2020 IEEE 27th International Conference on High Performance Computing, Data, and Analytics (HiPC) | 978-1-6654-2292-5/20/$31.00 ©2020 IEEE | DOI: 10.1109/HIPC50609.2020.00018

Exploring Task Parallelism for the Multilevel Fast Multipole Algorithm

Michael P. Lingg
Computer Science and Engineering
Michigan State University
East Lansing, MI
linggmic@msu.edu

Stephen M. Hughey
Electrical and Computer Engineering
Michigan State University
East Lansing, MI
hugheyst@msu.edu

Doğa Dikbayır
Computer Science and Engineering
Michigan State University
East Lansing, MI
dikbayir@msu.edu

Balasubramaniam Shanker
Electrical and Computer Engineering
Michigan State University
East Lansing, MI
bshanker@msu.edu

Hasan Metin Aktulga
Computer Science and Engineering
Michigan State University
East Lansing, MI
hma@cse.msu.edu

***Abstract*—The Multi-Level Fast Multipole Algorithm (MLFMA), a variant of the fast multiple method (FMM) for problems with oscillatory potentials, significantly accelerates the solution of problems based on wave physics, such as those in electromagnetics and acoustics. Existing shared memory parallel approaches for MLFMA have adopted the bulk synchronous parallel (BSP) model. While the BSP approach has served well so far, it is prone to significant thread synchronization overheads, but more importantly fails to leverage the communication/computation overlap opportunities due to complicated data dependencies in MLFMA. In this paper, we develop a task parallel MLFMA implementation for shared memory architectures, and discuss optimizations to improve its performance. We then evaluate the new task parallel MLFMA implementation against a BSP implementation for a number of geometries. Our findings suggest that task parallelism is generally superior to the BSP model, and considering its potential advantages over the BSP model in a hybrid parallel setting, we see it to be a promising approach in addressing the scalability issues of MLFMA in large scale computations.**

I. Introduction

Hyperbolic partial differential equations (PDEs) are used in a wide range of physics simulations, including several problems in electromagnetic and acoustics. These simulations can range from basic simulations to accelerate prototype development to very complex simulations to explore phenomena that are otherwise impossible to observe experimentally. Many modern technologies, such as as wireless communication devices, passive and active RF-IDs, optical and terahertz devices, sonar and radar emitters in automobiles, microwaves, medical diagnostic and imaging tools, rely on a strong understanding of electromagnetic interactions described by the Helmholtz equations and requires analysis on increasingly larger and finer structures. Such analysis are heavily reliant on high-fidelity computational tools which in turn are reliant on improvements in algorithmic and computational efficiency.

Many modern problems with interactions described by the Helmholtz equation involve simulation domains covering several thousand wavelengths. The most expensive component in solving the Helmholtz integral equation is evaluating the *oscillatory* interactions in the N-body problem corresponding to the PDE. The Multi-level Fast Multipole Algorithm (MLFMA) for Helmholtz equations reduces the $\mathcal{O}(N_s^2)$ cost of the direct potential evaluation down to $\mathcal{O}(N_s \log N_s)$ [1] for surface distributions (which constitute the most relevant problem types), with N_s being the number of degrees of freedom. While MLFMA provides a highly favorable algorithmic complexity, its overall computational and memory costs for applications requires leveraging the computational power of HPC systems through parallelization and performance optimizations. The closely related Laplace FMM (L-FMM) method, which has been developed for *non-oscillatory* potentials such as gravitational or electrostatic fields, preceded the MLFMA variant. Development of efficient L-FMM implementations has been highly successful in terms of performance and scalability [2]–[4], but this stands in stark contrast to MLFMA implementations [5]–[8], mainly due to the complex computation patterns and memory requirements of MLFMA.

MLFMA belongs to the broader class of tree algorithms [9]–[14] used to accelerate N-body computations. Broadly speaking, there are two types of MLFMA implementations, those using *local interpolation* [15]–[17] and those using *global interpolation* [18]. As demonstrated in several related works, in large-scale computations inter-process communication overheads significantly hamper the scalability of both versions, but more so of the global interpolation based version [19]. In this paper, we focus on the global interpolation version of MLFMA, as it provides arbitrary error control and is known to be favorable over the local interpolation

This work was supported by the National Science Foundation under Grant No. CCF-1822932. This research used resources of Michigan State University's High Performance Computing Center (MSU HPCC) and the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231.

978-1-6654-2292-5/20/$31.00 ©2020 IEEE
DOI 10.1109/HiPC50609.2020.00018

version in terms of computational and memory costs [20], [21]. As investigated in our recent study [19], despite careful optimizations, the strong scaling efficiency of an MPI-only parallel implementation of global interpolation based MLFMA drops precipitously beyond a few thousand cores. Obviously, on modern systems with massive on-node parallelism, a hybrid process and thread parallel approach can help alleviate the onset of communication overheads as threads on the same node can share data through the shared memory address space in place of inter-process communications. Furthermore, a hybrid parallel approach can lead to significant memory savings as it would be sufficient to store a single copy of interaction tables (which are static across all processes) per node, rather than a single copy per process as in an MPI-only approach.

MLFMA is a complex algorithm with several stages, each of which has different characteristics. While some stages are computation-intensive, others involve frequent communication operations. Existing hybrid parallelization approaches for MLFMA have adopted the bulk synchronous parallel (BSP) model where expensive loops within each stage are parallelized in a synchronous manner. While the BSP approach has served well so far, it fails to leverage the communication/computation overlap opportunities across different MLFMA stages or even within each stage for that matter (unless one opts for a complicated implementation that requires hand-tuning for the target architecture and problem). Furthermore, frequent synchronizations across all threads as required by the BSP model can hamper efficiency on many-core architectures. On the other hand, a task parallel approach based on the dataflow model (which is supported by several runtime systems) can alleviate both shortcomings outlined above. By enumerating all tasks with their data dependencies across all MLFMA stages, compute-intensive and communication-intensive parts of MLFMA can progress simultaneously which can potentially improve the scalability of MLFMA. Moreover, in a dataflow model, threads would need to synchronize only with those threads that are producers of the data on which they depend.

For reasons outlined above, exploring the pros and cons of task parallelism for MLFMA is of broad interest. In this paper, we take a step in this direction and develop a task parallel MLFMA implementation for shared memory architectures, and discuss optimizations to improve its performance. We then evaluate the new task parallel MLFMA implementation against a BSP implementation for a number of geometries. Our findings suggest that task parallelism is generally superior to the BSP model and considering its potential advantages over the BSP model in a distributed memory setting, we see it to be a promising approach in addressing the scalability issues of MLFMA in large scale computations.

II. Background and Related Work

A. Fast Multipole Method (FMM)

The first step of the FMM algorithm is to recursively subdivide the computational domain into cubes until the smallest desired box size, or a target number of particles per box, is reached. This subdivision is then used to create an octree structure to provide a hierarchical representation of the domain. To accelerate the computation, interactions among particles are approximated over the tree structure rather evaluating them directly between all pairs. Within the tree hierarchy, interactions are classified as *near field* and *far field* interactions. Two boxes are considered within each other's *near field*, if they share any face, edge or corner; interactions among particles inside such box pairs are evaluated directly. Two boxes are considered in each other's *far field*, if their parent boxes are in each other's *near field*, but the child boxes themselves are not touching each other. In case of far field boxes, interactions among particles are approximated through multipole expansions. More specifically, the procedure below is followed (further mathematical details can be found in [10], [11], [22], [23]):

1) Compute charge to multipole information for each leaf node based on the particles it encloses (C2M),
2) compute the multipole expansions for each node in the tree by traversing up the tree and interpolating from the multipole information of all of its children (M2M),
3) calculate interactions between far-field pairs by translating multipole expansions of sources to the observers' locations (M2L), these translated expansions are then referred to as *local expansions*
4) starting at the highest level nodes and traversing all the way down to the leaves, distribute (anterpolate) local expansions aggregated at non-leaf observer boxes as a result of M2L translations to their children (L2L),
5) convert the resulting local expansions at each leaf box to particles enclosed therein (L2O).

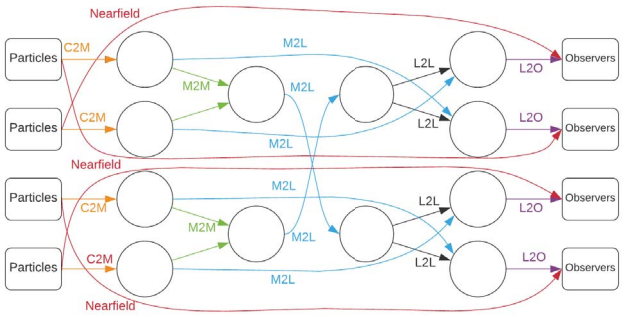

Fig. 1. Dependencies between boxes within an FMM octree due to the nearfield and farfield computation process.

Figure 1 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 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. 1 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.

B. 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. [24] and Yokota et al. [25] 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 [25]. 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 [26]. 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 [27], 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.

C. 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.

III. Methods

A. MLFMA with BSP

Applying the BSP model in MLFMA is relatively straightforward, 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.

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.

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.

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) 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.

B. 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.

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.

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 IV, 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. 1.

Algorithm 1 Task-based upward tree traversal

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

```
for each box p in post-order traversal do
    if p is leaf box then
        task Depend Out box p
            pmp ← C2M(p)
        end task
    else
        for each child box c do
            task Depend In all child box c Depend Out box p
                mp[c] ← interpolation(c)
                smp[c] ← shift(mp, p.center)
                aggregate(pmp,smp[c])
            end task
        end for
    end if
end for
```

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 paper 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 can be partitioned into block columns or rows. FFT-based interpolation of these partitions are also independent of each other [18], [19]. Hence, we create tasks for interpolations of partitions. We illustrate the fine-grained task parallel M2M method used for high level nodes in Alg.2.

Algorithm 2 Parallel Interpolation

Ensure: c is the child box being interpolated
1: $pts \leftarrow$ partition(c)
2: **for** each partition p in pts **do**
3: **task**
4: **for** each θ vector v in p **do**
5: $theta$[v] $\leftarrow$ interpolate(v)
6: $shift1$[v] $\leftarrow$ transposeandfold($theta$[v])
7: **end for**
8: **end task**
9: **end for**
10: **TaskWait**
11: $pts \leftarrow$ partition($shift1$)
12: **for** each partition p in pts **do**
13: **task**
14: **for** each ϕ vector v in p **do**
15: phi[v] $\leftarrow$ interpolate(v)
16: $shift1$[v] $\leftarrow$ transposeandfold(phi[v])
17: **end for**
18: **end task**
19: **end for**
20: **TaskWait**

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 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 IV for further details. We provide the pseudocode for our task-parallel M2L implementation with bundling in Alg. 3.

Algorithm 3 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: $int \leftarrow$ compute_interaction(fb,b)
5: $lp[b] \leftarrow$ add_interaction(int)
6: **end task**
7: **end for**
8: **end for**

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.

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.

IV. 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.

A. Tuning the Task-Parallel MLFMA Implementation

As mentioned in III, 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

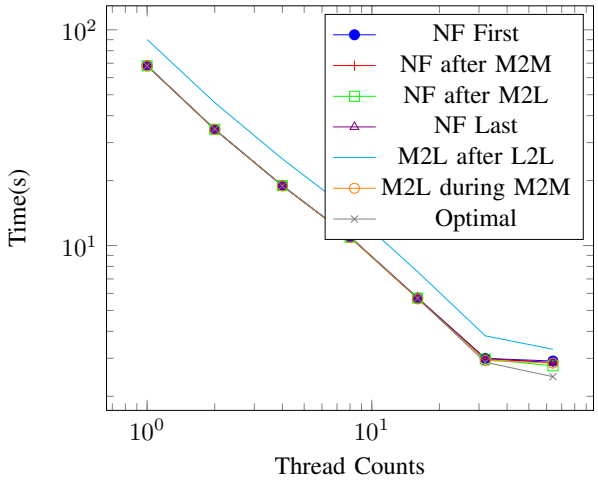

Fig. 2. Impact of task order on execution time.

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.

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

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

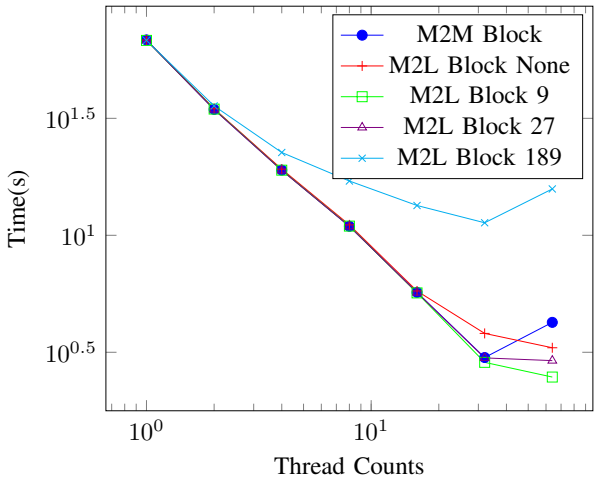

Fig. 3. Impact of bundling on execution time. Performance of M2M implementation with all children bundled into a single task (M2M with Bundling) and M2L with various bundling factors (none, 9, 27, and 189) are shown for different thread counts.

bundling all children into a single task during M2M performed significantly worse at 64 threads (see Fig. 3). 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. 3. 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 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.

B. 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.

1) Performance on a Multicore Architecture (Cori-Haswell): Figure 4 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.

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. 5). 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 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.

C. 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.

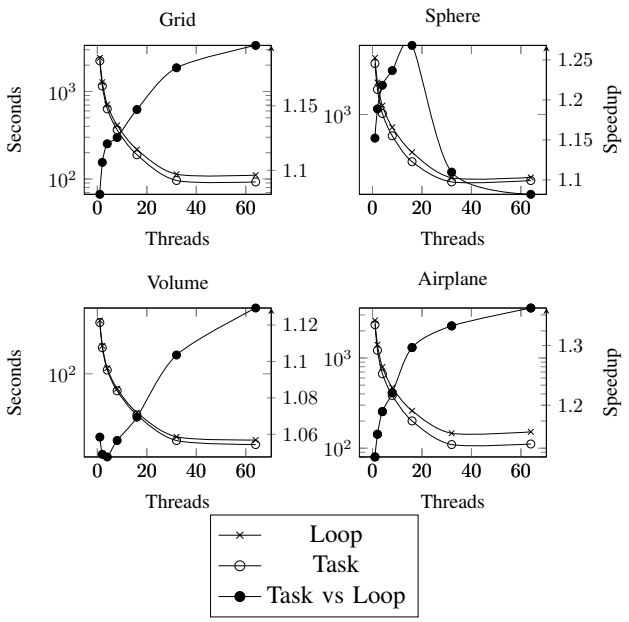

Fig. 4. Task vs Loop (BSP) parallel runs on Haswell compute nodes for four different geometries.

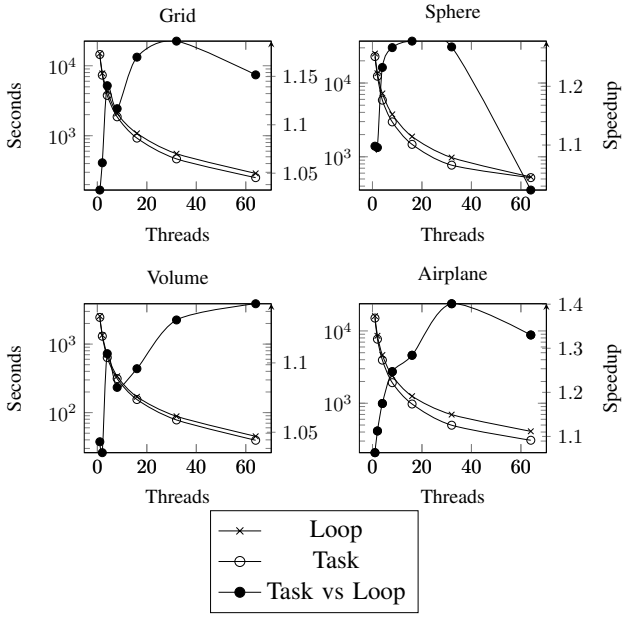

Fig. 5. Task vs Loop (BSP) parallel runs on KNL compute nodes for four different geometries.

1) Timeline Analysis: Figure 6 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 7 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.

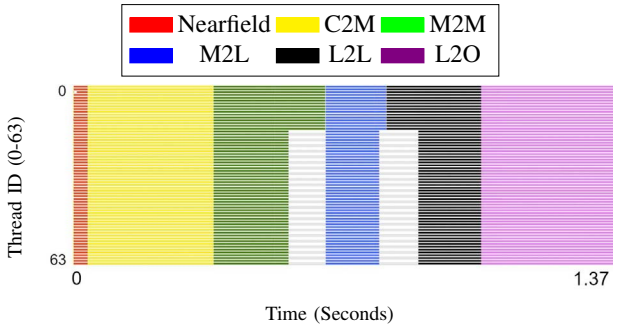

Fig. 6. BSP timeline on grid geometry.

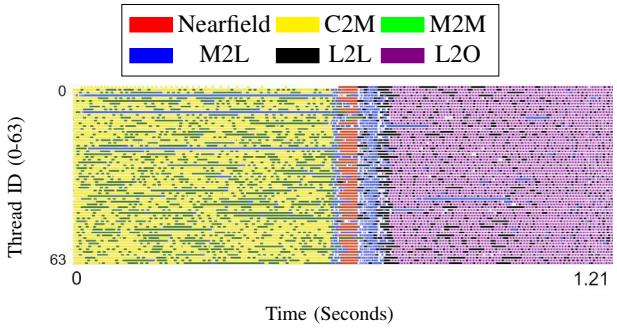

Fig. 7. Task parallel timeline on grid geometry.

Figures 8 and 9 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 executed until dependent tasks complete. As a result, the task parallel approach is not as efficient for this example.

Alternately, Figures 10 and 11 show how the BSP and task parallel timeline behave for an airplane geometry. Unlike the

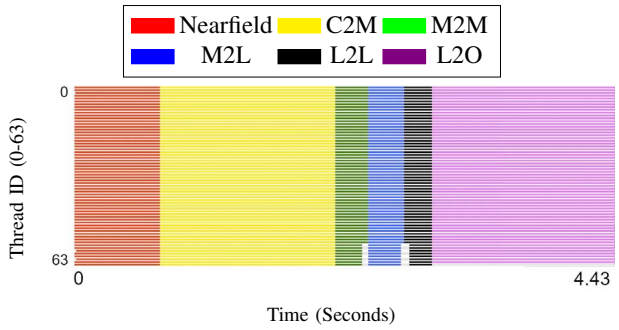

Fig. 8. BSP timeline on the sphere geometry.

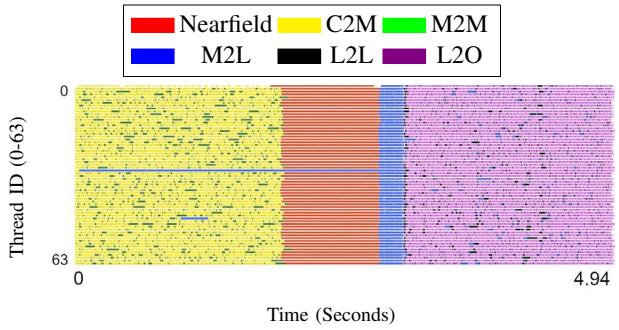

Fig. 9. Task parallel timeline on the sphere geometry.

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 10 where the top levels of M2M and L2L have fewer nodes that can be processed. Furthermore, the next level down still has a limited number of nodes to process. Figure 11 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.

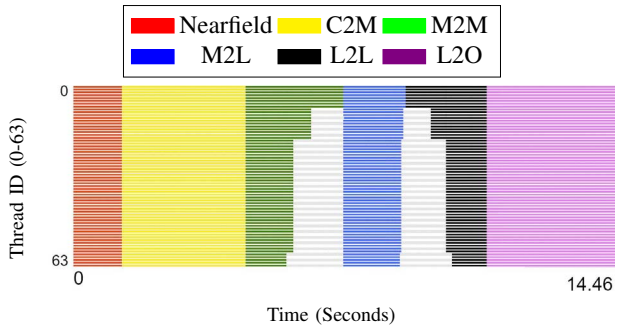

Fig. 10. BSP timeline on the airplane geometry.

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. 12 show that the ratio

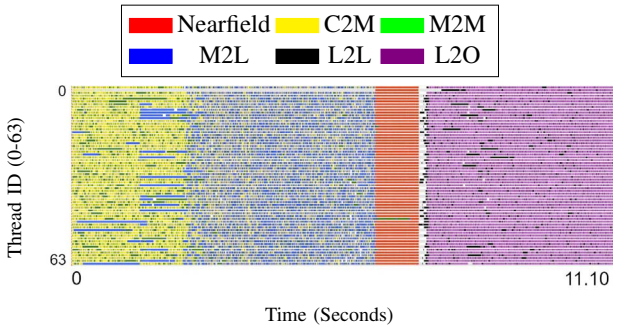

Fig. 11. Task parallel timeline on the airplane geometry.

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.

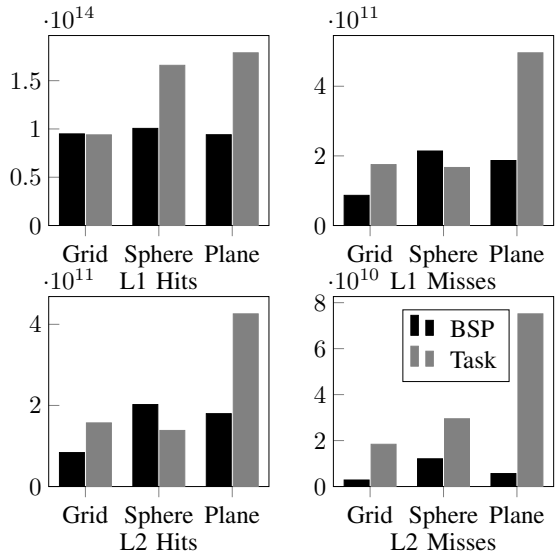

Fig. 12. Comparison of the cache performance of BSP and task parallel implementations on a Cori-Haswell node.

V. 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 paper, 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 provides 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.

REFERENCES

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

[2] I. Lashuk, A. Chandramowlishwaran, H. Langston, T.-A. Nguyen, R. Sampath, A. Shringarpure, R. Vuduc, L. Ying, D. Zorin, and G. Biros, "A massively parallel adaptive fast-multipole method on heterogeneous architectures," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. IEEE, 2009, pp. 1–12.

[3] T. Hamada, T. Narumi, R. Yokota, K. Yasuoka, K. Nitadori, and M. Taiji, "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*, ser. SC '09. New York, NY, USA: ACM, 2009, pp. 62:1–62:12. [Online]. Available: http://doi.acm.org/10.1145/1654059.1654123

[4] A. Rahimian, I. Lashuk, S. Veerapaneni, A. Chandramowlishwaran, D. Malhotra, L. Moon, R. Sampath, A. Shringarpure, J. Vetter, R. Vuduc *et al.*, "Petascale direct numerical simulation of blood flow on 200k cores and heterogeneous architectures," in *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2010, pp. 1–11.

[5] O. Ergul, "Parallel implementation of MLFMA for homogeneous objects with various material properties," *Progress In Electromagnetics Research*, vol. 121, pp. 505–520, 2011.

[6] V. Melapudi, B. Shanker, S. Seal, and S. Aluru, "A scalable parallel wideband MLFMA for efficient electromagnetic simulations on large scale clusters," *Antennas and Propagation, IEEE Transactions on*, vol. 59, no. 7, pp. 2565–2577, 2011.

[7] B. Michiels, J. Fostier, I. Bogaert, and D. De Zutter, "Performing large full-wave simulations by means of a parallel MLFMA implementation," in *Antennas and Propagation Society International Symposium (APSURSI), 2013 IEEE*. IEEE, 2013, pp. 1880–1881.

[8] M.-L. Yang, B.-Y. Wu, H.-W. Gao, and X.-Q. Sheng, "A ternary parallelization approach of mlfma for solving electromagnetic scattering problems with over 10 billion unknowns," *IEEE Transactions on Antennas and Propagation*, 2019.

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

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

[11] N. Nishimura, "Fast multipole accelerated boundary integral equation methods," *Applied mechanics reviews*, vol. 55, no. 4, pp. 299–324, 2002.

[12] A. Appel, "An efficient program for many-body simulations," *SIAM J. Sci. Comput.*, vol. 6, pp. 85–103, 1985.

[13] J. Barnes and P. Hut, "A hierarchical $((n \log n)$ force calculation algorithm," *Nature*, vol. 324, pp. 446–449, 1986.

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

[15] W. C. Chew, E. Michielssen, J. Song, and J.-M. Jin, *Fast and efficient algorithms in computational electromagnetics*. Artech House, Inc., 2001.

[16] Ö. Ergül and L. Gürel, "Hierarchical parallelisation strategy for multilevel fast multipole algorithm in computational electromagnetics," *Electronics Letters*, vol. 44, no. 1, pp. 3–5, 2008.

[17] B. Michiels, J. Fostier, I. Bogaert, and D. De Zutter, "Weak scalability analysis of the distributed-memory parallel MLFMA," *Antennas and Propagation, IEEE Transactions on*, vol. 61, no. 11, pp. 5567–5574, 2013.

[18] S. Hughey, H. M. Aktulga, M. Vikram, M. Lu, B. Shanker, and E. Michielssen, "Parallel wideband mlfma for analysis of electrically large, nonuniform, multiscale structures," *IEEE Transactions on Antennas and Propagation*, vol. 67, no. 2, pp. 1094–1107, Feb 2019.

[19] M. P. Lingg, S. M. Hughey, H. M. Aktulga, and B. Shanker, "High performance evaluation of helmholtz potentials using the multi-level fast multipole algorithm," 2020.

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

[21] M. Vikram, H. Huang, B. Shanker, and T. Van, "A novel wideband fmm for fast integral equation solution of multiscale problems in electromagnetics," *Antennas and Propagation, IEEE Transactions on*, vol. 57, no. 7, pp. 2094–2104, July 2009.

[22] L. Greengard, J. Huang, V. Rokhlin, and S. Wandzura, "Accelerating fast multipole methods for the helmholtz equation at low frequencies," *IEEE Computational Science and Engineering*, vol. 5, pp. 32–38, 1998.

[23] S. Wandzuraz, "The Fast Multipole Method for the Wave Equation: A Pedestrian Prescription," *IEEE Antennas and Propagation Magazine*, vol. 35, no. 3, pp. 7–12, 1993.

[24] E. Agullo, B. Bramas, O. Coulaud, E. Darve, M. Messner, and T. Takahashi, "Task-based fmm for multicore architectures," *SIAM Journal on Scientific Computing*, vol. 36, no. 1, pp. C66–C93, 2014.

[25] R. Yokota, "An fmm based on dual tree traversal for many-core architectures," *Journal of Algorithms & Computational Technology*, vol. 7, no. 3, pp. 301–324, 2013.

[26] W.-C. Pi, X.-M. Pan, and X.-Q. Sheng, "A parallel multilevel fast multipole algorithm based on openmp," in *2010 International Conference on Microwave and Millimeter Wave Technology*. IEEE, 2010, pp. 1356–1359.

[27] M. Abduljabbar, M. A. Farhan, N. Al-Harthi, R. Chen, R. Yokota, H. Bagci, and D. Keyes, "Extreme scale fmm-accelerated boundary integral equation solver for wave scattering," *SIAM Journal on Scientific Computing*, vol. 41, no. 3, pp. C245–C268, 2019.