A

A Distributed Memory Fast Multipole Method for Volume Potentials

DHAIRYA MALHOTRA, The University of Texas at Austin, Austin, TX 78712
GEORGE BIROS, The University of Texas at Austin, Austin, TX 78712

The solution of a constant-coefficient elliptic partial differential equation (PDE) can be computed using an integral transform: a convolution with the fundamental solution of the PDE, also known as a volume potential. We present a Fast Multipole Method (FMM) for computing volume potentials and use them to construct spatially-adaptive solvers for the Poisson, Stokes and Helmholtz problems. Conventional N-body methods apply to discrete particle interactions. With volume potentials, one replaces the sums with volume integrals. Particle N-body methods can be used to accelerate such integrals but it is more efficient to develop a special FMM. In this paper, we discuss the efficient implementation of such an FMM. We use high-order piecewise Chebyshev polynomials and an octree data structure to represent the input and output fields, enable spectrally accurate approximation of the near field, and the kernel independent FMM (KIFMM) for the far field approximation. For distributed memory parallelism, we use space filling curves, locally essential trees, and a hypercube-like communication scheme developed previously in our group. We present new near and far interaction traversals which optimize cache usage. Also, unlike particle N-body codes, we need a 2:1 balanced tree to allow for precomputations. We present a fast scheme for 2:1 balancing. Finally, we use vectorization, including the AVX instruction set on the Intel Sandy Bridge architecture to get over 50% of peak floating point performance. We use task parallelism to employ the Xeon Phi on the Stampede platform at the Texas Advanced Computing Center (TACC).

We achieve about 600GFlop/s of double precision performance on a single node. Our largest run on Stampede took 3.5s on 16K cores for a problem with 18E+9 unknowns for a highly nonuniform particle distribution (corresponding to an effective resolution exceeding 2E+23 unknowns since we used 23 levels in our octree).

Categories and Subject Descriptors: G.4 [**MATHEMATICAL SOFTWARE**]

Additional Key Words and Phrases: FMM, N-Body Problems, Potential theory

ACM Reference Format:
Dhairya Malhotra and George Biros, 2013. A distributed memory fast multipole method for volume potentials. *ACM Trans. Math. Softw.* V, N, Article A (January YYYY), 38 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

We consider the problem of rapidly evaluating integrals of the form

$$u(x) = \int_{\Omega} K(x-y) f(y), \tag{1}$$

where $f(y)$ is a given function (the "*source density*", Figure 1: Left), $u(x)$ is the sought function (the "*potential*", Figure 1: Right), and $K(x-y)$ is a given function (the "*kernel*"). In our context, K will be the fundamental solution of elliptic partial differential equation. The domain of integration Ω is the unit cube enclosing the support of $f(y)$.

Author's addresses: D. Malhotra and G. Biros, The University of Texas at Austin, Austin, TX 78712.

DOI: http://dx.doi.org/10.1145/0000000.0000000

If K is smooth for all x, then we can discretize (1) using a N-point quadrature with $\{w_j, y_j\}_{j=1}^N$ weights and points. That is to obtain $u_i = u(x_i)$ at N target points $\{x_i\}_{i=1}^N$ we compute

$$u_i = \sum_{j=1}^{N} K(x_i - y_j)\, w_j f(y_j) = \sum_{j=1}^{N} K_{ij} f_j, \tag{2}$$

with $f_j = w_j f(y_j)$. This calculation requires $\mathcal{O}(N^2)$ work. Depending on K it may be possible to use an N-body method to compute this sum.

This sum resembles the particle fast-multipole method (FMM). The particle FMM was first introduced for $K = 4\pi/\|x - y\|$, the fundamental solution of the Laplacian, and has been extended to many other kernels. The particle FMM evaluates sums of the form $\sum_{j=1}^N K_{ij} f_j$ in $\mathcal{O}(N)$ work to any specified accuracy. Notice that K has a singularity when $x_i = y_j$, which is the typical case for the fundamental solutions of PDEs. The particle FMM assumes that either $\|x_i - y_j\| \neq 0$ for all i, j, or if $\|x_i - y_j\| = 0$, it omits y_j from the summation for $u(x_i)$.

Applying directly the particle FMM on (1) using a smooth quadrature and omitting the singularity, results in a very slowly convergent scheme. Instead, we need product integration rules in which w_j depend on the evaluation point x. Product integration rules cannot be accelerated with FMM. To address this we observe that product integration is required only when $\|x-y\|$ is small. With this in mind, we select a parameter ζ and we split the integral to a near-x and a far-x part. That is, we write

$$u(x) = \int_{\mathcal{N}(x)} K(x-y) f(y) + \int_{\mathcal{F}(x)} K(x-y) f(y) \tag{3}$$

where $\mathcal{N}(x)$ is a ball of radius ζ around x and $\mathcal{F} := \Omega \backslash \mathcal{N}(x)$. The first integral corresponds to the near-interactions of the FMM, (see §2), but unlike particle summation its calculation is much more complicated as it requires special quadratures. The second integral is smooth and can be computed using a standard quadrature rule and can be accelerated using the FMM.

Note that an FMM particle code could be used to accelerate volume potentials as follows. We first compute a provisional u_p using (2), weights for smooth quadratures omitting j if $y_j = x_i$. Then we use a standard particle FMM to compute the sum. Finally, we perform a local correction $u = u_p - \sum_{j:y_j \in \mathcal{N}(x)} K_{ij} f_j + \int_{\mathcal{N}(x)} K(x-y) f(y)$. That is, we subtract the incorrect particle-FMM near interactions and add the correct near field by using accurate quadrature. (This approach is typical for accelerating boundary integrals.) Such an approach is easy to implement and it maintains linear complexity for the evaluation of (1). Its disadvantage is that it requires three near field calculations instead of just one. Most important a naive calculation of the singular integral using, say adaptive quadrature, on the fly, will result in an extremely slow scheme.

We opt not to use a particle FMM, but instead we directly approximate the integral and not the direct sum. This approach was first introduced in [Ethridge and Greengard 2001] in 2D and extended in 3D in [Langston et al. 2011]. The key aspect in the formulation in [Ethridge and Greengard 2001] is to represent $f(x)$ using a non-uniform octree and Chebyshev polynomials within each octant (Figure 1: Left). The near interactions ($\int_{\mathcal{N}(x)} K(x-y) f(y)$) require computing singular and near-singular integrals. To avoid costly integration, the near interactions are evaluated using precomputed linear transformations of octant source distribution in Chebyshev basis space. The latter is computationally feasible only if the tree is 2:1 balanced (for its definition, see §2.2.3).

Motivation and significance. Several applications require solutions of linear constant coefficient elliptic PDEs and their derivatives such as the Poisson, Stokes, Biot-

Savart, and Helmholtz problems. This motivates the need for high-order accurate, parallel and scalable techniques for solving such problems. Equation (1) provides the framework to build efficient solvers on the unit box. For example, for the Poisson problem

$$-\Delta u(x) = f(x), \quad \Rightarrow u(x) = \frac{1}{4\pi}\int_\Omega \frac{1}{\|x-y\|} f(y). \tag{4}$$

for free-space conditions for $u(x)$ and with $f(x) = 0$ for all $x \notin \Omega$. Equation (1) can also be used to solve more complex problems, for example indefinite vector problems like the Stokes problem:

$$\begin{aligned} -\Delta u(x) + \nabla p(x) = f(x),\ \operatorname{div} u(x) = 0, \quad \Rightarrow \\ u(x) = \frac{1}{8\pi}\int_\Omega \left(\frac{1}{\|x-y\|} + \frac{(x-y)(x-y)^T}{\|x-y\|^3} \right) f(y) \end{aligned} \tag{5}$$

where u is the velocity and p is the pressure. The Poisson solver finds application in incompressible fluids, and magnetohydrodynamics to enforce divergence free conditions, star formation and cosmological simulations, as well as quantum chemistry to approximate Gaussian distributions of charges. The Stokes solver is used for simulations of complex fluids in low Reynolds numbers with applications to polymers, biophysics, and geophysics. Besides free-space boundary conditions additional boundary conditions are possible to implement, depending on the kernel, for example Dirichlet, Neumann, or periodic boundary conditions.

Contributions. This paper builds on our previous work on the Kernel Independent FMM (KIFMM) [Lashuk et al. 2012; Ying et al. 2004] for particle N-body problems and the method of Greengard and Ethridge [Ethridge and Greengard 2001] for volume potentials. We discuss algorithmic modifications that significantly improve scalability.

— We present novel traversal schemes for the near and far interactions (also known as the FMM U, W, X, V lists) to optimize per-node performance (§3).
— We present integration of our with co-processors (Intel Xeon Phi and Nvidia GPU).
— We introduce a novel 2:1 balancing scheme (§3). This balancing is not needed in particle FMMs but it is a critical component of making scaling volume FMM codes possible.
— The single-node algorithmic refactoring and optimizations result in $7\times$ **speedup** over an optimized, multithreaded volume FMM implementation (see §4). We demonstrate the scalability of the method to thousands of cores for non-uniform distributions that use **25 levels of refinement**.

To our knowledge, our algorithm is among the fastest AMR constant-coefficient Poisson, Stoke and Helmholtz solvers. It achieves three main algorithmic goals: high-order approximation, linear work, excellent single-node performance, and parallel scalability.

Limitations. The main limitation of the proposed methodology is that it is appropriate for box geometries. Complex smooth geometries can be resolved using a combination of boundary integrals and volume potentials [Ying et al. 2006]. Variable coefficients also are not discussed here, but in a nutshell, one cannot simply use an integral transform, but instead one has to solve an integral equation using an iterative solver. A few additional optimizations were not implemented at the time of submission and they can potentially reduce the cost every further. For example overlapping of communication with computation when using message passing interface. Finally, our code is efficient up to 14-th order approximation of $f(x)$ at the leaf nodes. For higher-order

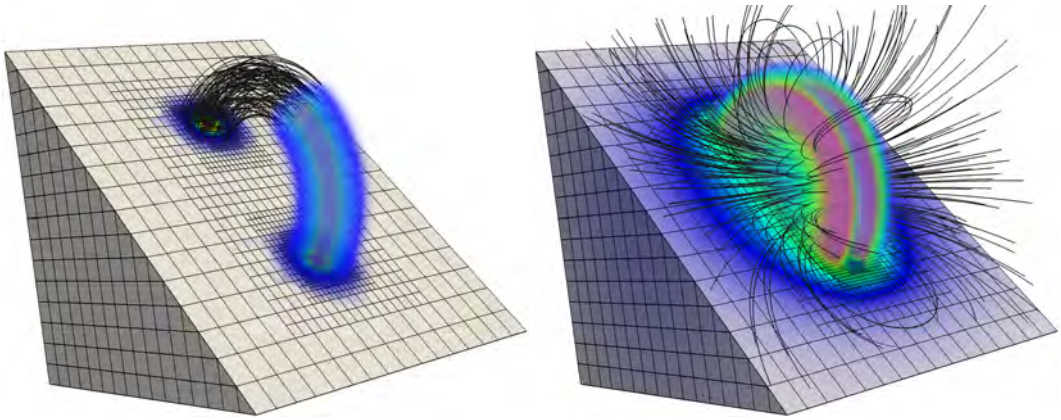


Fig. 1 *Left: Vorticity field for a vortex-ring resolved on an adaptive Chebyshev octree is the input to volume FMM. Right: Output velocity field obtained from FMM by computing convolution of the vorticity field with the Biot-Savart kernel.*

approximations, efficient implementation requires further compression techniques for the near and far interactions.

Related work. As mentioned, a volume potential FMM was first proposed in [Ethridge and Greengard 2001] and a basic 3D shared memory implementation (using OpenMP) was discussed in [Langston et al. 2011].

Due to the significance of Poisson and other boundary value problems, there has been a lot of effort to develop scalable solvers. Industrial vendors like Intel include Poisson solvers in the scientific computing libraries, but only for regular grid, for which Fast Fourier transforms can be used [Sbalzarini et al. 2006; Phillips et al. 2002; Phillips and White 1997; Dror et al. 2010; Nukada et al. 2012]. Here we are interested in solvers that enable highly nonuniform spatial resolution. Multigrid methods are effective and scalable [Sundar et al. 2012]. Yet no high-order accurate implementations exist that have scaled to thousands of cores (this is related to constructing efficient smoothers and aggregations schemes for high-order approximations). Other scalable approaches include hybrid domain decomposition methods [Lottes and Fischer 2005]. A very efficient Poisson solver is based on a non-iterative domain decomposition method [McCorquodale et al. 2006] using a low-order approximation scheme. In [Langston et al. 2011], that solver was compared with a high order volume potential FMM. The FMM solver required $4\times$–$100\times$ fewer unknowns. We discuss this more in §4. Also Multigrid and domain decomposition methods do not scale as well for indefinite vector operators like the Stokes problem [Benzi et al. 2005]. Another approach in solving Poisson problems using an integral transform is to smooth the kernel (for example, by replacing $1/\|x-y\|$ with $1/\sqrt{\varepsilon^2+\|x-y\|^2}$. This allows standard quadratures and the use of particle N-body codes. This approach works well for low accuracy computations but is somewhat problematic for high-accuracy calculations because ε has to be so small that the kernel becomes nearly singular [Lindsay and Krasny 2001].

There have been several works optimizing Fast Multipole Method on various architectures, including large distributed memory systems with accelerators. As we mentioned, our distributed memory FMM follows closely on our previous work on particle FMM [Lashuk et al. 2012]. Other scalable implementations of particle N-body codes include [Yokota et al. 2011; Hamada et al. 2009; Jetley et al. 2010; Hu et al. 2011]. The importance of blocking to explore locality was also discussed in [Chandramowlishwaran et al. 2010]. An efficient implementation for far-field interactions in black-box FMM was discussed in [Takahashi et al. 2012], however, this approach worked well

only for uniform particle distributions, using single precision computation on GPUs. None of the works on optimizing FMM performance discuss volume potentials.

Finally, let us discuss related work on 2:1 balancing. The challenge with 2:1 balancing is the distributed memory parallelization. Recent work includes the work of [Sundar et al. 2008; Burstedde et al. 2011; Isaac et al. 2012]. The main advantage of our scheme is that it is quite simple to implement.

Outline. In §2, we discuss the sequential algorithm for volume FMM, give complexity estimate and discuss the sources of errors. We also discuss the relevant background about particle FMM and kernel independent FMM. The material in §2 has appeared elsewhere but we include it so that the paper is self contained. Our main contributions are described in the remaining sections. In §3, we explain the parallel implementation of the method discussing distributed and shared memory parallelism, task parallelism in relation to co-processors, vector parallelism and several other optimizations. Finally, in §4, we present results for convergence, single node performance and scalability on Stampede and Titan supercomputers.

2. SEQUENTIAL ALGORITHMS

Particle N-body	
K	kernel function
Ω	computational domain: $[0,1]^3$
$y,\ q$	source: coordinates, density
$x,\ u$	target: coordinates, potential
$N_s,\ N_t$	number of source and target points
$\mathcal{T}$	octree
N_{oct}	number of octants
N_{leaf}	number of leaf octants
L_{max}	maximum tree depth
$\mathcal{B}$	tree node
$\mathcal{P}(\mathcal{B})$	parent of $\mathcal{B}$
$\mathcal{C}(\mathcal{B})$	children of $\mathcal{B}$
$\text{Level}(\mathcal{B})$	level of $\mathcal{B}$
Particle FMM	
m	multipole order: number of points on an edge of equivalent surface
$x^{u,\mathcal{B}},\ u^{u,\mathcal{B}}$	upward-check surface for $\mathcal{B}$: coordinates, potential
$y^{u,\mathcal{B}},\ q^{u,\mathcal{B}}$	upward-equivalent surface for $\mathcal{B}$: coordinates, density
$x^{d,\mathcal{B}},\ u^{d,\mathcal{B}}$	dnward check surface for $\mathcal{B}$: coordinates, potential
$y^{d,\mathcal{B}},\ q^{d,\mathcal{B}}$	dnward equivalent surface for $\mathcal{B}$: coordinates, density
$\mathcal{N}(\mathcal{B})$	near region of $\mathcal{B}$
$\mathcal{F}(\mathcal{B})$	far region of $\mathcal{B}$: $\Omega \setminus \mathcal{N}(\mathcal{B})$
Octant lists	
$\mathcal{J}(\mathcal{B})$	adjacent octants to $\mathcal{B}$ (arbitrary level)
$\mathcal{K}(\mathcal{B})$ (colleagues)	adjacent octants to $\mathcal{B}$, same level
$\mathcal{L}(\mathcal{T})$	all leaf nodes in $\mathcal{T}$
$\mathcal{U}(\mathcal{B})$	U-list: $\mathcal{J}(\mathcal{B}) \cap \mathcal{L}(\mathcal{T})$ (Note: $\mathcal{B} \in \mathcal{U}(\mathcal{B})$)
$\mathcal{V}(\mathcal{B})$	V-list: $\mathcal{C}(\mathcal{K}(\mathcal{P}(\mathcal{B}))) \setminus \mathcal{K}(\mathcal{B})$
$\mathcal{W}(\mathcal{B})$	W-list: $\mathcal{C}(\mathcal{K}(\mathcal{B})) \setminus \mathcal{J}(\mathcal{B})$
$\mathcal{X}(\mathcal{B})$	X-list: $\{\widehat{\mathcal{B}}:\ \ \widehat{\mathcal{B}} \in \mathcal{L}(\mathcal{T}),\ \ \mathcal{B} \in \mathcal{W}(\widehat{\mathcal{B}})\}$

Table I *Notation for N-body problems, tree data structure and particle FMM.*

In this section, we first introduce the particle N-body problem and briefly describe the basic idea behind the fast multipole method in this context. Then, in §2.1, we discuss the kernel independent variant [Ying et al. 2004] for particle N-body problems

and give the complete FMM algorithm. Finally, in §2.2, we describe the modifications to the particle FMM, for evaluating volume potentials.

In Table I, we summarize notation for N-body problems and particle FMM for easy reference. We will introduce and explain the notation in detail as we discuss these concepts in the following section.

N-body problem. We are given a set of N_s point sources at coordinates $(y_j)_{j=1}^{N_s}$ with charge strengths $(q_j)_{j=1}^{N_s}$ and N_t target points at coordinates $(x_i)_{i=1}^{N_t}$. A source charge $\{y_j, q_j\}$ interacts with a target point x_i and contributes a potential equal to $K(x_i, y_j)q_j$, where K is called the kernel function. Since the potential is additive, the total potential u_i at target points x_i is given by the sum:

$$u_i = \sum_{j=1}^{N_s} K(x_i, y_j)q_j, \quad \forall i = 1, \cdots, N_t$$

The cost of solving this N-body problem is $\mathcal{O}(N_s \times N_t)$. The Fast Multipole Method (FMM) computes an approximate (to any desired accuracy) solution for the potentials u_i in $\mathcal{O}(N_s + N_t)$ time.

To solve the N-body problem with FMM, we first partition the domain Ω using a tree data structure $\mathcal{T}$ (Figure 2). Then, for each target point $x_i \in \mathcal{B}$ (where $\mathcal{B}$ is a leaf node in $\mathcal{T}$), we need to compute interactions from each node in $\mathcal{T}$. In the FMM, these interactions are broadly classified into two parts, near interactions and far interactions:

$$u_i = \sum_{y_j \in \mathcal{N}(\mathcal{B})} K(x_i, y_j)q_j + \sum_{y_j \in \mathcal{F}(\mathcal{B})} K(x_i, y_j)q_j$$

The near interactions (from $\mathcal{N}(\mathcal{B})$, green region in Figure 2), are computed exactly through direct summation by adding contributions from each source point $y_j \in \mathcal{N}(\mathcal{B})$.

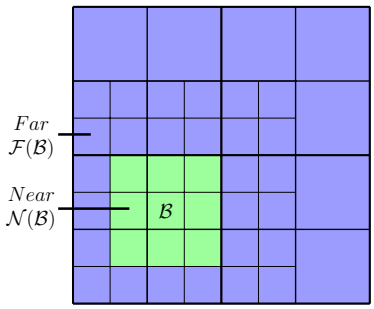


Fig. 2 *Source nodes for near (green) and far (blue) interactions for a target node B.*

The tree nodes further away from $\mathcal{B}$ (in $\mathcal{F}(\mathcal{B})$, blue region in Figure 2) are called *well-separated* from $\mathcal{B}$. The interaction matrix, for interactions from source points in a well-separated tree node to target points in $\mathcal{B}$, is low rank and can be approximated. A typical low rank decomposition has the form $M = U\Sigma V^*$, where Σ is a smaller matrix than M. In the context of FMM:

— V^* corresponds to computing a *multipole expansion* for the source node. The multipole expansion approximates the potential due to the source density of a tree node at target nodes which are well-separated from it. For example, the center-of-mass approximation is a simple first order approximation.
— Σ corresponds to the V-list or multipole-to-local translation. It involves evaluating contribution from the multipole expansion of the source node at the target node $\mathcal{B}$ and building a local expansion. The *local expansion* is a condensed representation approximating the potential at points within $\mathcal{B}$. For example, evaluating the potential at the corners of $\mathcal{B}$ as an approximation of the potential everywhere within $\mathcal{B}$.
— U corresponds to evaluating the local expansion of $\mathcal{B}$ to compute the potential at all target points within $\mathcal{B}$. For example, interpolating the potential from corners of $\mathcal{B}$ to each target point $x_j \in \mathcal{B}$.

The main difference between various variants of FMM is the calculation of multipole, and local expansions and the M2L translation. The definition of what interactions are considered well-separated generally follows from the definition of multipole and local expansions. We will discuss each of these operations in the context of Kernel Independent FMM in §2.1.

Furthermore, far interactions for a target node $\mathcal{B}$ are evaluated hierarchically, so that the sum over source points $y_j \in \mathcal{F}(\mathcal{B})$ is further broken down into interactions from sources in different regions corresponding to increasingly coarser length scales:

$$\sum_{y_j \in \mathcal{F}(\mathcal{B})} K(x_i, y_j) q_j = \sum_{y_j \in \mathcal{V}(\mathcal{B})} K(x_i, y_j) q_j + \sum_{y_j \in \mathcal{V}(\mathcal{P}(\mathcal{B}))} K(x_i, y_j) q_j + \cdots + \sum_{y_j \in \mathcal{V}(\mathcal{B}_0)} K(x_i, y_j) q_j$$

where, $\mathcal{V}(\mathcal{B}) \equiv \{\widehat{\mathcal{B}} \in \mathcal{F}(\mathcal{B}) \setminus \mathcal{F}(\mathcal{P}(\mathcal{B})), \quad \text{Level}(\widehat{\mathcal{B}}) = \text{Level}(\mathcal{B})\}$ (equivalent to definition in Table I) is the region well-separated from $\mathcal{B}$ but excluding the region well-separated from its parent as shown in Figure 3. At each level l in the tree, we compute V-list interactions to the target node $\mathcal{B}_l$ (the ancestor of $\mathcal{B}$ at level l) from source nodes in $\mathcal{V}(\mathcal{B}_l)$ (also at level l). This corresponds to a *hierarchically block-separable* [Gillman 2011] representation of the full N-body problem interaction matrix.

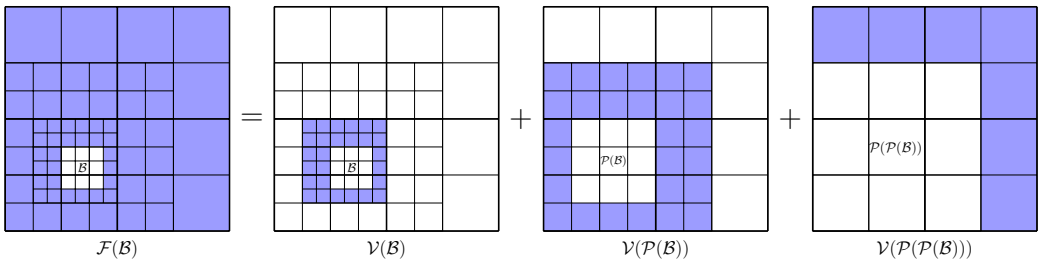


Fig. 3 *Far interactions broken into parts evaluated hierarchically at different levels in the tree.*

2.1. Kernel Independent FMM

Here we summarize the features of KIFMM. Additional details and an error analysis can be found in [Ying et al. 2004]. In KIFMM in 3D, the computational domain Ω is partitioned hierarchically using an octree data structure. Interactions of a leaf octant with adjacent (sharing a face, edge, or vertex) leaf octants and with itself are computed using direct summation. The remaining interactions are evaluated using multipole and local expansions as described below.

2.1.1. Multipole Expansion. The multipole expansion of an octant approximates the potential due to source points within the octant, at target points well-separated from it. In kernel independent FMM, the multipole expansion of an octant $\mathcal{B}$ is represented by a set of source points $(y_k^{u,\mathcal{B}}, q_k^{u,\mathcal{B}})$ on the upward-equivalent surface around $\mathcal{B}$ (Figure 4). The source points $y_k^{u,\mathcal{B}}$ are arranged in a regular $m \times m \times m$ grid with the interior points removed. We also define the upward-check surface as a surface surrounding the upward-equivalent surface, and the points $x^{u,\mathcal{B}}$ on this surface are arranged in an $m \times m \times m$ grid with interior points removed. The multipole order m, determines the accuracy of the multipole expansion.

We compute the upward-equivalent density $q_k^{u,\mathcal{B}}$ in two steps. First, we compute the upward-check potential $u_i^{u,\mathcal{B}}$, at points on the upward-check surface $x^{u,\mathcal{B}}$, due to the

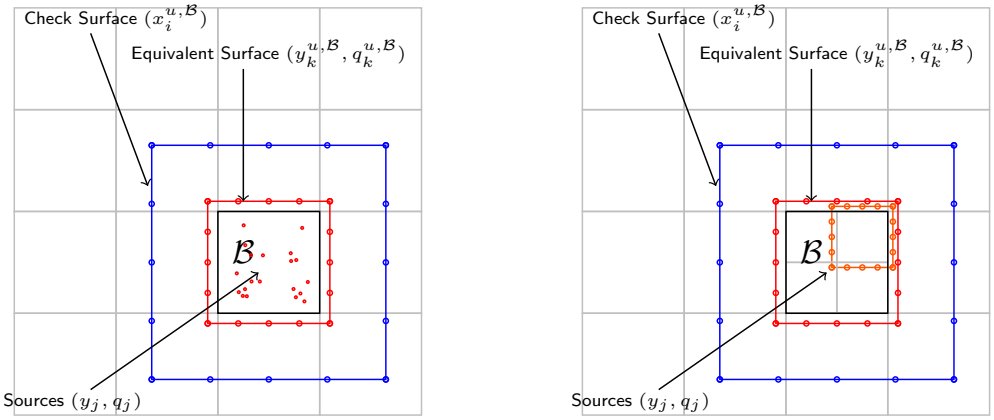


Fig. 4 *Left: Multipole expansion of a leaf octant computed directly from source points. Right: Multipole expansion of a non-leaf octant computed from the upward-equivalent density of its children.*

source points $(y_j,\ q_j)$ within the upward-equivalent surface of $\mathcal{B}$:

$$u_i^{u,\mathcal{B}} = \sum_{y_j \in \mathcal{B}} K(x_i^{u,\mathcal{B}}, y_j) q_j, \qquad \forall i$$

Next, we solve a linear system to obtain the upward-equivalent source density $q_k^{u,\mathcal{B}}$ such that it produces the same potential $u_i^{u,\mathcal{B}}$ at each points on the check surface:

$$u_i^{u,\mathcal{B}} = \sum_{k} K(x_i^{u,\mathcal{B}}, y_k^{u,\mathcal{B}}) q_k^{u,\mathcal{B}}, \qquad \forall i$$

Since this is an ill-conditioned system, we solve this by computing a pseudoinverse. The potential generated by the equivalent source density $q_k^{u,\mathcal{B}}$ is an accurate approximation of the potential generated by source points within the upward-equivalent surface of $\mathcal{B}$, at target points outside the check surface of $\mathcal{B}$. Below, we describe the process for computing the multipole expansion for leaf and non-leaf octants respectively.

Source-to-Multipole translation: For a leaf octant $\mathcal{B}$ (Figure 4: left), we evaluate the potential $u^{u,\mathcal{B}}$, at points $u_i^{u,\mathcal{B}}$ on its upward-check surface, due to its source points $(y_j,\ q_j) \in \mathcal{B}$. Then, we solve a linear system, to determine the source density $q^{u,\mathcal{B}}$ at points $y^{u,\mathcal{B}}$ on the upward-equivalent surface of $\mathcal{B}$, using a precomputed pseudoinverse.

Multipole-to-Multipole translation: For a non-leaf octant $\mathcal{B}$ (Figure 4: right), we evaluate the upward-check potential $u_i^{u,\mathcal{B}}$ due to the source points $(y^{u,\mathcal{B}_c},\ q^{u,\mathcal{B}_c})$ on the upward-equivalent surface of its children $\mathcal{B}_c \in \mathcal{C}(\mathcal{B})$, and then compute the equivalent source density $q^{u,\mathcal{B}}$ as explained before for leaf octants.

2.1.2. Local Expansion. The local expansion of an octant approximates the potential at target points within the octant due to source points well-separated from it. In kernel independent FMM, the local expansion of an octant $\mathcal{B}$ is represented by a set of source points $(y_k^{d,\mathcal{B}}, q_k^{d,\mathcal{B}})$ on the downward-equivalent surface around $\mathcal{B}$ (Figure 5). The source points $y_k^{d,\mathcal{B}}$ are arranged in a regular $m \times m \times m$ grid with the interior points removed. We also define the downward-check surface as a surface surrounding $\mathcal{B}$ but inside the downward-equivalent surface, and the points $x^{d,\mathcal{B}}$ on this surface are arranged in an $m \times m \times m$ grid with interior points removed. The multipole order m, determines the accuracy of the local expansion.

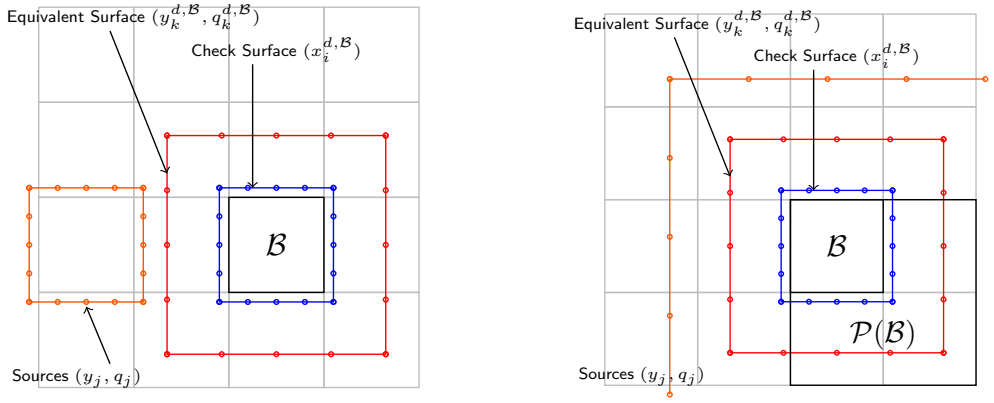


Fig. 5 *Left: Local expansion from upward-equivalent source distribution of a well-separated octant. Right: Local expansion from downward-equivalent source distribution of the parent octant.*

We compute the downward-equivalent density $q_k^{d,\mathcal{B}}$ in two steps. First, we compute the downward-check potential $u_i^{d,\mathcal{B}}$, at points on the downward-check surface $x^{d,\mathcal{B}}$, due to the source points $(y_j,\ q_j)$ outside the downward-equivalent surface of $\mathcal{B}$:

$$u_i^{d,\mathcal{B}} = \sum_{y_j \in \mathcal{B}} K(x_i^{d,\mathcal{B}}, y_j) q_j, \qquad \forall i$$

Next, we solve a linear system to obtain the downward-equivalent source density $q_k^{d,\mathcal{B}}$ such that it produces the same potential $u_i^{d,\mathcal{B}}$ at each points on the check surface:

$$u_i^{d,\mathcal{B}} = \sum_{k} K(x_i^{d,\mathcal{B}}, y_k^{d,\mathcal{B}}) q_k^{d,\mathcal{B}}, \qquad \forall i$$

Since this is an ill-conditioned system, we solve this by computing a pseudoinverse. The potential generated by the equivalent source density $q_k^{d,\mathcal{B}}$ is an accurate approximation of the potential generated by source points outside the upward-equivalent surface of $\mathcal{B}$, at target points inside the check surface of $\mathcal{B}$. The local expansion of an octant $\mathcal{B}$ is constructed from two contributions: the multipole expansion for well-separated octants $\mathcal{B}_V \in \mathcal{V}(\mathcal{B})$ and from the local expansion of the parent $\mathcal{P}(\mathcal{B})$. Constructing the local expansion from these two components is discussed below:

Multipole-to-Local or V-list interactions: include contributions to the local expansion of an octant $\mathcal{B}$ from source octants $\mathcal{B}_V \in \mathcal{V}(\mathcal{B})$ which are at the same level (depth in the octree) and well-separated from it (i.e. lying entirely outside the downward-equivalent surface of the octant) but not well-separated from the parent octant (Figure 5: left). We evaluate the potential $u^{d,\mathcal{B}}$, at points $x^{d,\mathcal{B}}$ on the downward-check surface of $\mathcal{B}$, due to the upward-equivalent density $(y^{u,\mathcal{B}_V}, q^{u,\mathcal{B}_V})$ (the multipole expansion) of each source octant $\mathcal{B}_V \in \mathcal{V}(\mathcal{B})$. Then, we solve a linear system, to determine the source density $q^{d,\mathcal{B}}$ at points $y^{d,\mathcal{B}}$ on the downward-equivalent surface of $\mathcal{B}$, using a precomputed pseudoinverse.

Local-to-Local translation: includes contributions from all octants outside the downward-check surface of the parent octant (Figure 5: right). These interactions are included in the local expansion of the parent octant $\mathcal{P}(\mathcal{B})$. To add this component of the potential to the local expansion of $\mathcal{B}$, we first evaluate the potential $u^{d,\mathcal{B}}$, at points $x^{d,\mathcal{B}}$ on the downward-check surface, due to the downward-equivalent density $(y^{d,\mathcal{P}(\mathcal{B})},\ q^{d,\mathcal{P}(\mathcal{B})})$ (the local expansion) of the parent $\mathcal{P}(\mathcal{B})$. Then, we compute the equiv-

alent density from the check potential using a precomputed pseudoinverse and add it to $q^{d,\mathcal{B}}$.

2.1.3. FFT Acceleration of V-list interaction. Each multipole-to-local (V-list) translation involves computing the downward-check potential for an octant from the upward-equivalent density of octants in its interaction list (Figure 5: left). With $m^3-(m-2)^3 \approx 6(m-1)^2$ points[1] each on the equivalent and check surfaces, each translation has a computational cost of $2\times 36(m-1)^4$ (counting multiplications and additions). Since the points are equispaced, this computation can be written as a convolution of the upward-equivalent density of the source octant with the kernel function in three dimensions, each represented by a three-dimensional $m \times m \times m$ array. We use Fourier transform (implemented using FFT) to convert this convolution to a complex Hadamard product in Fourier space. Since this corresponds to a circular convolution, we need to zero-pad the original arrays to size $2m \times 2m \times 2m$. The FFT reduces the complexity of the V-list interaction from $\mathcal{O}(m^4)$ to $\mathcal{O}(m^3)$.

The FFT of the kernel function evaluated at grid points is precomputed and the FFT of the upward-equivalent density is computed once for each octant. We sum the Hadamard product results from all V-list interactions for each target octant: $v_t = v_t + M_k \circ v_s$, where $\circ$ represents the operator for Hadamard product. This computation involves $4m^3$ complex multiplications and additions (each) or $32m^3$ floating point operations. Finally, we compute the inverse FFT (IFFT) of the result for each target octant to obtain the downward-check surface potential and from that we obtain the downward-equivalent density. For each octant, there are several V-list interactions (189 for uniform octree), and therefore the cost of computing Hadamard products is much larger than the cost of computing FFTs and IFFTs, which are computed once per octant. Neglecting the cost of computing FFTs and IFFTs, this brings down the cost of each V-list interaction from $72(m-1)^4$ floating point operations to $32m^3$ floating point operations.

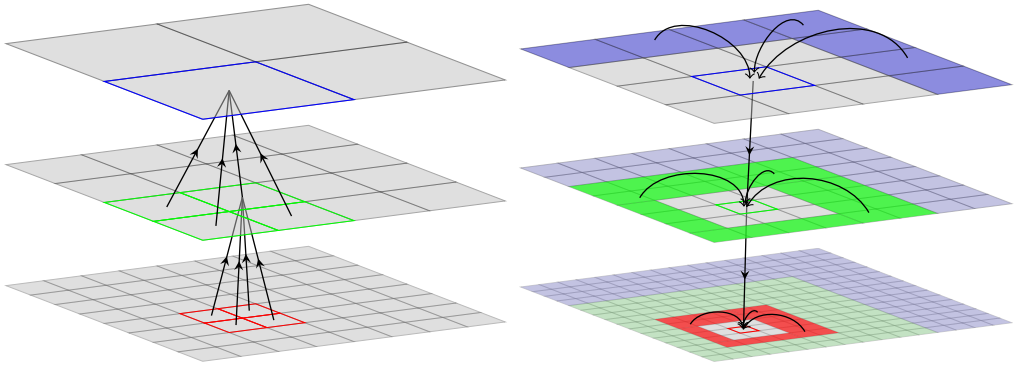


Fig. 6 *Upward Pass: constructing multipole expansions and Downward Pass: constructing local expansions, evaluating near interactions.*

2.1.4. FMM Algorithm. Starting from the source and target points, we construct an octree data structure by refining octants with more points than a pre-determined upper bound. For simplicity we first describe the algorithm assuming a uniform octree. The overall algorithm has two main stages:

[1] m^3 regular grid without the $(m-2)^3$ interior points gives $m^3-(m-2)^3$ surface points

Upward Pass: We visit each octant in bottom-up order (level-by-level from finest to coarsest level) and construct the multipole expansion of each octant as a representation of its far field potential (Figure 6: left). For the leaf octants, the multipole expansion is computed directly from the underlying source distribution (source-to-multipole or S2M translation), while for non-leaf octants it is computed from the multipole expansion of each of its children (multipole-to-multipole or M2M translation).

Downward Pass: We visit each octant in top-down order (level-by-level from coarsest for finest level) constructing their local expansions (Figure 6: right). For each octant, we add contributions from the local expansion of the parent (local-to-local or L2L translation) and the multipole expansion of well-separated octants (multipole-to-local or M2L or V-list interactions) which were not included in the parent's local expansion. At a leaf octant, we compute the final potential at target points within the octant by first computing direct interactions from sources in adjacent leaf octants (source-to-target or S2T or U-list interactions) and then adding contribution from the local expansion by evaluating the potential due to the downward-equivalent density of the octant (local-to-target or L2T translation).

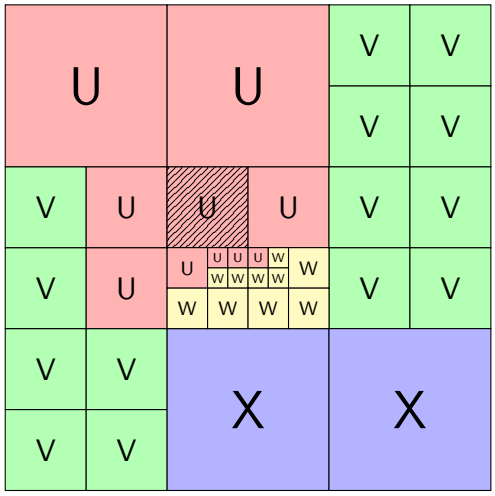


Fig. 7 *Source nodes for each interaction type for a target node (shaded).*

For non-uniform (or adaptive) octrees, the interactions are more complex. Figure 7 shows the various (U,V,W and X) types of interactions that we need to account for during the downward pass for a given target octant (shaded). For a target octant $\mathcal{B}$, the list of contributing source octants for U,V,W,X-list interactions are represented as $\mathcal{U}(\mathcal{B})$, $\mathcal{V}(\mathcal{B})$, $\mathcal{W}(\mathcal{B})$, $\mathcal{X}(\mathcal{B})$ respectively.

— **U-list** represents the near interactions of a target leaf octant $\mathcal{B}$ with itself and with adjacent leaf octants (source-to-target or S2T translations). As discussed before, these interactions are computed through direct summation of the potential contributed by each source point $(y_j, q_j) \in \mathcal{B}_s \in \mathcal{U}(\mathcal{B})$ to each target point $x_i \in \mathcal{B}$.
— **V-list** interactions are the multipole-to-local translations discussed before (multipole-to-local or M2L translations). For a target octant $\mathcal{B}$ we compute the downward-check potential $u^{d,\mathcal{B}}$ from the upward-equivalent density $q^{u,\mathcal{B}_s}$ of a source octant $\mathcal{B}_s \in \mathcal{V}(\mathcal{B})$. Then, from $u^{d,\mathcal{B}}$ we compute the downward-equivalent density $q^{d,\mathcal{B}}$ for $\mathcal{B}$.
— **W-list** includes source octants (marked W in Fig 7) which are at a finer level than the target leaf octant, such that the multipole expansion of the source octant is applicable (since the target octant is outside its upward-check surface), however a multipole-to-local translation is not applicable since the source octant is not outside the downward-equivalent surface of the target octant. For such interactions, we use the multipole expansion of the source octant to evaluate the potential directly at the target points (multipole-to-target translations).
— **X-list** includes source octants (marked X in Fig 7) which are leaf octants and at a coarser level than the target octant, such that the multipole expansion of the source octant is not applicable (since the target octant is not outside its upward-check sur-

face), however we can construct a local expansion at the target octant since the source octant is outside the downward-equivalent surface of the target octant. For such interactions, we evaluate the downward-check potential at the target octant directly from the points in the source octant, and then construct the downward-equivalent density (source-to-local translations).

The overall algorithm is summarized in Algorithm 2.1. The translation operators M_{S2M} and M_{M2M} compute the multipole expansion ($q^{u,\mathcal{B}}$) of target octant $\mathcal{B}$. Operators M_V, M_X and M_{L2L} together compute the local expansion ($q^{d,\mathcal{B}}$) of the target octant $\mathcal{B}$. Finally, Operators M_{L2T}, M_U and M_W compute the final potential within the target octant $\mathcal{B}$. As we have discussed before, all of these operators are linear oprators. The matrix for the operators M_{M2M}, M_V and M_{L2L} can be precomputed. The remaining operators depend on the position of either the source or the target points and can not be precompute for particle FMM. For volume FMM, as we will see in the next section, these operators can also be precomputed since for volume FMM we can think of representing the continuous source density and target potential on a lattice (the Chebyshev node points) within each octant.

Algorithm 2.1 FMM

Input: tree $\mathcal{T}$, target points x_i, source points y_j and density q_j
Output: target potentials u_i

Upward Pass: compute multipole expansion $q^{u,\mathcal{B}} \quad \forall \mathcal{B} \in \mathcal{T}$

1: $q^{u,\mathcal{B}} \longleftarrow M_{S2M}\left((y_j, q_j) \in \mathcal{B} \right)$ ▷ $\forall \mathcal{B} \in \mathcal{L}(\mathcal{T})$
2: $q^{u,\mathcal{B}} \longleftarrow M_{M2M}\left(q^{u,\mathcal{B}_k} : \mathcal{B}_k \in \mathcal{C}(\mathcal{B}) \right)$ ▷ in postorder $\forall \mathcal{B} \in \mathcal{T} \setminus \mathcal{L}(\mathcal{T})$

Downward Pass: compute local expansions $q^{d,\mathcal{B}} \quad \forall \mathcal{B} \in \mathcal{T}$

3: $q^{d,\mathcal{B}} \longleftarrow M_V\left(q^{u,\mathcal{B}_k} : \mathcal{B}_k \in \mathcal{V}(\mathcal{B}) \right)$ ▷ $\forall \mathcal{B} \in \mathcal{T}$
4: $q^{d,\mathcal{B}} \mathrel{+}= M_X\left((y_j, q_j) \in \mathcal{B}_k \in \mathcal{X}(\mathcal{B}) \right)$ ▷ $\forall \mathcal{B} \in \mathcal{T}$
5: $q^{d,\mathcal{B}} \mathrel{+}= M_{L2L}\left(q^{d,\mathcal{P}(\mathcal{B})} \right)$ ▷ in preorder $\forall \mathcal{B} \in \mathcal{T}$

compute target potential $u^{\mathcal{B}} \quad \forall \mathcal{B} \in \mathcal{L}(\mathcal{T})$

6: $u^{\mathcal{B}} \longleftarrow M_{L2T}\left(q^{d,\mathcal{B}} \right)$ ▷ $\forall \mathcal{B} \in \mathcal{L}(\mathcal{T})$
7: $u^{\mathcal{B}} \mathrel{+}= M_U\left((y_j, q_j) \in \mathcal{B}_k \in \mathcal{U}(\mathcal{B}) \right)$ ▷ $\forall \mathcal{B} \in \mathcal{L}(\mathcal{T})$
8: $u^{\mathcal{B}} \mathrel{+}= M_W\left(q^{u,\mathcal{B}_k} : \mathcal{B}_k \in \mathcal{W}(\mathcal{B}) \right)$ ▷ $\forall \mathcal{B} \in \mathcal{L}(\mathcal{T})$

9: **return** u

Volume FMM	
f	source density function
ϵ_{tree}	tolerance for adaptive refinement
q	maximum degree of Chebyshev polynomials
$T_i(x)$	Chebyshev polynomial of degree i in x
Symmetries	
$T_i \quad i = 1, \cdots, 5$	coordinate transforms: reflection about X,Y,Z axes; swap X,Y and X,Z axes
$P_i \quad i = 1, \cdots, 5$	equivalent surface permutations
$Q_i \quad i = 1, \cdots, 5$	Chebyshev coefficient permutations

Table II *Notation for volume FMM.*

2.2. Volume FMM

In this section we describe how the kernel independent FMM can be adapted to evaluate potential due to continuous source distribution f defined on a cubic domain Ω. Instead of summation, as for particle FMM, we now need to evaluate integrals over the computational domain Ω:

$$\begin{aligned} u(x) &= \int_{\Omega} K(x-y)f(y) \\ &= \int_{\mathcal{N}(x)} K(x-y)f(y) + \int_{\mathcal{F}(x)} K(x-y)f(y) \end{aligned}$$

As before, we use an octree based partitioning of the domain and split the computation into near and far field interactions. The far field interactions are approximated using multipole and local expansions and near interactions are evaluated exactly. Since the kernel function has a singularity at the origin, this leads to singular and near-singular integrals which are costly to evaluate.

In §2.2.1, we discuss the octree data structure used to represent the source density function f. Then, in §2.2.2, we describe algorithms for efficient computation of singular and near-singular integrals on this data structure. In §2.2.3 we discuss the 2:1 balance constraint on our octree, which enables us to precompute and store quadrature rules without requiring extremely large amounts of memory. Finally, in §2.2.4, we discuss the overall algorithm for evaluating volume potentials and discuss the main sources of error and give a complexity analysis for the volume FMM. In §2.2.5, we discuss some more techniques for reducing memory usage.

2.2.1. Octree Construction. We approximate the source distribution f, defined over the cubic domain $(x_1, x_2, x_3) \in [-1, 1]^3$, by Chebyshev polynomials of degree q as:

$$\hat{f}(x_1, x_2, x_3) = \sum_{i,j,k \geq 0}^{i+j+k \leq q} \alpha_{i,j,k} T_i(x_1) T_j(x_2) T_k(x_3) \tag{6}$$

where, $T_i(x)$ is the Chebyshev polynomial of degree i in x. Notice that this is not a complete tensor order approximation since we truncate the expansion, so that $i+j+k \leq q$. We take the absolute sum of the highest order coefficients in the Chebyshev approximation $\hat{f}$ as an estimate of the truncation error. If the truncation error ϵ_{tree} is larger than a prescribed threshold, then we subdivide the domain and try to approximate the source density f in each of the smaller cubic domains. We refine adaptively to obtain an octree based domain decomposition and piecewise polynomial approximation of the density function up to the desired accuracy (Figure 8).

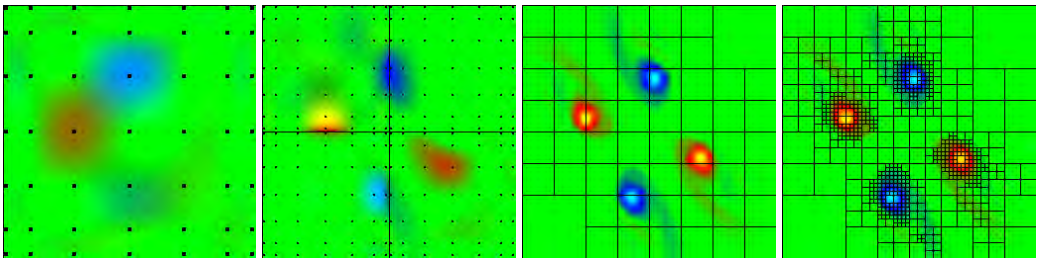


Fig. 8 *Adaptive refinement of Chebyshev quadtree starting from the root node showing the Chebyshev node points where the input function f is sampled and refining adaptively up to six levels.*

For highly non-uniform octrees, the initial samples of the density function using the above method may entirely miss small features. Therefore, we provide a way for the user to guide the initial tree construction and then adaptive refinement takes over. We do this by providing an initial particle distribution and constructing the initial octree by refining until each octant has one particle.

We also represent the volume potential solution using Chebyshev polynomials on this octree. We evaluate the volume potential at the Chebyshev node points within each leaf octant and from that construct a Chebyshev interpolation of the volume potential. We assume that the final potential is as smooth as the source distribution and can be represented accurately using the same octree discretization. This is however not strictly true, for example in the case of Helmholtz equation where it may be necessary to use uniform refinement.

2.2.2. Singular Quadratures. Given a piecewise polynomial approximation of the density function as above, we need to be able to evaluate the potential due to a single leaf octant $\mathcal{B}$ in the octree. If the target point is sufficiently far away from the octant, then the kernel function is smooth and standard Gussian quadratures can be applied. However, to evaluate the potential at a point inside or close to the boundary of the octant requires evaluating singular and near-singular integrals respectively. We can do this efficiently using precomputed quadratures. For a leaf octant $\mathcal{B}$ with Chebyshev approximation of the density function $\hat{f}(y) = \sum_{i,j,k} \alpha^{\mathcal{B}}_{i,j,k} T_{i,j,k}(y)$, the potential at a point x is given by:

$$
\begin{aligned}
u(x) &= \int_{y \in \mathcal{B}} K(x-y)\hat{f}(y) \\
&= \int_{y \in \mathcal{B}} K(x-y) \left[\sum_{i,j,k} \alpha^{\mathcal{B}}_{i,j,k} T_{i,j,k}(y) \right] \\
&= \sum_{i,j,k} \alpha^{\mathcal{B}}_{i,j,k} \left[\int_{y \in \mathcal{B}} K(x-y) T_{i,j,k}(y) \right]
\end{aligned}
$$

The integral term in the above expression can be precomputed using the method explained in Appendix A. Now, the potential due to any leaf octant $\mathcal{B}$ at a point x (coordinates relative to $\mathcal{B}$) can be evaluated using the following summation:

$$
u(x) = \sum_{i,j,k} \alpha^{\mathcal{B}}_{i,j,k} I_{i,j,k}
$$

We precompute these quadratures (once for each level in the octree) so that S2M, U-list and X-list interactions can then be represented as matrix-vector products. For S2M interaction, we precompute quadratures for evaluating potential at each point on the upward-check surface. For a leaf octant with source density represented by the Chebyshev coefficients $\alpha = [\alpha_1, \cdots, \alpha_n]^T$, the upward-check potential can be evaluated as: $u_{check} = M_{s2ch} \times \alpha$. We then evaluate the upward-equivalent density from the check potential as before. Similarly, for U-list interactions, we precompute quadratures for evaluating potential at Chebyshev node points in the target octant, for each possible direction of the target octant relative to a source octant, and then compute Chebyshev interpolation of the target potential. For X-list interactions, we precompute quadratures for evaluating potential at points on the downward-check surface of

target octants in each possible direction relative to the source octant and then compute the downward-equivalent density using precomputed pseudoinverse.

2.2.3. 2:1 Balance Constraint. For U-list interactions, for a target octant at a given level in the octree, the source octants can be at a coarser level, at the same level or at a finer level in the octree. Precomputing quadratures for every possible combination is not feasible. To limit the number of possible interaction directions, we constrain adjacent leaf octants to be within one level of each other and this is known as the 2:1 balance constraint. To enforce this constraint, we need to refine leaf octants which are adjacent to leaf octants at a much finer level and thus violating the 2:1 balance constraint. This refinement is called 2:1 balance refinement.

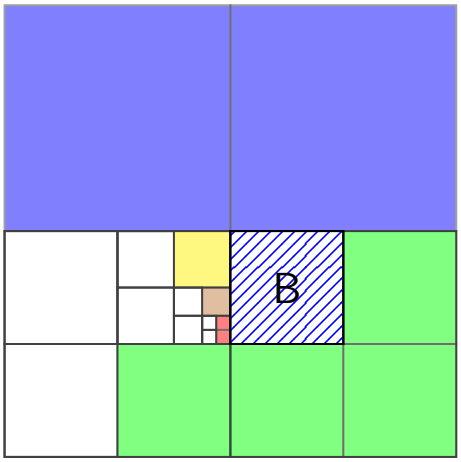


Fig. 9 *An unbalanced quadtree.*

The steps in 2:1 balance refinement are described in Algorithm 2.2. We start from the finest level of the octree and in each step move to the next coarser level. At each step, we collect the set S of parents of colleagues of non-leaf octants at that level. These are the non-leaf octants which must exist at the next coarser level for the 2:1 balance constraint to hold, and are therefore added to the set of non-leaf octants N in the next iteration. At each level, we add the set of children of the non-leaf octants N, to the final balanced octree $\widehat{\mathcal{T}}$. The data structure "std::set" is used to implement Algorithm 2.2. It allows addition, deletion and searching of octants in $O(logN)$ steps. The complexity of the overall algorithm is $T(N_{oct}) = O(N_{oct}logN_{oct})$ where, N_{oct} is the number of octants.

Algorithm 2.2 SEQUENTIALBALANCE

Input: $\mathcal{T}$ unbalanced octree, L_{max} maximum tree depth.
Output: $\widehat{\mathcal{T}}$ balanced octree.

1: $\widehat{\mathcal{T}} \leftarrow \emptyset$, $S \leftarrow \emptyset$
2: **for** $i \leftarrow L_{max}$ to 1 **do**
3: $N \leftarrow S + \{\mathcal{B} : \ \mathcal{B} \in \mathcal{T} \setminus \mathcal{L}(\mathcal{T}), \ \ \text{Level}(\mathcal{B}) = i\}$
4: $S \leftarrow \text{Parent}(\text{Colleagues}(N))$
5: $\widehat{\mathcal{T}} \leftarrow \widehat{\mathcal{T}} + \text{Children}(N)$
6: **end for**
7: **return** $\widehat{\mathcal{T}}$

2.2.4. Summary of Volume FMM. The volume FMM differs from the classical particle FMM in that the source density is represented by a polynomial approximation. Therefore, all translations involving the source term need to be modified. This includes the S2M, X-list and U-list interactions. The necessary modifications for these translations were discussed in §2.2.2. Furthermore, we want to represent the final result by a Chebyshev interpolation, therefore, we choose the target points to be the Chebyshev

node points within each octant and from that we compute the polynomial approximation using another linear transformation. Since, we know the positions of the target points, we can precompute the translation operators involving the target potential (L2T, W-list and U-list interactions). The remaining operators (M2M, L2L and V-list interactions) are exactly the same as for particle FMM.

Some translation operators involve a sequence (composition) of linear transformations. For such operators, we precompute the composition and store only a single matrix. For example, the U-list interactions involve computing the potential at Chebyshev node points and then computing the Chebyshev approximation at the target node, therefore we precompute the composition of these two linear transformations. Similarly, for M2M translations, we precompute the composition of the linear transformation to compute the check surface potential and the linear transformation to compute the equivalent density from the check surface potential.

Having precomputed all the translation operators: S2M, M2M, L2L, L2T, U,V,W,X-list for each direction of the source octant relative to the target octant and for each level in the octree, the volume FMM can now be summarized as follows:

— **Tree Construction**: Construct a piecewise Chebyshev approximation of the source density using octree based domain decomposition. Perform 2:1 balance refinement using Algorithm 2.2.
— **Upward Pass:** For all leaf octants apply S2M translation to construct the multipole expansion (upward-equivalent density). For all non-leaf octants apply M2M translations in bottom-up order, to construct multipole expansion from the multipole expansion of children.
— **Downward Pass:** For all octants, apply M2L (V-list) and S2L (X-list) translations to construct the local expansion (downward-equivalent density) of each octant. In top-down order apply the L2L translation to all leaf octants and add the results to their local expansions. For all leaf octants, apply L2T, M2T (W-list) and S2T (U-list) translations to construct the final target potential as piecewise Chebyshev interpolation.

Boundary Conditions: The FMM implementation discussed so far computes convolution of source density function with the free-space Green's function, and hence, gives a solution for free-space bound boundary conditions i.e. the potential decays to zero at infinity. We can also enforce periodic boundary conditions on the cubic computational domain. The basic idea is that if we could create an infinite tiling of the source density function, then the solution given by convolution with the free-space Green's function will be periodic. Since it is not feasible to create an infinite tiling, we just map neighbours of boundary octants to octants on the opposite end of the domain. This creates the effect of repeating the source density once on each side of the computational domain.

The remaining contribution must come from the local expansion of the root octant. We can approximate this component by creating a sequence of dummy octants with negative depths, and during the upward pass we continue past the root octant and through these negative depths. Then during the downward pass, we start from the octant with the least depth, and continue down, while adding M2L contributions from itself and L2L contributions from the parent, until we have constructed the local expansion of the root octant at level zero. This sequence of application of M2M, M2L and L2L translations is precomputed into a single translation operator to compute the local expansion of the root octant directly from its multipole expansion without constructing dummy octants.

Error Analysis: Here, we summarize the three main sources of error.

— *Polynomial Approximation Error:* This is the error in the input (source density) and output (potential) approximation using the piecewise polynomial approximation on an octree. This depends on the refinement of the octree (h-adaptivity) and the degree of the Chebyshev polynomials q used (fixed at the start of execution). We control the input approximation error by adaptively refining the octree until we attain the desired error tolerance ϵ_{tree}. For the output approximation, we assume (at least for non-oscillatory kernels) that the solution will be sufficiently smooth and therefore the error will be bounded by the desired tolerance ϵ_{tree}.
— *Numerical Integration Error:* Near interactions are computed by evaluating singular integrals numerically using special quadrature rules described in Appendix A. This integration is performed during the precomputation phase and can be computed to machine precision, and therefore we do not need to consider this component of the error in our analysis.
— *Multipole and Local Expansions:* The multipole and local expansion approximate far field interactions. In the kernel independent FMM, the accuracy of these approximations depends on the multipole order m, which determines the number of points on the upward and downward-equivalent surfaces.

Complexity Analysis: The cost of FMM evaluation is given by the number of interactions between the octants weighted by the cost of each translation. The cost of each translation depends on the parameters: m the multipole order and q the degree of Chebyshev polynomials. These parameters determine the number of Chebyshev coefficients per leaf octants ($(q+1)(q+2)(q+3)/6 \approx (q+2)^3/6$) and the number of points on the equivalent surface ($m^3-(m-2)^3 \approx 6(m-1)^2$). Let N_{oct} be the total number of octants and N_{leaf} be the number of leaf octants in the octree. Also, let N_U, N_V, $N_W, N_X(=N_W)$ denote the number of interactions of each type U,V,W and X-list respectively. in turn determine the cost of the translations. The overall cost for each interaction type is summarized in Table III.

Interaction Type	Cost (floating point operations)
S2M, L2T	$N_{leaf}(q+2)^3/6 \times \ 6(m-1)^2 \times 2$
M2M, L2L	$N_{oct} \ 6(m-1)^2 \times \ 6(m-1)^2 \times 2$
W-list, X-list	$N_W \ 6(m-1)^2 \times (q+2)^3/6 \times 2$
U-list	$N_U \ (q+2)^3/6 \times (q+2)^3/6 \times 2$
V-list	$N_V \ 32m^3 \ + \ 2 \times N_{oct}(2m)^3 \log(2m)$

Table III *Total computational cost for each interaction type.*

Note: a matrix-vector product with a matrix of dimensions $n_1 \times n_2$ is $2n_1n_2$, counting multiplications and additions. This accounts for the $\times 2$ factor for S2M, L2T, M2M, L2L, U,W,X-list costs in Table III.

For a uniform octree with N_{leaf} leaf octants, $N_{oct} \approx (8/7)N_{leaf}$, $N_U = 27N_{leaf}$, $N_V = 189N_{oct}$, and $N = N_{leaf}(q+2)^3/6$ total unknowns. Due to the large constant factors, the cost of U-list and V-list interactions dominate and the overall cost, as a function of the total number of unknowns N, the Chebyshev degree q and the multipole order m, is given by:

$$T_{FMM} = \mathcal{O}\left(9(q+2)^3 N\right) + \mathcal{O}\left(41472 \frac{m^3}{(q+2)^3} N\right) \tag{7}$$

For a given source density distribution we want to evaluate the volume potential accurate to $-log_{10}(\epsilon_{fmm})$ digits, in the least amount of time using available computational resources. We have three parameters that we can vary, to minimize the computational cost while achieving the desired accuracy:

— Refinement Tolerance (ϵ_{tree}): determines the accuracy of the polynomial approximation of the source density function by controlling the tree refinement.
— Chebyshev Degree (q): is the degree of the polynomials used to approximate the source density and the target potential within each leaf octant.
— Multipole Order (m): determines the number of points on the equivalent and check surfaces. This in turn determines the accuracy of the multipole and local expansions.

Since ϵ_{tree} and m also affect the accuracy, they are determined directly from ϵ_{fmm}. We choose q to minimize time to solution T_{FMM}. Assuming that the constants in the order notation for near and far interaction costs are the same in equation 7, for a uniform octree we have $q \approx 4.1\sqrt{m} - 2$. When using co-processor for near interactions, the constants in the order notation are not same and q must be larger.

2.2.5. Reducing Memory Requirement. Even after enforcing the 2:1 balance constraint, the memory required to store precomputed translation operators can still be much more than what is available on most HPC systems. For example, for Chebyshev approximation with polynomial degree 13, U-list interaction matrices take 333MB (139 matrices of dimension 560×560) of memory per level of the octree. For tensor kernels like Helmholtz (2×2 kernel) and 20 octree levels, this amounts to about 26GB of memory. Precomputing all interaction types (U,V,W,X-list) would take hundreds of GB of memory. Storing on disk and reading when needed would be too costly. In the following section we describe how we reduce the memory footprint of precomputed interaction operators.

Scale-invariant Kernels: Several kernel functions are scale-invariant i.e. when the distance between a source and a target point is scaled by a constant factor α, the interaction between them is scaled by a constant factor α^γ:

$$K(\alpha(x-y)) = \alpha^\gamma K(x-y)$$

For example, for the Laplace kernel,

$$K(\alpha(x-y)) = \frac{1}{2\pi}\frac{1}{|\alpha(x-y)|} = \alpha^{-1}K(x-y)$$

For such kernels, we precompute interaction operators for only one level of the octree and with appropriate scaling, the same operators can be applied to each level of the octree. For d-dimensional space and a kernel with scaling exponent γ, the interaction operators computed for level zero in the octree can be applied to interaction at level l by scaling appropriately for the change in volume (by a factor of $2^{-d\times l}$ for S2M, U,X-lists interactions) and the change in distance (by a factor of $2^{-\gamma\times l}$ for U,W-lists and L2T operators). The scaling factors for different interaction types are summarized in Table IV.

Interaction Type	Scaling Factor
M2M, L2L, V-list	1.0 (no scaling)
S2M, X-list	$2^{-d\times l}$
L2T, W-list	$2^{-\gamma\times l}$
U-list	$2^{-d\times l} \times 2^{-\gamma\times l}$

Table IV *Scaling factors for different interaction types for scale-invariant kernels.*

Symmetries: For each interaction type, we have interaction operators for every possible direction. Some of the interactions directions are equivalent if we take a reflection along a particular coordinate axis or swap two coordinate axes. Therefore, we can

group interaction directions into interaction classes and only store one interaction matrix for each interaction class. In Table V, we summarize the number of interaction directions for each interaction type. We also report the number of interaction classes required to be stored and the reduction in storage achieved by storing only one matrix per interaction class.

Interaction Type	Directions	# of Classes	Storage Reduction
S2M, L2T	1	1	1×
M2M, L2L	8	1	8×
W-list, X-list	152	6	25×
U-list	139	10	14×

Table V *The number of interaction directions for each interaction type and the number of interaction classes and the storage reduction factor due to symmetries.*

We represent interaction directions by an integer triplet (i, j, k), representing the relative coordinates of the source octant relative to the target octant. The representative class for an interaction direction is determined by taking the absolute value of each integer in the triplet and then sorting them in increasing order. The change of sign of an integer represents a reflection along the corresponding coordinate and the sorting can be accomplished by a sequence of swap operations: swapping the first and second integers (swapping X and Y axes) and swapping the first and third integers (swapping X and Z axes). We represent these five transformations (reflection along X, Y or Z axis and swapping {X,Y} or {X,Z} axes) by $T_1, T_2, \cdots, T_5$.

The domain and range of the precomputed translation operators is either the equivalent density data or the Chebyshev coefficient data. For equivalent density data, reflection or swapping axes corresponds to a rearrangement of the vector elements corresponding to the order of points on the equivalent surface after reflection or swapping axes. For Chebyshev coefficient data a reflection along an axis corresponds to a change of sign of all the odd order Chebyshev coefficients and swapping axes corresponds to a rearrangement of the Chebyshev coefficients.

For tensor kernels, these transformations can be more complex. For example, when the domain or range of a kernel function is a spatial vector field (Stokes velocity, Laplace gradient, Biot-Savart kernels), reflection along an axis will also cause the vector field component along that axis to reverse direction and swapping axes will also require a rearrangement of the components of the vector field. There may be other issues to consider, such as for Biot-Savart kernel, reflection along an axis or swapping axes leads to reversal of the field direction (due to the right-handed orientation of axes).

Nevertheless, these five transformations ($T_1, T_2, \cdots, T_5$) can be performed through permutation and scaling operations. For equivalent density data, we have: $P_1, P_2, \cdots, P_5$ operators and for Chebyshev data, we have: $Q_1, Q_2, \cdots, Q_5$ operators. These permutation and scaling operators are sparse matrices, represented by a permutation vector and a scaling vector, each with length equal to the length of the data that they operate on.

The transformation from a direction (i, j, k) to the representative direction in its class (i_0, j_0, k_0) is given by a sequence of transformations: $T_{\alpha_1}, T_{\alpha_2}, \cdots, T_{\alpha_n}$. An interaction in direction (i, j, k) between a source vector v_s and a target vector v_t through an interaction matrix M is given by:

$$v_t = v_t + M v_s$$

We can now represent this interaction using the interaction matrix M_0 for the direction (i_0, j_0, k_0) by suitable transformations on the source and target vectors. For example

for W-list interactions (transformation from equivalent data to Chebyshev data), this would correspond to:

$$v_t = v_t + Q^T_{\alpha_1} \cdots Q^T_{\alpha_n} \times M_0 \times P_{\alpha_n} \cdots P_{\alpha_1} v_s$$

The composition of permutation and scaling operators can be precomputed and the interaction operator M can then be computed from M_0 as:

$$M = Q^{i,j,k}_{i_0,j_0,k_0} \times M_0 \times P^{i_0,j_0,k_0}_{i,j,k}$$

Note: The transpose of a permutation and scaling operator is also a permutation and scaling operator and is also its inverse (since scaling here is either 1 or -1). Also, the composition of permutation and scaling operators is also a permutation and scaling operator.

3. PARALLEL ALGORITHM

We optimize our method for current state-of-the-art HPC systems. These systems are large clusters (many thousand node) with low latency, high bandwidth interconnect. Each node is a multi-core, shared memory system with or without additional accelerator devices. We support both Intel Phi and NVIDIA GPU. We start by discussing Intra-node parallelism and then discuss the distributed memory implementation.

3.1. Intra-node Parallelism

We maximize intra-node performance of our algorithm by effectively utilizing parallelism at each level of the architecture. We first discuss parallelism in the context of co-processors, by concurrently solving different parts of the problem on the co-processor and the CPU. We then discuss our main contribution in this paper, the redesign of U,W,X-list and V-list computations by rearranging data to optimize cache usage. Furthermore, we discuss multithreading and vector intrinsics to extract maximum intra-node performance.

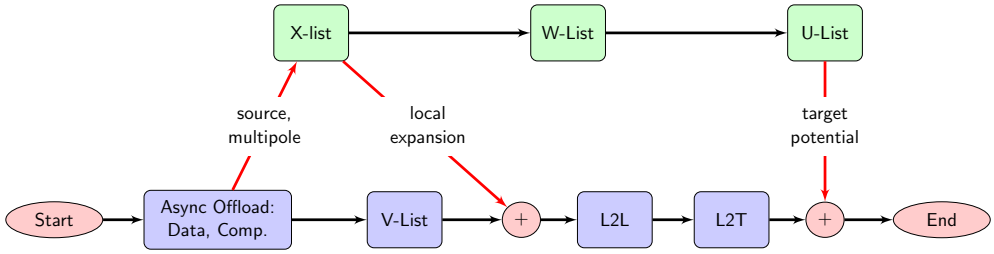


Fig. 10 *Asynchronous computation of different interaction types on co-processor (green) and CPU (blue), and data transfer (red arrows) between host and device memory in the downward pass of FMM. The source density and multipole data is the input and output is the target potential.*

3.1.1. Asynchronous Execution on co-processor. In heterogeneous architectures it is essential that we overlap computation on CPU with computation on co-processor. In FMM there is a natural decomposition of the downward pass: we compute U,W,X-list interactions on co-processor and and the rest (V-list, L2L and L2T) are computed on CPU (Figure 10). At the start of the downward pass, we initiate asynchronously the following operations: transfer source Chebyshev data from host memory to co-processor, execute X-list interactions on co-processor, transfer downward-equivalent density (local expansion) from co-processor to host, transfer upward-equivalent densities (multipole expansion) from host to co-processor, execute W-list, U-list on co-processor and transfer target Chebyshev potential from co-processor to host. These operations are

non-blocking, so the CPU can continue its execution; on co-processor, each of these operations execute in sequence.

On the CPU, we compute V-list interactions. We wait for the downward-equivalent densities to finish transferring from co-processor to host and then add the V-list contributions to it. On the CPU, we continue by evaluating L2L and L2T interactions and then wait for the target Chebyshev potential to complete transferring from co-processor to CPU before adding the contributions from L2T to the target potential.

The next two sections summarize the most important aspects of our work, reorganizing the data structure to optimize cache performance. The near interaction optimizations are possible only for volume potential methods (not for particles). The V list optimizations can be applied to both particle and volume FMM.

3.1.2. U,W,X-List Optimizations. We precompute the translation operators for U,W,X-list translations as matrices. For example: a source octant $\mathcal{B}_s$ interacts with a target octant $\mathcal{B}_t$ through an interaction matrix M_k (k depends on the relative position of $\mathcal{B}_s$ with respect to $\mathcal{B}_t$). Depending on the type of interaction (U, W or X), the corresponding source vector (v_s) may be the upward-equivalent density (multipole expansion) vector or the Chebyshev source density coefficients. Similarly, the target vector (v_t) may be the downward-equivalent density (local expansion) vector or the Chebyshev coefficients of the output potential. The contribution from v_s evaluated through M_k is added to v_t as: $v_t = v_t + M_k v_s$.

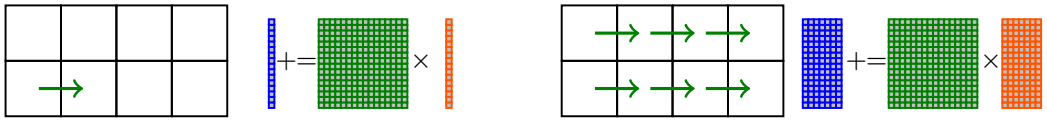


Fig. 11 *The left figure shows how a naive implementation would compute one interaction at a time using DGEMV. The right figure shows interactions in the same direction, i.e. with the same interaction matrix, combined into a single DGEMM.*

Now, consider several source octants $\mathcal{B}_{s_i}$ interacting with target octants $\mathcal{B}_{t_i}$, where $(t_i, s_i) \in I_k$ and I_k is the list of index pairs for source and target octants interacting through the interaction matrix M_k as shown in Figure 11:right. We can now combine these matrix-vector products into a single matrix-matrix multiplication as:

$$[v_{t_1}, v_{t_2}, \ldots v_{t_n}] = [v_{t_1}, v_{t_2}, \ldots v_{t_n}] + M_k \, [v_{s_1}, v_{s_2}, \cdots v_{s_n}]$$

where, $(s_i, t_i) \in I_k, \forall i = 1, .., n$. By doing so, we can now use matrix-matrix multiplication function which is a level-3 BLAS function, instead of matrix-vector multiplication (a level-2 BLAS function) and achieve better performance.

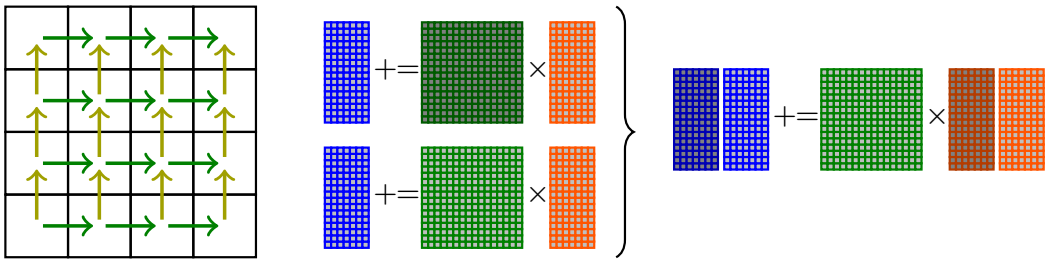


Fig. 12 *Using symmetries to combine interactions in the same interaction class, to produce larger matrices, achieving high Flop rates even for small problems.*

Many of the interaction matrices can be derived from another matrix through suitable permutation and scaling of its rows and columns, as discussed in §2.2.5. We choose one matrix as representative of the set of matrices which can be derived from each

other. We derive a matrix $M_{k'}$ belonging to the class of matrices M_k using suitable permutation operators P and Q (sparse matrices) to permute the rows and columns respectively: $M_{k'} = PM_kQ^T$. We can therefore compute interactions through $M_{k'}$ as:

$$\begin{aligned} v_t &= v_t + (PM_kQ^T)v_s \\ &= v_t + P(M_k(Q^T v_s)) \end{aligned}$$

where, Q^T (acting on v_s) is also a permutation operator. So, we can now construct larger matrices and compute all interactions belonging to the same interaction class (Figure 12) using a single matrix-matrix multiplication:

$$\begin{aligned} \left[w_1^1, w_2^1, \cdots w_1^2, \cdots w_n^m\right] &= M_k \left[Q_1^T v_{s_1^1}, Q_1^T v_{s_2^1}, \cdots Q_2^T v_{s_1^2}, \cdots Q_m^T v_{s_n^m}\right] \\ \left[v_{t_1^1}, v_{t_2^1}, \cdots v_{t_1^2}, \cdots v_{t_n^m}\right] &= \left[P_1 w_1^1, P_1 w_2^1, \cdots P_2 w_1^2, \cdots P_m w_n^m\right] \end{aligned}$$

where, $(t_i^j, s_i^j) \in I_{k_j}$ and $M_{k_j} = P_j M_k Q_j^T \quad \forall j = 1, \cdots, m$.

Matrix-matrix multiplication can achieve a very high percentage of the theoretical peak performance for most architectures for sufficiently large matrices. However, the performance degrades for smaller matrices. It is even more important in the case of the co-processor which requires the matrices to be even larger to achieve good performance. Using symmetries significantly improves performance for U,W,X-list interactions on both the CPU and on co-processor particularly for small problems.

3.1.3. V-List Optimizations. For a source octant $\mathcal{B}_s$ interacting with a target octant $\mathcal{B}_t$ such that $\mathcal{B}_s \in \mathcal{V}(\mathcal{B}_t)$, the Hadamard product for the interaction is represented as: $v_t = v_t + M_k \circ v_s$. Since Hadamard product has $\mathcal{O}(n)$ floating point operations, and $\mathcal{O}(n)$ memory accesses for vectors of length n, a naive scheme will be bound by the memory bandwidth. However, we note that V-list interactions have spatial locality, i.e. the same set of source and target vectors are used when evaluating interactions for a compact region in space. Therefore, if we can keep data in cache then we can significantly improve performance.

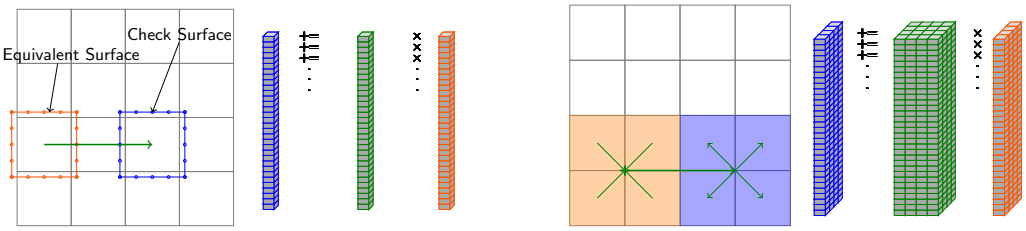


Fig. 13 *Left: interaction list for an octant using the conventional organization of the V-list interaction. Right:Interaction between two sibling groups.*

The first optimization that we make is to interleave the source and target vectors for sibling octants. We compute interactions between adjacent sibling groups by loading the first eight elements (one from each sibling) in the interleaved source and target vectors and computing all interactions between these (Figure 13), represented as multiplication with an 8×8 matrix, then load the next eight elements from the vectors and so on for the length of the vectors. By doing so, we also compute interactions between adjacent octants which do not actually appear in V-list interactions, so the corresponding entry in the 8×8 matrix is zero. As a result, we perform some extra (about 10%) computation, however the increased efficiency justifies the additional computational cost.

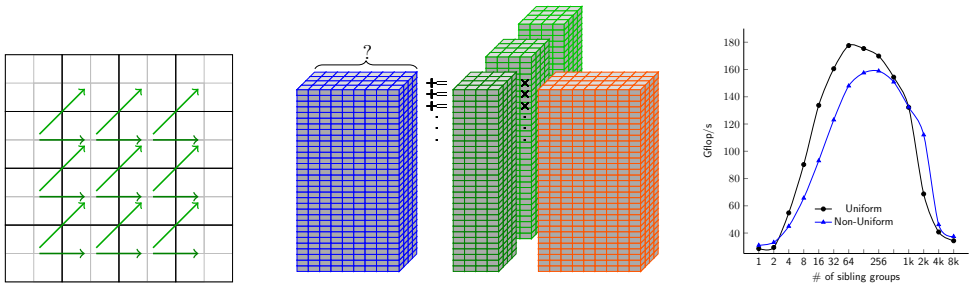


Fig. 14 *Combining multiple sibling group interactions and determining the optimal block size for interactions.*

Next, we note that as in the case of U,W,X-lists, we can combine several interactions (Figure 14) and replace matrix-vector multiplications by a single matrix-matrix multiplication. Since, these matrices are small it is not efficient to use BLAS and therefore, we implement our own matrix-matrix multiplication routine for 8×8 matrices optimized for the Sandy Bridge architecture by using AVX vector intrinsics. We can also look at this computation as a stack of matrix-matrix products. Each layer in the stack can be computed independently and we use OpenMP parallelism to distribute work across cores.

We further optimize cache usage by taking a Morton-sorted list of target octants (at the same level) and splitting into blocks which have spatial locality. By doing so, we ensure that we can keep the first eight elements from the source and target vectors of each sibling group (one layer of the stack) in cache when we loop over different interaction directions. In Figure 14, the plot shows the performance in GFLOP/s on 16 CPU cores ($2\times$Xeon E5-2680) for different block sizes. A block size of about 128 sibling groups worked best in our experiments, achieving nearly $180\text{GFLOP}/s$ or about 50% of the theoretical peak. In Table VI, we give the arithmetic intensity (defined as the number of floating point operations per word (8-bytes) of memory transfer) using our scheme on a uniform octree assuming that the block of data fits in the cache. As we increase the block size, the arithmetic intensity increases, however, the required cache also increases. For a block size of 128, we already require more more memory than what is available in L1 cache and this prevents us from achieving higher performance.

BlockSize	CacheSize (kB)	FLOP	Mem.Transfer	Arith.Intensity
32	22	4.3E+5	6.6E+3	65
64	36	8.5E+5	8.8E+3	97
128	61	1.7E+6	1.3E+4	131
256	106	3.4E+6	2.1E+4	162
512	190	6.8E+6	3.6E+4	189
∞	∞	∞	∞	277

Table VI *Arithmetic intensity (defined as* FLOP/*word) and the required cache size for different block sizes in V-list computation. Memory transfers are the number of words (8-bytes) transferred*

The overall algorithm (for one block of data) to compute Hadamard products is described in Algorithm 3.1. In step 1, we interleave source data for siblings. The outermost loop is over the height of the stack and this is parallelized using OpenMP. Then, we loop over each of the 26 interaction directions. In the innermost loop, we compute the matrix-vector products for each source-target interaction pair. We then de-interleave target data for siblings to get the downward-check potential in Fourier space for each octant.

Algorithm 3.1 VLISTHADAMARD

Input: v_s source vectors in Fourier space of length $(2m)^3$; M_k^i translation operators for $k = 1, .., 26$ (each sibling group interaction direction) and $i = 1, \cdots, (2p)^3$
Output: v_t target vectors in Fourier space of length $(2m)^3$

```
w_s ← Interleave(v_s, ∀siblings)
parfor i ← 1 to (2m)^3 do                                   ▷ vector length
    for k ← 1 to 26 do                                      ▷ directions
        for each (s, t) pair in direction k do
            w_t[8i, ··· , 8i + 7] ← M_k^i × w_s[8i, ··· , 8i + 7]
        end for
    end for
end parfor
(v_t ∀siblings) ← DeInterleave(w_t)
return v_t
```

p	number of processes
p_r	rank of current process
$\mathcal{T}_{p_r}$	local tree of process p_r
$\mathcal{P}_u$	user processes of $\mathcal{B}$
M_i	$\min \mathcal{L}(\mathcal{T}_i)$: first MortonId in local linear octree of process i.
$\text{Send}(S, p_i)$	send S to process i
$\text{Recv}(R, p_i)$	receive R from process i
Complexity Analysis	
t_w	per-word transfer time (1/bandwidth)
t_s	interconnect latency
T_{FMM}	FMM runtime

Table VII *Notation for distributed memory parallelism.*

3.2. Distributed Memory Parallelism

We first discuss the distributed memory tree construction and explain the partitioning of the domain across processes. We also discuss the parallel 2:1 balance algorithm on this distributed octree. Finally, we explain the communication steps in the parallel FMM algorithm. In Table VII, we list some notation which is used in this section.

For each algorithm in this section, we also provide the communication cost of the algorithm. For the analysis of communication cost, we assume an uncongested network and therefore assume that the cost of point-to-point communication between any two compute nodes is given by the sum of the latency t_s and the message transfer time $t_w N_m$ where, t_w is the per-word transfer time and N_m is the message size [Grama et al. 2003].

3.2.1. Tree Construction. On a distributed memory system we use Morton Ids for tree construction and load balancing [Warren and Salmon 1993]. We sort the initial seed points by their Morton Id using a distributed sort and partition the points equally between processes. Each process constructs a linear octree (a linear array of leaf octants sorted by their Morton Id) using its local point set. We then collect the Morton Ids of the first octant of each local octree and build the array $M_0, \cdots, M_{p-1}$. The domain belonging to a process with process id i is given by the region between M_i and M_{i+1} on the Morton curve.

We then proceed with the adaptive Chebyshev refinement. During the adaptive refinement process, we may need to load balance after each level of refinement (if the load imbalance exceeds a threshold). We load balance by redistributing the leaf octants equally across processes, while maintaining the Morton ordering of the octants (and updating M). This requires only point-to-point communication, where

each process communicates with at most eight other processes and we communicate data of size at most $N_{oct}q^3/6$ for each redistribution. For an octree with n_d levels, N_{oct} octants and assuming an uncongested network, the total communication cost is: $\mathcal{O}(t_s n_d + t_w n_d N_{oct} q^3/6)$.

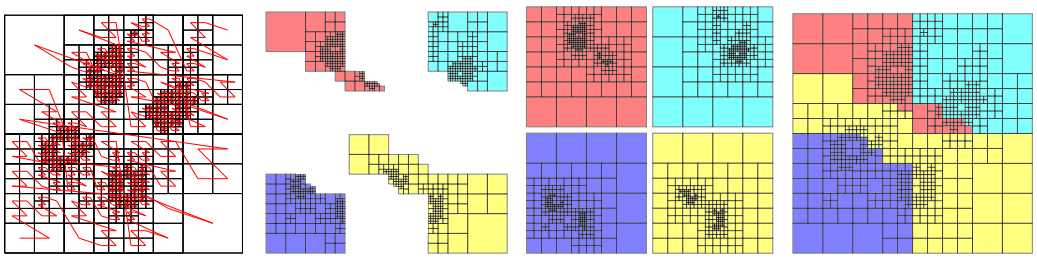


Fig. 15 *Steps in the distributed 2:1 balance refinement process: (1) The Morton sorted linear octree is (2) partitioned between the participating processes. (3) each process performs a local 2:1 balance refinement. (4) Now, the set of octants is sorted and duplicates are removed to obtain the final 2:1 balanced tree.*

3.2.2. 2:1 Balance Refinement. We use the sequential 2:1 balance algorithm (Algorithm 2.2) to obtain a distributed memory algorithm for 2:1 balance refinement. Starting with the distributed linear octree, i.e. the tree is partitioned across processes using the Morton Id of the octants. We perform the local 2:1 balancing (Algorithm 2.2) to generate the balancing octants. Next, we globally sort the octants using a variation of the hyperquick sort algorithm that we recently developed called HykSort [Sundar et al. 2013]. This is followed by removing duplicate octants, which is trivial given a sorted set of octants. The overall algorithm is described in Algorithm 3.2 and Figure 15 shows the steps involved for a 2D example.

Algorithm 3.2 PARALLELBALANCE

Input: $\mathcal{T}_{p_r}$ local octree.
Output: $\widehat{\mathcal{T}}_{p_r}$ globally balanced octree.

1: $\widehat{\mathcal{T}}_{p_r} \leftarrow$ SequentialBalance($\mathcal{T}_{p_r}$) ▷ Algorithm 2.2
2: HykSort($\widehat{\mathcal{T}}_{p_r}$) ▷ [Sundar et al. 2013]
3: RemoveDuplicates($\widehat{\mathcal{T}}_{p_r}$)
4: **return** $\widehat{\mathcal{T}}_{p_r}$

3.2.3. Distributed Memory FMM. In the upward pass of FMM, we compute the multipole expansions for each octant. For the octants which are entirely owned by a single process, the multipole expansion can be computed independently of the other processes. For non-leaf octants which are shared between processes, we need to perform a reduction to add up the contributions from regions owned by different processes.

In the downward pass of the FMM, we build the local expansion of each octant by adding contributions from V,X-lists and L2L interactions. We also compute U,W-list and L2T interactions to build the target potential. For a distributed octree, the interacting octants may belong to different processes. We, therefore, build a Locally Essential Tree (LET) by communicating the octants needed by a process for the downward pass. These new octants are called ghost octants. The number of such octants is proportional to the number of boundary leaf octants owned by a process.

The number of boundary octants is minimized since the octants owned by a process are a contiguous chunk of the Morton sorted array of leaf octants and since Morton

curves preserve locality, the set of leaf octants owned by a process are spatially adjacent. Once we have constructed the Locally Essential Tree, the downward pass of the FMM can proceed independently of all other processes.

Multipole Reduce: At the end of the FMM upward-pass, we need to add up the multipole expansions of the octants which span regions belonging to more than one process. The communication between processes required for the reduction of multipole expansion is mapped to a hypercube network topology. Each process identifies the list of octants it shares with other processes. These will be the ancestors of either the first or the last octant in the local Morton sorted list of leaf octants. Now, in each step, a process exchanges shared octants with another process that it is directly connected to in the hypercube topology, moving in the order of least significant dimension to the most significant dimension of the hypercube. In each step, shared octants among pairs of adjacent regions are merged forming a bigger region such that the only octants which still need to be updated are the octants shared across the new bigger regions and the octants contained entirely within a region now have the correct multipole expansions. At any stage in this process, each process maintains a list of octants which it's region (the region to which the process belongs) shares with adjacent regions, i.e. a maximum of $2L_{max}$ octants, where L_{max} is the maximum depth of the tree. In each communication step, the shared octants are exchanged and then each process independently adds up the multipole expansions of the octants shared between the two regions and builds the list of octants shared by the new region with adjacent regions. The pseudocode for the reduction is given in Algorithm 3.3.

Algorithm 3.3 MULTIPOLEREDUCE

Input: p_r process rank, p process count, $\mathcal{T}_{p_r}$ local tree.
Output: $\mathcal{T}_{p_r}$ with correct multipole expansions.

$S_1 \leftarrow \{\mathcal{B} : \mathcal{B} \in$ **Ancestors**$(\min \mathcal{L}(\mathcal{T}_{p_r}))\}$
$S_2 \leftarrow \{\mathcal{B} : \mathcal{B} \in$ **Ancestors**$(\max \mathcal{L}(\mathcal{T}_{p_r}))\}$
for $i \leftarrow 0$ to $\log p$ **do**
  $p_0 \leftarrow p_r$ XOR 2^i
  Send$([S_1, S_2], p_0)$
  Recv$([R_1, R_2], p_0)$
  if $p_r \leq p_0$ **then**
    Reduce(S_2, R_1)
    $S_2 \leftarrow R_2$
  else
    Reduce(S_1, R_2)
    $S_1 \leftarrow R_1$
  end if
end for

Time Complexity:. $T(n) = \mathcal{O}(t_s \log p + t_w \log p \log N_{oct} + \log p \log N)$
Where, t_s, t_w are the communication latency and the per-word transfer time respectively, p is the number of processes, N_{oct} is the number of octants in the complete octree.

Multipole Broadcast: After the Multipole Reduce step, each octant has its correct multipole expansion. Now, we build the Locally Essential Tree (LET) by sending octants from its owner to each of its users. For each process (p_r process rank), we identify shared local octants Q, which have user processes other than the current process itself. We then proceed with the hypercube communication scheme, where we split the

process set into two halves ($\{p_1, \cdots, p_1 + 2^i - 1\}$ and $p_r \in \{p_2, \cdots, p_2 + 2^i - 1\}$) each with 2^i processes. Each process communicates with a process $p_0 \in \{p_1, \cdots, p_1 + 2^i - 1\}$ in the other half and sends those octants from its shared set Q, which have user processes in $\{p_1, \cdots, p_1 + 2^i - 1\}$. Next, we retain the new received octants and only those shared octants which will be used in subsequent communication steps. We stop when the process set contains only p_r. Then, Q contains all the ghost octants which together with the local octants in $\mathcal{T}_{p_r}$ make up the LET. The pseudocode for the hypercube broadcast is given in Algorithm 3.4.

Algorithm 3.4 CONSTRUCTLET

Input: p_r process rank, p process count, $\mathcal{P}_u(\mathcal{B})$ user processes of octant $\mathcal{B}$, $\mathcal{T}_{p_r}$ local tree.
Output: $\widehat{\mathcal{T}}_{p_r}$ local tree with ghost octants added.

$Q \leftarrow \{\mathcal{B} : \ \mathcal{B} \in \mathcal{T}_{p_r},\ |\mathcal{P}_u(\mathcal{B})| > 1\}$ ▷ all shared octants
for $i \leftarrow (\log p - 1)$ to 0 **do**
  $p_0 \leftarrow p_r$ XOR 2^i
  $p_1 \leftarrow p_0$ AND $(p - 2^i)$
  $p_2 \leftarrow p_r$ AND $(p - 2^i)$
  $S \leftarrow \{\mathcal{B} : \ \mathcal{B} \in Q,\ \mathcal{P}_u(\mathcal{B}) \cap \{p_1, \cdots, p_1 + 2^i - 1\} \neq \emptyset\}$
  $\text{Send}(S,\ p_0)$
  $\text{Recv}(R,\ p_0)$
  $Q \leftarrow \{\mathcal{B} : \ \mathcal{B} \in Q,\ \mathcal{P}_u(\mathcal{B}) \cap \{p_2, \cdots, p_2 + 2^i - 1\} \neq \emptyset\}$
  $Q \leftarrow Q \cup R$
end for
return $\widehat{\mathcal{T}}_{p_r} \leftarrow \mathcal{T}_{p_r} \cup Q$

Time Complexity:. The communication cost for the hypercube communication scheme is discussed in detail in [Lashuk et al. 2012]. For an uncongested network, that work provides a worst case complexity which scales as $\mathcal{O}(t_s \log p + t_w N_s (q^3 + m^2)\sqrt{p})$, where N_s is the maximum number of shared octants owned by any process. However, assuming that the messages are evenly distributed across processes in every stage of the hypercube communication process, we get a cost of $\mathcal{O}(N_s(q^3 + m^2) \log p)$. For our experiments with uniform octrees, the observed complexity appears to agree with this estimate. Also, since the shared octants are near the boundary of the process domains, we expect $N_s \sim (N_{oct}/p)^{2/3}$, where N_{oct} is total number of octants.

4. NUMERICAL EXPERIMENTS

In this section we present results to demonstrate the convergence of the scheme, the single node efficiency, the performance of the new V-list for the KIFMM, and the overall scalability of our solver.

Platforms and architectures. For the majority of our experiments we used **TACC's Stampede** system in both strong and weak scaling regimes. Stampede entered production in January 2013 and is a high-performance Linux cluster consisting of 6400 compute nodes, each with dual, eight-core processors for a total of 102,400 available CPU-cores. The dual-CPUs in each host are Intel Xeon E5 (Sandy Bridge) processors running at 2.7GHz with 2GB/core of memory and a three-level cache. The nodes also feature the new Intel Xeon Phi coprocessor. Stampede has a 56GB/S FDR Mellanox InfiniBand network connected in a fat tree configuration that carries all high-speed traffic (including both MPI and parallel file-system data). We have focussed on the CPU+Phi tests.

We also ran experiments on **ORNL's Titan** a Cray XK7 with a total of 18,688 nodes consisting of a single 16-core AMD Opteron 6200 series processor, for a total of 299,008 cores. Each node has 32GB of memory. It is also equipped with a Gemini interconnect and 600 terabytes of memory across all nodes. Most nodes are also equipped with NVIDIA GPUs but we have not used them in the experiments we report here.

4.1. Convergence Analysis

In §2.2.4, we discussed the sources of errors in our algorithm and discussed how those errors vary with different parameters. In this section, we conduct experiments to measure errors as a function of various FMM parameters and show that these errors converge as predicted by the theory. All errors reported in this section are are relative errors. We show convergence for Laplace, Stokes and Helmholtz kernels. We also report time to solution (T_{FMM}) and time per unknown (T_{FMM}/N) on a single CPU core and compare the sequential cost as a function of various paramters and kernels.

In our first experiment, we solve the Poisson's equation, with free space boundary condition and $f(x) = -(4\alpha^2|x|^2 - 6\alpha)e^{-\alpha|x|^2}$ where, $\alpha = 160$, $x \in [-0.5, 0.5]^3$. The solution $u(x)$ and its gradient $\nabla u(x)$ are given by $e^{-\alpha|x|^2}$ and $-2\alpha x e^{-\alpha|x|^2}$ respectively.

ϵ_{tree}	q	N_{leaf}	$\|\|e_f\|\|_\infty$	m	$\|\|e_u\|\|_\infty$	$\|\|e_{\vec{\nabla} u}\|\|_\infty$	T_{FMM}	T_{FMM}/N
1E-0	9	176	2.9E-5	10	1.5E-5	1.9E-4	0.19	4.8E-6
1E-1	9	512	1.9E-6	10	1.8E-6	3.2E-5	0.52	4.6E-6
1E-2	9	1072	2.7E-7	10	2.4E-7	5.6E-6	1.07	4.5E-6
1E-3	9	2080	7.0E-8	10	2.9E-8	1.6E-6	2.10	4.6E-6
1E-4	9	3480	3.1E-9	10	2.8E-8	1.9E-7	3.52	4.6E-6
1E-2	14	120	3.1E-7	10	6.5E-8	1.8E-6	0.30	3.7E-6
1E-3	14	176	3.9E-8	10	2.8E-8	7.2E-7	0.44	3.7E-6
1E-4	14	512	3.7E-9	10	2.8E-8	1.7E-7	1.21	3.5E-6

Table VIII *Here we report the convergence of our scheme for the Laplace kernel as a function of the approximation error for a smooth right hand side f. We specify relative error ϵ_{tree}. This controls the depth of the Chebyshev tree. The multipole order m is kept fixed. T_{FMM} is the evaluation time (excluding gradient computation).*

In Table VIII, we report convergence as we vary the input error tolerance ϵ_{tree} from 1E-0 to 1E-4. As we reduce ϵ_{tree}, the input error $||e_f||_\infty$ also reduces. Note that we are reporting the relative error in f and not the absolute error which is of the order of ϵ_{tree}. Here, we compute the output potential $u(x)$ and then differentiate the polynomial representation to obtain $\nabla u(x)$. We report relative errors $||e_u||_\infty$ and $||e_{\vec{\nabla} u}||_\infty$. We note that the error $||e_u||_\infty$ is of the same order as $||e_f||_\infty$, however $||e_{\vec{\nabla} u}||_\infty$ is about an order of magnitude larger. This is expected since numerical differentiation reduces the order of the method. We show results for two different values of q and observe that the higher order method requires fewer unknowns and is therefore more efficient. Also note that for a fixed value of q and m, the cost per unknown T_{FMM}/N is relatively constant for different problem sizes, and this shows $\mathcal{O}(N)$ complexity for FMM. Note that T_{FMM} does not include the time to compute the gradient $\vec{\nabla} u$, which is computed in a post-processing.

In Table IX we show convergence as we vary the order of multipole expansion m. Here, we choose a very small ϵ_{tree} so that input approximation error does not affect convergence. As we vary m from 4 to 12, we observe spectral convergence and we get about eight digits of accuracy. As before, here also we observe that for the same accuracy, we require fewer octants for $q = 14$ than for $q = 9$. Also, note that for larger values of m, the FMM cost is dominated by the cost of V-list interactions, and we can clearly see the $\mathcal{O}(m^3)$ complexity for a fixed q and ϵ_{tree}.

ϵ_{tree}	q	N_{leaf}	$\|\|e_f\|\|_\infty$	m	$\|\|e_u\|\|_\infty$	$\|\|e_{\vec{\nabla} u}\|\|_\infty$	T_{FMM}	T_{FMM}/N
1E-4	9	3480	3.1E-9	4	4.8E-4	9.1E-3	0.75	9.8E-7
1E-4	9	3480	3.1E-9	6	6.4E-6	1.4E-4	1.17	1.5E-6
1E-4	9	3480	3.1E-9	8	1.8E-7	2.1E-6	2.39	3.1E-6
1E-4	9	3480	3.1E-9	10	2.9E-8	1.9E-7	3.52	4.6E-6
1E-4	9	3480	3.1E-9	12	9.6E-9	9.9E-8	6.79	8.9E-6
1E-4	14	512	3.7E-9	4	4.2E-4	5.5E-3	0.63	1.8E-6
1E-4	14	512	3.7E-9	6	6.4E-6	5.8E-5	0.73	2.1E-6
1E-4	14	512	3.7E-9	8	1.8E-7	2.0E-6	0.94	2.7E-6
1E-4	14	512	3.7E-9	10	2.8E-8	1.7E-7	1.21	3.5E-6
1E-4	14	512	3.7E-9	12	1.0E-8	4.5E-8	1.77	5.1E-6

Table IX *Here we report the convergence of the scheme for the Laplace kernel with increase multipole order $multorder$ while keeping the right hand side approximation fixed. The right hand side f is smooth. T_{FMM} is the evaluation time (excluding gradient computation).*

We also test our code for a discontinuous RHS as follows:

$$-\Delta u(x) \;=\; \begin{cases} 1.0 & \text{if } |x| < R \\ 0.0 & \text{otherwise} \end{cases}$$

where, $R = 0.1$. The analytical solution for this problem is given by:

$$u\,(x) \;=\; \begin{cases} R^2/2 - x^2/6 & \text{if } |x| < R \\ R^3/3|x| & \text{otherwise} \end{cases}$$

Since, for a discontinuous distribution, it does not make sense to measure $||e_f||_\infty$ in its

L_{max}	q	N_{leaf}	$\|\|e_f\|\|_2$	m	$\|\|e_u\|\|_\infty$	$\|\|e_u\|\|_2$	T_{FMM}	T_{FMM}/N
4	9	120	3.5E-1	10	1.3E-2	7.5E-3	0.10	3.9E-6
5	9	176	2.1E-1	10	3.0E-3	4.9E-4	0.16	4.1E-6
6	9	1240	1.4E-1	10	1.0E-3	4.0E-4	1.12	4.1E-6
7	9	3480	9.0E-2	10	7.4E-4	2.3E-4	3.37	4.4E-6
8	9	9808	6.6E-2	10	2.0E-4	8.4E-5	9.76	4.5E-6
9	9	37136	4.7E-2	10	4.2E-5	8.6E-6	41.5	5.1E-6
4	14	120	1.8E-1	10	3.5E-3	2.1E-3	0.25	3.0E-6
5	14	176	1.1E-1	10	6.0E-3	3.2E-3	0.39	3.2E-6
6	14	1240	1.1E-1	10	1.1E-3	3.4E-4	2.60	3.1E-6
7	14	3480	7.3E-2	10	3.2E-4	1.8E-5	7.78	3.3E-6
8	14	9808	5.1E-2	10	9.8E-5	3.1E-5	22.6	3.4E-6
9	14	37136	3.7E-2	10	6.4E-5	3.7E-6	88.2	3.5E-6

Table X *Here we report the convergence with tree depth L_{max} for discontinuous input for Laplace kernel. Since the solution is not smooth, specifying an error tolerance will not work, we need to specify both tolerance (for the smooth regions of f) and a maximum tree depth (for octants that contain the discontinuous part of f).*

polynomial approximation, we instead report $||e_f||_2$ for the input. Note also that here the refinement of the octree is controlled by the maximum allowed depth L_{max}, and at the discontinuity, the octants will be refined to this depth. The theory predicts that for a discontinuous $f(x)$, the input error $||e_f||_\infty$ converges as $2^{-L_{max}/2}$ and this is the observed convergence rate in Table X. We also report the output error norms $||e_u||_\infty$ and $||e_u||_2$, and observe convergence to about 4 and 5 digits of accuracy respectively. In this case, there are no benfits to using a higher polynomial order and the low order scheme is about two times faster compared to the high order scheme.

ϵ_{tree}	q	N_{leaf}	$\|\|e_f\|\|_\infty$	m	$\|\|e_u\|\|_\infty$	$\|\|e_u\|\|_2$	T_{FMM}	T_{FMM}/N
1E-4	9	13952	9.2E-10	4	2.8E-3	3.1E-3	22.1	2.4E-6
1E-4	9	13952	9.2E-10	6	4.5E-5	6.1E-5	36.2	3.9E-6
1E-4	9	13952	9.2E-10	8	9.0E-7	1.4E-6	75.7	8.2E-6
1E-4	9	13952	9.2E-10	10	6.4E-8	7.4E-8	118	1.3E-5
1E-4	9	13952	9.2E-10	12	2.4E-8	1.6E-8	237	2.6E-5
1E-4	14	1128	1.0E-10	4	1.9E-3	3.0E-3	10.3	4.5E-6
1E-4	14	1128	1.0E-10	6	3.4E-5	6.1E-5	11.7	5.1E-6
1E-4	14	1128	1.0E-10	8	9.1E-7	1.5E-6	14.9	6.5E-6
1E-4	14	1128	1.0E-10	10	6.3E-8	8.4E-8	18.5	8.1E-6
1E-4	14	1128	1.0E-10	12	2.3E-8	1.8E-8	27.5	1.2E-5

Table XI *Convergence with multipole order m for Stokes kernel.*

In our third experiment we solve for the velocity field $u(x)$ for a Stokes flow, with free space boundary condition:

$$\begin{aligned} -\mu\Delta u + \nabla p &= 4L^2(5 - 2L\,|x|^2)e^{-L|x|^2}(x_3e_2 - x_2e_3) \\ u(x) &= \frac{2L}{\mu}e^{-L|x|^2}(x_3e_2 - x_2e_3) \end{aligned}$$

where e_2, e_3 are unit vectors along Y and Z axes respectively, $L = 125$, and the viscosity $\mu = 1$. The analytical solution for the velocity field $u(x)$ is used to compute the output relative error. The results are presented in Table XI.

ϵ_{tree}	q	N_{leaf}	$\|\|e_f\|\|_\infty$	m	$\|\|e_u\|\|_\infty$	$\|\|e_u\|\|_2$	T_{FMM}	T_{FMM}/N
1E-5	9	4152	8.1E-10	8	7.7E-3	2.4E-2	10.0	5.5E-6
1E-5	9	4152	8.1E-10	10	4.7E-5	1.1E-4	15.7	8.6E-6
1E-5	9	4152	8.1E-10	12	5.6E-8	3.3E-7	29.0	1.6E-5
1E-5	14	512	7.5E-10	8	6.4E-3	2.7E-2	3.65	5.2E-6
1E-5	14	512	7.5E-10	10	2.7E-5	1.1E-4	5.26	7.6E-6
1E-5	14	512	7.5E-10	12	4.6E-8	3.3E-7	8.29	1.2E-5

Table XII *Convergence with multipole order p for Helmholtz kernel.*

In Table XII, we demonstrate the case of an oscillatory kernel by solving the Helmholtz equation with wavenumber 10 and free space boundary condition:

$$\begin{aligned} \Delta u + \mu^2 u &= (4\alpha^2\,|x|^2 - 6\alpha + \mu^2)e^{-\alpha|x|^2} \\ u(x) &= e^{-\alpha|x|^2} \end{aligned}$$

where, $\alpha = 160$, $\mu = 20\pi$. The analytical solution u is used to compute the output error. The results are presented in Table XII. Since Helmholtz is an oscillatory kernel, it is much more difficult to solve with high accuracy. Consequently, we get larger errors compared to Laplace and Stokes for the same multipole order. However, we are still able to achieve about seven digits of accuracy with $m = 12$.

4.2. Scalability Analysis

In the following experiments, we use two different types of input octrees. **Uniform:** A uniform octree with a specified depth d with number of leaf octants $N_{leaf} = 8^d$. **Non-uniform:** A highly non-uniform octree with maximum depth ranging from 7 levels in our smallest test with 900 leaf octants to 25 levels in our largest experiment with 131 million octants corresponding to 73.6 billion unknowns.

4.2.1. Single node scalability. We now show some performance results on a single node of Stampede, which as 16 CPU cores and an Intel Phi co-processor.

core	N_{leaf}	HADAMARD	FFT+IFFT	ALL
1	512	0.308 (13.9)	0.127 (3.6)	0.434 (10.9)
4	512	0.081 (13.3)	0.033 (3.5)	0.114 (10.4)
16	512	0.021 (12.7)	0.010 (2.8)	0.035 (8.5)
1	4096	2.761 (12.4)	1.013 (3.6)	3.774 (10.1)
4	4096	0.720 (11.9)	0.258 (3.6)	0.977 (9.7)
16	4096	0.190 (11.1)	0.071 (3.2)	0.265 (8.9)

Table XIII *Timing (GFlop/s per core) results for shared memory strong scaling of V-List for uniform octree with Laplace kernel and* $m = 10$.

core	N_{leaf}	HADAMARD	FFT+IFFT	ALL
1	904	0.265 (11.1)	0.224 (3.6)	0.488 (7.7)
4	904	0.074 (9.9)	0.057 (3.5)	0.132 (7.1)
16	904	0.021 (8.6)	0.017 (3.1)	0.044 (5.3)
1	62483	24.989 (11.4)	15.439 (3.6)	40.428 (8.4)
4	62483	6.836 (10.4)	3.940 (3.6)	10.776 (7.9)
16	62483	1.883 (9.5)	1.040 (3.4)	2.925 (7.3)

Table XIV *Timing (GFlop/s per core) results for shared memory strong scaling of V-List for non-uniform octree with Laplace kernel and* $m = 10$.

	m	q	N_{leaf}	U,W,X-LIST	V-LIST	L2L+L2T	WAIT	ALL
Uniform Octree								
CPU	6	7	512	0.018 (200.7)	0.062 (146.6)	0.004 (109.5)	0.000	0.090 (146.6)
CPU+PHI	6	7	512	0.013 (285.2)	0.062 (148.0)	0.004 (109.5)	0.000	0.090 (146.6)
ASYNC	6	7	512	0.001 (–NA-)	0.063 (145.2)	0.004 (107.1)	0.001	0.077 (171.0)
CPU	6	7	4096	0.120 (238.7)	0.496 (147.7)	0.017 (200.4)	0.000	0.639 (164.8)
CPU+PHI	6	7	4096	0.070 (411.8)	0.494 (148.2)	0.017 (197.3)	0.000	0.601 (175.3)
ASYNC	6	7	4096	0.001 (–NA-)	0.502 (146.0)	0.017 (194.4)	0.001	0.531 (198.2)
CPU	6	7	32768	0.941 (244.4)	3.991 (147.0)	0.144 (183.3)	0.000	5.098 (165.3)
CPU+PHI	6	7	32768	0.534 (430.5)	3.940 (148.9)	0.137 (192.8)	0.000	4.694 (179.6)
ASYNC	6	7	32768	0.000 (–NA-)	3.984 (147.2)	0.138 (191.4)	0.001	4.157 (202.8)
Non-uniform Octree								
CPU	8	13	904	0.700 (274.2)	0.172 (92.2)	0.045 (94.9)	0.000	0.921 (230.0)
CPU+PHI	8	13	904	0.329 (583.9)	0.174 (91.1)	0.044 (99.3)	0.000	0.560 (378.2)
ASYNC	8	13	904	0.003 (–NA-)	0.173 (91.6)	0.046 (95.7)	0.122	0.349 (606.9)
CPU	8	13	9654	6.161 (304.7)	2.121 (108.6)	0.268 (172.3)	0.000	8.556 (251.8)
CPU+PHI	8	13	9654	2.820 (665.8)	2.114 (108.9)	0.265 (174.7)	0.000	5.268 (409.0)
ASYNC	8	13	9654	0.003 (–NA-)	2.123 (108.5)	0.275 (167.8)	0.574	3.001 (717.9)
CPU	8	13	62483	41.064 (295.4)	13.303 (111.8)	1.301 (229.7)	0.000	55.683 (249.9)
CPU+PHI	8	13	62483	18.263 (664.3)	13.241 (112.3)	1.279 (234.0)	0.000	33.190 (419.3)
ASYNC	8	13	62483	0.003 (–NA-)	14.356 (103.6)	1.306 (229.0)	3.469	19.284 (721.7)

Table XV *Timing (GFlop/s) results for Downward Pass for Stokes kernel.*

In Tables XIII and XIV , we demonstrate intra-node strong scalability for **our new V-list algorithm** for uniform and non-uniform octrees respectively. We only report OpenMP results since we get best performance with 16 OpenMP threads and one MPI process per compute node and this is also the mode of operation in all runs with more than one compute node. As a result of our new optimized algorithm for Hadamard product, we achieve 203Gflop/s per compute node or 60% of theoretical peak (340Gflop/s) on one node and achieve 90% efficiency for intra-node strong scaling for the uniform case. This is a significant improvement over a naive Hadamard product,

	m	q	N_{leaf}	U,W,X-LIST	V-LIST	L2L+L2T	WAIT	ALL
Uniform Octree								
CPU	8	7	512	0.005 (75.2)	0.019 (128.0)	0.001 (78.8)	0.000	0.029 (101.0)
CPU+PHI	8	7	512	0.012 (35.2)	0.019 (128.7)	0.001 (78.8)	0.000	0.040 (73.1)
ASYNC	8	7	512	0.003 (–NA-)	0.019 (124.7)	0.001 (78.8)	0.000	0.029 (100.7)
CPU	8	7	4096	0.024 (132.0)	0.147 (132.0)	0.006 (171.6)	0.000	0.182 (130.5)
CPU+PHI	8	7	4096	0.019 (165.7)	0.149 (130.0)	0.006 (174.3)	0.000	0.189 (125.3)
ASYNC	8	7	4096	0.003 (–NA-)	0.149 (130.6)	0.006 (163.8)	0.000	0.166 (143.2)
CPU	8	7	32768	0.178 (144.4)	1.171 (132.5)	0.061 (146.7)	0.000	1.417 (133.9)
CPU+PHI	8	7	32768	0.102 (250.6)	1.169 (132.8)	0.062 (142.6)	0.000	1.380 (137.5)
ASYNC	8	7	32768	0.004 (–NA-)	1.169 (132.7)	0.062 (144.1)	0.000	1.250 (151.8)
Non-uniform Octree								
OLD	10	13	904	0.315 (85.0)	0.214 (14.3)	0.009 (106.0)	0.000	0.570 (55.9)
CPU	10	13	904	0.123 (216.1)	0.037 (101.4)	0.008 (113.0)	0.000	0.169 (185.9)
CPU+PHI	10	13	904	0.076 (346.9)	0.037 (101.9)	0.008 (114.0)	0.000	0.128 (245.8)
ASYNC	10	13	904	0.000 (–NA-)	0.037 (101.1)	0.008 (114.0)	0.012	0.078 (404.0)
OLD	10	13	9654	1.294 (194.1)	2.724 (16.5)	0.069 (153.8)	0.000	4.258 (74.5)
CPU	10	13	9654	0.976 (257.4)	0.457 (115.8)	0.066 (159.3)	0.000	1.508 (208.6)
CPU+PHI	10	13	9654	0.537 (467.9)	0.451 (117.4)	0.063 (169.1)	0.000	1.085 (290.0)
ASYNC	10	13	9654	0.003 (–NA-)	0.455 (116.4)	0.063 (167.0)	0.046	0.580 (542.8)
OLD	10	13	62483	6.919 (234.6)	18.494 (15.7)	0.341 (200.0)	0.000	25.771 (76.8)
CPU	10	13	62483	5.993 (270.3)	2.784 (122.9)	0.344 (198.3)	0.000	9.111 (222.9)
CPU+PHI	10	13	62483	3.264 (496.4)	2.875 (119.1)	0.334 (204.1)	0.000	6.723 (302.0)
ASYNC	10	13	62483	0.000 (–NA-)	2.914 (117.5)	0.347 (197.0)	0.382	3.409 (595.7)

Table XVI *Timing (GFlop/s) results for Downward Pass for Laplace kernel.*

which was limited by the main-memory bandwidth and attained roughly 1Gflop/s per core with 16 threads per node.

In Table XV we show performance for various stages in the **downward pass** for the Stokes kernel. We show results for various values of parameters m (order of multipole expansion), q (degree of polynomial approximation) and for various problem sizes (number of leaf octants, N_{leaf}). For each case, we show the performance for three configurations: 1) CPU only configuration, 2) CPU+XEON PHI: with U,W,X-lists executing on Xeon Phi and everything else on CPU, 3) ASYNC: with Xeon Phi executing asynchronously and overlapped with CPU execution.

In Table XV we first show performance with $m = 6$ and $q = 7$ and uniform octree. In this case, U,W,X-lists make up only a small fraction of the total work. Even though for larger problems, the Xeon Phi is almost twice as fast as CPU for the same computation, the total time is dominated by the V-list time. In the ASYNC case, the Xeon Phi computation is completely overlapped with CPU computation and there is no WAIT time for CPU and we observe speed up of 1.23x.

For higher accuracy with $m = 8$ and $q = 13$ with non-uniform octree, U,W,X-list evaluation dominates the execution time for the CPU only case and for CPU+XEON PHI configuration, it is comparable to V-list execution time. In the ASYNC mode, the CPU is idle for some time as it waits for computation on Xeon Phi to complete. Here we observe a significant speedup ($2.9\times$ for large problems) because we are able to keep the Xeon Phi busy, and since Xeon Phi provides high flop rates.

In Table XVI, we provide similar results for Laplace kernel. For the non-uniform case, we also show the performance for the OLD code i.e. without the optimizations mentioned in this paper. We see speedup of about $3\times$ for CPU version and $6.5 - 7\times$ for the $Async$ case.

4.2.2. Strong Scaling. We now demonstrate strong or fixed size scalability of our method, i.e. we fix a problem size and we increase the number of processors. In each experiment, we show two cases. In the first, we use symmetries for U,W,X-list ineractions and in the second we compute without using symmetries. We show that for large problems, there is verry little difference between the two. However, for smaller grain

size (for large processor counts), using symmetries gives much better performance and we get better scalability. In each test case, we report a breakdown of the time spent in each stage of the method: Upward Pass, Communication and Downward Pass in the form of a bar graph. We also report the performance as GFLOP$/s$ per node and this is proportional to the parallel scaling efficiency. In Figure 16 we demonstrate strong

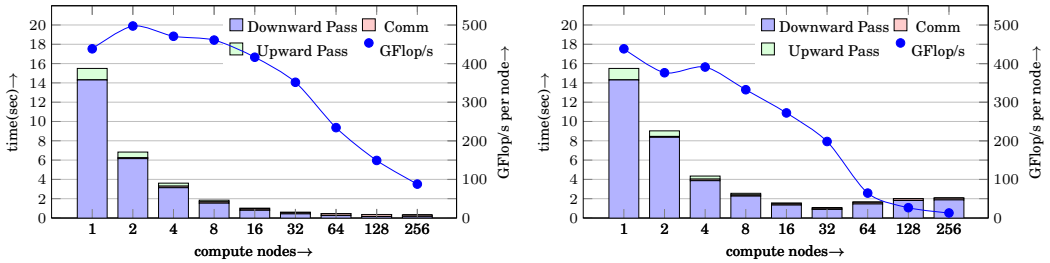


Fig. 16 *Strong scaling on Stampede (asynchronous execution on Phi) with and without symmetries for Laplace kernel with* $m = 10$, $q = 13$ *and non-uniform octree with problem size of 200k octants.*

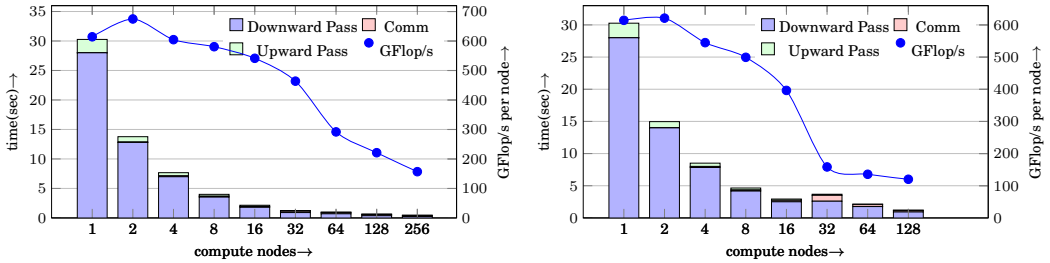


Fig. 17 *Strong scaling on Stampede (asynchronous execution on Phi) with and without symmetries for Stokes kernel with* $m = 8$, $q = 13$ *and non-uniform octree with problem size of 82k octants.*

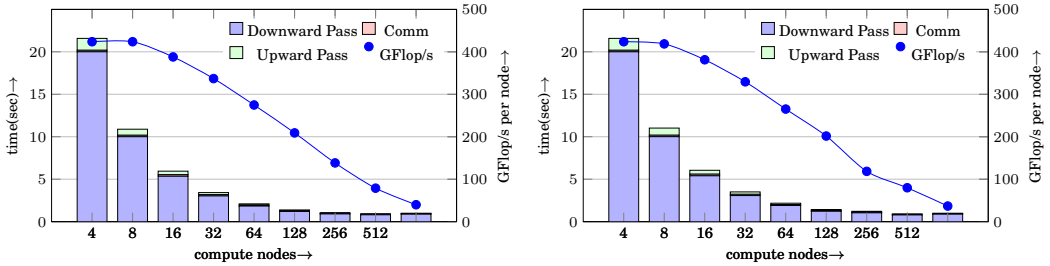
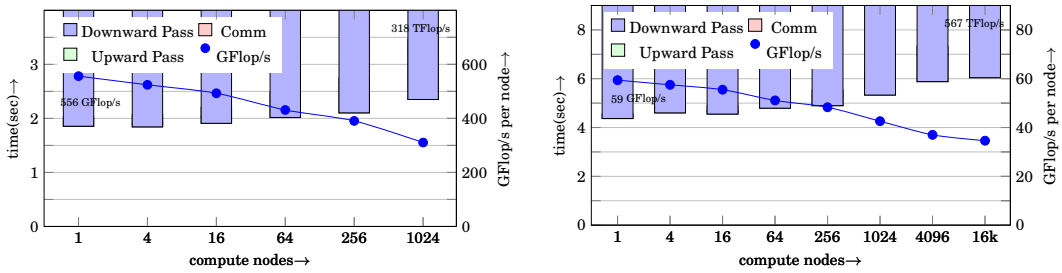


Fig. 18 *Strong scaling on Stampede (asynchronous execution on Phi) with and without symmetries for Helmholtz kernel with* $m = 10$, $q = 14$ *and uniform octree with problem size of 65k octants.*

scalability for the Laplace kernel with and without using symmetries to improve performance of U,W,X-lists. For the one compute node case, the problem is large and we are not left with any additional memory on the Phi to make effective use of symmmetries and therefore, we have lower performance than with two compute node case. We observe that with symmetries, we continue to get nearly 50% efficiency with 64 compute nodes, whereas without symmetries the efficiency has dropped below 15%. Similarly, in Figures 17 we show results for Stokes kernel, where we get nearly 75% efficiency with 32 compute nodes, and without symmetries it has dropped to about 25%. Finally, in Figure 18 we show results for the Helmholtz kernel with a uniform octree.

Since here, we only have U-list computation and no W,X-list interactions, even without symmetries, we are able to completely overlap computation of the Phi with V-list computation on the CPU, and therefore, we do not see any significant difference between the two cases (with and without symmetries). Here, however we loose efficiency since the Helmholtz kernel is not scale invariant and we have to compute interactions level-by-level. The efficiency drops to about 49% as we scale from 4 nodes to 128 compute nodes.



(a) Stampede: 23 levels, $m = 10$, $q = 13$ with 32k octants per compute node.

(b) Titan: 25 levels, $m = 10$, $q = 13$ with 8k octants per compute node.

Fig. 19 *Weak scaling for non-uniform octrees on Stampede (asynchronous execution on Phi) and Titan.*

4.2.3. Weak Scaling. In Figure 19(a) we demonstrate weak scalability for the non-uniform octree case. Here, the octree is highly non-uniform with a maximum tree depth of 23 levels. However, we still achieve about 72% efficiency for the upward and downward passes combined. For the overall FMM evaluation, we achieve about 56% efficiency with about 318 TFLOP/s or 23% of peak theoretical performance. In Figure 19(b), we present weak scalability results from ORNL's Titan, for the same problem as above, going up to 25 levels and 73.6 billion unknowns, while scaling from 1 MPI process to $16k$ MPI processes on $16k$ nodes of Titan and achieve 567 TFLOP/s.

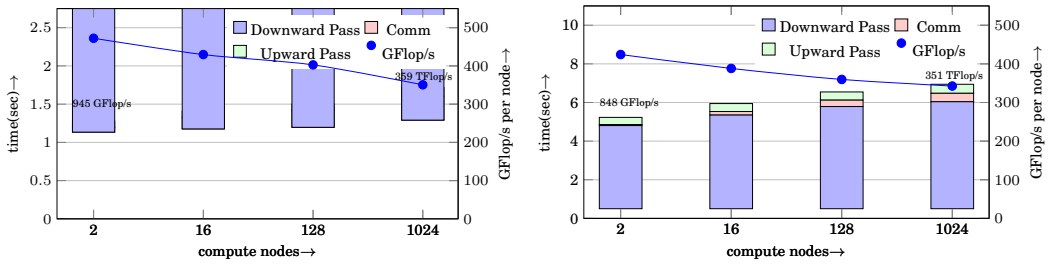


Fig. 20 *Weak scaling on Stampede (asynchronous execution on Phi) for Laplace (left) and Helmholtz (right) kernels with $m = 10$, $q = 14$ and uniform octree with grain size of 16k octants per compute node.*

In Figure 20 we show weak scalability for Laplace kernel (left) and Helmholtz kernel (right) for a grain size of 16k octants per compute node. In both cases, the communication cost appears to scale logarithmically as predicted for a network without congestion. The remaining stages, the upward pass and the downward pass, also scale well with about $85-90\%$ efficiency. Overall, for 1024 compute nodes, we have 74% efficiency and 359TFLOP$/s$ for Laplace kernel and 80% efficiency and 351TFLOP$/s$ for Helmholtz kernel. We have better scalability for Helmholtz, since it is a 2×2 tensor kernel, therefore it has $4\times$ times the computation but only $2\times$ the communication compared to the

Laplace problem for the same number of octants and there is less communication overhead for Helmholtz. This can be clearly seen from the bar graph.

4.2.4. Scalability of 2:1 Balance. Although 2:1 balance refinement is not part of the evaluation phase, it is an important component of the setup phase for our solver. In the past we have used an algorithm developed earlier in our group and described in [Sundar et al. 2008]. However, that algorithm did not take into account the load imbalance arising from local refinement and therefore, its performance degraded rapidly for large, highly non-uniform octrees, which we use in our scalability experiments. Table XVII shows weak scalability results for the old and new algorithms. The new algorithm is over 50 times faster for this input case.

$cores$	$N_{leaf}/core$	OLD	NEW
16	16752	0.2271	0.0126
64	15664	0.5969	0.0152
256	15143	1.6212	0.0186
1024	15107	5.5016	0.0255
2048	15101	9.2205	0.0291
4096	15106	12.9548	0.0441
8192	15113	27.7878	0.4992

Table XVII *Weak scaling for 2:1 balance refinement for old and new (3.2) algorithms.*

5. CONCLUSIONS

We have presented a high-order accurate, adaptive, scalable solver from boundary value problems in the unit box. Using this framework we created fast Poisson, Stokes and Helmholtz solvers (with either far field or periodic boundary conditions). We measured its efficiency by looking at Flop rates, μ secs per unknown, and the most generic metric, time per digit of accuracy for problems with continuous and discontinuous right hand sides.

Besides Stokes, Poisson and Helmholtz many other problems which fit in this category with either minor modifications (e.g., elasticity) We are currently extending the code by overlapping communication and computation for the distributed memory implementation and by considering more advanced all-to-all exchanges.

APPENDIX

A. EVALUATING SINGULAR INTEGRALS

For direct interactions, we need to evaluate integrals of the following form.

$$u(r_0) = \int_{\mathcal{B}} K(r - r_0)\, p(r)$$

where, $K(r - r_0)$ is the Green's function, $\mathcal{B}$ is the cubic domain of an octant and $p(r)$ is the polynomial approximation of the source density within the octant. This is a near-singular integral when r_0 is on the boundary of $\mathcal{B}$ and a singular integral when r_0 is inside $\mathcal{B}$. A simple tensor-product Gauss quadrature rule will converge very slowly for values of r_0 within or close to $\mathcal{B}$. To evaluate such integrals, we make use of the Duffy transformation [Duff 1982] followed by a tensor-product Gauss quadrature rule.

Consider the following integral (with $K(r - r_0) = \frac{1}{|r-r_0|}$ for Poisson's equation) over a regular pyramid.

$$\begin{aligned} u &= \int_0^1 \int_{-x}^{x} \int_{-x}^{x} K(x,y,z)\, p(x,y,z)\, \mathrm{d}z\, \mathrm{d}y\, \mathrm{d}x \\ &= \int_0^1 \int_{-x}^{x} \int_{-x}^{x} \frac{1}{\sqrt{x^2+y^2+z^2}}\, p(x,y,z)\, \mathrm{d}z\, \mathrm{d}y\, \mathrm{d}x \end{aligned}$$

We perform the following change of variables and transform the integration domain to a cuboid.

$$y = u\,x, \qquad z = v\,x$$

$$\begin{aligned} u &= \int_0^1 \int_{-1}^{1} \int_{-1}^{1} x^2\, K(x, u\,x, v\,x)\, p(x, u\,x, v\,x)\, \mathrm{d}v\, \mathrm{d}u\, \mathrm{d}x \\ &= \int_0^1 \int_{-1}^{1} \int_{-1}^{1} \frac{x}{\sqrt{1^2+u^2+v^2}}\, p(x, u\,x, v\,x)\, \mathrm{d}v\, \mathrm{d}u\, \mathrm{d}x \end{aligned}$$

This transformed equation does not have a singularity and can now be integrated using a tensor-product Gauss quadrature rule. Moreover, the integral with respect to x can be computed exactly by choosing the order of the rule appropriately in the x-direction.

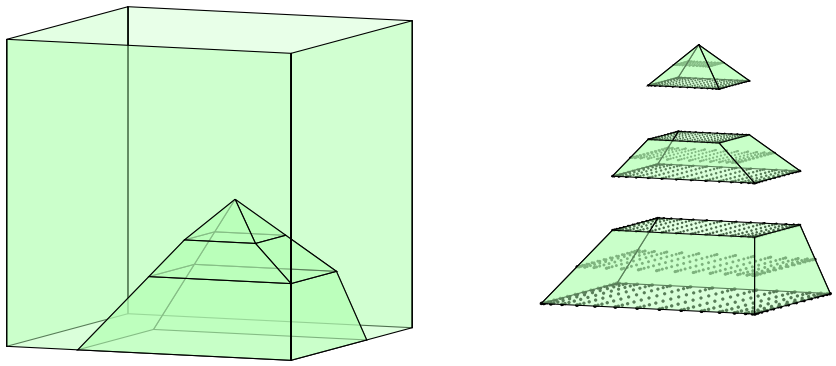


Fig. 21 *Intersection of a regular pyramid (apex at r_0) with a cubic octant and decomposition into frustum stack and smaller pyramid with node points for the Gauss-quadrature rule.*

To evaluate the singular integral over a cubic domain (an octant), we partition the domain into six regular pyramidal regions with the apex of the pyramids at the singularity. The intersection of each pyramid with the volume of the cube can be represented as stacks of rectangular frustums and a smaller pyramid (Figure 21). The integral over each of these components is individually evaluated using the technique described above using Duffy transformation.

We use the above scheme to precompute integrals $I_i^{r_0} = \int K(r - r_0)\, T_i(r)$ for each T_i in the Chebyshev basis set, and target location r_0. Then for any arbitrary (smooth) function, approximated by Chebyshev polynomials $p(r) = \sum \alpha_i T_i(r)$, we compute $u(r_0) = \int K(r - r_0)\, p(r)$ by the sum $u(r_0) = \sum \alpha_i I_i^{r_0}$.

REFERENCES

Michele Benzi, Gene H Golub, and Jörg Liesen. 2005. Numerical solution of saddle point problems. *Acta numerica* 14, 1 (2005), 1–137.

Carsten Burstedde, Lucas C Wilcox, and Omar Ghattas. 2011. p4est: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees. *SIAM Journal on Scientific Computing* 33, 3 (2011), 1103–1133.

Aparna Chandramowlishwaran, Kamesh Madduri, and Richard Vuduc. 2010. Diagnosis, tuning, and redesign for multicore performance: A case study of the fast multipole method. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, 1–12.

Ron O Dror, J.P. Grossman, Kenneth M Mackenzie, Brian Towles, Edmond Chow, John K Salmon, Cliff Young, Joseph A Bank, Brannon Batson, Martin M Deneroff, Jeffrey S Kuskin, Richard H Larson, Mark A Moraes, and David E Shaw. 2010. Exploiting 162-Nanosecond End-to-End Communication Latency on Anton. In *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–12. DOI:http://dx.doi.org/10.1109/SC.2010.23

Michael G Duff. 1982. Quadrature Over a Pyramid or Cube of Integrands with a Singularity at a Vertex. *SIAM J. Numer. Anal.* 19 (1982), 1260–1262.

Frank Ethridge and Leslie Greengard. 2001. A new fast-multipole accelerated Poisson solver in two dimensions. *SIAM Journal on Scientific Computing* 23, 3 (2001), 741–760.

A Gillman. 2011. *Fast direct solvers for elliptic partial differential equations*. Ph.D. Dissertation. University of Colorado.

A. Grama, A. Gupta, G. Karypis, and V. Kumar. 2003. *An Introduction to Parallel Computing: Design and Analysis of Algorithms* (second ed.). Addison Wesley.

T. Hamada, T. Narumi, R. Yokota, K. Yasuoka, K. Nitadori, and M. Taiji. 2009. 42 TFlops hierarchical N-body simulations on GPUs with applications in both astrophysics and turbulence. In *Proceedings of SC09 (The SCxy Conference series)*. ACM/IEEE, Portland, Oregon.

Qi Hu, Nail A Gumerov, and Ramani Duraiswami. 2011. Scalable fast multipole methods on distributed heterogeneous architectures. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 36.

Tobin Isaac, Carsten Burstedde, and Omar Ghattas. 2012. Low-cost parallel algorithms for 2: 1 octree balance. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*. IEEE, 426–437.

Pritish Jetley, Lukasz Wesolowski, Filippo Gioachin, Laxmikant V Kalé, and Thomas R Quinn. 2010. Scaling hierarchical N-body simulations on GPU clusters. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, 1–11.

Harper Langston, Leslie Greengard, and Denis Zorin. 2011. A free-space adaptive FMM-based PDE solver in three dimensions. *Communications in Applied Mathematics and Computational Science* 6, 1 (2011), 79–122.

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

Keith Lindsay and Robert Krasny. 2001. A particle method and adaptive treecode for vortex sheet motion in three-dimensional flow. *J. Comput. Phys.* 172, 2 (2001), 879–907.

James W Lottes and Paul F Fischer. 2005. Hybrid multigrid/Schwarz algorithms for the spectral element method. *Journal of Scientific Computing* 24, 1 (2005), 45–78.

Peter McCorquodale, Phillip Colella, Gregory T Balls, and Scott B Baden. 2006. A Local Corrections Algorithm for Solving Poisson's Equation in Three Dimensions. *Communications in Applied Mathematics and ComputationalScience* 2, LBNL–62377 (2006).

Akira Nukada, Kento Sato, and Satoshi Matsuoka. 2012. Scalable multi-GPU 3-D FFT for TSUBAME 2.0 Supercomputer. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*. IEEE, 1–10.

James C. Phillips, Gengbin Zheng, Sameer Kumar, and Laxmikant V. Kalé. 2002. NAMD: Biomolecular Simulation on Thousands of Processors. In *Proceedings of Supercomputing (The SCxy Conference series)*. ACM/IEEE, Baltimore, Maryland.

Joel R. Phillips and Jacob K. White. 1997. A Precorrected-FFT Method for Electorstatic Analysis of Complicated 3-D Structures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 16, 10 (1997), 1059–1072.

IF Sbalzarini, JH Walther, M. Bergdorf, SE Hieber, EM Kotsalis, and P. Koumoutsakos. 2006. PPM–A highly efficient parallel particle–mesh library for the simulation of continuum systems. *J. Comput. Phys.* 215, 2 (2006), 566–588.

Hari Sundar, George Biros, Carsten Burstedde, Johann Rudi, Omar Ghattas, and Georg Stadler. 2012. Parallel geometric-algebraic multigrid on unstructured forests of octrees. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 43.

Hari Sundar, Dhairya Malhotra, and George Biros. 2013. HykSort: A New Variant of Hypercube Quicksort on Distributed Memory Architectures. In *ICS'13: Proceedings of the International Conference on Supercomputing, 2013*.

Hari Sundar, Rahul S. Sampath, and George Biros. 2008. Bottom-Up Construction and 2:1 Balance Refinement of Linear Octrees in Parallel. *SIAM Journal on Scientific Computing* 30, 5 (2008), 2675–2708. DOI:http://dx.doi.org/10.1137/070681727

Toru Takahashi, Cris Cecka, William Fong, and Eric Darve. 2012. Optimizing the multipole-to-local operator in the fast multipole method for graphical processing units. *Internat. J. Numer. Methods Engrg.* 89, 1 (2012), 105–133.

Michael S Warren and John K Salmon. 1993. A parallel hashed oct-tree n-body algorithm. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*. ACM, 12–21.

Lexing Ying, George Biros, and Denis Zorin. 2004. A kernel-independent adaptive fast multipole method in two and three dimensions. *J. Comput. Phys.* 196, 2 (2004), 591–626.

Lexing Ying, George Biros, and Denis Zorin. 2006. A high-order 3D boundary integral equation solver for elliptic PDEs in smooth domains. *J. Comput. Phys.* 219, 1 (2006), 247–275.

R. Yokota, J.P. Bardhan, M.G. Knepley, LA Barba, and T. Hamada. 2011. Biomolecular electrostatics using a fast multipole BEM on up to 512 gpus and a billion unknowns. *Computer Physics Communications* (2011).