can be used to increase the precision of the simulation. In gravity simulations, for example, simulating large galaxies by only simulating larger stars, or clusters of stars, allows for simulations that run in less time, but loses some of the complex interactions of a larger simulation. Simulating each star in the entire Milky Way galaxy would require 100 billion particles, and the Andromeda Galaxy would require 1 trillion particles, requiring much more computational power to simulate in a reasonable amount of time. Helmholtz simulations have similar considerations that lead to a large

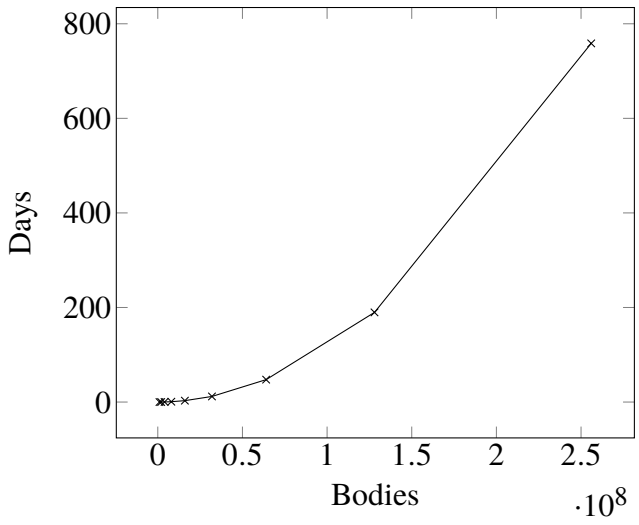

Figure 1.1: n^2 interaction scaling, assuming 1 billion interactions per second.

number of particles interacting. Precisely simulating small features on large wavelength surfaces, such as the previously mentioned antenna on a naval vessel, requires small wavelengths to precisely model the antenna, but ship itself is very large in terms of the same wavelengths. In addition to the potential problem space being very large, the $O(N_S^2)$ cost of a complete simulation, where N_S is the number of bodies, or degrees of freedom, means the computation cost increases quadratically with problem size. Figure 1.1 shows how computing n^2 interactions, even at an assumed 1 billion interactions computed per second, quickly increases from a little over 16 minutes to compute 1 million particles, to over two years to compute 256 million interactions.

To reduce the amount of computation necessary, an approximation can be computed for groups of bodies, and interactions can be computed between these approximations. This approach is very similar to how hub and spoke communication can be used to reduce the number of connections required vs point to point communication, as shown in figure 1.2.

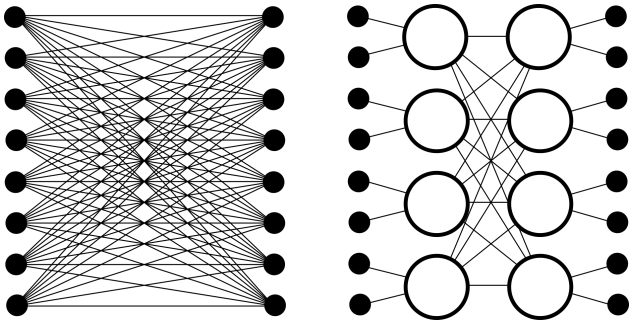

Figure 1.2: Point to point vs hub interactions.

1.1 Building a Tree Algorithm

Farfield vs Nearfield The primary method of producing group interactions for N-Body solution has been to separate the interactions into nearfield and farfield interactions. As shown by Appel (1985), interactions between any group of particles that is well separated from another group can be computed between approximations of the groups with controllable accuracy, Dembart & Yip (1998). While Appel (1985) used monopole approximations, since Greengard & Rokhlin (1987), the approximations are typically created using multipole expansions, thus the common name for the algorithm, the Fast Multipole Method (FMM). The multipole expansion is a mathematical series created by sampling the outgoing radiation pattern of all particles contained within a unit sphere, Engheta et al. (1985). Interactions using these multipole expansions provide controllable accuracy based on the number of multipole samples, more samples means more accuracy, but at an increased computational cost. The result of the group interaction is then distributed to the particles within the group, while bodies within each group still interact with each other directly. The top of figure 1.3 illustrates how all bodies would interact with a single body in a standard N-Body solution. The middle of figure 1.3 illustrates the farfield interaction process:

1. The group of source particles (on the left) is approximated.

2. The source group interacts with the observer group (on the right).

3. The observer group distributes the farfield interaction to the particles in the group.

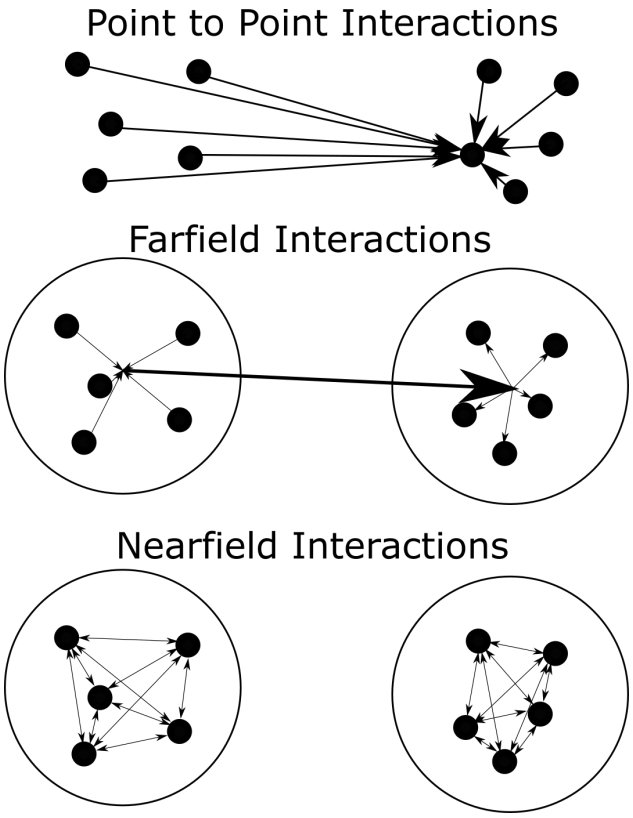

Figure 1.3: Point to point interactions shown on the top, while farfield interactions are shown in the middle, and nearfield interactions on the bottom.

The bottom of figure 1.3 the illustrates how the nearfield interactions of the bodies within each well separated group would be calculated. The farfield and nearfield interactions are summed together to compute the potential observed by each particle due to the interaction with all other particles. If this approach is constrained to a single level of approximations, it is shown to reduce the complexity to $O(N_S)^{\frac{4}{3}}$ for a volume and $O(N_S)^{\frac{3}{2}}$ for a surface, Coifman et al. (1993b).

This method of computing interactions via hub-like group approximations can be extended further. Interactions with a group of approximations that are well separated from another group of approximations can themselves be approximated as a single radiation pattern, providing multiple levels of approximations. The multi-level approach allows the farfield approximations to be organized into a tree. The leaf level contains nodes representing the approximation of the particles located in the area of the problem space occupied by the leaf node, and higher levels contain nodes representing radiation pattern approximations of groups of child nodes.

Tree Construction While higher level approximations are created from groups of lower level approximations, the tree structure is typically created from top down, before approximations, and interactions between approximations, are calculated, Barnes & Hut (1986). First the entire problem space is placed inside of a cube. This cube forms the root node of the FMM tree. The cube is evenly divided into 8 smaller cubes, forming the second level of the tree. Cubes are further subdivided, creating more levels and producing an octree. The same method is used with both volume and surface geometries, but a 2-d plane of particles cutting across the cube will typically populate 4 nodes per level with particles, rather than 8, creating a quadtree.

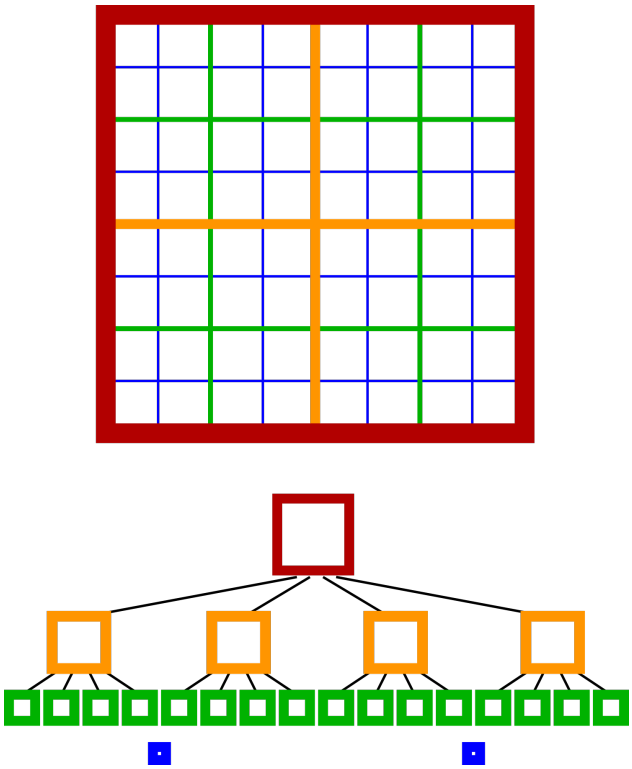

Figure 1.4: An FMM tree corresponding with a 2 dimensional problem space square.

Figure 1.4 illustrates a 2 dimensional representation of the tree construction. The entire geometry is contained by the root node in red. The second level evenly divides each dimension in half, producing four orange nodes in level 2. The subdivision continues, producing 16 green nodes in level 3 and 64 blue nodes in level 4 (all nodes are not displayed in the tree as it would exceed the page width). In the example in figure 1.4, level 4 would represent the leaf level, but an FMM tree could continue subdividing the geometry. Creation of lower levels of the tree typically

ends when the lowest level nodes are around 0.25λ, the point where the multipole expansion begins to appear non-oscillatory, however tree memory use and leaf node particle density also need to be considered. Too many points per node increases the computation time of nearfield, while too few points per node increases the computation time of farfield. With the tree constructed, the tree traversal process can be used to compute the farfield interactions, while nearfield interactions are computed directly.

Multi-Level FMM Figure 1.5 illustrates an example of 2 dimensional multi-level node interactions. At the highest level, Level 3, the source node in red performs farfield interactions with all other nodes that are not an immediate neighbor. Moving down a level, farfield interactions can be performed with nodes which are not immediate neighbors of the child node. Child nodes of nodes which were farfield nodes for the parent have already computed farfield interactions at the higher level, so do not need computation at the lower level. This process continues until the lowest level of computation is reached, and all opportunities for farfield interactions have been exhausted. At the lowest level, the leaf level of the tree, direct interactions must be computed between particles within the source box, as well as between these particles and the particles within immediate neighbor boxes. The multi-level approach reduces the computation cost to $O(N_S \log N_S)$ for a surface geometry, and $O(N_S)$ for a volume geometry. While the nearfield interactions have been limited to particles only interacting within their own box and with particles in the immediate neighboring boxes, and now cost $O(N_S)$ to process.

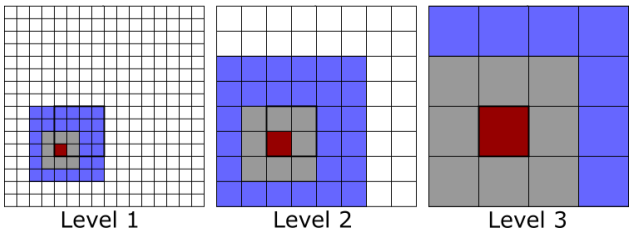

Figure 1.5: Multi-level interactions. The source node is red, nodes with nearfield interactions are grey and nodes with farfield interactions are blue.

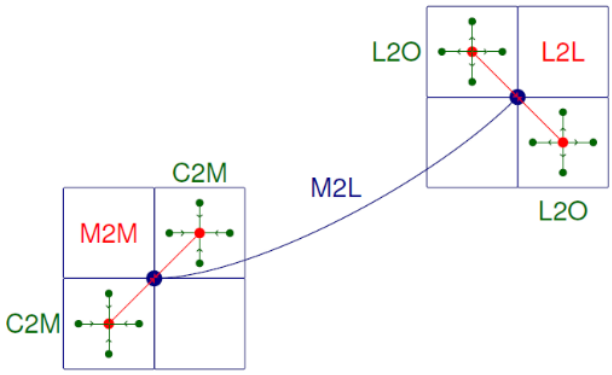

Figure 1.6: Top-down view of the lowest two levels of an FMM tree, illustrating the key computational kernels. Particles (green dots) are mapped onto leaf nodes (small squares) in an octree partitioning.

1.2 Tree Traversal

The stages of farfield computation during tree traversal are illustrated in Fig. 1.6 and defined as follows:

1. **Charge-to-multipole (C2M)**: This stage calculates leaf node multipole expansions. Particles inside each leaf node are summarized into the multipole expansion for that node.

2. **Multipole-to-multipole interpolation (M2M)**: M2M stage calculates the multipole expansions (approximations) in parent nodes. Starting at the leaf nodes and traversing up the tree, child nodes interpolate and shift their multipole expansions to the centers of their parent boxes. Once all of its children nodes are interpolated and shifted, the parent node aggregates the interpolated multipole expansions of its children.

3. **Multipole-to-local translation (M2L)**: M2L stage translates the multipole expansions of a source box to the center of observer boxes within its far field as defined according to the well separateness criteria. Multipole contributions at each observer box is then aggregated to account for the effect of distant particles.

4. **Local-to-local anterpolation (L2L)**: L2L distributes the far field contributions in parent nodes (as calculated by the M2L stage) to each of its children. L2L is essentially the reverse of the M2M stage; it is also called the downward tree traversal or the anterpolation stage.

5. **Local-to-observer (L2O)**: Once multipole expansions are percolated all the way down to the leaf nodes, they must be mapped back onto the particles. This is done in the L2O stage, which involves calculation of the potential observed by each particle based on the multipole information summed up in its enclosing box.

Nearfield The nearfield interactions can be computed during any phase of farfield computations as this operation computes the point to point interaction of particles in the same, or neighboring, leaf boxes. The only consideration is that the results of nearfield interactions must be summed with the results of L2O for each particle.

Laplace vs Helmholtz FMM As discussed earlier, common FMM variants can be grouped into two categories: Laplace FMM (L-FMM), where interactions are between particles are non-oscillatory particle interactions, such as gravitational and electrostatic particles, and Helmholtz FMM (MLFMA), Dembart & Yip (1995); Nishimura (2002); Shanker & Huang (2007), where interactions are oscillatory (*i.e.*, wave problems such as electromagnetic and acoustic particles). In L-FMM, since the interaction is non-oscillatory, only the magnitude information needs to be approximated in the multipole expansion. As such, the number of terms, or samples, of the multipole expansion is constant for all nodes across all levels of the tree. Consequently, most of the work for L-FMM is at the lowest level of the tree, Ying & Zorin (2004); Sundar et al. (2008); Agullo et al. (2014); Salmon (1991); Wang et al. (2019); Ying et al. (2004), because a majority of the tree nodes are located at the leaf level.

In contrast, in MLFMA, both magnitude and phase information must be stored due to the oscillatory nature of the interaction. To ensure controllable accuracy, the amount of multiple expansion data in a tree node must be approximately quadrupled as one ascends the tree. In addition, most MLFMA problems involve surface geometries (which implies that the number of tree nodes decreases roughly by a factor of 4 at each level). When combined with the increase in node size, all levels of the MLFMA tree have similar computational costs. In practice, higher tree levels often require more computations in MLFMA.

1.3 Thesis Outline

Existing work on FMM, and specifically MLFMA, has provided significant performance improvements by converting the $O(N_S^2)$ N-Body problem into an $O(N_S \log N_S)$ or $O(N_S)$ (depending on the problem geometry) tree algorithm. The continued growth of the size of N-Body problems has led to the need for efficient parallel FMM algorithms. Laplace FMM methods have shown excellent scalability, Hamada et al. (2009); Ishiyama et al. (2012); Rahimian et al. (2010); Lashuk et al. (2012), but methods to parallelize Helmholtz MLFMA continue to lag behind due to the added complexity of the oscillatory interactions, Ergül (2011); Ergul & Gurel (2013); Michiels et al. (2013a); Taboada et al. (2013); Waltz et al. (2007). In the remainder of this report, we present a number of algorithms that provide performance and parallel scalability improvements for MLFMA applications. The content is organized as follows.

Chapter 2 describes a step by step method to improve the performance of the spherical harmonics transform. These improvements focus primarily on data locality and cache optimizations of the matrix vector operations. Numerical results show the performance improvements of the optimizations nearing 10x speedups over the original algorithm.

Chapter 3 describes a fine grain parallel method that divides the samples and work of the high level nodes of the MLFMA tree evenly among the processes. This method improves the balance of process loads when the number of nodes in a level is less than the number of process, and balances tree storage across processes. The chapter also includes a detailed complexity model of the algorithm. Finally, numerical results are included that show the load balancing effect of the algorithm and the accuracy of the complexity model, and a comparison showing up to 164x speedup when compared with with an existing MLFMA algorithm.

Chapter 4 performs a comparison of bulk synchronous parallel and task parallel approaches for implementing MLFMA. Task parallel implementation is expected to best handle process communication if integrated with the distributed memory parallel code, and this comparison demonstrates that

task parallel implementation is efficient. The chapter includes a description of the two implementations, as well as considerations of performance trade offs with the task parallel implementation. The chapter wraps up with a performance comparison of the two shared memory parallel methods, comparing thread activity, cache utilization analysis, and execution time comparisons showing task parallel outperforming the bulk synchronous parallel version under most scenarios, with up to 1.4x speedup.

Chapter 5 looks at possible future research of balancing global and local interpolation, integrating thread based parallel methods into the MLFMA algorithms and improving load balancing for unbalanced trees.

CHAPTER 2

OPTIMIZATION OF THE SPHERICAL HARMONICS TRANSFORM BASED TREE TRAVERSALS IN THE HELMHOLTZ FMM ALGORITHM

2.1 Introduction

In this chapter, a step by step method of improving the performance of the spherical harmonics transform is provided. A naive implementation of spherical harmonics transforms results in data alignment that is very inefficient for data locality. An updated spherical harmonics transform algorithm will be provided that arranges the data to optimize cache use.

Vertical Tree Traversal (M2M/L2L) in MLFMA Multipole expansions are simply functions defined on the unit sphere $(\phi, \theta) \in [0, 2\pi] \times [0, \pi]$ and sampled at locations (ϕ_i, θ_j) for $i = 1, \ldots, N_\phi$, and $j = 1, \ldots, N_\theta$. The sampling rates for a given tree node at some level l are governed by its bandwidth $K(l)$, defined by $K(l) = \lfloor \chi \sqrt{3} k D(l) \rfloor + 1$, where $D(l)$ is the box diameter at level l, k is the wave number defined in (1.3), $\chi \geq 1.0$ is a parameter that can be tuned to control the accuracy of the MLFMA evaluation (higher χ values yield more accurate simulations at the expense of higher computational cost). Two main techniques used in traversing up/down the MLFMA tree are local interpolation Song & Chew (1995); Chew et al. (1997); Zhao & Chew (2000); Ergul & Gurel (2006) and global interpolation techniques Jakob-Chien & Alpert (1997); Shanker et al. (2003); Sarvas (2003).

Local Local interpolation focuses on interpolating a local region of the multipole data. This method requires a boundary samples from neighboring regions, usually stored on other processes, to perform the interpolation. The boundary samples can be as few as only the samples immediately neighboring the local region. This means that it is relatively easy to limit the bandwidth, as shown in Michiels et al. (2013b) during interpolation. Also the he number of messages between processes can be reduced, as shown in Yang et al. (2019), during interpolation. A further possible benefit

of the Yang et al. is processes are assigned a slice of samples from all node of a given level, then during translation the process owns all samples that must be translated so no communication is required. The trade off is that local interpolation introduces errors in the interpolation result. This error can be reduced by increasing the number of multipole samples in both the θ and ϕ direction. The error can also be reduced by calculating the local interpolation using boundary samples beyond the immediate neighboring samples. The increase of boundary samples can result in needing to communicate with processes beyond the neighboring ones. The best results are found by combining these two methods, but any improvement in error here is at the cost of additional memory or communication requirements.

Global In contrast to global interpolation requires all of the samples during interpolation. This means if the samples are partitioned between processes, all processes storing some samples of a node must be involved in communication to interpolate the node. Thus partitioning options such as assigning slices of samples from all nodes of a level is less efficient as interpolation with this partitioning would require communication between all processes to perform interpolation. The benefit of global interpolation is due to interpolating with all of the samples, the interpolation result can be exact with a minimal number of multipole samples. For this reason, we focus on global interpolation methods.

Problem Statement and Contribution In this chapter, we focus on the MLFMA algorithm. While the literature on optimizing the performance of L-FMM implementations is abundant, existing work in Laplace FMM can not readily be transferred to an MLFMA implementation due to the reasons outlined above. In MLFMA, interpolation and anterpolation operations (which together are also referred to as tree traversal operations) related to the creation of multipole expansions across the tree constitute one of the most expensive parts. Therefore optimization of these tree traversal operations is important, and provides the main motivation for this work. We begin by describing the fast spherical filter based inter/anterpolation operators in Section 2.2. Then in Section 2.3, we provide a basic implementation of tree traversals with spherical filtering and present

a series of techniques that successively leads up to our fully optimized tree traversal implementation for MLFMA. In Section 2.4, we demonstrate the impact of the presented techniques through an extensive set of tests.

Related Work The MLFMA literature is quite extensive, and the works discussed in this chapter represent only a sampling of those discussing the relevant issues. The single-level MLFMA (without inter/anterpolation) was introduced in Coifman et al. (1993a), providing a prescription for its implementation. Implementation issues for the multilevel version are discussed, e.g., in Song et al. (1997); Darve (2000); Vikram et al. (2009). The problem of inter/anterpolation is addressed in depth in Jakob-Chien & Alpert (1997); Shanker et al. (2003); Sarvas (2003); Ergul & Gurel (2006); Cecka & Darve (2013a), yielding a number of different methods of varying cost and accuracy. In fact, the techniques described in these studies provide the background for the present work. Wedi *et al.* Wedi et al. (2013) describe performance optimization techniques for a similar problem the field of numerical weather prediction. However, the algorithm presented in Wedi et al. (2013) involves a complicated compression technique which trades accuracy for lower computational complexity, but is only efficient at regimes different than those tackled in this chapter. Additionally, the optimization techniques we present do not compromise the accuracy of the underlying operations.

2.2 Background

2.2.1 Globally Interpolated Tree Traversal (M2M/L2L) in MLFMA

Previously mentioned, our chosen method of interpolation is global interpolation. Among the global techniques, one can choose the spherical harmonics transforms (SHT) Jakob-Chien & Alpert (1997) or the fast Fourier transforms (FFT) Sarvas (2003) based inter/anterpolations. Spherical harmonics and Fourier transforms require $O(K(l)^3)$ and $O(K(l)^2 \log K(l))$ operations, respectively. Spherical harmonics based inter/anterpolations, while asymptotically more expensive, utilize optimal (minimal) sampling rates on the unit sphere by sampling at Gauss-Legendre nodes in the θ direction, and picking uniformly spaced samples in the ϕ direction Jakob-Chien & Alpert (1997). More

specifically, it requires $N_\theta(l) = K(l) + 1$ samples in the θ direction, and $N_\phi(l) = 2K(l) + 1$ samples in the ϕ direction for a given level l. *As such, spherical harmonics transforms are advantageous in terms of memory (up to a factor of 2) and computational costs (by about a factor of 4 to 8 depending on inputs and hardware) for several levels above the leaf nodes, and therefore our MLFMA implementation is based on this technique*. Also as discussed in our recent work Hughey et al. (2018), one can easily transition from spherical harmonics sampling to the full uniform sampling as required by the fast Fourier transforms (FFT), when the asymptotic costs of spherical harmonics transforms start taking over.

2.2.2 Spherical Harmonics based Tree Traversal

Before we move to implementation and optimization details, we review the steps of the spherical harmonics transform (SHT) method to perform interpolation (M2M) and anterpolation (L2L) in the MLFMA tree. Interpolation is used to increase the sampling rate of multipole expansions to that of their parents' so that diagonal shift operators may be employed as one moves up the tree. In anterpolation, the reverse is done, i.e. the parent box multipole expansions are shifted to their children and decimated to the child's sampling rate. The steps outlined below focus on the upward traversal (M2M) operation; we note the nuances for downward traversal (L2L) at the end of this subsection.

1. Compute Fourier coefficients $f^m(\theta)$ by performing FFTs for each set of samples in the θ direction from $\theta_1, \ldots, \theta_{N_\theta(l)}$:

$$\mathbf{f}^m(\theta) = \frac{\sqrt{2\pi}}{N_\phi(l)} \sum_{i=1}^{N_\phi(l)} f(\phi_i, \theta) e^{im\phi_i} \tag{2.1}$$

   where, $\phi_i = 2\pi i / N_\phi(l)$ for $i = 1, \ldots, N_\phi(l)$ and $f(\phi_i, \theta)$ is the multipole expansion for a tree node along the ϕ direction for a given θ.

2. Compute the spherical harmonic coefficients f_n^m for $|\ m\ | \leq n \leq K' = K(l)$ by applying Legendre transforms to the Fourier coefficients calculated in the previous step:

$$\mathbf{f}_n^m = \sum_{j=1}^{N_\theta(l)} w_j \bar{P}_n^m(\cos\theta_j) f^m(\theta_j), \tag{2.2}$$

where $w_1, \ldots, w_j$ denote the corresponding Gauss-Legendre quadrature weights, and $\bar{P}_n^m(\cos\theta)$ denote the normalized associated Legendre functions.

3. Calculate the filtered Fourier coefficients $\widetilde{f}^m(\theta_j)$ from spherical harmonic coefficients (f_n^m) up to degree K' by a backward Legendre transform:

$$\widetilde{\mathbf{f}}^m(\theta) = \sum_{n=|m|}^{K'} f_n^m \bar{P}_n^m(\cos\theta). \tag{2.3}$$

4. Finally, the filtered multipole values $\widetilde{f}(\phi_i, \theta_j)$ are calculated by an inverse FFT, using Fourier coefficients up to degree K':

$$\widetilde{\mathbf{f}}(\phi, \theta) = \sqrt{2\pi} \sum_{m=-N_\theta(l-1)}^{N_\theta(l-1)} \widetilde{f}^m(\theta) e^{im\phi} \tag{2.4}$$

The L2L operation consists of the same steps but with $K' = K(l+1) < K(l)$ to accomplish the filtering. In M2M, the shift operator is applied post-interpolation, whereas in L2L the shift is applied before the anterpolation.

2.3 Algorithms & Numerical Methods

2.3.1 Implementation of the Spherical Harmonics Transform Method

First, we discuss in detail the base algorithm used to implement the SHT method. This will provide a detailed explanation of why certain design decisions were made, allowing for a step by step description of the improvements from this base algorithm.

Overview In SHT, for a given tree level l, the multipole expansion has $N_\phi(l)$ samples in the ϕ direction and $N_\theta(l)$ samples in the θ direction. To aid in the presentation, $N_\phi(l)$ will be referred to

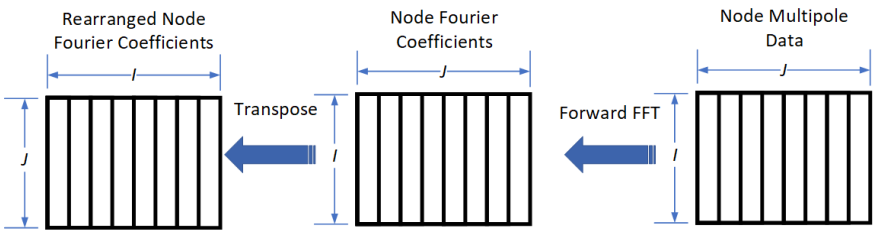

Figure 2.1: Spherical Harmonics Filter Step 1.

as I and $N_\theta(l)$ will be referred to as J. The multipole expansion information is stored as a matrix of size $I \times J$; we denote this matrix by $\mathcal{M}$. To exploit data locality due to reasons that will be discussed below, our base code (written in FORTRAN) stores ϕ samples as columns of $\mathcal{M}$.

During interpolation, the number of samples is increased to $N_\phi(l-1)$ samples (*i.e.*, roughly doubled) in the ϕ direction, and to $N_\theta(l-1)$ samples (*i.e.*, again roughly doubled) in the θ direction. Recall that anterpolation reduces the number of samples in the ϕ and θ directions by about half to $N_\phi(l+1)$ and $N_\theta(l+1)$, respectively. In both cases the *target* number of ϕ samples will be referred to as $\hat{I}$ and the target number of θ samples will be referred to as $\hat{J}$. Thus in both interpolation and anterpolation, a multipole expansion matrix $\mathcal{M}$ of size $I \times J$ is resampled to a multipole expansion matrix $\hat{\mathcal{M}}$ of size $\hat{I} \times \hat{J}$.

SHT performs calculations over $m = -J, \ldots, J$. In our implementation, we use ϕ indexes from $0, \ldots, I$ to represent the range of m as shown in the formula below. The reverse can be used to find the ϕ index corresponding to a given m index:

$$m = \begin{cases} \phi - 1 & \text{if } \phi \leq J+1 \\ \phi - 2 \times J & \text{if } \phi > J+1 \end{cases} \tag{2.5}$$

Precalculated Data In step 2 of the SHT method, the normalized associated Legendre functions and the Gaussian quadrature weights are multiplied together. Because the tree structure is static throughout the entire simulation in Helmholtz problems, the bandwidth at each level and the resulting sampling rates remain constant.

As such, we can precalculate the operators effecting this operation, yielding what we call the *forward transform matrices*. At each level there are $-J, \ldots, J$ forward matrices, corresponding with the range of m. Each forward matrix is $(J - |m|) \times J$ in size. Additionally, as described in Jakob-Chien & Alpert (1997), when $m < 0, P_n^{-m} = P_n^m$. This allows us to only store the forward matrices only in the range $m = 0, \ldots, J$.

In step 3 of SHT, the normalized associated Legendre functions are used again. To save important computational time, these can also be precalculated and stored as the *backward transform matrices*. The backward matrices project the number of θ samples for a given level l to the number of θ samples of a target level $\hat{l}$. Thus the matrix size is $\hat{J} \times (J - |m|)$. Finally, to save space, both the backward matrix for interpolation and the backward matrix for anterpolation to the target level are given by the same matrix. With the combination, there are $-\hat{J}, \ldots, \hat{J}$ backward matrices, again corresponding with the range of m for the target level. Each backward matrix is $\hat{J} \times (\hat{J} - |m|)$ in size.

Step 1 Figure 2.1 shows the first step in the SHT algorithm used in our base MLFMA implementation. In this step the Fourier coefficients are calculated by performing FFTs on the ϕ samples in each column, indexed from $0, \ldots, J$, of the multipole expansion. Each FFT produces a new column of ϕ sample Fourier coefficients. In the second step of the SHT method, calculations will be performed over θ samples. Since calculations would be most efficient by placing the θ samples along columns (in FORTRAN), the resulting vector from each FFT along ϕ samples is transposed and stored in rows of the new Fourier coefficients matrix. This produces a $J \times I$ matrix of ϕ-Fourier coefficients. In our implementation, all FFTs use the FFTW library Frigo & Johnson (2005) for best performance.

Step 2 The right side of Fig. 2.2 shows the second step in our base SHT implementation where the spherical harmonic coefficients are calculated from the Fourier coefficients matrix (of step 1) by using the precalculated Legendre transform matrices. More specifically, the base implementation loops over ϕ indexes from $\phi_i = 0, \ldots, I$ and performs a matrix-vector multiplication between the

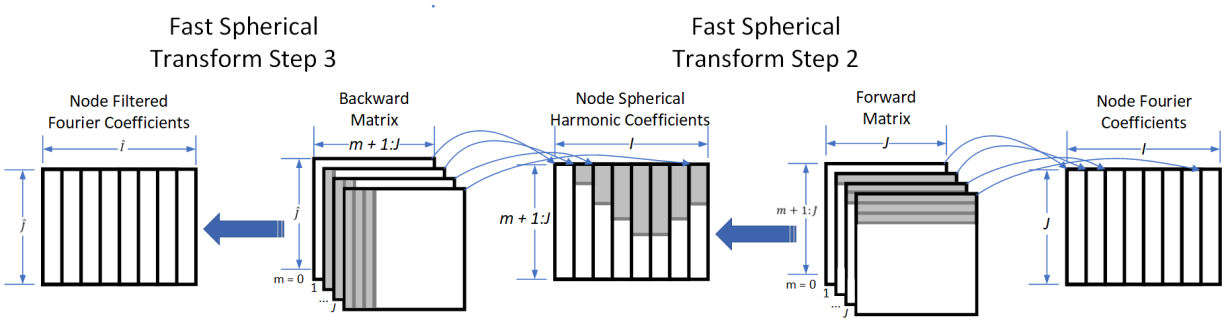

Figure 2.2: Spherical Harmonics Filter Steps 2 and 3.

forward matrix corresponding to index ϕ_i according to Eq. (2.5), and the ϕ_ith column of the Fourier coefficients matrix (produced in step 1). Note that in step 2 of the SHT, Legendre transforms are applied to Fourier coefficient vectors from m to K'. Figure 2.2 represents the vectors that are NOT used in matrix vector multiplication as light grey cells. While these values are not calculated, their inclusion in the figure helps show how the matrices and vectors align during multiplication.

Step 3 The left side of Fig. 2.2 shows the third step in our base SHT implementation where the filtered Fourier coefficients are calculated from the spherical harmonics coefficients matrix by performing backward Legendre transforms. This is done by looping over ϕ indexes of $\phi_i = 1, \ldots, I$ and performing a matrix-vector multiplication between the *backward matrix of the target level* corresponding to ϕ_i according to (2.5), and the ϕ_ith column of the spherical harmonics coefficients matrix (produced in step 2). Because the backward matrix for M2M and L2L have been combined, when performing M2M the backward matrix will have more columns than the number of rows in the filtered Fourier coefficients matrix. To perform the matrix vector multiplication, only the number of columns in the backward matrix up to the number of rows in the filtered Fourier coefficients matrix is used. This adjusts the use of the backward matrix to match the format shown in Fig. 2.2.

The number of rows of each backward matrix in the target level is $\hat{J}$, projecting the J length columns of spherical harmonics coefficients to $\hat{J}$ length columns of filtered Fourier coefficients. The initial number of columns of filtered Fourier coefficients matrix is I, the number of columns of the spherical harmonics coefficients. When the target node has more ϕ samples than the source

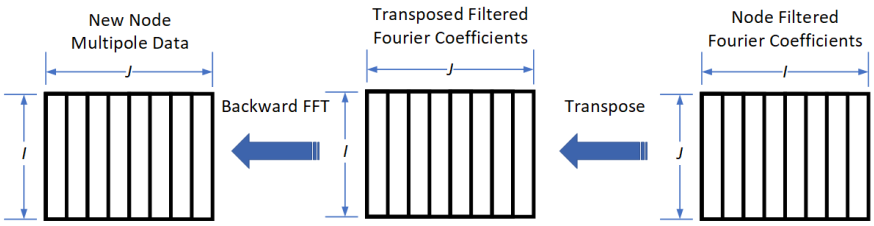

Figure 2.3: Spherical Harmonics Filter Step 4.

node, the filtered Fourier coefficients matrix must be padded to the desired $\hat{I}$ columns for the target level. To do this, the filtered Fourier coefficients matrix is split vertically down the middle, columns $1, \ldots, J$ on the left and columns $J+1, \ldots, I$ on the right. The matrix is then padded in the middle with columns of 0s to widen the matrix to $\hat{I}$, where $\hat{I}$ is the number of ϕ samples of the target node. When the target node has fewer ϕ samples than the source node, (2.5) is used to identify any column index producing an m value greater than $\hat{\theta}$ and these columns are ignored.

Step 4 Fig. 2.3 shows the fourth and final step in our base SHT implementation where inverse FFTs are performed on each θ index row, from $\theta_j = 0, \ldots, J$, of the filtered Fourier coefficients matrix for the node. To properly provide the data to the FFTW library, each row is transposed into a contiguous vector. The FFT results can then be stored in columns of the target multipole matrix, $\mathcal{M}$.

Operational Costs As evident from the above discussion, main computational costs associated with SHT are a series of FFTs (regular and inverse) and matrix-vector multiplications. More specifically, for a given node, J forward FFTs (each of which cost $I \log I$) and $\hat{J}$ inverse FFTs (each of which cost $\hat{I} \log \hat{I}$) must be performed. In terms of matrix-vector multiplications, there are a total of I matrix-vector multiplications involving forward matrices of size $(J-m) \times J$ depending on the value of m corresponding to the I index. The portion of the backward matrices used are of size $\hat{J} \times (J-m)$ (where $\hat{J}$ denotes the number of samples in the target level), and there again are a total of I matvecs involving backward matrices. Consequently, the cost of matrix vector multiplications for a particular m is $I \times (J-m) \times J + I \times \hat{J} \times (J-m)$. Given that m ranges from $0, \ldots, J$, the total

matvec cost can be approximated as

$$\sum_{1}^{I} \frac{1}{2} J \times J \times I + \frac{1}{2} \hat{J} \times J \times I. \tag{2.6}$$

2.3.2 Vector Reuse in Forward/Backward Matrix Multiplications

A simple optimization that can be applied to the base SHT implementation is reuse of vectors between forward and backward matrix multiplications. In the base implementation of SHT that we started with in our performance optimization efforts, all I forward matrix multiplications are performed, producing a set of I vectors containing spherical harmonic coefficients for a given node. Due to the total size of the forward matrices that one must go through before starting the backward matrix-vector multiplications, it is highly likely that the intermediate vectors with spherical harmonic coefficients will be evicted from cache (even for leaf nodes or nodes close to the leaf level). In this variant of the SHT implementation, instead of storing the spherical harmonics coefficients as an intermediate matrix, each vector, once calculated, is immediately multiplied by the corresponding backward matrix and the result is stored in the filtered Fourier coefficients matrix for the node. This avoids the need for intermediate storage of the entire spherical harmonics coefficients matrix for the node, but more importantly it allows reuse of the spherical harmonics coefficient vector data. As will be demonstrated through numerical results in Sect. 2.4, this optimization gives improved performance over the base version for lower level tree nodes. However, towards the top of the tree, it actually leads to slightly worse performance than the base version, as the individual spherical harmonics coefficient vectors themselves grow larger than the L1 cache, wiping off any benefits.

2.3.3 Combining Forward/Backward Matrices

The next optimization we pursue is combining forward and backward matrices in the SHT method. Note that the base code applies the forward and backward matrices one after the other in the form of back to back matrix-vector multiplications. In the proposed optimization, the first step is

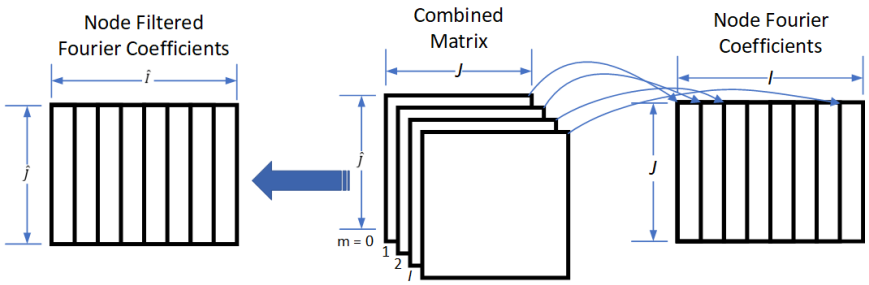

Figure 2.4: Spherical Harmonics Filter with combined Forward/Backward matrices.

multiplying the forward and backward matrices together, and then applying the combined matrices to Fourier coefficient vectors in a node, directly calculating the filtered Fourier coefficient vectors at the target node level.

A comparison of the computational costs of the combined method and the base method reveals that for a given node, by itself this optimization would actually increase computational costs. More precisely, for a given m index, we first need to multiply a backward matrix of size $\hat{J} \times (J-m)$ with a forward matrix of size $(J-m) \times J$, and the resulting matrix must be applied to a Fourier coefficient vector of size J. The total operational cost then is $\hat{J} \times (J-m) \times J + \hat{J} \times J$, which is more than the cost of the base algorithm.

However, a key observation made when combining the forward and backward matrices is that the forward and backward matrices are defined according to the spacial relationship between the nodes in the tree, and the structure of an MLFMA tree does not change over the course of a simulation as discussed previously. Moreover, the set of forward and backward matrices used in SHT is the same for all tree nodes at a given level l. This allows us to precalculate the combination of the forward and backward matrices. With precalculation, we only have to multiply the combined matrices with the corresponding Fourier coefficient vectors at a cost of $\hat{J} \times J$ (independent of the value of m), opening a door to save on the total number of calculations.

Note that the combined forward/backward matrix optimization would work well for certain values of m, *i.e.,* when m is small. As m grows, the forward and backward matrices have more zero-filled columns, but omitting calculations with the zero-filled columns as in the base algorithm

is not possible in the combined approach. By comparing the operational costs based on the value of m, it is possible to identify the value of m for which the number of operations in the base implementation becomes larger than that of the combined matrices approach:

$$\hat{J} \times (J - m) \times I + (J - m) \times J \times I = \hat{J} \times J \times I \tag{2.7}$$

As mentioned in Sect. 3.2, for M2M the number of ϕ samples for the current level, I, and the number of θ samples for the next level up, $\hat{J}$, are approximately two times the number of θ samples for the current level, J. We can use this relation to simplify and solve for the value of m where the base method becomes costlier than using the combined matrix approach:

$$\begin{aligned} 2J \times (J - m) \times 2J + (J - m) \times J \times 2J &= 2J \times J \times 2J \\ m &= \frac{2}{6}J \end{aligned} \tag{2.8}$$

This provides an estimation of when the base algorithm out performs the combined matrix for a given tree level. In fact, one can utilize the combined matrices approach only for values of m in a level where it provides an advantage over the base implementation. Also instead of relying on operation counts to judge which approach is better for a specific m value, an alternative method is an empirical tuning where one runs a number of matrix multiplications of appropriate sizes before starting the MLFMA solver, and determines for the given architecture at which point it becomes advantageous to switch to the combined matrix approach from the base implementation.

One additional trade off with the combined matrices approach is memory usage. In the base code, each level has a set of forward and backward matrices which are of size $J \times J$. With combined matrices, we need a matrix for the M2M stage and another one for the L2L stage each of which are of size $\hat{J} \times J$ where $\hat{J}$ is the number of θ samples of the target node. The θ samples are roughly doubled when moving up the tree and roughly halved when moving down the tree, which gives us an approximate memory ratio of:

$$\frac{2J^2 + \frac{1}{2}J^2}{2J^2} = 1.25 \tag{2.9}$$

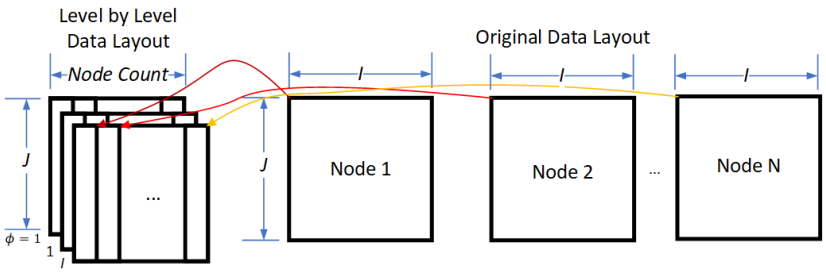

Figure 2.5: Reorganizing data for level by level processing.

or 25% more memory usage for the combined matrices over the separate forward and backward matrices. In order to use the fastest calculation of the base and combined matrix versions, we need to keep the forward backward matrices used by the base code, and additionally store the combined matrices. This results in approximately 125% more memory than the base version for storage of these matrices.

2.3.4 Level by Level Processing

Optimizations presented in the two preceding subsections focus on performing SHT operations for tree traversal on a node-by-node basis. While they try to improve data locality (through vector reuse) and reduce total computational costs (combined forward/backward matrices), their effectiveness is still limited. As a final optimization, we investigated level by level processing to perform the SHT operations. The key observation here is that a given forward/backward matrix pair (or their combined version) must be applied to the Fourier coefficient vectors of several nodes found in the same tree level (according to the ϕ to m mapping described in Eq.(2.5)). Moving from a node-by-node level processing to level-by-level processing is expected to yield significantly improved performance because i) the set of forward/backward matrices must now be read from memory *once per tree level* as opposed to once per node (hence significantly improving data locality), ii) the matrix-vector operations of node-by-node processing scheme can now be performed as matrix-matrix multiplications (hence significantly improving arithmetic intensity). That being said, level-by-level processing introduces new challenges in terms data reorganization/movement

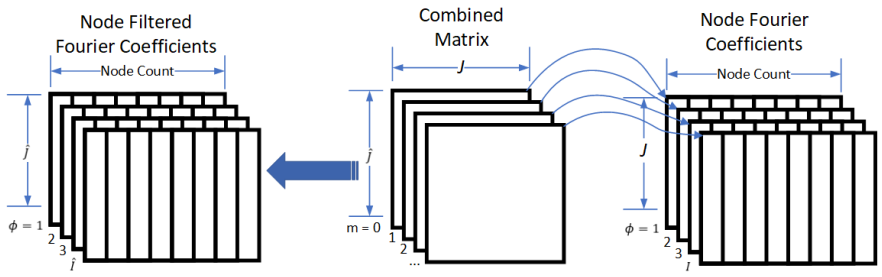

Figure 2.6: Spherical Harmonics Transform operation applied on a level by level basis (combined forward/backward version is shown).

as well as the size of intermediate storage space needed. Below we discuss the implementation details related to level-by-level processing and address the associated challenges.

For SHT method to utilize matrix-matrix operations, the Fourier coefficient matrices belonging to all nodes within the same tree level can be arranged into a 3D matrix, where ϕ is moved to the outer-most index so that for all nodes, all θ vectors can be multiplied with the same forward/backward matrix pair (or their combined matrix version). θ is left as the inner index to improve data locality during matrix-matrix multiplications, leaving the node index as the middle index. This data rearrangement is shown in Fig.2.5.

With such rearrangement of the Fourier coefficient vectors, matrix operations in SHT have a lot more opportunity for data reuse. As shown in Fig. 2.6, each ϕ indexed slice of the Fourier coefficients matrix can be multiplied by the corresponding single combined forward/backward matrix (or a single forward followed by backward matrix, if it is more efficient to do so depending on the value of m and tree level). This allows forward/backward matrices to be loaded once per tree level, multiplied by the corresponding θ vector of all nodes, and then stored in the filtered Fourier coefficients matrices.

How the algorithm loops through ϕ indexes to perform matrix multiplications can be further optimized in this scheme. Per Eq.(2.5) there are two ϕ indexes that map to the same m index, with the exception of $\phi = 1$. This means that when looping through all ϕ indexes, all but one combined forward/backward matrix must be loaded twice. Instead of looping through ϕ indexes, the algorithm can loop from $m = 0, \ldots, J$ and perform a matrix multiplication for each of the ϕ

indexes that correspond with the current m index. This allows each combined forward/backward matrix to finally only be loaded from memory once.

Matrix Multiplication Since matrix-matrix multiplication is a very common kernel in scientific computing and data analytics, there exists highly optimized libraries functions for this operation. Two options for matrix multiplication in BLAS are DGEMM (for multiplying real matrices) and ZGEMM (for multiplying complex matrices). Neither fits perfectly with the SHT method as the Fourier coefficients data is complex doubles, while the forward and backward matrices are real doubles.

To multiply a complex matrix with a real matrix, one method is to split the complex matrix into two matrices, one containing the real part and the other containing the imaginary part. Then each matrix can be multiplied against the real-valued multipole matrix using two DGEMMs. The results of these two multiplications would then need to be combined back into a complex matrix for later parts of the SHT operation. This option has a benefit that the DGEMM multiplication is being used exactly as intended, multiplying two full real matrices. The trade off is that extra time must be spent splitting the complex matrix apart and putting the results back together, and extra memory must be used to store the complex data in real matrices.

The second method to multiply a complex matrix with a real valued matrix would be to cast the real matrix into complex, with all imaginary values set to 0. Now the two matrices can by multiplied as complex matrices using ZGEMM. With all data stored as complex matrices, no extra work needs to be done to split apart the real and imaginary components or putting them back together. The downside to this method is that the full complex matrix multiplication must be performed, even though all of the imaginary data of one matrix is set to 0, resulting in extra processing, and the combined forward/backward matrices now require twice as much memory to store the imaginary 0s.

As shown in Sect. 2.4, we observe that the DGEMM version in general outperforms the ZGEMM option, and therefore our optimized level-by-level implementation uses DGEMMs.

Cache Blocking in Data Reorganization Because FFTs are performed across ϕ samples, the optimal multipole expansion data arrangement is to place ϕ data in columns (in FORTRAN) for SHT steps 1 and 4. This prevents data striding while performing the FFTs. On the other hand, the optimal data layout for matrix multiplication is columns of θ data, nodes in the rows and ϕ as the outer-most index (slices of 2D matrices). This conflict between FFTs and matrix multiplication means there is no one optimal data layout.

For this reason, we perform the FFTs in the same manner as outlined in Section 2.3.1. After performing FFTs, the resulting block of ϕ vectors with Fourier coefficient must be moved to the three dimensional matrix layout for DGEMM/ZGEMMs. While performing this copy operation, we utilize cache blocking on θ indexes to limit the amount of memory striding; otherwise the overheads associated with this copy operation can easily wipe out the benefits from reorganizing data in the ideal form for FFTs, then DGEMMs, and then again inverse FFTs.

Node Blocking Due to the rearrangement of of the matrices to to be $\theta \times nodecount \times \phi$, with θ as the inner most index, each 2D slice of the matrix is now has a width of the number of nodes, which can be on the order of millions for lower tree levels in large-scale computations. These larger 2D matrices can exceed the available cache. By passing a limited number of nodes to the SHT algorithm, the size of the 2D matrix can be limited to an amount of data that better fits in cache. This provides a performance improvement when there is a large number of nodes, however matrix multiplication is not as efficient with extremely narrow matrices. The node blocking is kept balanced by limiting how small the blocks can be, in order to keep the matrix multiplication efficient. For all performance tests presented in the next section, level-by-level implementations include FFTW and node blocking were used as these were the optimal scenarios found below.

2.4 Numerical Results and Performance Evaluation

In this section, the performance of the presented spherical harmonics transform optimizations are analyzed. Performance results were gathered using the Cori supercomputer in the National Energy Research Scientific Computing Center (NERSC). Runs were performed on the two socket

$K(l)$	3	6	12	24	48	96	192	384	768	Total
Base (sec)	3.10	1.93	1.62	3.11	3.93	4.01	17.79	40.01	104.75	180.25
Vector Pipeline	1.13	1.13	1.11	1.04	1.04	1.03	0.99	0.98	1.01	1.01
Matrix Pre Calculated	1.37	1.29	1.21	0.99	0.89	0.70	0.58	0.57	0.55	0.58
Optimal Combined	1.46	1.38	1.34	1.13	1.08	1.15	1.92	1.40	1.17	1.27
Level by Level	2.03	1.05	0.91	0.92	1.02	1.02	2.45	2.98	3.99	2.85
LbyL/DGEMM	1.48	0.97	0.68	1.00	1.28	1.71	3.99	5.30	9.32	4.71

Table 2.1: Speedup times for each Spherical Harmonics Transform implementation relative to the base code. Each column gives relative speedup for a particular level, the last column gives the overall speedup results. The first line for the base implementation reports the total execution time in seconds.

Intel Xeon “Haswell" processor nodes with 2.3Ghz clock rate and 128 GB DDR4 2133 MHz memory. Code was compiled using the Intel Fortran compiler with flags -O3, -no-vec and -mkl, and run as a single thread. Intel’s MKL library was set to utilize a single thread for ZGEMM and DGEMM operations to obtain fair comparisons against other versions described.

2.4.1 Overall Performance

To provide a consistent comparison of the optimizations of the SHT method, all methods were provided the same input data and verified to output identical resulting data as the base implementation (described in Sect. 2.3.1, which is known to produce accurate results). The base implementation’s execution time provides the baseline which each optimized version is compared against. In the results shown in Fig. 2.7 and Fig. 2.1, a full 11 level quad tree (4 children per node) was created to simulate a surface geometry. In all figures and tables presented, we provide measurements for each level of the tree to analyze the impact of changing samples sizes on the methods used. These different levels are marked with the bandwidth $K(l)$ in that level, from which the number of ϕ and θ samples are derived. In our synthetic quad tree, each leaf node contains different generated multipole data to ensure realistic memory usage and different interpolation/anterpolation results for each node, pinpointing any code errors that may produce different results from the base code.

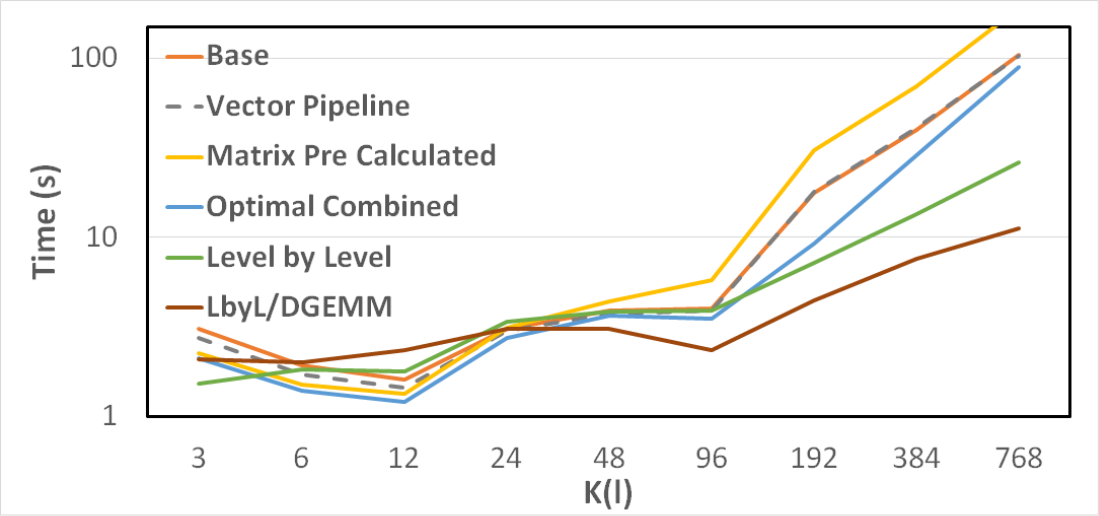

Figure 2.7: Execution time for different Spherical Harmonics Transform implementations for an 11 level surface geometry.

The initial optimization was to calculate the Spherical Harmonics Coefficients immediately followed by calculating the Filtered Fourier Coefficients for each θ vector, rather than calculating all of the Spherical Harmonics Coefficients before calculating the Filtered Fourier Coefficients. We denote this version as “vector pipelining" in Fig. 2.7. At lower levels of the tree (denoted by smaller $K(l)$ values), this optimization provides a minor improvement as it reduces intermediate storage and uses less operations by not having to store off the intermediate results. At higher tree levels, vector pipelining does not perform as well, when the forward and backward matrices no longer fit in cache.

The next optimization is precalculating the combined forward/ backward matrices, denoted as “Matrix Precalculated" in Figures 2.7 and 2.1. As discussed in the algorithms section, this precalculation is expected to outperform the base version only for lower m values. As we move up the tree, *i.e.*, as the number of required samples J and I increases, the value of m where matrix vector multiplications using the precalculated matrix outperforms the base code decreases. In other words, an implementation which uses the combined matrix approach for all values of m starts significantly underperforming against the base implementation. Adopting a hybrid approach where the best performing parts of the base code are intermixed with the best performing parts of using

the combined matrix approach, the improvement over the base code is higher in all tested cases. We refer to this hybrid approach as “Optimal Combined", and as can be seen in Figure 2.1 this hybrid approach delivers a 1.27x overall speedup over the base code.

Performance is further improved by moving to “Level by Level" computation method. This optimization which uses a naive 3-loop matrix matrix multiplication scheme still outperforms the base code overall by 2.85x. Rearranging the data in the ideal form for FFTs and then for matrix-matrix computations does impose an overhead cost. This is best seen for $K(l)$ from 6 through 96, where the overhead cost at times erases all the performance gains obtained from converting matrix-vector operations to matrix-matrix multiplications. At very low tree levels, the rearrangement cost is not significant with smaller numbers of ϕ and θ samples in the multipole data, and for higher levels of the tree, the gains from matrix-matrix multiplications far outweigh the cost of data rearrangements. For instance for $K(l)$ = 768, the “Level by Level" scheme achieves 3.99x speedup over the base SHT implementation.

Finally, given that the level by level scheme uses matrix/matrix multiplications, optimized library kernels can be used for this purpose. We observe that the best performance is obtained when using DGEMM to perform the matrix multiplication, as opposed to ZGEMMs. In our tests, our best implementation, named “LbyL/ DGEMM" achieves an overall speedup of 4.71x over the base SHT implementation. In fact, for $K(l)$=768, we observe speedups over 9x as a result of significantly improved matrix-matrix multiplication performance. At higher tree levels the performance improvements of “LbyL/DGEMM" increase at a faster rate.

While the “LbyL/DGEMM" approach shows significant speedup starting at $K(l)$=48, and continuing to increase as $K(l)$ increases, the performance at $K(l)$ from 6 to 24 significantly underperforms. The “Level by Level" performance shows the initial overhead is due to rearrangement of the data for matrix-matrix computations. The second overhead, as noted in Sect. 2.3.4, is the conversion of complex data matrices into one matrix of the real data, and a second of the imaginary data, so that the DGEMM multiplication can be used. While this overhead cost is significant at lower levels, the increase in performance results in “LbyL/DGEMM" being the fastest algorithm

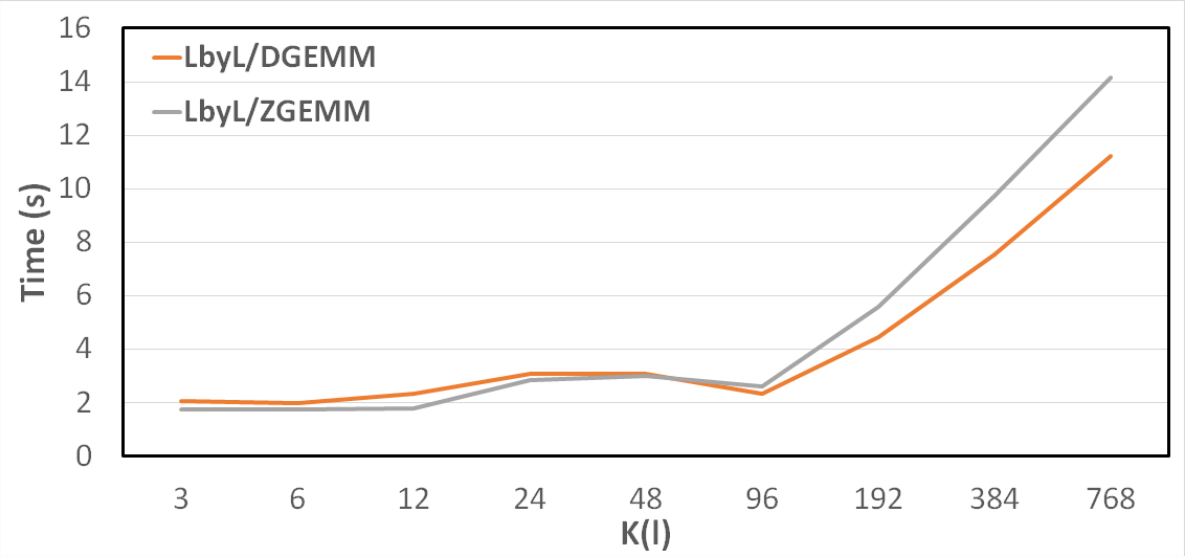

Figure 2.8: DGEMM Vs ZGEMM optimization.

in overall time. A probably faster approach could be to use the “Optimal Combined" method and “LbyL/DGEMM" together, using the method that is fastest for the current $K(l)$.

DGEMM vs ZGEMM Finally, in Section 2.3.4, it was discussed that both DGEMM and ZGEMM could be an option for matrix multiplication given that the multipole data is complex data and the forward/backward matrices are real data. Fig. 2.8 shows the resulting run times for DGEMM and ZGEMM based scheme as compared to the base code. At lower tree levels, ZGEMM appears to outperform DGEMM slightly due ZGEMM not having to split the multipole data into two real arrays. At higher levels though, DGEMM begins to notably outperform ZGEMM when the size of the matrices being multiplied become larger. Due to the minimal improvement of ZGEMM at lower levels and the significant improvement of DGEMM over ZGEMM at higher levels, we have chosen to use DGEMM in the other parts of the chapter.

Impact of Cache Blocking Recall that rearranging the data to be more optimal for matrix multiplication conflicted with the data arrangement needed by the FFTW library. This overhead was limited by cache blocking the data movement during the rearrangements back/forth. In particular, we perform a set of FFTs (8-16) at once for each θ index of the data, and move them in chunks to contiguous locations in the 3D matrix of Fourier coefficients. Figure 2.9 shows that

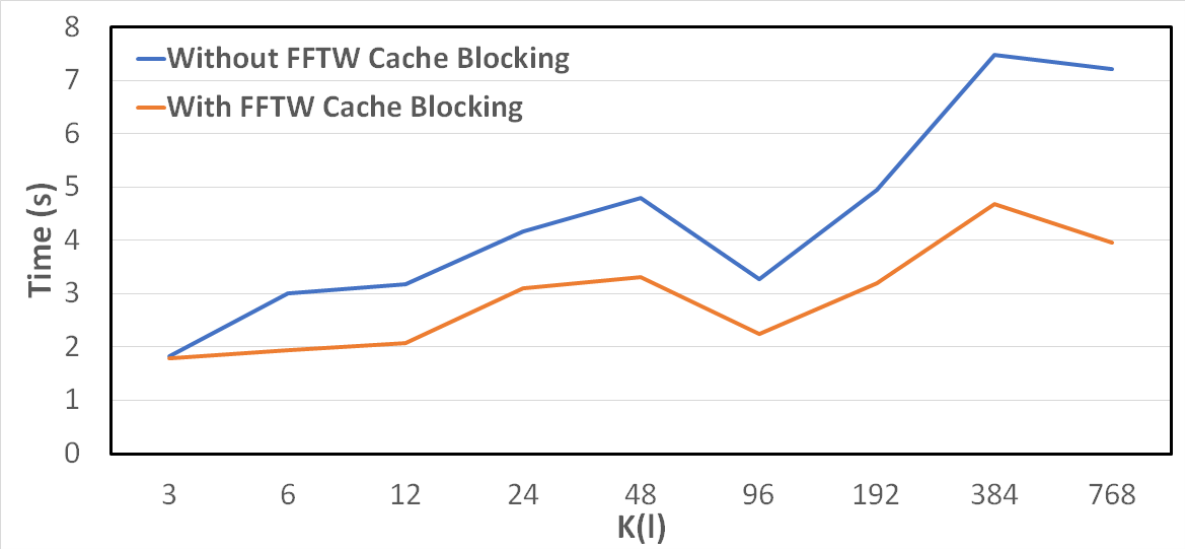

Figure 2.9: Analysis of blocking data around Forward/Backward FFTs.

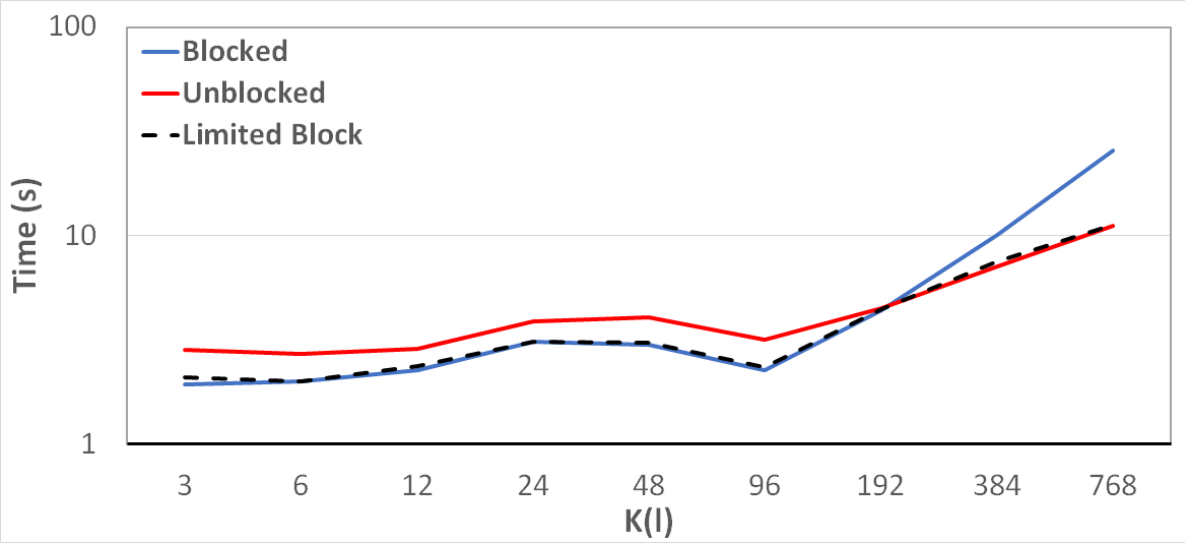

Figure 2.10: Analysis of DGEMM blocking on nodes (note, updated graph needed)

the improvement obtained from cache blocking the data rearrangement stage continues increasing as $K(l)$ increases, with a minor exception at $K(l)$=96. At this level, all methods used show a decrease in FFTW execution time, indicating that 96 is a sweet spot for the FFTW library. If cache blocking had not been used, the benefits of the LbyL/DGEMM scheme would have been significantly reduced.

Node Blocking Rearranging for optimal matrix multiplication also results in the potential of extremely large multipole matrices being multiplied against the combined forward/backward matrices. At the leaf level of an 11 level surface geometry for instance, there are 1 million nodes. Instead of passing all nodes to the SHT algorithm, they can be passed in blocks to save on the total memory requirements. Fig. 2.10 compares the LbyL/DGEMM scheme's execution time with no-blocking, when nodes are grouped into 16 block and when using an adaptive blocking scheme. As seen in the figure, dividing the total nodes into 16 blocks provides a significant improvement over unblocked at lower levels, but blocking at higher tree levels limits the improvement as this shrinks the size of the multipole matrix too much. At higher levels, passing all nodes to the SHT algorithm provides the best performance.

2.5 Conclusions and Future Work

Due to the need for representing both magnitude and phase information, Helmholtz variant of the FMM algorithm is significantly challenging compared to the commonly studied Laplace variant. In this chapter, due to its computational and storage cost advantages over alternatives, we adopted a Spherical Harmonic Transform based tree traversal scheme for the computationally expensive M2M and L2L stages of the MLFMA algorithm. We presented a series of optimization techniques to improve the performance of the SHT method. We demonstrated that the performance of SHT performance can be significantly improved by moving to a level-by-level scheme from the commonly used node-by-node processing scheme, and by carefully considering data rearrangement and cache utilization issues. This improvement has been shown in detail, step by step, with the performance improvements laid out. The performance improvements obtained (up to 9.3x) provide a clear demonstration of benefit of the optimizations described here.

CHAPTER 3

HIGH PERFORMANCE EVALUATION OF HELMHOLTZ POTENTIALS USING THE MULTI-LEVEL FAST MULTIPOLE ALGORITHM

3.1 Introduction

In the previous chapter we focused on a single node optimization to Spherical Harmonics Transform interpolation. In practice, full Helmholtz FMM simulations need to be executed on large computer clusters due to the high amounts of computation and memory required. As mentioned previously, Laplace FMM has a large portion of the tree data stored in the leaf levels, allowing for excellent scaling with leaf level optimizations. The Helmholtz FMM higher level tree costs mean data partitioning and communication of these higher levels are important bottlenecks to scaling. In this chapter, we address data partitioning and communication issues for high level tree node through novel algorithms, and provide an algorithmic complexity analysis of computation and communication overheads.

Data Partitioning How the tree is constructed, as discussed in section 1.1, has implications for parallel implementation trade offs. In Helmholtz FMM, the lowest levels of the tree have a large number of nodes with a small number of samples per node, while the highest levels of the tree have a small number of nodes with a large number of samples per node, such that the total data stored at any level is close to equivalent, or slightly larger at the high levels for a surface geometry.

Looking at the low levels, even a fairly small surface geometry that produces a 10 level tree has 262k leaf nodes. It is easy to assign a portion of the nodes to each of the 64, 8192 or even more processes. This method is known as spatial partitioning. If the data can be divided such that all descendant nodes of a high level node reside on a single process, all of these descendants can be interpolated using only local data. The primary question is how much communication is required when child nodes are split between processes, requiring communication when the interpolated data must be aggregated together, and what the performance impact will be. This question will be

investigated further in this chapter.

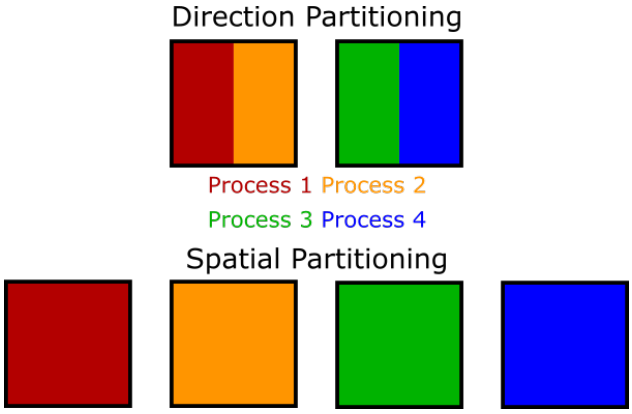

Figure 3.1: Spatial vs Direction Partitioning.

At the high levels, the number of nodes per level is far lower, fewer nodes in the level than the number of processes that might be assigned to the execution. At tree level 4 of any surface geometry, more than 64 processes means there will be more processes than nodes. The more processes assigned, the lower the level where we have more processes than nodes. At these high levels we can use a method known as direction partitioning. Figure 3.1 illustrates the how spatial partitioning assigns whole nodes to processes and directional partitioning divides nodes in a given level up among unique groups of processes. This method assigns each node in the level to unique groups of processes so that every process is assigned. Melapudi et al. (2011) describe an adaptive direction partitioning algorithm, where the switch from spatial partitioning to direction partitioning can occur at different levels, for different partitions to account for non-uniform trees. See section 3.3.1 for a more detailed description of this partitioning method. The primary weakness of the described method is how the native processes, we will refer to them as resident processes, must be sent the entire interpolated node data to be aggregated and then communicated back to the other processes sharing the node. In this chapter we will investigate a method of distributing all of the work of the shared nodes among the processes that share the node and analyze the complexity of this approach.

3.2 Background

Related Work The aforementioned work profile of the H-FMM octree suggests that any efficient parallelization must strike a balance between distributing the many lightweight boxes at lower levels and distributing the work of the few heavyweight boxes at higher levels across processes. Furthermore, the mathematics used to effect each stage of the process dictate the intricacies of the algorithms developed for parallelization. The existing algorithms address these different scenarios with different trade-offs. For the purposes of the ensuing discussion, we note that multipole and local expansions may be viewed as two-dimensional arrays of sampled function values.

At scale, the multipole and local expansions of octree boxes at the uppermost levels of the tree must be distributed across processes to achieve load balance Velamparambil et al. (2000). To reduce the costs of communication in M2M and L2L for these distributed nodes, several authors have employed *local* interpolation techniques Chew et al. (2001); Ergül & Gürel (2008); Michiels et al. (2013b), in which only a small "halo" of nearby samples are required to calculate each new sample. However, despite a slightly lower M2M and L2L asymptotic complexity of $O(N_s \log N_s)$, when compared to global interpolation's complexity of $O(N_s \log^2 N_s)$ Chien & Alpert (1997); Sarvas (2003); Cecka & Darve (2013b), this approach increases memory and computational costs stemming from the need to significantly oversample multipole and local expansions, in addition to reducing the numerical accuracy of the H-FMM. Alternatively, we propose the use of a parallel version of the *global* interpolation method, which has typically remained restricted to the serial M2M/L2L operations at lower levels of the tree. Though the global algorithm obviously has higher communication costs, it does not introduce additional numerical errors, and it facilitates optimal (minimum) sampling rates for multipole/local expansions Hughey et al. (2019); Lingg et al. (2018).

Local interpolation based hierarchical partitioning (HiP) approach distributes expansions hierarchically at the uppermost levels in block columns, or strips Ergül & Gürel (2008). However, the M2M and L2L communication costs scale as $O(\sqrt{N_s})$ per process, hampering the scalability of the H-FMM evaluation Ergül & Gürel (2013). The blockwise HiP (B-HiP) strategy Michiels et al. (2011, 2013b) alleviates this bottleneck by distributing expansions in blocks, whose much

lower surface-to-volume ratio results in $O(1)$ communication costs per process, hence improving scalability Michiels et al. (2015). In both methods, M2L operations are carried out in parallel by collecting on each process samples of the remote multipole expansions required to compute the local expansions it owns. It should also be noted that the increased sampling required with local interpolations hampers the scalability of the M2L phase, as collecting remote multipole expansions requires a significantly higher communication bandwidth compared to a global scheme with optimal sampling rates.

Building on the HiP approach, Yang et al. Yang et al. (2019) transition from hierarchical partitioning to plane-wave partitioning (PWP) Velamparambil et al. (2000) for the highest levels of the tree, using a binary tree decomposition of the MPI communicator to flexibly load balance the computation. The PWP approach achieves zero communication overhead for M2L operations by distributing expansions at the uppermost levels of the tree by assigning each process a fixed window of samples for *all* expansions at a given level. However, the transition from HiP to PWP requires expensive communications in M2M and L2L phases to rearrange the expansions, though this cost may be justified by recognizing that each node interacts with at most 8 other nodes to perform the M2M/L2L operations, while the maximum number of nodes for M2L operations is 189 (with a volumetric problem).

As previously stated, local interpolation methods are convenient for parallelization but result in an H-FMM that is **not** strictly error-controllable. The principal challenge to a scalable H-FMM with error control is the communication cost of distributed global (exact) interpolation. In Melapudi et al. (2011), Melapudi et al. describe an error-controllable H-FMM based on global interpolation using a bottom-up partitioning which gives great flexibility for load balanced partitioning of the tree. This work has continued to this day with the most recent research done by Hughey et al Hughey et al. (2019), which discusses the current state of the art in globally interpolated H-FMM. Scalability of this implementation is hampered by the coarse-grained parallelization of the M2L phase and redundant M2M/L2L calculations associated with high-level tree nodes shared by multiple processes.

Contribution In this chapter, we build upon our earlier efforts Melapudi et al. (2011); Hughey et al. (2019) toward a scalable, error-controllable H-FMM based on global interpolation. We address several challenges regarding parallelization and communication, and we demonstrate an efficient and scalable method for evaluation of the Helmholtz potential. In particular, our contributions can be summarized as follows:

1. Development of a fine-grain parallel algorithm with bottom-up partitioning that enables scalable evaluation of deep uniform MLFMA trees,

2. maintain the high level of controllable accuracy shown in previous global interpolation implementations,

3. a detailed analytical model to characterize the complexity and memory use of the parallel algorithm for far-field interactions,

4. and, demonstration of the overall algorithm performance and validation of this performance against our analytical model for different test scenarios.

3.3 Algorithms

In what follows, we describe details of each stage of the algorithm. For completeness, we replicate some of the concepts introduced in Melapudi et al. (2011); Hughey et al. (2019), before delving into details of our specific contributions. Specifics on mathematical formulae and operators used in this algorithm can be found in Melapudi et al. (2011); Hughey et al. (2019).

3.3.1 Tree Construction and Setup

Let N_p denote the number of processes used in the computation. We initially distribute the N_s particles evenly across all processes and determine the diameter D_0 of the cube bounding the entire computational domain. Given the finest box diameter $D(L)$, the number of levels in the tree is calculated as the smallest integer L such that $L \geq \log_2(D_0/D(L)) + 1$. Once the number of levels

and therefore the position of the leaf nodes are known, every particle is assigned a key based on the Morton-Z order traversal of the MLFMA tree Warren & Salmon (1993). A parallel bucket sort on the Morton keys is then used to roughly equally distribute particles across processes at the granularity of leaves. This is done by selecting $N_p - 1$ Morton keys, or "splitters", which chop the Morton Z-curve into N_p contiguous segments. Whole leaves are uniquely assigned to processes using these splitters. Given a contiguous segment of leaf nodes, each process determines all ancestor keys of its leaves up to the root. The leaf through ancestor nodes are used to construct the *local subtree*. A simple method of storing the local subtree is as a post-order traversal array. To quickly access a random node, we use an indexer into this local subtree array.

Plural Nodes Despite the non-overlapping partitioning of leaf nodes, overlaps among different processes at the higher level nodes are inevitable (and in fact, are desirable) as illustrated in Fig. 3.2. Details and associated proofs on such partitioning can be found in Melapudi et al. (2011). We call such nodes shared by multiple processes as *plural nodes*. While there are no limitations to the number of processes that can share a plural node, we designate a particular process, i.e., the right-most process sharing the plural node in the MLFMA tree, as its *resident process*. We refer to the resident process' copy of a plural node as a *shared node* and all other copies of this node residing on other processes as *duplicate nodes*. We call the set of processes that own these duplicate nodes as *users* of the shared node, denoted by $U(s)$, where s is the shared node.

One notable advantage provided by plural nodes is that storage of the node is split between multiple processes. In this case, the indexer additionally stores which *slice* of a tree node the current process actually stores in its local subtree array. As the information content for a node is not available to any single process in its entirety, a fine grain parallelization is necessary to perform computations associated with plural nodes. We note that a process can have at most two plural nodes per level in its local tree (essentially one to the "left", and another to the "right"); for proof see Melapudi et al. (2011)

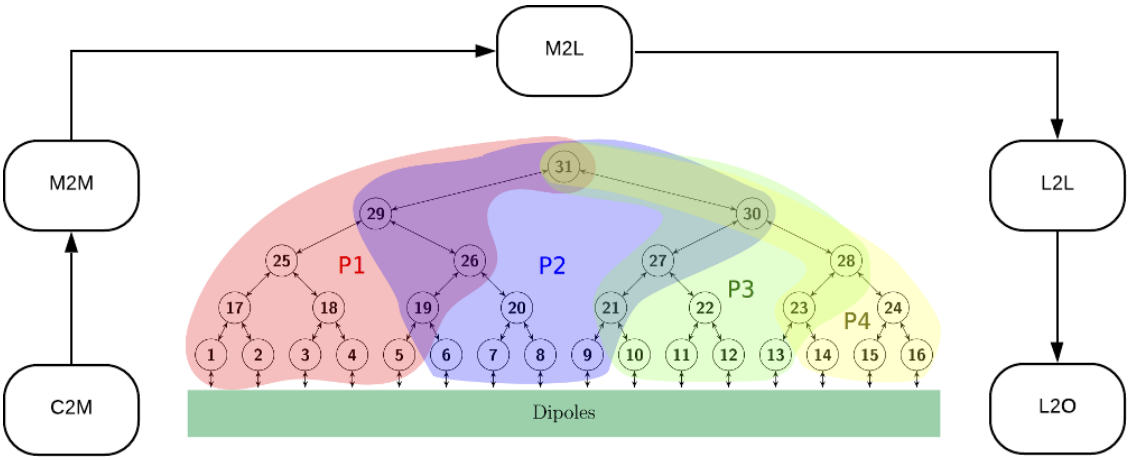

Figure 3.2: Parallel Farfield MLFMA.

3.3.2 Parallel Evaluation

3.3.2.1 C2M

C2M is unchanged from our previous works Melapudi et al. (2011); Hughey et al. (2019). As each process is assigned a contiguous segment of whole leaf nodes, each process handles the C2M phase for its assigned leaf nodes in parallel independently.

3.3.2.2 M2M

In a nutshell, M2M creates multipole expansions of non-leaf boxes from the multipole expansions of their children. This first requires multipole expansions of all children to be *interpolated* to the size of the parent box. Next, each interpolated child box is *shifted* from the center of the child box to the center of the parent box. Finally, multipole contributions of every shifted child box are *aggregated* to form the multipole expansion for the parent box.

M2M computations start at the leaf level and proceed upwards in the tree following a post-order traversal of the local tree. Our parallelization of *M2M* depends on the level of the node and is described in Alg. 4.3. The approach is as follows: i) Non-plural tree nodes (that are typically found at lower levels of the tree) are handled by their owner processes in parallel independently, ii) for plural nodes without any plural children, *interpolation* and *shift* steps for child nodes are performed

sequentially, and the aggregation step is performed as a reduce-scatter operation among processes sharing the plural node, iii) plural nodes with plural children (which typically are located at the higher levels of the tree and incur significant computational and storage costs) are processed using a fine-grained parallel algorithm that we discuss in more detail below.

Algorithm 3.1 Multipole-to-multipole interpolation

Require: $p.center$ coordinates of the parent box center
Ensure: pmp is parent's multipole representation

```
1:  for each box p in post-order traversal do
2:      for each child box c do
3:          if c is plural then
4:              mp[c] ← parallel_interpolation(c)
5:          else
6:              mp[c] ← serial_interpolation(c)
7:          end if
8:          smp[c] ← shift(mp, p.center)
9:      end for
10:     if p is plural then
11:         reduce_scatter(smp, users(p))
12:     else
13:         for each child box c do
14:             aggregate(pmp,smp[c])
15:         end for
16:     end if
17: end for
```

Fine-grained Parallel Interpolation For plural nodes that necessitate fine-grained parallelization of M2M, the multipole data of the child nodes needed for FFTs are themselves split among multiple processes as indicated in Alg. 4.3. Prior to elucidating our parallel algorithm, we note that our M2M implementation uses a Fast Fourier Transform (FFT)-based interpolation over the uniformly spaced multipole expansions of the child nodes Sarvas (2003). FFT-based interpolation on the Fourier sphere requires equispaced samples along θ (vertical) and ϕ (horizontal) directions. Due to the inter-dependencies of the FFT algorithm, there is no way to partition the data so as to avoid communication.

Our approach is as follows: First, each process is assigned a (roughly) equal number of con-

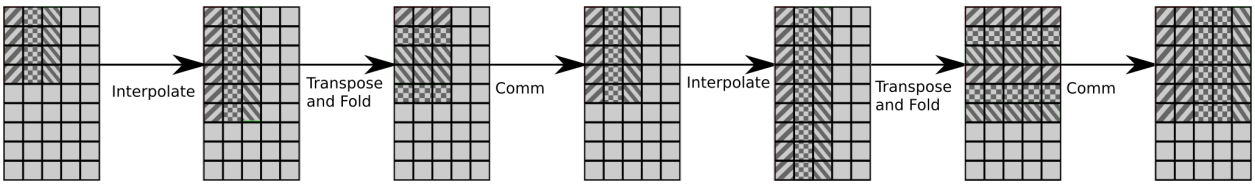

Figure 3.3: Graphical illustration of the transposition and folding operations during fine grained parallel interpolation of multipole data of child node c to parent node p for $N_\theta = 3, N_\phi = 4$ (for c) and $M_\theta = 5, M_\phi = 6$ (for p) using 3 processes each of which owns a column of the initial multipole data as indicated by the hashmarks. Multipole data from (a) is interpolated along the ϕ dimension locally, leading to the multipole expansions in (b). The folding operation acting on the interpolated data is shown by the repositioning of the data as in (c). The hash marks show how the folded data is stored on the wrong processes, and must be communicated to the correct process as shown in (d). With the entire multipole data columns in the θ direction being available on each process, another set of interpolations are performed locally (e), which is then transposed and folded to yield the final multipole expansions (f). A final communication step is needed to send each θ vector to their owner processes (g) which can then be shifted to the center of the parent box and added to the parent's multipole expansions in accordance with the spherical symmetry condition.

tiguous columns of multipole data (which correspond to groups of samples along the ϕ direction). The operation begins with a set of independent FFTs along these columns for interpolation in the ϕ direction, performed the same as the basic Sarvas approach. Then, the interpolated columns are shifted into rows (see Fig. 3.3), transposing and folding the samples in the θ direction into individual columns. The next step with serial processing would be FFT interpolation in the θ direction, but this data is split between processes sharing the plural node. Therefore, each process is communicated the θ samples they need to complete their assigned columns using an `MPI_Alltoallv` collective call. Now that each process is storing full columns of θ samples, these multipole data can be interpolated. The fully-interpolated multipole data is transposed and folded back to its original form (ϕ samples along columns, θ samples along rows). The data are again communicated back to the processes that own the corresponding multipole samples via another `MPI_Alltoallv`. These major steps are illustrated in Figure 3.3.

Shifting of Interpolated Data Shifting of multipole data is simply the translation of the interpolated samples from the child node's center to the parent node's center. Translation of each multipole

data is independent of others and trivially parallelizable even in the case of fine-grained parallel M2M.

Aggregation Aggregation requires adding all corresponding samples from each interpolated and shifted child node together to form the multipole expansion of a parent node (step (g) in Fig. 3.3). When children are owned by separate processes (as is the case for plural nodes), reduction communications are required. Note that in fine-grained parallel M2M for a plural node, each process owns only a portion of the parent node's multipole data. A reduce/scatter operation (for instance using `MPI_Reduce_scatter`) could perform both the aggregation and distribution of the appropriate portions of the aggregated multipole data to the processes sharing a plural node. One complication here is that the reduce/scatter operation would require memory to be allocated to the full-size of the parent node by each user process through padding the parts not owned by a process with 0s. Clearly, this would lead to significant memory and computational overheads, especially at the highest levels of the H-FMM tree (due to the large sizes of the plural nodes there). Therefore, we opt for a custom point-to-point aggregation scheme where the interpolated and shifted multipole samples from child nodes are communicated directly (via `MPI_Send` and `MPI_Recv`s) to the process that owns the corresponding samples of the parent node. If the source and destination are the same process, obviously no communication is performed. Each process sharing the parent node then adds up the corresponding multipole samples it receives from each child node, local or communicated. This method reduces both the amount of temporary memory necessary for aggregation and the overall communication volume.

Process Alignment In fine-grained parallel M2M, performance impact of how the multipole data of child and parent plural nodes are mapped to processes sharing those nodes also needs to be considered. From the description of our custom point-to-point aggregation scheme above, it is evident that increasing the overlap of the multipole sample regions owned by a process in the child and parent nodes is critical for reducing the communication volume. As an extreme example, if a process owns no samples in the parent node that correspond with any of the samples it owns in the

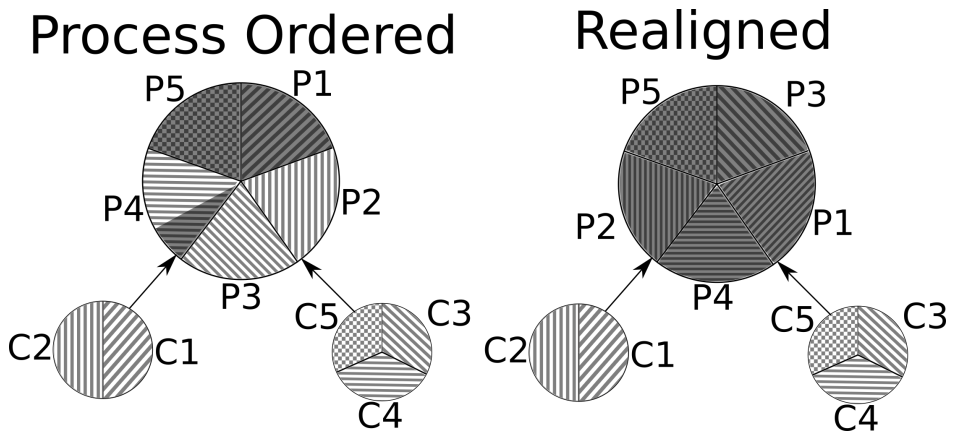

Figure 3.4: This image shows how multipole samples, ordered clockwise starting at 12 o'clock, are assigned to processes owning the children nodes (C1-C5) overlap with processes owning the parent node (P1-P5) when assigned in process rank order on the left, and with our realignment scheme on the right. The darker portions on the parent samples show the regions where the parent node data overlap with the child node data, and are essentially local data that do not require communication.

child node, all its interpolated and shifted child node samples would have to be communicated to another process for aggregation. In fact, this extreme example is not uncommon when multipole data of a node is simply partitioned into blocks and mapped to user processes according to their process ranks. This situation is illustrated on the left side of Fig. 3.4; some processes own samples of a parent node that has no overlap with the samples they own in the child node. Specifically, while process 1 owns overlapping samples in the child and parent nodes, process 2 and 3 own no overlapping samples.

As a heuristic to minimize the communication volume, we order processes within a parent node such that the parent node samples are assigned by following the priorities below to ensure maximal overlap with their child node samples:

1. Index of the lowest sample they own in the child nodes (lower comes first),

2. number of samples they own in the child nodes (fewer comes first),

3. process rank.

In the example given in Fig. 3.4, both process 1 and process 3 own samples with index 0 in the

child nodes, but process 3 has a smaller number of samples so it is assigned the first portion of the parent node samples with process 1 being assigned the second portion. Following processes 3 and 1, process 4 owns the multipole sample with the lowest index in the child nodes, followed by process 2, and then process 5. As such, remaining portions of the parent node samples are assigned in this order. As can be seen in the figure, all samples each process owns in the parent node fully overlap with samples that they own in the child nodes, despite the non-uniform layout. With the proposed process alignment scheme, process 3 will still need to communicate some samples to process 1 for aggregation, but over half of the child samples interpolated by process 3 remains local. Note that with the straight-forward ordering of processes by their ranks, the entire child data interpolated by process 3 would have to be communicated to processes 1 and 2. In the new scheme, all processes use interpolated samples from their part of a child node without having to communicate. While this scheme would work best with a perfectly balanced tree, this approach will still be effective in reducing the communication volume during aggregation with any tree structure.

3.3.2.3 M2L

The M2M step builds the multipole expansions of all tree nodes owned/shared by a process, starting from the leaves all the way up to the root (or the highest level of computation). During M2L each observer node loops through all source nodes in its far-field and translates the source multipole data to its locale, aggregating the effects from all its far-field interactions in the process. When the source-observer node pair is on the same process, this interaction is handled purely locally. However, when the source node data is on another process (or a set of processes), one needs a load balanced algorithm that is communication efficient.

To understand the scope of the problem, consider Figure 3.5. Here, the source node is a plural node shared by three processes (S1, S2 and S3); the observer node is shared by two processes (O1 and O2). Multipole samples for both are shared starting at the top of each circle and increasing clockwise (consistent with the process alignment scheme utilized during M2M). Process S1 and O1 both own multipole samples with the lowest indices, with S1 having less samples than O1. For

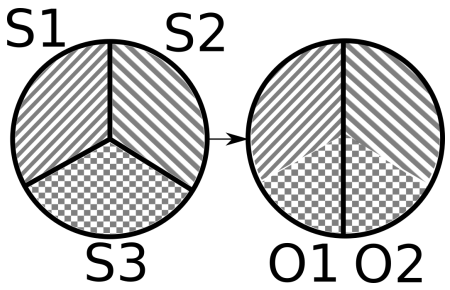

Figure 3.5: A translation operation between two plural nodes shared by different numbers of processes.

this far-field interaction, S1 would need to send all samples that it owns to O1. S3 and O2 both own samples with the highest indices; here S3 would need to send all of its samples to O2. Finally, S2 owns samples that are needed by both O1 and O2, therefore S2 must send half of its samples to O1 and the other half to O2. Since each node in the H-FMM tree interacts with several others ($\approx$27 for surfaces and up to 189 for volumes) each of which may be shared by a varying number of different processes, it is evident that coordination of all communications that must take place during an H-FMM evaluation is non-trivial.

In an initialization step before the actual M2L operations, all processes discover the owner process(es) of the tree nodes (i.e., observers) which will need the multipole data for the source nodes they own, given the partitioning of leaf nodes (for load balancing purposes) and process alignments for plural nodes. This pre-calculated list is formed to carry out the actual communications that will take place during the ensuing H-FMM potential evaluations. If a source node or the corresponding observer node is plural, then the pre-calculated communication list will include only the intersection of the multipole data owned by both the source and target processes. When a source node on a process is needed by multiple observers on another process, it is sufficient to include the source node in the communication list to that other process only once. Also, multipole data for multiple source nodes residing on one process that are needed by another can be combined into a single message in this communication list, even if the source nodes are at different levels. Note that the tree structure in most H-FMM applications are fixed. As such, the overheads associated with such an initialization stage is minimal.

Algorithm 3.2 M2L Translation

```
1: Determine the intersection of data owned by both source and target.
2: procedure M2L
3:     for each source box do
4:         for each farfield interaction target do
5:             if Target box is not local then
6:                 for each process sharing target box do
7:                     Add source multipole data in union of data owned by both source and target
   to buffer.
8:                 end for
9:             end if
10:        end for
11:    end for
12:    Communicate buffers between processes.
13:    for each target box do
14:        for each farfield interaction source do
15:            if Source box is local then
16:                Load source multipole data from local memory.
17:                Translate source multipole data to target and add to existing translated data for
   target node
18:            else
19:                for each process sharing source box do
20:                    Read source multipole data from communication.
21:                    Translate source multipole data to target and add to existing translated data
   for target node
22:                end for
23:            end if
24:        end for
25:    end for
26: end procedure
```

While plural node to plural node far field interactions (which constitute the most expensive communications in an H-FMM evaluation) could actually be carried out using all-to-all communications that involve only the users of the two corresponding plural nodes, due to the excessive number of M2L interactions present in large-scale computations (and hence the large number of different communicators that must be created), we choose to perform these communications using non-blocking point-to-point send/recv operations (i.e., MPI-Isends and MPI-Irecvs) in the default global communicator. Another reason for opting for a point-to-point scheme is that there are significant differences in the amount of data that must be sent to one process compared to another. As

part of the initialization step then, each process allocates a message buffer for every other process that it will communicate with. The size of the send buffer is limited to avoid excessive memory use and maximize communication overlap.

To perform communications during M2L, every process first fills their send buffers for each process based on the pre-calculated communication list and initiates the message transmission using `MPI_Isend`s. Immediately after initiating the sends, each process starts waiting for their expected messages using `MPI_Irecv`s. The status of these communications are checked periodically. Computations are overlapped with M2L communications in two ways. First, blocks of translations that are entirely local (which is actually common at the lower tree levels) are processed while the non-blocking send/recvs are taking place. Second, translation data that is detected as received during the periodic checks are applied immediately, overlapping the corresponding computational task with communications underway. Due to the limit we impose on the message buffer sizes, communications with processes that involve a large amount of data transfer need to be performed in multiple phases. Hence, upon reception/delivery of a message from another process, if there is more data to be transferred, a new non-blocking recv/send operation is initiated.

Translation Operators Source node data is translated to the target node by multiplying it with a translation operator. The translation operators can be pre-calculated to reduce computational costs. As these can potentially take significant memory, we limit such memory use by each process by having them store the pre-calculated operators only for translation of the local and remote samples that they will actually need. This information can be determined from the pre-calculated M2L communication list. In case the memory available is not sufficient to store the needed operators, we use techniques outlined in Hughey et al. (2019) to sample and interpolate translation operators.

3.3.2.4 L2L

To anterpolate and distribute the translated local expansions down to the child nodes, L2L applies the operations in M2M in reverse order. First, local expansions at the parent node are shifted to

the center of each child node, they are then anterpolated and percolated down the tree. Finally, the anterpolated data is aggregated with local expansions previously translated to the child node during the M2L stage.

Much like M2M, L2L operation is parallelized in three different ways: i) Non-plural parent tree nodes at the lower tree levels are processed independently in parallel by their owner processes, ii) for a plural node with a non-plural child, shifts involve communications, but the ensuing anterpolation and aggregation (with translated local expansions) are performed sequentially by the process owning the child node, iii) plural nodes with plural children require fine-grained parallelization.

Shift In a parallel shift operation, parent node samples corresponding to those of the child node must be communicated by the processes sharing the parent node to the process(es) owning the child node. This is most easily done before the data has been shifted, as the parent will not need to know the position of the child node. In case of plural parent and non-plural child, this communication would essentially be a *gather*, and in case of plural parent with a plural child, it would be an *all-to-all*, in both cases involving all processes sharing the parent node. However, only a subset of the processes sharing the parent node will actually share the child node. To avoid non-trivial issues that would arise from having to coordinate several collective calls among different subsets of processes, we again resort to point-to-point communications instead. Consequently, messages are only sent from processes owning a piece of the parent node to a process owning the corresponding piece of the child node. Once a process gathers all of necessary samples of the parent node, it applies the shift operation to all its multipole samples independently.

Anterpolation Anterpolation would have to be performed in parallel only if the child node is a plural node. The procedure for parallel anterpolation is exactly the same as that of the parallel interpolation, except that the number of multipole samples is reduced (rather than increased).

Aggregation Aggregating the shifted and anterpolated parent data with translated local expansions is trivial. Even in the case of plural child nodes, all required data is already available

locally.

3.3.2.5 L2O

As in C2M, each process handles the L2O computations of its assigned leaf nodes in parallel independently.

3.4 Numerical Methods

In this section, we analyze the asymptotic computational and communication complexity of the parallel H-FMM algorithm described above. To simplify the analysis, we focus on two extreme cases, a 2D surface represented by points on a regularly spaced planar grid (dimension d=2) and a 3D volumetric structure represented by points on a regularly spaced cubic grid (d = 3). These represent extreme cases, and hence are ideal for asymptotic analysis.

Let $K(l)$ denote the number of samples in the θ and ϕ directions for a node at level l. All FMM algorithms are constructed such that the only operators that depend on particles are C2M and L2O, the operations of M2M, M2L and L2L only depend on the existence of the leaf node Greengard et al. (1998); Coifman et al. (1993b); Ergin et al. (1998, 1999). As such we assume that each leaf node contains $\bar{O}(1)$ samples. It follows that the number of leaf nodes is $\propto N_s$, the number of source points. For simplicity and with no loss in generality, we assume that the constant of proportionality is 1. Next, we denote the number of nodes at level l by $G(l)$. The total number of levels is given by N_L. As one moves up the octree, we observe that the number of groups per level is reduced by roughly 4 times for the 2D surface and 8 times for the 3D volume. Leveraging the relation between the dimensionality of a structure and the rate of decrease in the number of nodes per level, one can write $G(l)$ as follows:

$$G(l) = \frac{N_s}{(2^d)^{(l-1)}}. \tag{3.1}$$

Since $K(l)$ doubles at each level, given $K(1) = C_k$, it follows that

$$K(l) = 2^{(l-1)} C_k. \tag{3.2}$$

Finally, we define P as the number of processes, P_L as the level where (almost) all nodes in a level start becoming plural and $P_N(l)$ as the average number of processes sharing a plural node at level l. Equivalently, P_L is the level when $G(P_L) < P$ for the first time, and this remains true from hereon to the root. Given P, N_s and d,

$$P_N(l) = \frac{P}{G(l)} = \frac{P(2^d)^{(l-1)}}{N_s} \tag{3.3}$$

3.4.1 Interpolation (M2M)

Computational Complexity M2M is performed for each node, starting from the leaf level up to the highest level N_L. The dominant component in the computational complexity for $M2M$ is FFT-based interpolation. Shifting and aggregation are $O(K(l)^2)$ operations each, while interpolation for a given node costs $O(K(l)^2 \log^2(K(l)))$. This gives a total computational complexity of

$$C \propto \sum_{l=1}^{N_L} G(l)K(l)^2 \log^2(K(l)) \tag{3.4}$$

Plugging in the equations (3.1) and (3.2) and simplifying the summation, we obtain the computational complexity for a surface to be:

$$C \propto O(N_s \log^2 N_s), \tag{3.5}$$

and for a volume to be:

$$C \propto O(N_s) \tag{3.6}$$

Number of Messages (Latency) Communication in $M2M$ happens during aggregations for both coarse-grained and fine-grained parallel $M2M$s, as well as the FFTs of the fine-grained parallel $M2M$s. As described in Sect. 3.3.2.5, we perform aggregations (which are effectively reduce-scatter operations) using point-to-point communications. In an ideal tree, every source and observer node

will be divided among the same number of nodes. This means the portion of a source node owned by any process will only be owned by a single process in the observer node, limiting the communication for each source node process to one process in each observer node. Since this is done for each group at each level, the total message count for aggregations can be written as:

$$M_{Ag} \propto \sum_{P_L}^{N_L} G(l) P_N(l+1). \tag{3.7}$$

Using expressions for $P_N(l+1)$ and $G(l)$, yields the number of messages for aggregation

$$M_{Ag} = O(P \log(N_S) 2^d). \tag{3.8}$$

Here, we ignore aggregations that would be needed for plural nodes (located at process boundaries) below level P_L. Note that there may only be two such plural nodes per level for each process and these aggregations will involve only two processes. As such, their contribution to the number of messages during aggregations is of a lower order term.

Next, consider the parallel FFTs in fine-grained parallel $M2M$s. In this case, an all-to-all communication is performed after each of the two fold and transpose operations. As we implement these all-to-all communications using point-to-point calls, the message count for FFTs is then:

$$M_{FFT} = \sum_{l=P_L}^{N_L} G(l) P_N(l)^2 \propto O(P^2). \tag{3.9}$$

Consequently, the total number of messages for $M2M$ is:

$$M_{M2M} = O(P^2 + P \log(N_s)). \tag{3.10}$$

Note, the N_s portion of the equation above is only going to matter when P_L is greater than the number of levels in the tree. In all other cases, increasing the height of the tree does not increase the number of levels with plural nodes. Given that it is practically useless to have more processes than the number of leaf nodes (which is the condition required for P_L to be more than the tree height), the message count can be simplified to $M_{M2M} = O(P^2)$.

Communication Volume (Bandwidth) Bandwidth during interpolation is due to all to all communications during interpolation, and a reduce scatter during the aggregation. Each of these operation communicates up to the entire node, resulting in a bandwidth that can be written as:

$$B \propto \sum_{l=1}^{N_L} G(l)K(l)^2 \tag{3.11}$$

Applying the previous definitions for $G(l)$ and $K(l)$ yields a communication bandwidth of $B \propto N_s \log N_s$ for the surface geometry, and $B \propto N_s$ for the volume geometry.

3.4.2 Translation (M2L)

Computational Complexity The complexity for the translation operation at a given level is directly proportional to the number of multipole samples for nodes, the average number of interactions per node (denoted by $I(l)$ for level l), and the number of nodes at that level. Summing these costs across all levels, we obtain:

$$C \propto \sum_{l=1}^{N_L} K(l)^2 I(l) G(l). \tag{3.12}$$

While the number of interactions for a node changes based on its exact position in the geometry (for instance, corner or edge nodes), the upper limit is the constant $6^d - 3^d$. Using the equations for $K(l)$ and $G(l)$, computational complexity of the translation step can be simplified to $O(N_s \log N_s)$ for the surface structure, and to $O(N_s)$ for the volume structure.

Number of Messages (Latency) At level P_L or above, a process can have multipole samples for only one node. Since a process owns at most part of a single node, each of its interactions will require a separate communication because the nodes in its far-field will all reside on different processes. Assuming an ideal tree partitioning where the source and target nodes are shared among the same number of processes, the kth process for the target node will only need the source node data from the kth process of the source node. As we limit the size of each translation message, the number of messages will then be proportional to the communication volume between a pair of processes divided by the message buffer size M_S. At levels below P_L, a process can own multiple

nodes. Here groups of nodes can be communicated to the same process, if all source nodes reside on one process and all observer nodes reside on another. In this case, the interaction count is going to be based on the total amount of data communicated between the two interacting processes, divided by the message buffer size, summed up for all interacting processes.

Considering contributions at/above P_L and below P_L gives a total message count of:

$$M \propto \sum_{l=P_L}^{N_L} PI(l)\frac{K(l)^2}{P_N M_S} + PI(l)\lceil\frac{\sum_{l=1}^{P_L-1} K(l)^2\frac{G(l)}{P}}{M_S}\rceil \tag{3.13}$$

where M_S is the size of message buffers. For the surface geometry, this can be simplified to $M \propto O(N_s \log N_s) + O(N_S)$ (where the first term is for levels $\geq P_L$ and the second term is for levels $< P_L$), and for the volume geometry it can be simplified to $M \propto O(N_s)$ (with both below and above P_L having the same impact).

Communication Volume (Bandwidth) Similarly, communication volume can be analyzed in two parts as well. At and above P_L, all multipole data for every source node must essentially be communicated to every target node as no process contains any multipole data other than its own. Even if the number of processes increases, still the same amount of data needs to be transmitted, just among an increased number of nodes. Therefore, for level at or above P_L, the communication volume is independent of the number of processes:

$$B \propto \sum_{l=P_L}^{N_L} K(l)^2 G(l) I(l) \tag{3.14}$$

This expression simplifies to $O(N_s \log N_s)$ for the surface geometry and to $O(N_s)$ for the volume geometry.

Below P_L, each process will own more than one node, nodes will be interacting with nodes on the same process, or multiple nodes owned by a neighboring process. In fact, only nodes within two nodes off the edge of process boundaries will require communications with other processes. Total communication bandwidth can then be expressed as:

$$B \propto \sum_{l=1}^{P_L-1} PK(l)^2(S_N) \tag{3.15}$$

where S_N is the number of nodes that have nodes in its far-field from at least one (out of the 8 possible neighboring processes for the surface and 26 for the volume) other processes touching them and can be represented as $S_N = \frac{G(l)}{P}^{\frac{d-1}{d}} = \frac{1}{P}(\frac{N_s}{(2^d)^{(l-1)}})^{\frac{d-1}{d}}$. With this definition of S_N, the total communication volume for M2L below P_L becomes

$$B \propto O(N_S) \tag{3.16}$$

for the surface, and

$$B \propto O(N_S^{\frac{2}{3}}) \tag{3.17}$$

for the volume, due to the lower portion of the tree dominating.

3.4.3 Anterpolation (L2L)

As mentioned before, L2L is the reverse operation for M2M. Similar to M2M, anterpolation dominates the computational complexity for L2L. Computational complexity for anterpolations is the same as that of interpolations, so L2L's computation complexity is the same as M2M's. Likewise, communications performed are the same but in reverse order. Therefore, the latency and bandwidth costs of L2L are the same as those of M2M.

3.5 Numerical Results and Performance Evaluation

In this section, we evaluate the performance of the parallel H-FMM algorithm described. All results were obtained on the Cori-Haswell supercomputer at National Energy Research Scientific Computing Center (NERSC). Each node on this system contains two sockets, populated by Intel Xeon E5-2698 v3 (Haswell) processors with a clock speed of 2.3 GHz. Each node has 32 cores and 128 GB 2133MHz DDR4 RAM. The Haswell system uses the Cray Aries with Dragonfly topology interconnect network. The code is implemented in Fortran 90 using only MPI parallelization and was compiled with the Intel compiler, version 19.0.3.199. The Cray-FFTW library, version 3.3.8.4, is used for all FFT operations.

The runs here focus on the timing of the M2M, M2L and L2L phases of the tree traversal. As discussed in section 3.4, these operations of these phases only depend on the existence of the leaf node. Thus, each leaf node is only populated with a single unknown, effectively bypassing the near-field, C2M and L2O processing steps. This also makes the number of unknowns being processed much smaller than what could be processed by M2M, M2L and L2L in the same amount of time when analyzing actual physics as in Hughey et al. (2019). In a typical 0.25λ leaf box (as we use in the runs below) with a $0.1\ \lambda$ discretization rate, one could potentially have anywhere between 100-180 particle per box. Our largest tree being processed is 14 levels with 42 million points for one point per leaf box. If the leaf nodes were fully populated, this tree would be equivalent to processing a tree with 4.2 to 7.5 billion points. Populating leaf nodes would increase the time of the C2M and L2O steps, but have no impact on the execution times of the M2M, M2L and L2L phases. Additionally all runs use the same θ and ϕ discretization as set by the 0.25λ leaf box size and a $\chi = 1.1$, see Lingg et al. (2018); Vikram et al. (2011) for use of χ

For all runs testing an increasing number of processes, the first run is performed with the lowest number of nodes that can execute the geometry without an out of memory exception using 32 processes per node (1 process per core). Processes are assigned to cores using srun, with –cpus-per-task set to 1 and –nodes set to the number of processes divided by 32. The number of processes are increased by increasing the number of nodes used, while maintaining 32 processes per node.

Numerical results from all runs were compared against results with our previous work Hughey et al. (2019) to verify the only differences were due to floating point precision, and the previous work showed the results to be error controllable with respect to the analytical solution.

3.5.1 Load Balance with the Fine Grain Parallel Algorithm

The intent of the fine-grain parallel algorithm is to provide improved balance at the upper levels of the tree where a lower number of much larger nodes reside. First, we look at the performance of a planar grid of particles (in the $z = 0$ plane) of dimensions $512\lambda \times 512\lambda$ with a grid spacing of $\lambda/4$ and 4,194,304 particles in total. The box size is chosen to be 0.25λ, resulting in a 12-level tree with

10 levels of computation. As can be seen in the left subfigure of Fig. 3.6, the resulting execution profile is very balanced across process ranks. Execution time of the fastest to the slowest process varies by only 1.43%. Balance of total time can be a little misleading as M2L cannot progress until all processes that a given process interacts with have completed their M2M processing. However, the M2M execution times are also very balanced, varying from slowest to fastest process by 5.23%.

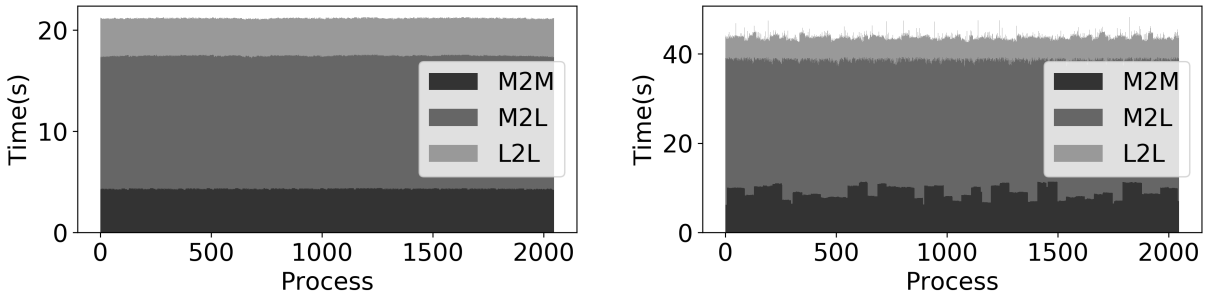

Figure 3.6: Process execution times for a grid and a sphere geometry.

Next, we look at the performance for a sphere of diameter 384λ discretized using 4,542,208 dipoles on the surface with a leaf box size of $d_0 = 0.25\lambda$, yielding an 11-level tree. This geometry is less balanced than the grid geometry (see the right subfigure of Fig. 3.6) as high level nodes can range from having no children due to no particles being in that part of the geometry at the leaf level, up to having a completely filled quad tree from the leaf level up to a high level node. This results in notable imbalance at the M2M level, which as discussed before, results in delays in the M2L execution. Note that while there are no explicit barriers between the phases, there is an implicit barrier at the beginning of M2L processing, as an M2L interaction communication cannot proceed until both interacting processes have completed their M2M phases (though the faster process can perform local translations while waiting). Another (less significant) implicit barrier occurs at the beginning of L2L where nodes that are fully owned by a process must have all of the data from the translated parent node to perform anterpolation on this parent node. The M2M execution range from the fastest to the slowest processes varies by as much as 84.5%. Despite this noticeable imbalance, the long execution times are not clustered among a small group of processes as one would see if each of the highest level tree nodes were to be handled by a single process without

fine-grain parallelization. So even in this unbalanced geometry, the fine-grain parallel algorithm is helping to maintain a good load balance across process ranks.

3.5.2 Scalability

Next, we investigate the strong scaling efficiency of our parallel Helmholtz FMM algorithm, first on a surface and then on a volumetric structure.

For the 2D surface structure, we use the same $512\lambda \times 512\lambda$ planar grid as above. As our base case for strong scaling efficiency, we use the performance on 128 cores because this is the smallest number of cores that this problem can be executed on due to its memory requirements. As seen in Table 3.1, both the interpolation and anterpolation phases (M2M and L2L) perform very well with the increasing process counts, while M2L's performance falls off rapidly (down to 25% efficiency on 2048 cores). There are a couple of factors that contribute to this difference we observe in scaling characteristics. First factor is that M2M and L2L incur significant communications only at the highest level nodes, while M2L communications occur at every level where the source and observer nodes are on separate processes, which may essentially happen all the way down to the leaf nodes. Secondly, and more importantly, M2M and L2L computations involve relatively computation-heavy FFTs in between its communication steps. When the number of nodes in a level exceeds the number of processes, no process can own both a source and observer node of any translation, so all node data must be communicated. As the number of processes approaches the number of leaf nodes, the M2L communication bandwidth asymptotically approaches the worst case estimate. This means the increase in M2L bandwidth exceeds the worst case estimate increase as the number of processes approaches the number of leaf nodes. Despite M2L not scaling very well, the fine grained parallel algorithm presented still provides good speedups, nearly an 8x speedup when going from 128 to 2048 processes without showing any performance stagnation.

Next, we examine strong scaling on a $32\lambda \times 32\lambda \times 32\lambda$ volumetric structure (Table 3.2). From 128 to 512 processes, we observe very good scaling (80% overall efficiency), but then parallel efficiency drops off quickly (down to 50% overall at 2048 cores). In a volumetric problem, each

	Grid (s)				**Speedup**	**Par Eff.** (%)			
N_p	M2M	M2L	L2L	Tot	Tot	M2M	M2L	L2L	Tot
128	5.80	5.30	5.30	18.55	1.00	1.00	1.00	1.00	1.00
256	3.06	4.13	2.66	11.21	1.65	0.95	0.64	0.99	0.83
512	1.52	2.76	1.31	6.42	2.89	0.95	0.48	1.01	0.72
1024	0.81	2.12	0.69	4.05	4.58	0.89	0.31	0.95	0.57
2048	0.43	1.31	0.37	2.38	7.78	0.84	0.25	0.89	0.49

Table 3.1: Performance of the MLFMA algorithm on the 512λ grid geometry.

	Volume (s)				**Speedup**	**Par Eff.** (%)			
N_p	M2M	M2L	L2L	Tot	Tot	M2M	M2L	L2L	Tot
128	0.527	2.15	0.526	3.26	1.00	1.00	1.00	1.00	1.00
256	0.266	1.10	0.263	1.68	1.94	0.99	0.97	0.99	0.97
512	0.14	0.679	0.135	0.99	3.27	0.93	0.79	0.97	0.82
1024	0.079	0.380	0.084	0.574	5.68	0.83	0.71	0.78	0.71
2048	0.051	0.271	0.058	0.406	8.03	0.65	0.50	0.57	0.50

Table 3.2: Performance of the MLFMA algorithm on the 32λ volumetric geometry.

tree node has a large number of nodes in its far-field (up to 189). Therefore the overall execution time is largely dominated by the M2L stage which does not manifest good scaling. The ideal scenario for our fine-grained parallel algorithm is when the nodes of a given level are distributed evenly among the processes, i.e., when the number of processes divides evenly into the number of nodes in a level or vice versa. This does not occur at 1024 or 2048 processes for this particular volumetric problem. Nevertheless, the overall speedup remains at around 8x when going from 128 to 2048 processes.

Finally, we look at scaling on the 384λ sphere (Table 3.3). As seen in the load balance analysis of the previous subsection, load imbalances result in the faster processes having to wait for slower processes. This results in a noticeable drop in scaling efficiency of the M2M phase, where the imbalance has the greatest impact, as well as the M2L phase, where some processes that are already in their M2L phase have to wait for others that are still in their M2M phase. This also has an impact

	Sphere (s)				**Speedup**	**Par Eff.** (%)			
N_p	M2M	M2L	L2L	Tot	Tot	M2M	M2L	L2L	Tot
128	6.71	13.54	5.29	26.76	1.00	1.00	1.00	1.00	1.00
256	3.86	10.05	2.7	18.00	1.49	0.87	0.67	0.97	0.74
512	2.34	6.20	1.46	11.04	2.42	0.72	0.55	0.91	0.61
1024	1.23	4.32	0.72	7.70	3.47	0.68	0.39	0.92	0.43
2048	0.92	3.19	0.416	5.58	4.79	0.46	0.26	0.79	0.30

Table 3.3: Performance of the MLFMA algorithm on the 384λ diameter sphere geometry.

on the overall speedup. While increasing the number of processes continues to improve execution times, the speedup when going from 128 to 2048 processes is just under 5x in the sphere case.

3.5.3 Complexity Analysis

To help validate the complexity analysis presented in Sect. 3.4, the software was instrumented to report the computational cost, the number of messages sent and the size of these messages. In accordance with the geometries analyzed in Sect. 3.4, data was collected on the grid geometries ranging from 64λ to 1024λ and volume geometries ranging from 16λ to 16λ to as these geometries produce perfect quadtrees of heights ranging from 9 to 13 levels and octrees of heights ranging from 7 to 11 levels, respectively. As complexity estimates are asymptotic, they are scaled by least-squares fit to help visualize how well the estimates match the actual measurements.

Figure 3.7 shows the actual vs. the estimated overall computational complexities for the surface and volume geometries. The actual complexities match the estimates very closely. This indicates that the implementation of this algorithm does not have any unnecessary overhead costs in computation as computation is near to the ideal for Helmholtz FMM.

Figure 3.8 shows the actual vs the estimated communication volumes for each phase separately. Of note is how the measured communication volume drops off relative to the estimate. We believe this is due to the number of samples producing a tree with more nodes at lower levels than the number of processes. Hence, many nodes are fully owned by a single process and require no

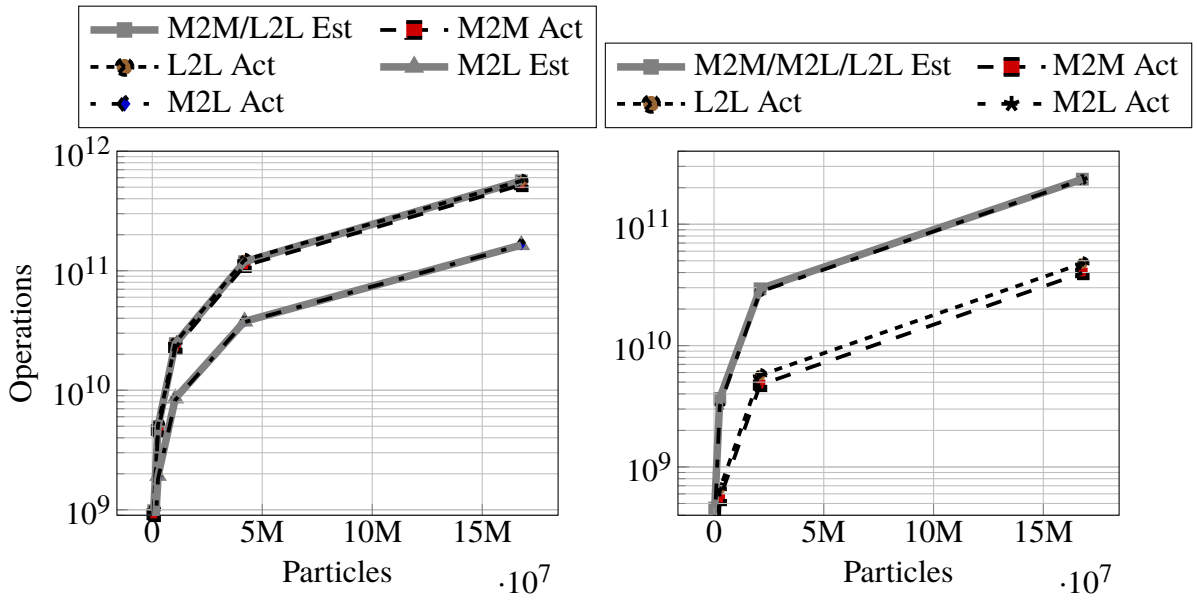

Figure 3.7: Actual vs. estimated computational complexity. The left subfigure shows results for the surface geometry, while the right subfigure is for the volume geometry.

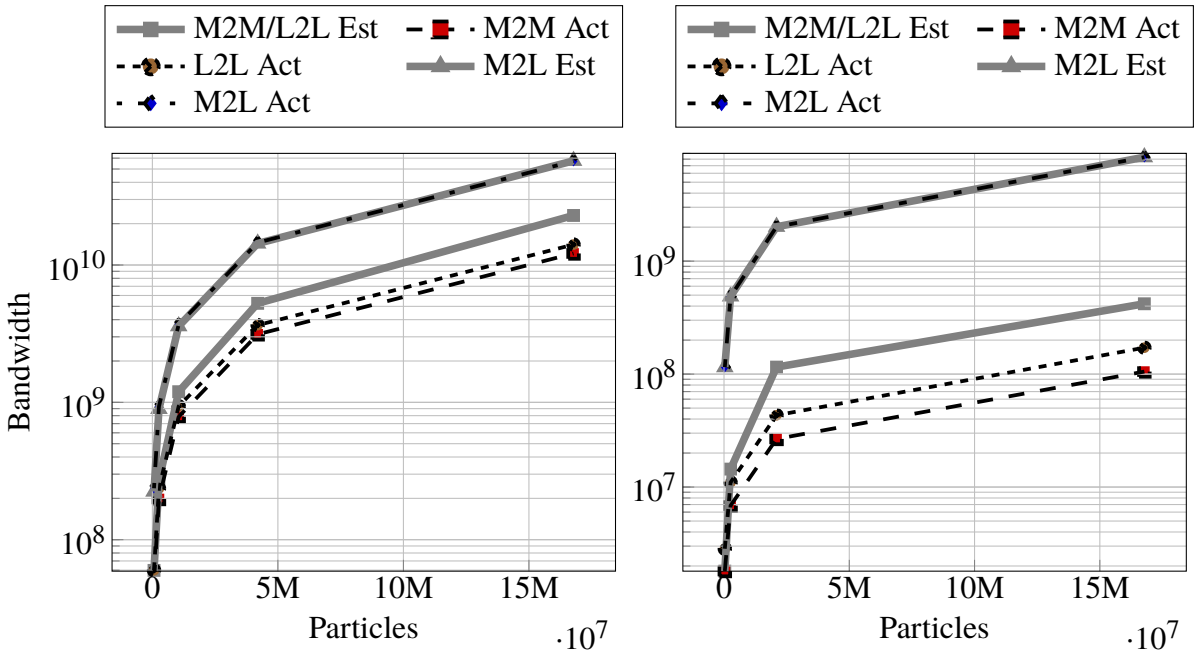

Figure 3.8: Actual vs estimated communication volume. The left subfigure shows results for the surface geometry, while the right subfigure is for the volume geometry.

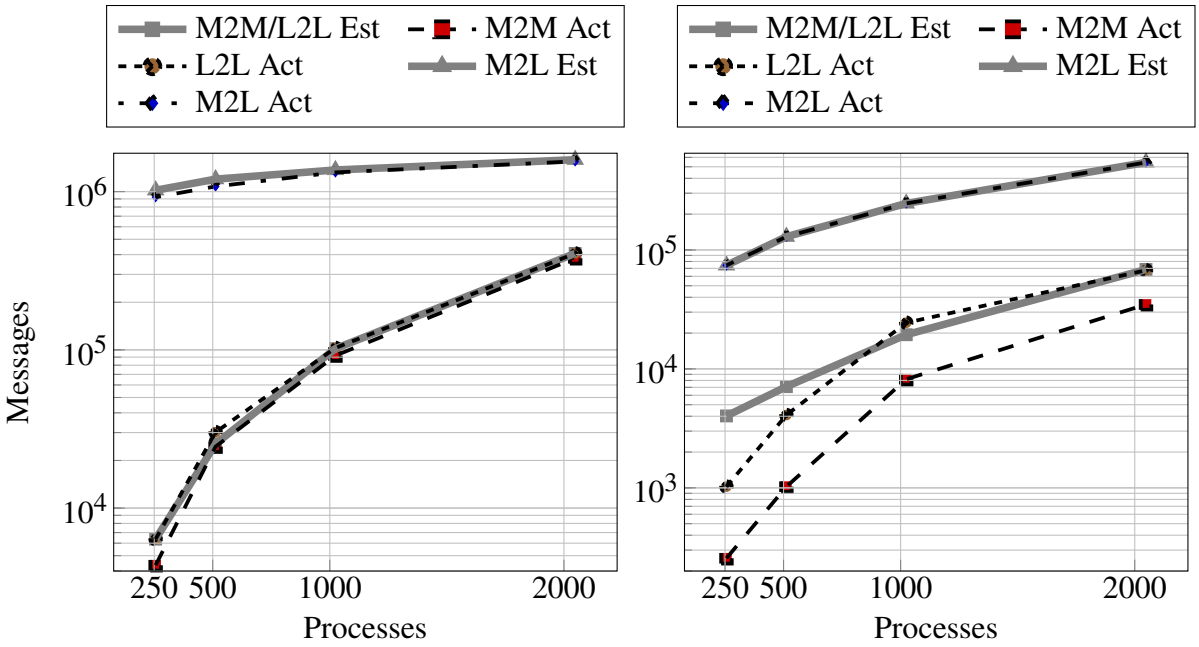

Figure 3.9: Message counts vs expected Big-O message counts. The left diagram shows analysis of a surface geometry, while the right diagram shows analysis of a volume geometry.

communication during M2M or L2L. Increasing the number of processes would lead to more levels with plural nodes, bringing the communication volume closer to our estimates. M2L does not show the same communication volume falloff as M2M and L2L compared to the estimated volume because fully owned nodes still require data from the source nodes to be communicated to the process owning the observer node. Such communications will be required all the way down to the leaf nodes.

Figure 3.9 shows the measured worst case messages vs the estimated worst case messages for M2M and L2L. M2M and L2L Message counts are dominated by the P^2 complexity of the all to all communications and the actual message count reflects this. The M2L prediction simplifies a very complex process that results in the number of M2L messages that are communicated.

3.5.4 Process Alignment

In Table 3.4, we compare the number of packets sent between Rank Ordered and Process Aligned schemes during M2M and L2L phases for the 512λ grid geometry. We observe a notable reduction

N_P		128	256	512	1024	2048
Rank Ordered	M2M Bandwidth	1714	1872	3009	3126	4246
	L2L Bandwidth	1206	1166	2429	2382	3649
	Combined Bandwidth	2920	3038	5438	5508	7895
Process Aligned	M2M Bandwidth	1468	1662	2711	3272	4412
	L2L Bandwidth	960	955	2131	2104	3338
	Combined Bandwidth	2428	2617	4842	5376	7750
Delta		-492	-421	-596	-132	-145

Table 3.4: Comparison of the number of packets sent between Rank Ordered and Process Aligned schemes for the 512λ grid geometry in millions of packets sent.

N_p	S/R Buffs	Trans Ops	Tree Mem
128	1.7	107.5	52.7
256	4.5	141.0	62.0
512	102.7	162.2	52.8
1024	257.0	199.0	62.0
2048	431.8	221.4	52.8

Table 3.5: Total memory utilization (in GBs) by the three largest data structures for the 512λ grid geometry.

in the number of messages exchanged, and hence the overall bandwidth, for lower process counts and continued reduction at higher process counts as expected. This reduction is likely to be effective in the relatively good scaling characteristics of M2M and L2L phases.

3.5.5 Memory Utilization

Table 3.5 shows the total program memory utilization of the three data structures with largest memory needs with increasing process counts. As expected, the memory used for tree storage (Tree Mem) does not increase with process count, despite some fluctuations due to different partitionings of the leaf nodes. This shows that the tree data structure is being nicely partitioned across processes. Size of the translation operators (Trans Ops) increase slowly with process count,

N_p	S/R Buffs	Trans Ops	Tree Mem
128	3.1	6.6	5.0
256	8.4	10.1	5.2
512	22.3	16.3	5.5
1024	31.9	21.4	5.1
2048	51.7	30.4	5.4

Table 3.6: Total memory utilization (in GBs) by the three largest data structures for the 32λ volume geometry.

slightly more than doubling going from 128 to 2048 processes. This is due to the spatial distribution of the tree nodes; multiple source observer pairs with the same translation in the tree may belong to different processes. Particularly, as the process count increases and the number of nodes in a process decreases. This results in some processes storing some of the same translation operators as the other processes. The greatest memory increase is in the message buffers (S/R Buffs). The translation send and receive buffers (S/R Buffs) are used to communicate the data for source nodes that interact with nodes in another process. Single node communications for each source and observer pair would eliminate the need for this buffer, but would result in drastically more translation messages which would degrade performance. So the translation message buffers are maximized to use any remaining memory to limit the number of translation messages that must be sent.

On the other end of what can be performed with H-FMM is the volume geometry. Here the number of nodes per level is significantly increased due to the underlying full oct-tree structure (as opposed to a quad tree for a surface geometry), but the maximum height of a tree that can be computed is reduced. Most memory is reduced due to the shorter height of the tree, which reduces the size of the nodes at the top of the tree. However, the translation message buffers still use up as much memory as possible to improve translation communication performance.

Processes	2	4	8	16	32	64	128	256	512
BEMFMM (s)	108.92	100.85	148.69	91.03	60.45	35.72	29.42	26.72	24.72
This work (s)	8.29	4.28	2.37	1.26	0.68	0.39	0.23	0.17	0.15
Speedup	13.1	23.6	62.7	72.2	88.9	91.6	127.9	157.1	164.8

Table 3.7: Comparison of BEMFMM vs our parallel MLFMA implementation (referred to as "this work").

3.5.6 Performance Comparison with Other Codes

Finally, we seek to compare our approach against those code that are available in the public domain. We note that open-source H-FMM codes are almost non-existent, with Abduljabbar et al. Abduljabbar et al. (2019) being a recent exception. In their 2019 paper, Abduljabbar et al. (2019), BEMFMM is tested with a 1 meter sphere, up to 17.9 λ and 2.3 billion unknowns, which suggests a 7 or 8 level tree, with 5 or 6 levels of computation. Our code has been run on a 14 level tree with 12 levels of computation, which covers geometries up to 2048 λ. If the leaf nodes were fully populated, this tree would be equivalent to processing a tree with 4.2 to 7.5 billion points. Their BEMFMM code discretizes a mesh; the discretized points are used as particle inputs to our H-FMM code in order to compare processing of the same geometry. We ran a spherical geometry with 240 thousand mesh elements that produces 1.44 million points. Both codes are configured to produce an 8 level tree with exactly 65471 leaf nodes with this sphere geometry. With this configuration, our fine grain parallel Helmholtz FMM algorithm shows significantly faster performance in comparison as seen in Table 3.7.

3.6 Conclusions and Future Work

We have demonstrated a novel method for parallel computation of large, upper level tree nodes in large-scale Helmholtz FMM which helps alleviate a key performance bottleneck associated with node dependency. The complexity of this method has been characterized. The results presented support the provided characterization and show the balance provided by this method. The performance of the algorithm has been shown to compare favorably with an existing Helmholtz

FMM implementation.

Beyond the improvements presented, further work can be performed to improve memory usage, as well as the M2L communication overhead. One possible method is a hybrid approach of MPI parallel with thread parallel. Using thread parallel within a given node provides the opportunity to exploit shared memory parallelism. All cores within a node can be assigned to shared memory threads, rather than MPI processes, eliminating the need to communicate between these threads, and reducing duplicate memory allocation.

CHAPTER 4

EXPLORING TASK PARALLELISM FOR THE MULTILEVEL FASTMULTIPOLE ALGORITHM

4.1 Introduction

In the previous chapter, we showed how using a fine grain parallel distribution of the larger, higher level nodes helps to balance the overall work of traversing the tree. Traversal up and down (M2M and L2L) the tree showed excellent scalability, while translating the node data (M2L) quickly saw reduced performance as the number of processes increased. An analysis of the performance of translations found that computation scaled well as the number of processes increased, but adding more processes did not lead to a reduction in communication time as process counts increased to one thousand and beyond.

One way of reducing M2L communications is to partition data among multiple processes, such that each process owns the same range of samples for all nodes at a given level, which Yang et al. (2019) call plane wave partitioning. Figure 4.1 shows how direction partitioning splits each node between unique subsets of processes, while plane wave partitioning divides each node up between all processes equally. This partitioning method eliminates the need to communicate any data while processing translations. With local interpolation, this method can perform well because there is limited impact to interpolation and anterpolation. In our chosen method of global interpolation, the entire node data is needed for interpolation and anterpolation. As a result, this data partitioning will perform more poorly when using global interpolation, as each process will have to communicate with all other processes during interpolation and anterpolation, rather than a subset of other processes.

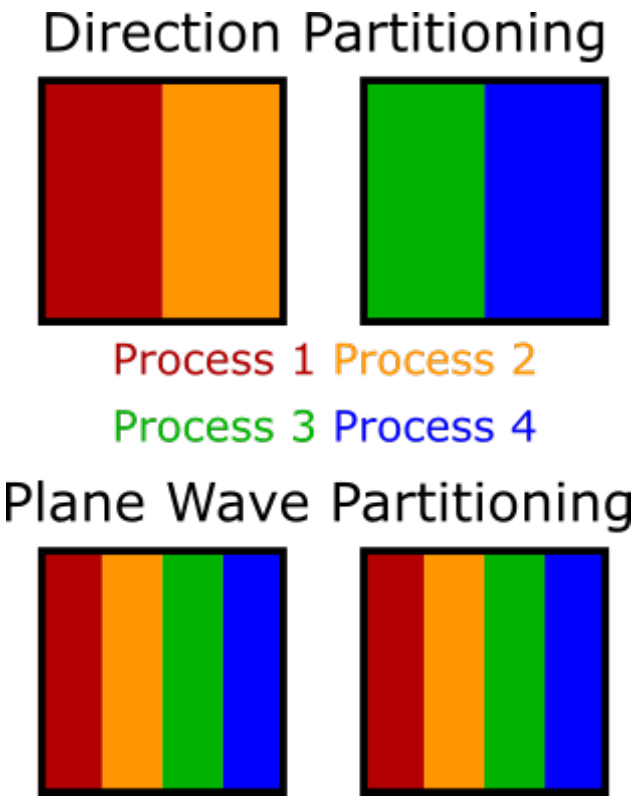

Figure 4.1: Direction Vs Plane Wave Partitioning.

Rather than changing how the node data is partitioned to reduce translation communication, we can instead look into maximizing the number of nodes stored in a given process. In a pure MPI scheme, increasing the number of processes reduces the work for each process, but at the same time, the number of source and observer node pairs residing on the same process decreases. Each source and observer node pair that reside in the memory of the same process requires no communication. Thus if we can divide up the workload further, while keeping a higher number of source and observer node pairs in the memory of the same process, we can reduce communication costs.

Shared memory parallel adds an excellent complement to the MPI parallel approach. While assigning each core of a node maximizes the amount of physical processing power, it also means processes running on the same core must communicate with each other to share data. Instead, a shared memory parallel scheme can be used where each core within a node executes a separate thread, and each MPI process is assigned to a separate node. Now the shared memory threads can

process all of the nodes that previously resided on separate cores for a given node, without any communications necessary. Figure 4.2 illustrates how M2L interactions, when partitioning 16 cores each to a separate process, with the red node would require communicating with 8 other processes. While partitioning the 16 cores to 16 threads, and 1 process, would not require any communication, as all nodes are stored in shared memory. In this chapter we compare two common methods of shared memory parallel, Bulk Synchronous Parallel (BSP) and task parallel to determine what method would be preferable to integrate with the existing MPI parallel implementation.

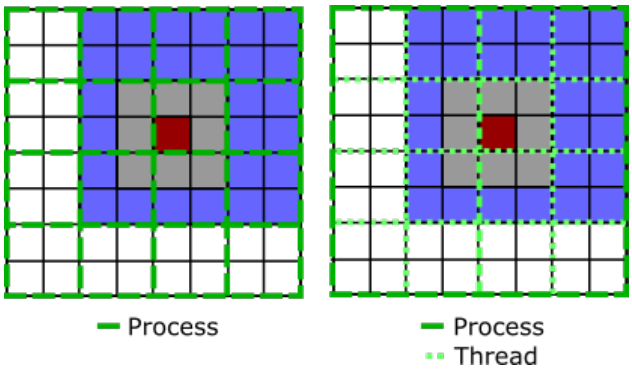

Figure 4.2: Interactions with 16 processes (green dashed lines) Vs 1 process and 16 threads (lighter green dotted lines).

4.2 Background and Related Work

4.2.1 Fast Multipole Method (FMM)

Figure 4.3 shows how the interaction information flows from the multipole expansion tree on the left side through the local expansion tree on the right side, through different stages of the FMM algorithm (for illustration purposes, only a small subset of interactions/information flow is shown). In MLFMA, memory and computation associated with each node quadruples at each level as one ascends in the tree. Consequently, for surface geometries that are typical in electromagnetics and acoustics applications, each level has approximately the same amount of memory and computation costs. Note that all nodes within a given level can be processed independently, while traversing up (M2M) or down (L2L) the tree. Therefore, it is relatively straight-forward to apply the BSP model to MLFMA, as one can loop through the tree level by level and divide up the nodes at each level

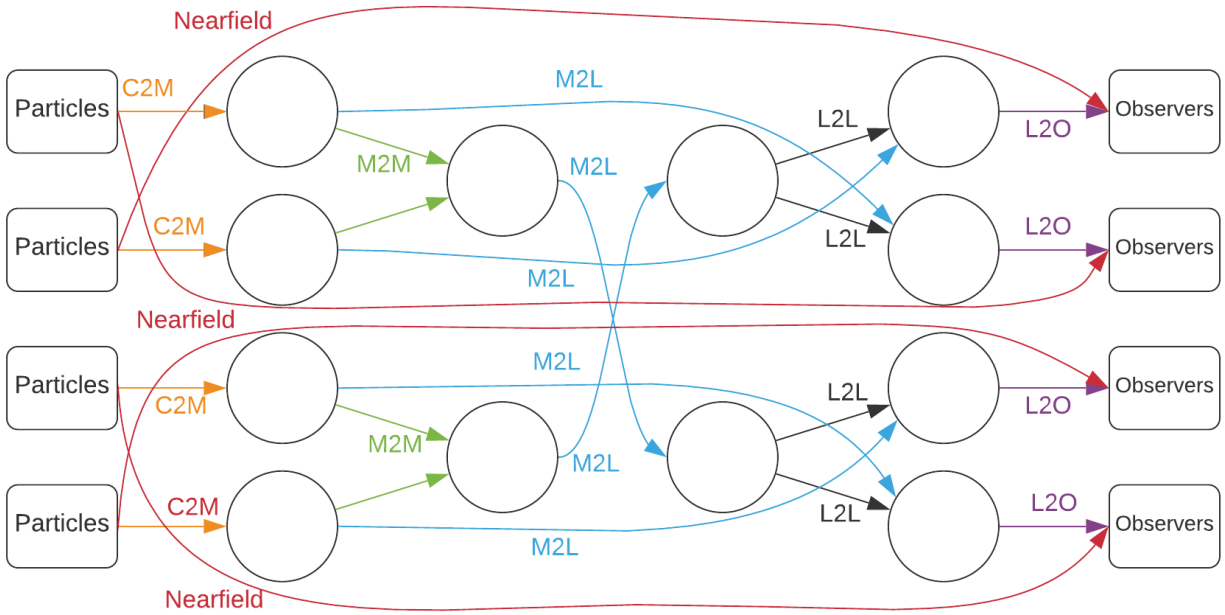

Figure 4.3: Dependencies between boxes within an FMM octree due to the nearfield and farfield computation process.

among threads using *parallel-for* loops. This method may run into a bottleneck as there are fewer (but significantly heavier) nodes available while moving up the tree, leading to the possibility of more threads being available to work than the number of nodes above a certain level. In levels where this occurs, one can parallelize over operations within each tree node at the expense of finer-grained synchronization overheads among threads.

The information flow shown in Fig. 4.3 nicely illustrates the dependencies among different computational steps associated with the tree nodes. Dependencies among these tasks form a directed acyclic graph (DAG) which can easily be expressed through a runtime system with dataflow dependency support. A task parallel approach is less prone to thread idling as it can "fill in" any voids with useful work from other stages of the computation, and finer parallelization of heavier few nodes towards the top of the tree does not necessarily require participation (and synchronization) by all threads. In this sense, task parallelism provides a flexible and potentially effective solution.

4.2.2 Related Work

To the best of our knowledge, task parallelism has not been explored in detail in the context of MLFMA before, but there are several prior works on task parallel L-FMM. Of those, studies by Agullo et al. Agullo et al. (2014) and Yokota et al. Yokota (2013) are similar to this work. Agullo et al. evaluate multiple methods of thread parallel approaches; in the first method they split all tree nodes for each level between threads using a *parallel-for*, they then expand this method by investigating a single thread only processing of some of a parent node's children or a portion of a node's far-field interactions. Finally, they interleave different steps of FMM by using tasks with different DAG orderings and priorities. Each approach shows good strong scaling of up to 91% efficiency on a shared memory architecture, when a geometry with a large number of particles is chosen. The efficiency falls off when using a smaller number of particles. This high efficiency is in part due to a majority of L-FMM processing being at the lowest level of the tree where there are a large number of tree nodes that can be parallelized independently.

Yokota presents an L-FMM implementation using a dual tree traversal scheme and task based parallelism Yokota (2013). The dual tree approach provides greater flexibility in tree partitioning and consequently in load balancing. The implementation is shown to scale well on a shared memory system, and performs better than other algorithms on the same hardware.

Pi et al. analyze a BSP implementation for MLFMA Pi et al. (2010). The implementation simply loop parallelizes the creation of the near-field interaction matrix, and uses parallel-loops to process nodes during each level of the far-field tree traversal. With runs up to 16 threads on the Deep-Comp 7000 HPC at the Chinese Academy of Sciences, the near-field parallel portion shows efficiencies above 95%, while the far-field parallel portion shows lower efficiencies of under 75%.

Abduljabbar et al. describe a solver for low-frequency 3D Helmholtz soft body acoustic problems Abduljabbar et al. (2019), which is probably the closest work reported in the literature to our work. They outline the shared memory optimizations they have performed on MLFMA to maximize node performance. They break down these optimizations into two categories: Data-level and thread-level parallelism. In the context of data-level parallelism, they exploit the vectorization

units in modern multi-core processors, mostly through compiler-aided techniques. Their thread-level parallelization scheme extends the task based dual-tree approach proposed by Yokota, but it lacks details in regards to how they adopt the dual-tree approach to MLFMA. Even though theirs is a distributed memory parallel implementation, it is also not detailed if/how communications are performed along with computational tasks being performed by multiple threads. For these reasons, effective task parallelization strategies for MLFMA warrant further in-depth analysis.

4.2.3 Contributions

Our contributions in this work can be summarized as follows:

1. We develop an efficient task parallel implementation of MLFMA,
2. we explore ideal task orderings and task granularities for optimal performance, and
3. we present an in-depth comparison of BSP and task parallel MLFMA implementations on modern shared memory architectures.

4.3 Methods

4.3.1 MLFMA with BSP

Applying the BSP model in MLFMA is relatively straight-forward, as it mainly amounts to parallelizing over tree nodes for each phase of MLFMA using *parallel-for* loops. Nevertheless, we provide some details to facilitate the performance analysis and comparison discussions presented in the next section. As the base MLFMA implementation is written in Fortran, OpenMP was used for thread parallel development for both BSP and task parallel.

4.3.1.1 Near-field Computations (NF)

In this phase, point-to-point interactions for all particles in a given leaf box with particles in nearby leaf boxes are processed using direct interactions. In doing so, we choose to sweep through all pairs

in an observer-first parallel loop, i.e., the effects of all source particles on an individual observer particle is calculated by a single thread. This avoids the write-after-write contention that would have risen had we chosen to sweep through all pairs in a source-first way.

4.3.1.2 Upward Tree Traversal (C2M and M2M)

For the upward tree traversal, we choose a level-by-level approach over a post order traversal approach because it 1) can easily exploit the independent parallel processing opportunity among nodes in a particular level, and 2) does not suffer from load imbalances among threads as all nodes in a level have similar computational costs. In the upward tree traversal, first all leaf nodes are processed in parallel, performing the C2M operations for each leaf node. Then during M2M, the multipole information of previously processed child nodes is interpolated, shifted and aggregated to form the multipole information of their parent nodes. This process is repeated moving up one level at a time until all levels have been processed. This scheme requires synchronization among all threads at the end of each tree level.

4.3.1.3 Translations (M2L)

M2L is very similar to near-field computation, after all, these are the two MLFMA phases where actual interactions take place. Observer boxes are looped over in parallel and the translations from each source box which has far-field interactions with the current observer are computed and aggregated to the observer boxes. In this phase, observer boxes are processed in a post-order traversal order as our implementation has evolved from a serial implementation. For M2L, there is no clear advantage of level-by-level processing over post-order processing or vice versa, because all nodes across the entire tree are fully independent of each other. The only dependency for any observer box is that the upward traversal phase (C2M and M2M) must be completed for all source boxes before the M2L translation can safely be performed.

4.3.1.4 Downward Tree Traversal (L2L and L2O)

The downward tree traversal is almost the reverse operation of the upward tree traversal. We loop though the tree level-by-level in a top-down manner, and perform a parallel loop over nodes in each level.

4.3.2 Task Parallel MLFMA

Creation of tasks in Helmholtz FMM requires a balance between task granularity versus flexibility. For instance, for a coarse granularity partitioning, a geometry with 16 nodes to compute at its highest level of computation could have each of the 16 nodes along with all their children defined as a task and have them assigned to one of 16 threads available. While such a partitioning provides coarse grained tasks, an unbalanced tree would result in some threads completing their tasks at much different times from others. Conversely, tasks can be limited in scope to the interpolation of a single child node, or the translation of one source to observer node. Tasks of this scale would be fine-grained, but would have far fewer dependencies within the tree. The reduced dependencies mean more tasks would be available to threads for execution at any given time. However, this would also mean more scheduling overheads at runtime. As a guiding principle, we try to balance between the flexibility of fine-grained tasks vs. their scheduling overheads.

4.3.2.1 Near-field Computations (NF)

We have chosen to keep the task parallel near-field implementation simple and straightforward. Much like the loop parallel implementation which performs a parallel loop through all observer nodes, we make near-field computations of each observer node a task. Near-field computations implemented in this way only has output dependencies with the L2O phase, thus they can be executed at any other time. This provides great scheduling flexibility and potential performance improvements as near-field computations can help fill-in the thread idlings during execution of the far-field interactions that have complex dependencies.

4.3.2.2 Upward Tree Traversal (C2M and M2M)

The C2M step generates the multipole expansion of a leaf box from all particles within it. We create a task for the C2M operation of each leaf node. Even for a small geometry, the number of leaves far exceeds the number of threads available on a typical shared memory architecture. Thus, there is little point in making C2M tasks finer grained than creating the entire multipole expansion for a single leaf. Creating a task from groups of leaves would yield larger granularity tasks, but it would also increase the number of M2M and M2L tasks dependent on each C2M task, restricting parallelism up the tree.

The M2M step generates the multipole expansion of a parent node from all its children. We create a separate task for each child being interpolated, shifted and aggregated to create a parent node. This means each task has a single input dependency on the child node's multipole data being ready and a single output dependency on the parent node. The M2M operation to produce the entire multipole data for a parent node could be a single task as well, but then such a task would depend on multipole data for all child nodes being ready, instead of just one. As we demonstrate in Section 2.4, coarse-graining M2M tasks does not perform as well as the fine-grain approach we adopt.

We provide the pseudocode for this initial version of our task-parallel upward tree traversal algorithm in Alg. 4.3.

One of the drawbacks of the above described upward tree traversal scheme is interpolation of the nodes at the higher levels of the tree. For instance, in a typical surface geometry, there are likely to be 16 nodes at the highest level. Due to output dependencies, only up to 16 threads can be actively working on the interpolation of these high-level nodes. Therefore, we apply a further refinement of M2M tasks for the high level tree nodes. All samples within a node are fully independent during the shifting and aggregating operations, therefore we split these operations into many tasks for individual nodes. Interpolation is more complex though. While it is beyond the scope of this chapter to go into too much detail, in MLFMA multipole data take the form of functions sampled in two angular dimensions; the data can be viewed as a rectangular array of function samples which

Algorithm 4.3 Task-based upward tree traversal

Require: $p.center$ coordinates of the parent box center
Ensure: pmp is parent's multipole representation

```
1:  for each box p in post-order traversal do
2:      if p is leaf box then
3:          task Depend Out box p
4:              pmp ← C2M(p)
5:          end task
6:      else
7:          for each child box c do
8:              task Depend In all child box c Depend Out box p
9:                  mp[c] ← interpolation(c)
10:                 smp[c] ← shift(mp, p.center)
11:                 aggregate(pmp,smp[c])
12:             end task
13:         end for
14:     end if
15: end for
```

can be partitioned into block columns or rows. FFT-based interpolation of these partitions are also independent of each other Hughey et al. (2019); Lingg et al. (2020). Hence, we create tasks for interpolations of partitions. We illustrate the fine-grained task parallel M2M method used for high level nodes in Alg.4.4.

4.3.2.3 Translations (M2L)

The M2L phase translates the multipole expansion of each source box to the local expansions of all observer boxes in its far-field. Following our previous strategy of minimal dependencies would mean each translation of source to observer box should be a separate task as in fine-grained parallelization of M2M phase. On the other extreme, all translations for a source node could be defined as a single task which could potentially reduce the number of times a source node needs to be loaded from memory. We have found that a middle ground between the two, i.e., performing translations in chunks, is the most efficient approach for M2L.

In MLFMA, the number of translations (interactions) required for a node changes significantly from a geometry to another - while the average number of translations per node is about 27 for

Algorithm 4.4 Parallel Interpolation

Ensure: c is the child box being interpolated

```
1: pts ← partition(c)
2: for each partition p in pts do
3:     task
4:         for each θ vector v in p do
5:             theta[v] ← interpolate(v)
6:             shift1[v] ← transposeandfold(theta[v])
7:         end for
8:     end task
9: end for
10: TaskWait
11: pts ← partition(shift1)
12: for each partition p in pts do
13:     task
14:         for each φ vector v in p do
15:             phi[v] ← interpolate(v)
16:             shift1[v] ← transposeandfold(phi[v])
17:         end for
18:     end task
19: end for
20: TaskWait
```

a surface geometry, this number goes up to 189 for a volume geometry (which is not common in practice). Therefore, we experimented with various bundling factors (*bf*) for M2L, see Section 2.4 for further details. We provide the pseudocode for our task-parallel M2L implementation with bundling in Alg. 4.5.

Algorithm 4.5 Task-parallel translations

Ensure: bf is translation bundling factor

Ensure: lp is the local expansions of the box

```
1: for each box b do
2:     for each box fb interacting with b in groups of bf do
3:         task Depend In box b Depend Out box fb
4:             int ← compute_interaction(fb,b)
5:             lp[b] ← add_interaction(int)
6:         end task
7:     end for
8: end for
```

4.3.2.4 Downward Tree Traversal (L2L and L2O)

As mentioned before, L2L and L2O steps are almost the reverse of M2M and C2M operations, respectively. As such, their task parallelization follows the same strategy as upward tree traversal outlined above, albeit with some simplifications. For L2L, the highest level nodes are *read-only*. Output dependencies are on nodes the next level down, which will have a minimum of 64 nodes. This represents a sufficient degree of parallelism for existing multi-core and many-core architectures, therefore we have not adopted the fine-grained parallelization method of M2M here, but it certainly can be done relatively easily.

4.3.2.5 Task ordering

A further consideration is the impact of the order of tasks. Being able to influence the scheduling of tasks is important for performance reasons because tasks from different phases of the computation that do not have dependencies between them may “fill-in" the voids encountered during execution. Most task-based runtime systems, including OpenMP which we have used for implementation of our ideas described above, allow programmers to specify task priorities. In OpenMP though, task priorities are only suggestions for the runtime system and we have observed in general that these priorities have little to no effect in terms of the scheduling of tasks; at least, that has been the case for our task-parallel implementation. However, we have found that the order in which tasks are generated affects their execution order and that is what we have used to modify the scheduling of tasks.

In this regard, near-field tasks provide the greatest flexibility because they can only conflict with the L2O tasks writing the tree-generated potential values. Therefore, near-field tasks can be performed without race conditions at any time before or after L2O. The chosen time to perform nearfield processing of our algorithm is after translations (M2L) and before starting the downward traversal (L2L). At the top of the MLFMA tree, the number of nodes is typically smaller than the number of threads, but each node is very large and requires significant amount of computation. As a result, there is a good chance that some threads will be left without tasks to perform until

the upward traversal (M2M) and translations (M2L) of these highest level nodes are completed. Performing near-field computations during this time-frame fills in any potential gaps.

The remaining stages of the tree traversal have more dependencies to deal with. A node cannot start its M2M computations until the M2M computations of its child nodes have finished. A node cannot perform its M2L translations until its own M2M computations are completed. Finally, a node cannot start its L2L phase until its parents have completed theirs and the node has completed its M2L interactions. This limits task ordering, but still allows some flexibility. The simplest implementation is generating all upward traversal tasks first, then all far-field interaction tasks, and finally all downward traversal tasks. Alternatively, one can do the same thing but at the level of individual nodes. As soon as a node has interpolated, shifted and aggregated all of its children, farfield interactions can be computed for that node. On the opposite end, a high level node can perform its L2L operations as soon as M2L has translated all of its source nodes, but before any of its children have performed M2L translations. This approach can be repeated, computing anterpolations before translations where possible. The first method was chosen as it was empirically found to perform better.

4.4 Results

In this section, we evaluate the performance of the task-parallel MLFMA algorithm described. All results were obtained on the Cori supercomputer at National Energy Research Scientific Computing Center (NERSC). Each Haswell node on this system contains two sockets, populated with Intel Xeon E5-2698 v3 (Haswell) processors with a clock speed of 2.3 GHz. Each node has 32 cores, plus hyperthreading, 128 GB 2133MHz DDR4 RAM, and 40M Cache. The code is implemented in Fortran 90 using only OpenMP parallelization and was compiled with the Intel compiler version 19.0.3.199. The Cray FFTW library version 3.3.8.4 is used for all FFT operations.

Performance was also measured using Cori's KNL nodes. Each KNL node contains a single socket, populated with an Intel Xeon Phi Processor 7250 ("Knights Landing") processor with a clock speed of 1.4 GHz. Each node has 68 cores, with 4 hardware threads per node, 96 GB 2400

MHz DDR4 RAM, and 64 KB L1 cache per core, plus 1MB L2 cache per tile (2 cores per tile). The lower processor speed vs Haswell leads to longer execution times.

4.4.1 Tuning the Task-Parallel MLFMA Implementation

As mentioned in 4.3, there are two optimizations we used for our task-parallel MLFMA implementation. These are ordering of the creation of tasks, which in turn alters the scheduling of tasks, and bundling of tasks. For tuning our implementation, we chose a 7-level sphere geometry, as spheres are a commonly used benchmark for MLFMA codes. Our tuning is empirical, certainly relying on the specific architecture and geometry. However, we note that in applications, the MLFMA is used as an inner kernel in long-running iterative solvers that can take hundreds to thousands of iterations to converge for large problems. Since our tuning parameter space is relatively small, it is practical to tune the performance for the particular geometry and architecture before the actual solver is launched.

4.4.1.1 Task Generation Ordering

Since the near-field (NF) phase is the most flexible phase within MLFMA, we created different flavors of task-parallel MLFMA where NF tasks are generated between all tree computation phases. Starting with “NF First", these are “NF after M2M", “NF after M2L", “NF Last". There are two other finer grain reorderings; they interleave the execution of M2L with M2M (“M2L during M2M") or M2L with L2L (“M2L after L2L"), rather than executing each phase entirely separately.

As can be seen in Fig. 4.4, for most thread counts, generating NF tasks at different phases has minimal impact, but for 64 threads “NF after M2L" results in a 5% performance improvement over the others. Executing L2L wherever possible before M2L produces good scaling, but poor execution times overall. Executing M2L as soon as possible during the M2M execution shows a slight improvement in performance. Finally, combining the best of the two task orders, “NF after M2L" and “M2L during M2M", produces a 4% execution improvement at 32 threads and over 18%

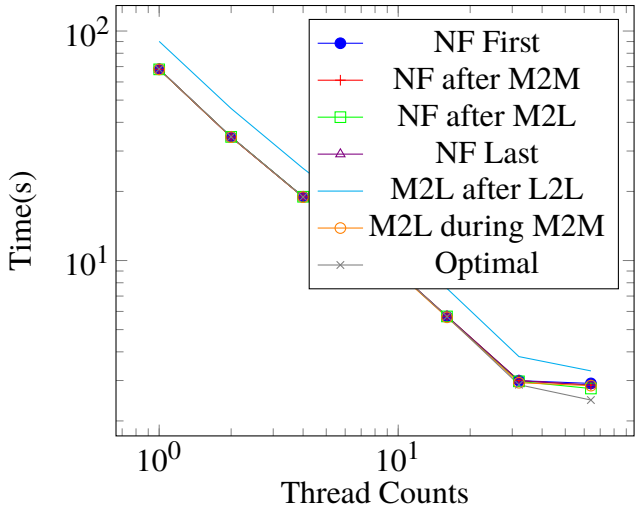

Figure 4.4: Impact of task order on execution time.

improvement at 64 threads. This method is labeled on the graph as “Optimal", and is used for the task-parallel MLFMA results reported.

4.4.1.2 Task Bundling for M2L

The second optimization we implemented is bundling tree operations together in each task. For the same sphere geometry, we experimented with different bundling schemes. This included the extreme cases of bundling all M2M operations of children of a single parent node together on one side and creating a separate task for each child on the other side. Both methods performed on par with each other for small thread counts, but we observed that bundling all children into a single task during M2M performed significantly worse at 64 threads (see Fig. 4.5). This is likely due to the small number of tasks created at higher tree levels which contain computationally expensive nodes. Therefore, we define all children during M2M as individual tasks.

For M2L interactions, we experimented with different bundling factors such as 9, 27 (which is the expected number of interactions for surface geometries), 189 (theoretical maximum for M2L for any geometry) and compared them with regular (non-bundled) M2L in terms of performance, see Fig. 4.5. Grouping the translations of 9 observer nodes with a common source node into a single task provides a notable benefit. Any impact is barely noticeable through 16 threads, but at 32 and

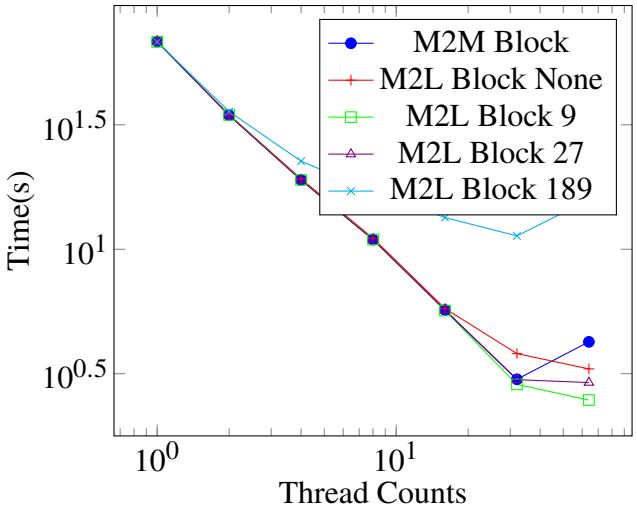

Figure 4.5: Impact of bundling on execution time. Performance of M2M implementation with all children bundled into a single task (M2M with Bundling) and M2L with various bundling factors (none, 9, 27, and 189) are shown for different thread counts.

64 threads, the performance improvement over the non-bundled version is nearly 33%. Increasing the bundling factor to 27 interactions of a common source node decreases the performance slightly, and the extreme case of bundling 189 interactions results in a significant performance falloff even at small number of threads. All results presented in the rest of this manuscript uses a bundling factor of 9 for the M2L phase.

4.4.2 Performance Comparison between BSP and Task Parallel Implementations

In this subsection, we compare the performance of our task parallel MLFMA implementation against the BSP version on a number of geometries. Both versions use the same tree construction methods (so the amount of work performed by both methods is identical) and they use the same OpenMP compilation and runtime settings. The potentials computed by both versions were compared to ensure that the only differences are due to floating point arithmetic precision.

For benchmarking, we used four different geometries. The first geometry is a simple planar grid of particles (in the $z = 0$ plane). The grid dimensions are $128\lambda \times 128\lambda$, with 5,242,880 points uniformly distributed over the geometry and smallest FMM box size of $\lambda/4$. This produces a 10-level tree with 20 points in each leaf box. The second geometry is a sphere whose radius is

128λ, with 7,264,954 points uniformly distributed over the geometry and smallest FMM box size of $\lambda/4$. This also produces a 10-level tree, with an average of 18 points in each leaf box. The third geometry is a 3D volumetric distribution of particles. The box dimensions are $8\lambda \times 8\lambda \times 8\lambda$, with 1,048,576 points randomly distributed over the cube and smallest FMM box size of $\lambda/4$. This produces a 6-level tree with an average of 32 points in each leaf box. The last geometry is an airplane model which is of size 256λ in length. It is discretized with over 4,459,776 points and the smallest FMM box size is $\lambda/4$. This produces an 11-level tree with an average of 15 points in each leaf box, albeit with a highly non-uniform distribution of points across leaves.

4.4.2.1 Performance on a Multicore Architecture (Cori-Haswell)

Figure 4.6 compares the execution times of BSP and task-parallel MLFMA versions using 1 to 64 threads. Note that the Haswell processors only have 32 physical cores (on two sockets), so 64 thread executions use hyperthreading. The airplane model, which is a real application, shows the strongest performance advantage for task-parallel MLFMA as it attains as much as 1.35x speedup over the BSP version. Both the grid and volume geometries also show that the task parallel version achieves consistently increasing speedups over the BSP version with increasing number of threads. While we initially observe significant gains with task parallelism over the BSP version for the sphere geometry as well, to our surprise these gains fade away at high number of threads. We try to provide a more detailed insight into these results in the next subsection.

4.4.2.2 Manycore Architecture (Cori-KNL)

We performed the same performance analysis using Cori-KNL nodes which have a significantly different architecture than Cori-Haswell nodes. We observe that for 2 to 32 threads, task parallel MLFMA shows performance gains similar to those of Cori-Haswell experiments (see Fig. 4.7). However, its scalability falls off slightly at 64 cores, which is potentially due to two cores sharing the L2 cache on a tile when the number of threads is increased from 32 to 64. Beyond 64 threads, KNL effectively employs hyperthreading. In this regime (not shown in plots), while the BSP

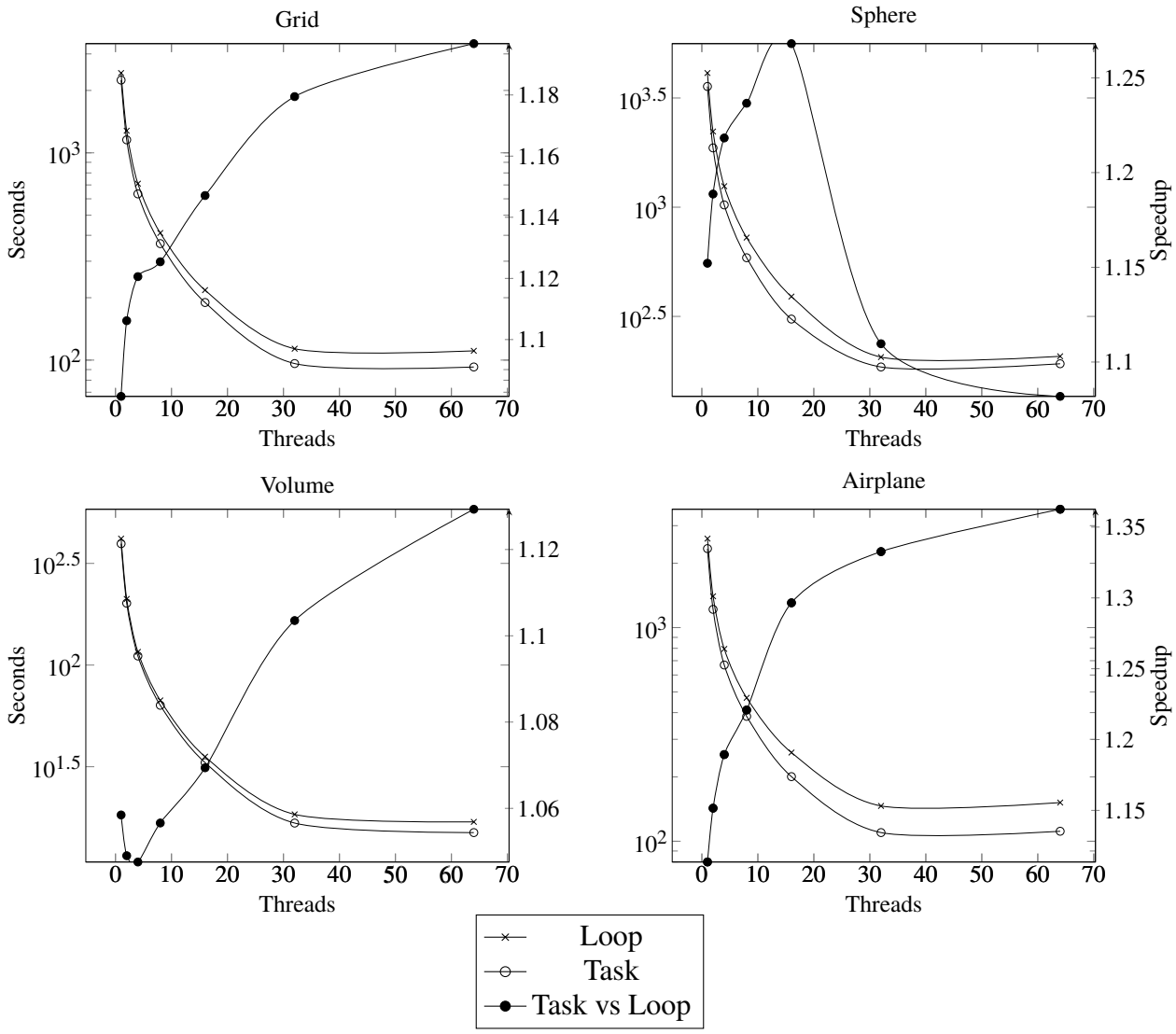

Figure 4.6: Task vs Loop (BSP) parallel runs on Haswell compute nodes for four different geometries.

implementation is able to keep performing at a similar level, the performance of the task parallel MLFMA actually starts dropping. This is likely because the scheduling of tasks which must be done sequentially starts becoming a bottleneck with the increase in the number of threads. The use of many slow cores on KNL (as opposed to multiple high performance cores like Xeon CPUs) can have a compounding effect on this bottleneck, too.

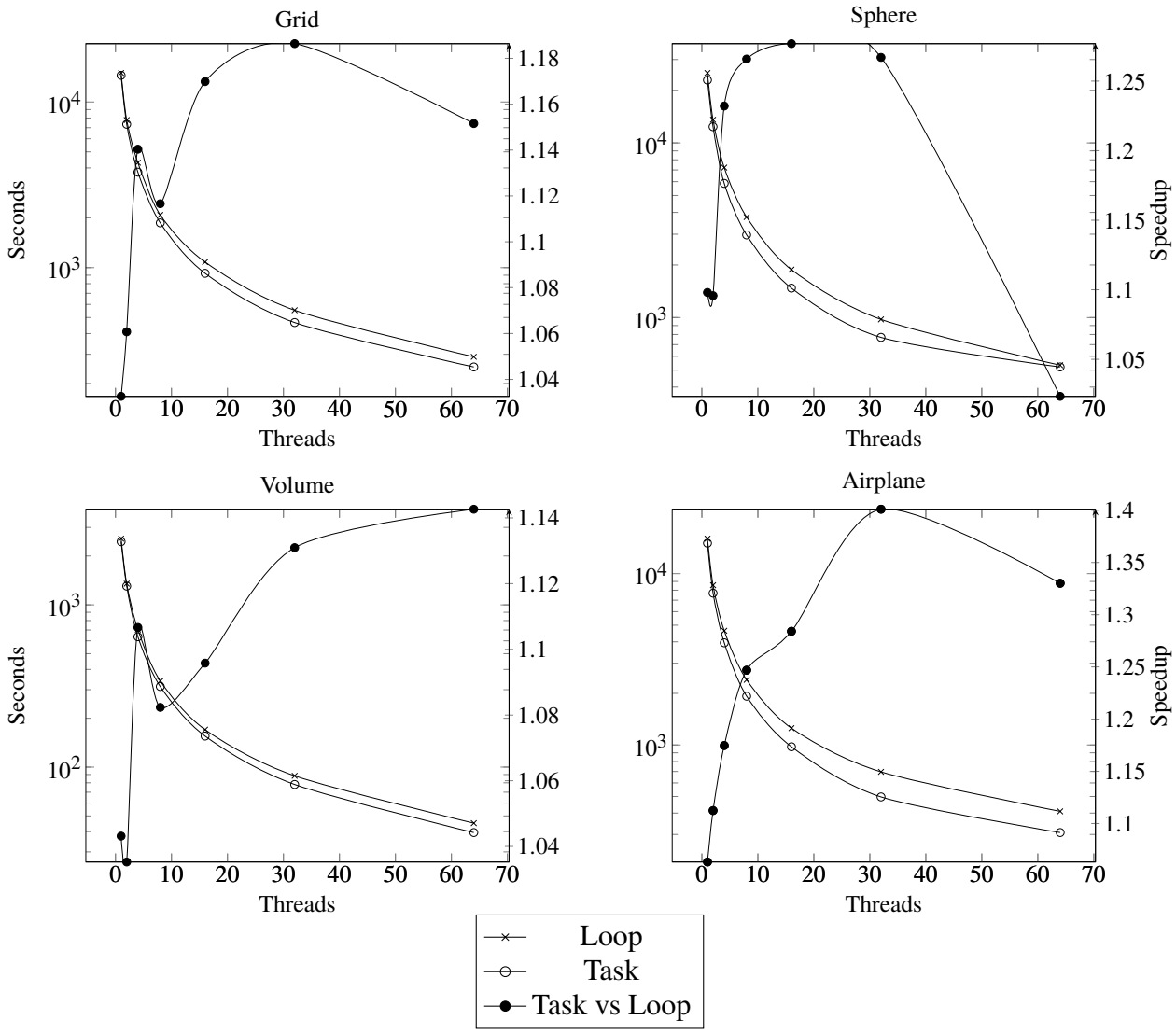

Figure 4.7: Task vs Loop (BSP) parallel runs on KNL compute nodes for four different geometries.

4.4.3 Understanding the Reasons behind Observed Differences

To understand the performance benefits of task parallel MLFMA over BSP version, we conducted timeline and cache performance analyses, for which we used the *perf-stat* tool.

4.4.3.1 Timeline Analysis

Figure 4.8 shows the order of execution of threads in the BSP version execution for a 7-level grid geometry using 64 threads (hyperthreaded) on a Cori-Haswell node. NF, C2M and L2O all perform

well. Each of these operations has a very large number of nodes that can all be processed in parallel. M2M begins showing load balance issues which become very significant at the highest level where only 16 nodes can be processed. M2L shows a lesser extent of thread idleness, likely due to thread dependencies as there are a large number of M2L nodes that can be processed in parallel, up to the highest level where we again see an issue with there only being 16 nodes at the top level. Finally, L2L shows similar thread inactivity as M2M, but in reverse.

Figure 4.9 shows the order of execution of the tasks during task parallel MLFMA. Unlike the BSP version, C2M, M2M and M2L tasks are mixed together as dependencies allow. Further, NF is mixed in with other tasks, filling in some the empty space during M2M computation of the top level and M2L helping fill in more of the rest. The start of L2L also shows the benefit of fine grain parallel at the top level interpolation and anterpolation operations where more threads are able to participate.

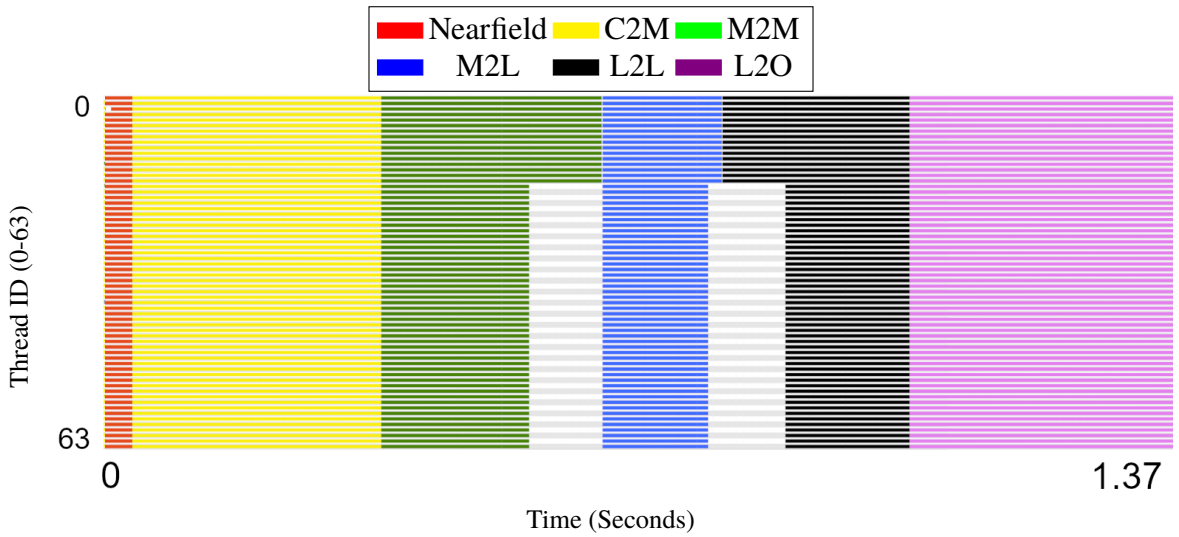

Figure 4.8: BSP timeline on grid geometry.

Figures 4.10 and 4.11 show how the BSP and task parallel timelines change for a spherical geometry. The sphere fills more of the highest level tree nodes. This means the BSP approach has more nodes to process at the top level and is more efficient at keeping all threads active. The dependencies caused by lower level nodes having fewer child boxes than higher level nodes, as the sphere acts more like a surface, mean the task parallel approach has more tasks that cannot be

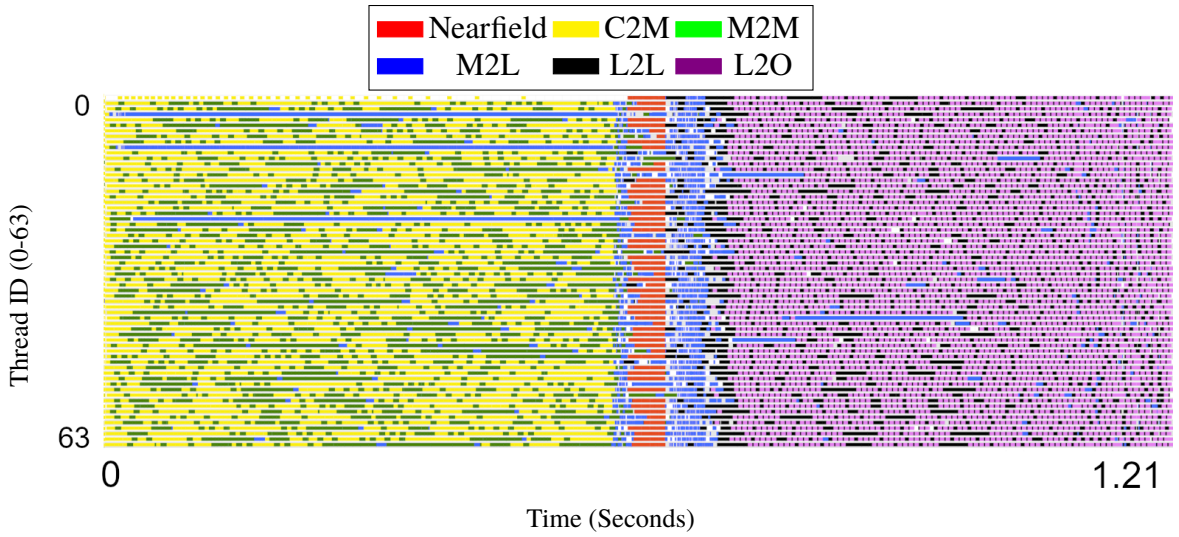

Figure 4.9: Task parallel timeline on grid geometry.

executed until dependent tasks complete. As a result, the task parallel approach is not as efficient for this example.

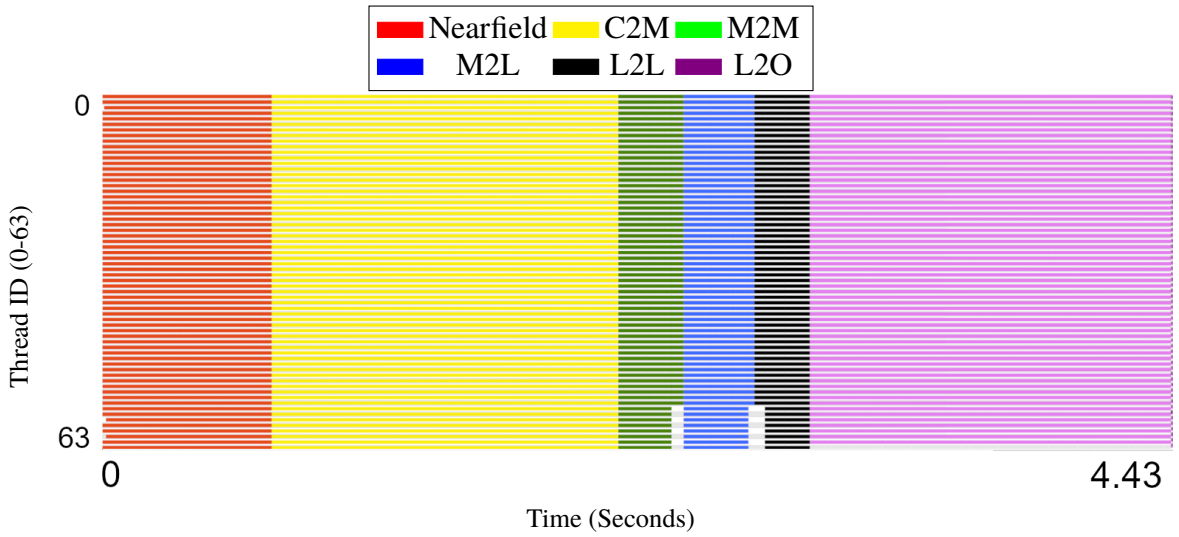

Figure 4.10: BSP timeline on the sphere geometry.

Alternately, Figures 4.12 and 4.13 show how the BSP and task parallel timeline behave for an airplane geometry. Unlike the previous examples, this geometry is non-uniform, so many of the particles are clustered in fewer nodes. The impact of this can be seen in Figure 4.12 where the top levels of M2M and L2L have fewer nodes that can be processed. Furthermore, the next level down

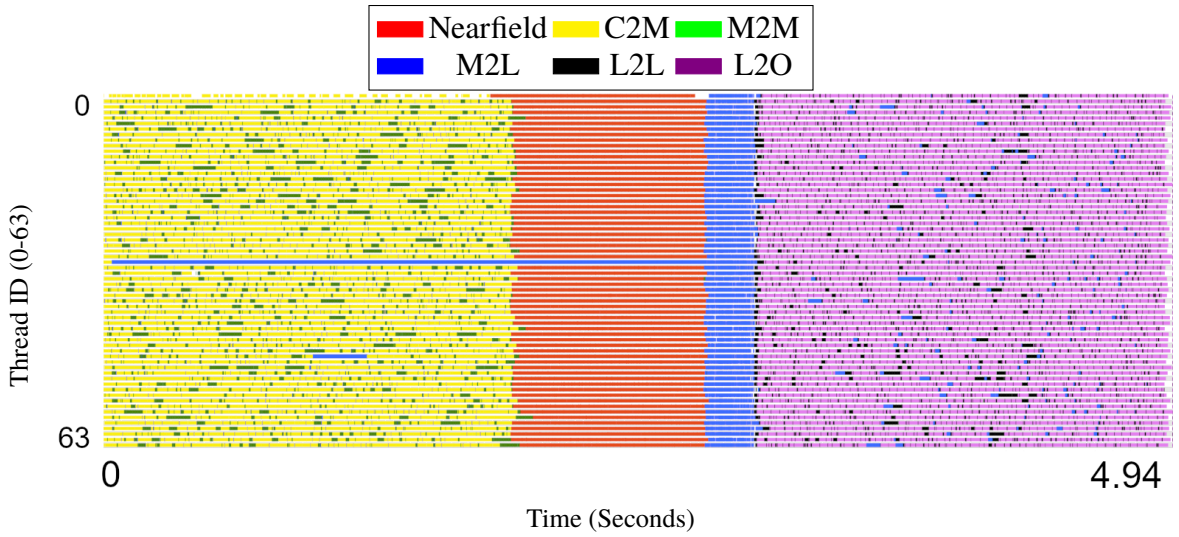

Figure 4.11: Task parallel timeline on the sphere geometry.

still has a limited number of nodes to process. Figure 4.13 shows that tasks keep more threads active by performing M2L and NF tasks during the times when there are not enough high level nodes to occupy all threads.

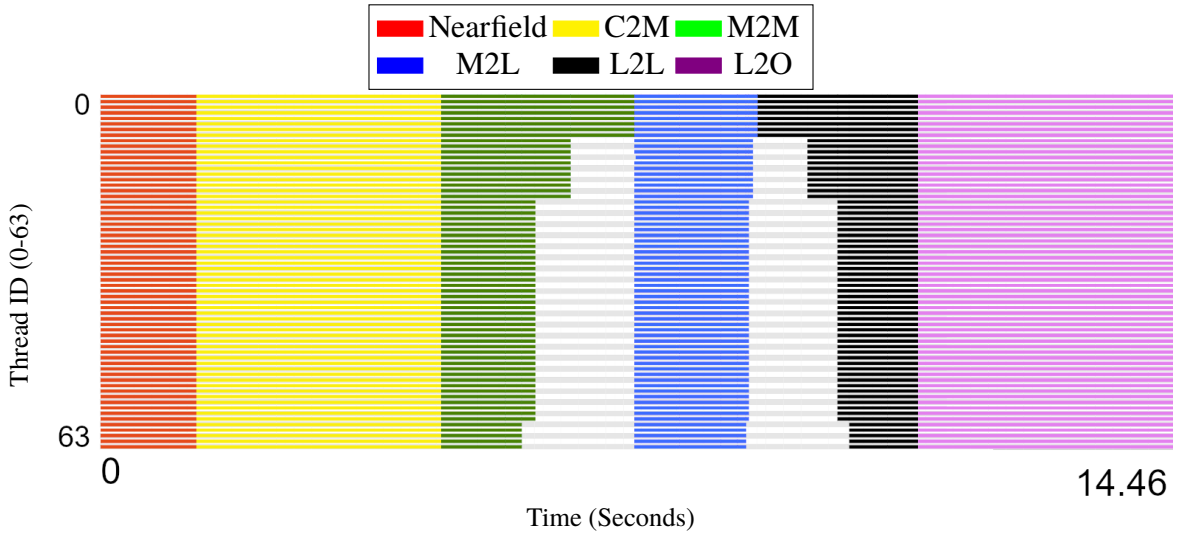

Figure 4.12: BSP timeline on the airplane geometry.

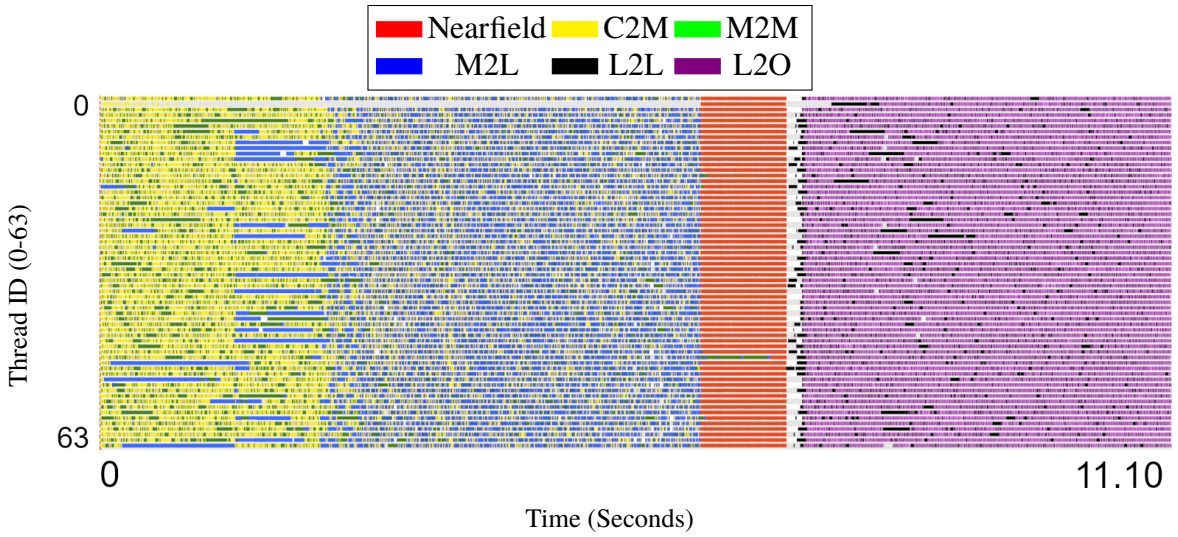

Figure 4.13: Task parallel timeline on the airplane geometry.

4.4.3.2 Cache Analysis

To look further at why the task parallel approach is more efficient, we analyzed the cache utilization of the two versions. Cache analysis was performed using VTune and 64-thread executions of the grid geometry on Cori-Haswell nodes. The cache analysis runs in Fig. 4.14 show that the ratio of cache hits to misses is not always more favorable for task parallel vs the BSP version. However, since L1 cache hits ratio is as high as 99.8%, any differences are effectively a rounding error. As such, we conclude that while task parallel MLFMA makes less effective use of cache, this does not negatively impact its performance at a significant degree.

4.5 Conclusions

Due to the near constant amount of processing necessary per level with the number of nodes per level decreasing while moving up the tree, Helmholtz FMM presents challenges to parallelization that are not present in Laplace FMM. In this chapter, we presented a task parallel MLFMA implementation to address these parallelization challenges. Results on various geometries have shown that in most cases, particularly for the real world application case of an airplane geometry, the task parallel implementation shows improved performance and scalability for shared memory architectures compared to a bulk synchronous parallel MLFMA implementation. Our study pro-

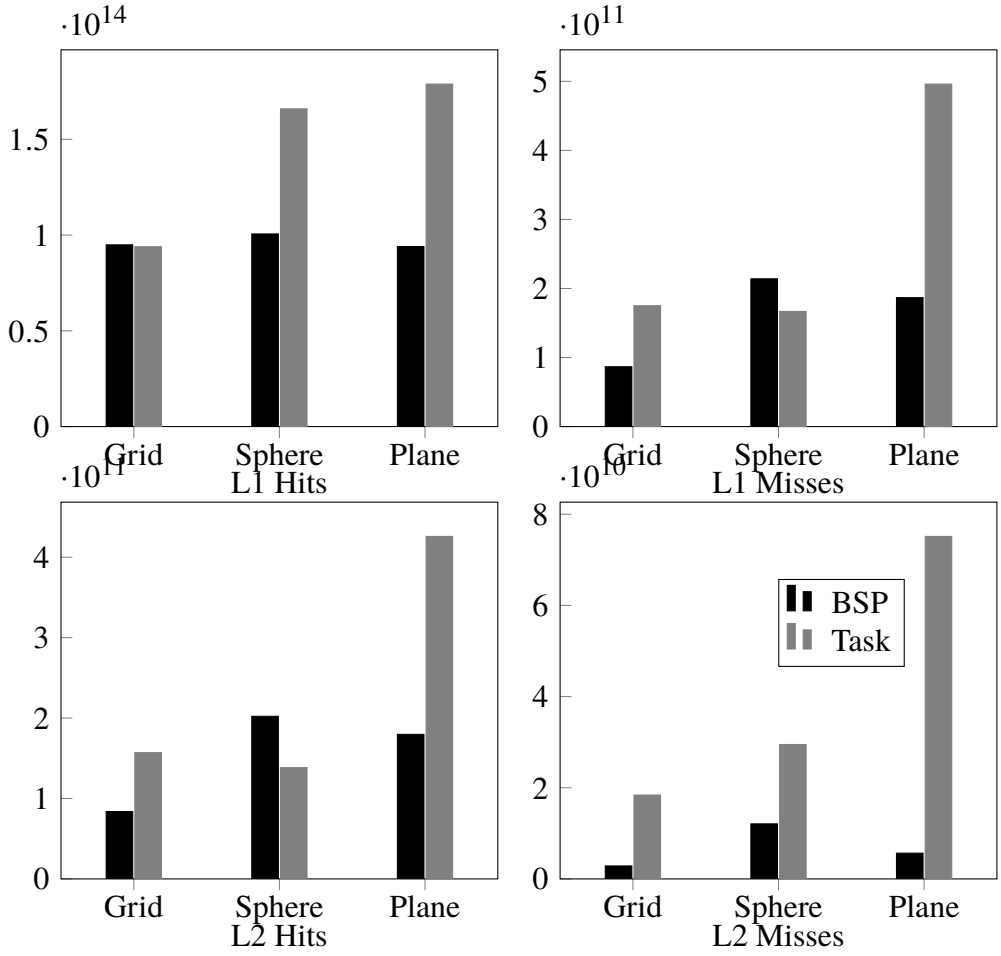

Figure 4.14: Comparison of the cache performance of BSP and task parallel implementations on a Cori-Haswell node.

vides evidence that task parallelism is a promising approach for MLFMA, and it can be even more useful in a hybrid shared and distributed memory parallel context because it would allow great flexibility in terms of overlapping the execution of communication-intensive parts of MLFMA with its computation-intensive parts so as to minimize idle times and achieve scaling to a large number of compute nodes.

CHAPTER 5

FUTURE WORK

While looking at the results of chapter 3, it was noticed in table 3.1 that the scaling of M2M and L2L was quite strong through 2048 processes, while M2L's performance faded much faster. The most likely cause is too much data communication during translation. At upper levels of the tree, a process will own at most a full node for a given level, or only part of a node. To perform translations, the observer node needs all samples from the source node, or if the node is split across processes, the samples of the source node corresponding to the samples the process owns for the observer node. Since no process can own more than one node, all data from every node must be communicated between processes owning source nodes, and processes owning observer nodes. Worse, every observer node is guaranteed to be owned by a separate process. So each source node must be communicated to up to the 27 processes owning observer nodes for a surface, or 189 for a volume. At lower levels, multiple observer nodes may reside on a single process, or a process may own both a source and observer node, reducing or eliminating some of the communication at this level. However, when the number of processes increase, the level where all node data must be communicated moves down the tree, increasing communication requirements.

A possible improvement to the communication requirements lies in local interpolation as shown in Yang et al. (2019). In this paper, nodes at the highest levels of the tree are divided into groups based on the number of processes. Then each group of samples from every node for that level are assigned to corresponding processes. Because each process owns corresponding samples of each node for a level, translation requires no communication. With this approach, global interpolation could still be used at lower levels, taking advantage of the exact interpolation results and lower sampling rates, while local interpolation is used only at the higher level where translation communication costs are the highest. In addition to managing the errors induced by local interpolation and the increased sampling costs, the impact to the M2M and L2L stages will need to be considered.

A method with more immediate improvements would be to test the performance benefits of

combining the MPI parallel approach in chapter 3 with the shared memory parallel implementation of chapter 4. In shared memory parallel, the ratio of nodes in a level to process count would be the same as MPI parallel, but all processes on the same processor could access the same memory. This means when a source node is owned by one process and an observer node is owned by another process, in MPI parallel the source data must be communicated to the process owning the observer. With a hybrid of MPI and shared memory parallel, a process would reside on a single node, while each core of that node could be running a separate thread. All threads of the same process can used shared memory, rather than more expensive communication, during operations where interacting nodes are on separate threads in the same process.

Further work can also be done to improve load balancing for non-uniform trees. Hughey (2018) describes a bottom up load balancing method. The work done in chapter 3 providing a fine grain parallel algorithm at the top level of the tree results in a different work load from the cited thesis so a new method would be necessary. In an unbalanced tree, there would likely be a higher level where all nodes at the level must be fully processed, but some of these nodes may have few descendants with any particles. The unbalance could range from all high level nodes having the same number of descendants, to one high level node having a full sub tree with other nodes only having a unary subtree. Bottom up partitioning could properly balancing the spacial partitioning at the low level, but assign the previously mentioned high level node with a unary subtree a single process. Top down partitioning could assign a large number of processes to the same high level node with a unary subtree while all these processes have to share a single leaf node. Instead the data could be partitioned among processes in a two stage process. In the first stage full high level nodes at an appropriate level would be assigned to groups of processes. The size of the groups could be balanced between the number of descendants of the high level node, and the amount of work processing the high level node and the share of the node's parents. The second stage would assign descendants of the shared nodes to processes sharing the ancestor node. This method would balance direction partitioning among the process groups, then balance spatial partitioning within the groups.

BIBLIOGRAPHY

BIBLIOGRAPHY

Abduljabbar, Mustafa, Mohammed Al Farhan, Noha Al-Harthi, Rui Chen, Rio Yokota, Hakan Bagci & David Keyes. 2019. Extreme scale fmm-accelerated boundary integral equation solver for wave scattering. *SIAM Journal on Scientific Computing* 41(3). C245–C268.

Agullo, Emmanuel, Bérenger Bramas, Olivier Coulaud, Eric Darve, Matthias Messner & Toru Takahashi. 2014. Task-based fmm for multicore architectures. *SIAM Journal on Scientific Computing* 36(1). C66–C93.

Appel, A. 1985. An efficient program for many-body simulations. *SIAM J. Sci. Comput.* 6. 85–103.

Barnes, J. & P. Hut. 1986. A hierarchical $((n \log n)$ force calculation algorithm. *Nature* 324. 446–449.

Burresi, Matteo, Lorenzo Cortese, Lorenzo Pattelli, Mathias Kolle, Peter Vukusic, Diederik S Wiersma, Ullrich Steiner & Silvia Vignolini. 2014. Bright-white beetle scales optimise multiple scattering of light. *Scientific reports* 4.

Cecka, Cris & Eric Darve. 2013a. Fourier-based fast multipole method for the helmholtz equation. *SIAM Journal on Scientific Computing* 35(1). A79–A103.

Cecka, Cris & Eric Darve. 2013b. Fourier-based fast multipole method for the helmholtz equation. *SIAM Journal on Scientific Computing* 35(1). A79–A103.

Chaigne, Sebastien, Guillaume Sylvand, Eric Duceau & Julien Simon. 2007. On the use of the fast multipole method for accurate automotive body panel acoustic load predictions. Tech. rep. SAE Technical Paper.

Chew, W.C., V. Jandala, C.C. Lu, E. Michielssen, B. Shanker, J.M. Song & J.S. Zhao. 1997. Fast multilevel techniques for solving integral equations in electromagnetics. In *Microwave conference proceedings, 1997. apmc '97., 1997 asia-pacific*, vol. 1, 457–460vol.1. doi:10.1109/APMC.1997.659422.

Chew, Weng Cho, Eric Michielssen, JM Song & Jian-Ming Jin. 2001. *Fast and efficient algorithms in computational electromagnetics*. Artech House, Inc.

Chien, R. J. & B. K. Alpert. 1997. A fast spherical filter with uniform resolution. *J. Comput. Phys.* 136. 580–584.

Coifman, R., V. Rokhlin & S. Wandzura. 1993a. The fast multipole method for the wave equation: A pedestrian prescription. *IEEE Antennas Propagat. Mag.* 35(3). 7–12.

Coifman, Ronald, Vladimir Rokhlin & Stephen Wandzura. 1993b. The fast multipole method for the wave equation: A pedestrian prescription. *IEEE Antennas and Propagation magazine* 35(3). 7–12.

Darve, Eric. 2000. The fast multipole method: numerical implementation. *Journal of Computational Physics* 160(1). 195–240.

Dembart, B. & E. Yip. 1995. A 3d fast multipole method for electromagnetics with multiple levels. In *Proceedings of the 11th annual conference on applied computational electromagnetics*, vol. 1, 621–628. Monterey, CA.

Dembart, B. & E. Yip. 1998. The accuracy of fast multipole methods for maxwell's equations. *IEEE Computational Science and Engineering* 5. 48–56.

Ding, Hong-Qiang, Naoki Karasawa & William A Goddard III. 1992. Atomic level simulations on a million particles: The cell multipole method for coulomb and london nonbond interactions. *The Journal of chemical physics* 97(6). 4309–4315.

Engheta, Nader, William D Murphy, Vladimir Rokhlin & Marius Vassiliou. 1985. The fast multipole method for electromagnetic scattering computation. *IEEE Transactions on Antennas and Propagation* 40. 634–641.

Ergin, A Arif, Balasubramaniam Shanker & Eric Michielssen. 1998. Fast evaluation of three-dimensional transient wave fields using diagonal translation operators. *Journal of Computational Physics* 146(1). 157–180.

Ergin, A Arif, Balasubramaniam Shanker & Eric Michielssen. 1999. The plane-wave time-domain algorithm for the fast analysis of transient wave phenomena. *IEEE Antennas and Propagation Magazine* 41(4). 39–52.

Ergül, Ö & L Gürel. 2008. Hierarchical parallelisation strategy for multilevel fast multipole algorithm in computational electromagnetics. *Electronics Letters* 44(1). 3–5.

Ergul, Ozgur. 2011. Parallel implementation of MLFMA for homogeneous objects with various material properties. *Progress In Electromagnetics Research* 121. 505–520.

Ergül, Özgür. 2011. Solutions of large-scale electromagnetics problems involving dielectric objects with the parallel multilevel fast multipole algorithm. *JOSA A* 28(11). 2261–2268.

Ergul, Ozgur & Levent Gurel. 2006. Enhancing the accuracy of the interpolations and anterpolations in mlfma. *IEEE Antennas and Wireless Propagation Letters* 5(1).

Ergul, Ozgur & Levent Gurel. 2013. Accurate solutions of extremely large integral-equation problems in computational electromagnetics. *Proceedings of the IEEE* 101(2). 342–349.

Ergül, Özgür & Levent Gürel. 2013. Fast and accurate analysis of large-scale composite structures with the parallel multilevel fast multipole algorithm. *JOSA A* 30(3). 509–517.

Frigo, Matteo & Steven G. Johnson. 2005. The design and implementation of FFTW3. *Proceedings of the IEEE* 93(2). 216–231. Special issue on "Program Generation, Optimization, and Platform Adaptation".

Greengard, L. 1988. *The rapid evaluation of potential fields in particle systems*. Cambridge, MA: MIT Press.

Greengard, L. & V. Rokhlin. 1987. A fast algorithm for particle simulations. *Journal of Computational Physics* 20. 63–71.

Greengard, Leslie, Jingfang Huang, Vladimir Rokhlin & Stephen Wandzura. 1998. Accelerating fast multipole methods for the helmholtz equation at low frequencies. *IEEE Computational Science and Engineering* 5(3). 32–38.

Hamada, Tsuyoshi, Tetsu Narumi, Rio Yokota, Kenji Yasuoka, Keigo Nitadori & Makoto Taiji. 2009. 42 tflops hierarchical n-body simulations on gpus with applications in both astrophysics and turbulence. In *Proceedings of the conference on high performance computing networking, storage and analysis* SC '09, 62:1–62:12. New York, NY, USA: ACM. doi:10.1145/1654059.1654123. http://doi.acm.org/10.1145/1654059.1654123.

von Hoerner, S. 2001. How it all started. In *Dynamics of star clusters and the milky way*, vol. 228, 11.

Hughey, S., H. M. Aktulga & B. Shanker. 2018. Scalable and accurate tree traversal operators for helmholtz fmm. *under submission* .

Hughey, S., H. M. Aktulga, M. Vikram, M. Lu, B. Shanker & E. Michielssen. 2019. Parallel wideband mlfma for analysis of electrically large, nonuniform, multiscale structures. *IEEE Transactions on Antennas and Propagation* 67(2). 1094–1107. doi:10.1109/TAP.2018.2882621.

Hughey, Stephen Michael. 2018. *Efficient parallelization of non-uniform fast multipole algorithms*.

Huttunen, Tomi, Matti Malinen, Jari P Kaipio, Phillip Jason White & Kullervo Hynynen. 2005. A full-wave helmholtz model for continuous-wave ultrasound transmission. *IEEE transactions on ultrasonics, ferroelectrics, and frequency control* 52(3). 397–409.

Ishiyama, Tomoaki, Keigo Nitadori & Junichiro Makino. 2012. 4.45 pflops astrophysical n-body simulation on k computer: The gravitational trillion-body problem. In *Proceedings of the international conference on high performance computing, networking, storage and analysis* SC '12, 5:1–5:10. Los Alamitos, CA, USA: IEEE Computer Society Press. http://dl.acm.org/citation.cfm?id=2388996.2389003.

Jakob-Chien, R & B K Alpert. 1997. A fast spherical filter with uniform resolution. *J. Comp. Phys.* 136. 580–584.

Lashuk, Ilya, Aparna Chandramowlishwaran, Harper Langston, Tuan-Anh Nguyen, Rahul Sampath, Aashay Shringarpure, Richard Vuduc, Lexing Ying, Denis Zorin & George Biros. 2012. A massively parallel adaptive fast multipole method on heterogeneous architectures. *Communications of the ACM* 55(5). 101–109.

Lingg, Michael P, Stephen M Hughey & Hasan Metin Aktulga. 2018. Optimization of the spherical harmonics transform based tree traversals in the helmholtz fmm algorithm. In *Proceedings of the 47th international conference on parallel processing*, 1–11.

Lingg, Michael P., Stephen M. Hughey, Hasan Metin Aktulga & Balasubramaniam Shanker. 2020. High performance evaluation of helmholtz potentials using the multi-level fast multipole algorithm.

LóPez-PortuguéS, Miguel, JesúS A LóPez-FernáNdez, Jonatan MenéNdez-Canal, Alberto RodríGuez-Campa & José Ranilla. 2012. Acoustic scattering solver based on single level fmm for multi-gpu systems. *Journal of Parallel and Distributed Computing* 72(9). 1057–1064.

Melapudi, Vikram, Balasubramaniam Shanker, Sudip Seal & Srinivas Aluru. 2011. A scalable parallel wideband MLFMA for efficient electromagnetic simulations on large scale clusters. *Antennas and Propagation, IEEE Transactions on* 59(7). 2565–2577.

Michiels, Bart, Jan Fostier, Ignace Bogaert & Daniel De Zutter. 2013a. Performing large full-wave simulations by means of a parallel MLFMA implementation. In *Antennas and propagation society international symposium (apsursi), 2013 ieee*, 1880–1881. IEEE.

Michiels, Bart, Jan Fostier, Ignace Bogaert & Daniël De Zutter. 2013b. Weak scalability analysis of the distributed-memory parallel MLFMA. *Antennas and Propagation, IEEE Transactions on* 61(11). 5567–5574.

Michiels, Bart, Jan Fostier, Ignace Bogaert & Daniel De Zutter. 2015. Full-Wave Simulations of Electromagnetic Scattering Problems With Billions of Unknowns. *Antennas and Propagation, IEEE Transactions on* 63(2). 796–799.

Michiels, Bart, Jan Fostier, Ignace Bogaert, Piet Demeester & Daniël De Zutter. 2011. Towards a scalable parallel MLFMA in three dimensions. In *Computational electromagnetics international workshop (cem), 2011*, 132–135. IEEE.

Nishimura, Naoshi. 2002. Fast multipole accelerated boundary integral equation methods. *Applied mechanics reviews* 55(4). 299–324.

Pi, Wei-Chao, Xiao-Min Pan & Xin-Qing Sheng. 2010. A parallel multilevel fast multipole algorithm based on openmp. In *2010 international conference on microwave and millimeter wave technology*, 1356–1359. IEEE.

Potter, Douglas, Joachim Stadel & Romain Teyssier. 2017. Pkdgrav3: beyond trillion particle cosmological simulations for the next era of galaxy surveys. *Computational Astrophysics and Cosmology* 4(1). 2.

Rahimian, Abtin, Ilya Lashuk, Shravan Veerapaneni, Aparna Chandramowlishwaran, Dhairya Malhotra, Logan Moon, Rahul Sampath, Aashay Shringarpure, Jeffrey Vetter, Richard Vuduc, Denis Zorin & George Biros. 2010. Petascale direct numerical simulation of blood flow on 200k cores and heterogeneous architectures. In *Proceedings of the 2010 acm/ieee international conference for high performance computing, networking, storage and analysis* SC '10, 1–11. Washington, DC, USA: IEEE Computer Society. doi:10.1109/SC.2010.42. `http://dx.doi.org/10.1109/SC.2010.42`.

Salmon, John K. 1991. *Parallel hierarchical n-body methods*: California Institute of Technology dissertation.

Salmon, John K & Michael S Warren. 1994. Fast parallel tree codes for gravitational and fluid dynamical n-body problems. *The International Journal of Supercomputer Applications and High Performance Computing* 8(2). 129–142.

Sarvas, J. 2003. Performing interpolation and anterpolation by the fast fourier transform in the 3d multilevel fast multipole algorithm. *SIAM J. Numer. Anal.* 41. 2180–2196.

Shanker, B., A.A. Ergin, Mingyu Lu & E. Michielssen. 2003. Fast analysis of transient electromagnetic scattering phenomena using the multilevel plane wave time domain algorithm. *Antennas and Propagation, IEEE Transactions on* 51(3). 628–641. doi:10.1109/TAP.2003.809054.

Shanker, B. & H. Huang. 2007. Accelerated cartesian expansions - a fast method for computing of potentials of the form r^{- ν} for all real ν. *Journal of Computational Physics* 226. 732–753.

Shimada, Jiro, Hiroki Kaneko & Toshikazu Takada. 1994. Performance of fast multipole methods for calculating electrostatic interactions in biomacromolecular simulations. *Journal of Computational Chemistry* 15(1). 28–43.

Song, J. M. & W. C. Chew. 1995. Multilevel fast-multipole algorithm for solving combined field integral equations of electromagnetic scattering. *Microwave and Optical Technology Letters* 10(1). 14–19.

Song, J. M., C. C. Lu & W. C. Chew. 1997. Mlfma for electromagnetic scattering by large complex objects. *IEEE Transactions on Antennas and Propagation* 45. 1488–1493.

Sundar, Hari, Rahul S Sampath & George Biros. 2008. Bottom-up construction and 2: 1 balance refinement of linear octrees in parallel. *SIAM Journal on Scientific Computing* 30(5). 2675–2708.

Taboada, Jose Manuel, Marta G Araujo, Fernando Obelleiro Basteiro, José Luis Rodríguez & Luis Landesa. 2013. MLFMA-FFT parallel algorithm for the solution of extremely large problems in electromagnetics. *Proceedings of the IEEE* 101(2). 350–363.

Velamparambil, S, Jiming Song & Weng Cho Chew. 2000. On the parallelization of electrodynamic multilevel fast multipole method on distributed memory computers. In *Innovative architecture for future generation high-performance processors and systems, 1999. international workshop*, 3–11. IEEE.

Vikram, M., He Huang, B. Shanker & T. Van. 2009. A novel wideband fmm for fast integral equation solution of multiscale problems in electromagnetics. *Antennas and Propagation, IEEE Transactions on* 57(7). 2094–2104. doi:10.1109/TAP.2009.2019926.

Vikram, M., C. Knowles, B. Shanker & L.C. Kempel. 2011. An ultra-wideband fmm for multiscale electromagnetic simulations. *27th Annual Review of Progress in Applied Computational Electromagnetics* .

Vikram, M. & B. Shanker. 2009. An incomplete review of fast multipole methods from static to wideband as applied to problems in computational electromagnetics. *Applied Computational Electromagnetics Society Journal* 27. 79.

Waltz, Caleb, Kubilay Sertel, Michael Carr, Brian C Usner, John L Volakis & Others. 2007. Massively parallel fast multipole method solutions of large electromagnetic scattering problems. *Antennas and Propagation, IEEE Transactions on* 55(6). 1810–1816.

Wang, Ruoxi, Chao Chen, Jonghyun Lee & Eric Darve. 2019. Pbbfmm3d: a parallel black-box fast multipole method for non-oscillatory kernels. *arXiv preprint arXiv:1903.02153* .

Warren, M.S. & J.K. Salmon. 1993. A parallel hashed oct-tree n-body algorithm. In *Proc. supercomputing*, 1–12.

Wedi, Nils P, Mats Hamrud & George Mozdzynski. 2013. A fast spherical harmonics transform for global nwp and climate models. *Monthly Weather Review* 141(10). 3450–3461.

Yang, Ming-Lin, Bi-Yi Wu, Hong-Wei Gao & Xin-Qing Sheng. 2019. A ternary parallelization approach of mlfma for solving electromagnetic scattering problems with over 10 billion unknowns. *IEEE Transactions on Antennas and Propagation* .

Ying, Lexing, George Biros & Denis Zorin. 2004. A kernel-independent adaptive fast multipole algorithm in two and three dimensions. *Journal of Computational Physics* 196(2). 591–626.

Ying, Lexing & Denis Zorin. 2004. A simple manifold-based construction of surfaces of arbitrary smoothness. *ACM Transactions on Graphics* 23. 271–275. doi:http://doi.acm.org/10.1145/1015706.1015714. `http://doi.acm.org/10.1145/1015706.1015714`.

Yokota, Rio. 2013. An fmm based on dual tree traversal for many-core architectures. *Journal of Algorithms & Computational Technology* 7(3). 301–324.

Zhao, Jun-Sheng & Weng Cho Chew. 2000. Integral equation solution of maxwells equations from zero frequency to microwave frequencies. *IEEE Transactions on Antennas and Propagation* 48. 1635–1645.