Algorithm XXX: 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 low-frequency 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 3E+23 unknowns since we used 23 levels in our octree).

CCS Concepts: •**Mathematics of computing** → **Mathematical software;**

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

ACM Reference Format:
Dhairya Malhotra and George Biros, 2016. Algorithm XXX: A Distributed-Memory Fast Multipole Method for Volume Potentials. *ACM Trans. Math. Softw.* 9, 4, Article XXXX (March 2016), 27 pages.
DOI: 0000001.0000001

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 an elliptic partial differential equation. The domain of integration Ω is the unit cube enclosing the support of $f(y)$. We use the volume fast multipole method (FMM) to approximate this inte-

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

gral. This approach was first introduced in [Ethridge and Greengard 2001] in 2D and extended to 3D in [Langston et al. 2011]. The key aspect in the formulation 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 tables. We expand the source f using an adaptive piecewise Chebyshev basis, and we precompute the near field of the convolution of the Green's function with Chebyshev polynomials. The latter is computationally feasible only if the tree is 2:1 balanced (for its definition, see §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

$$-\Delta u(x) = f(x), \quad \Rightarrow u(x) = \frac{1}{4\pi}\int_\Omega \frac{1}{\|x-y\|} f(y). \tag{2}$$

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:

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

where u is the velocity and p is the pressure. Finally equation (1) can be used for scattering problems, for example for acoustic scattering by solving the Helmholtz problem with wave number k:

$$-k^2 u(x) - \Delta u(x) = f(x), \quad \Rightarrow u(x) = \frac{1}{4\pi}\int_\Omega \frac{\exp^{-ik\|x-y\|}}{\|x-y\|} f(y). \tag{4}$$

The Poisson solver and its variants (for example, the modified Poisson solver) finds application in incompressible fluids [Cottet and Koumoutsakos 2000; Gunzburger 1989; Gunzburger and Nicolaides 1993], plasma physics and magnetohydrodynamics to enforce divergence-free conditions [Christlieb et al. 2005; Dedner et al. 2002], star formation and cosmological simulations [Kravtsov et al. 1997; Bode and Ostriker 2008; Springel et al. 2001], as well as quantum chemistry to approximate Gaussian distributions of charges [Harrison et al. 2004]. The Stokes solver is used for simulations of complex fluids in low Reynolds numbers with applications to polymer flows [Doi and Edwards 1988], biophysics [Pozrikidis 2001], and geophysics [Burstedde et al. 2013]. The Helmholtz problem appears in acoustic, seismic, and electromagnetic scattering [Colton and Kress 1998]. Besides free-space boundary conditions additional boundary conditions are possible to implement, 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 volume FMM [Langston et al. 2011; 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).

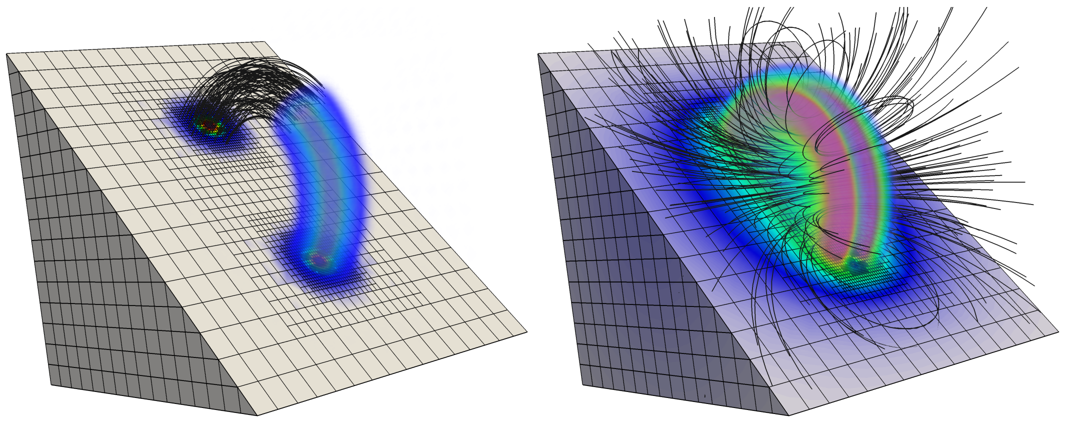


Fig. 1. Left: Vorticity field for a vortex-ring resolved on an adaptive Chebyshev octree is the input to volume FMM. Right: Output velocity field obtained from FMM by computing convolution of the vorticity field with the Biot-Savart kernel.

— We present integration of our method with coprocessors (Intel Xeon Phi and NVIDIA GPU).
— 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**.

We have implemented our method in the PVFMM software library. The library is open source and can be downloaded from the library homepage (http://pvfmm.org). To our knowledge, our algorithm is among the fastest AMR constant-coefficient Poisson, Stokes and low-frequency Helmholtz solvers. It achieves four 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 even 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 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.

There are many alternatives to using an integral equation formulation. Multigrid methods are also very effective and scalable [Sundar et al. 2012] and are applicable to Stokes and low-frequency Helmholtz problems. 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. Another approach for integral equations is to use wavelet decomposition [Harrison et al. 2004; Fann et al. 2007]. A comparison between the different methods for solving elliptic PDEs on the unit box is beyond the scope of this paper; however, we present some performance results for the wavelet approach in §4.3.1. Let us only mention that the two important features that modern solvers should provide is non-uniform discretization and high-order accuracy. For example, in [Langston et al. 2011] the authors compare the low-order solver of [McCorquodale et al. 2006] with a high-order volume potential FMM. The FMM solver required up to $100\times$ fewer unknowns. We discuss this more in §4. 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].

Regarding performance analysis, there have been several works optimizing the 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].

Outline. In §2, we briefly discuss the sequential algorithm for volume FMM. Our main contributions are described in the remaining sections. In §3, we explain the parallel implementation of the method discussing distributed-memory and shared-memory parallelism, task parallelism in relation to coprocessors, 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

The potential u at each point $x \in \Omega$ due to a continuous source distribution f defined on a cubic domain $\Omega = (0,1)^3$ is given by,

$$u(x) = \int_{\Omega} K(x-y)f(y) = \int\limits_{\mathcal{N}(x)} K(x-y)f(y) + \int\limits_{\mathcal{F}(x)} K(x-y)f(y) \tag{5}$$

In the volume FMM, we partition the domain using an adaptive octree. For each target evaluation point x, we have split the integral over Ω into the near interactions from $\mathcal{N}(x)$ (the set of octants containing or adjacent to x) and the far-field interactions from all the remaining octants $\mathcal{F}(x)$. The far-field interactions are approximated us-

Table I. Notation for tree data structure and volume FMM.

$\mathcal{T}$	octree
N_{oct}	number of octants
N_{leaf}	number of leaf octants
L_{max}	maximum tree depth
$\mathcal{B}$	tree node (octant)
$\mathcal{N}(\mathcal{B})$	near region of $\mathcal{B}$
$\mathcal{F}(\mathcal{B})$	far region of $\mathcal{B}$: $\Omega \setminus \mathcal{N}(\mathcal{B})$
$\mathcal{L}(\mathcal{T})$	leaf nodes in $\mathcal{T}$
K	kernel function
f	source density function
u	volume potential solution
$T_i(x)$	Chebyshev polynomial of degree i in x
ϵ_{tree}	tolerance for adaptive refinement
q	maximum degree of Chebyshev polynomials
m	multipole order

ing multipole and local expansions. The near interactions are evaluated through direct numerical integration. Since the kernel function has a singularity at the origin, this leads to singular and near-singular integrals which are costly to evaluate. In the remainder of this section we briefly describe the volume fast multipole method. A more detailed discussion of the method can be found in [Langston et al. 2011].

In Table I, we have summarized the notation used in this section. We discuss the octree data structure used to represent the source density function f and the volume potential u in §2.1. Then, in §2.2, we describe algorithms for efficient computation of singular and near-singular integrals on this data structure. In §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 §2.4, we discuss the overall algorithm for evaluating volume potentials. Then, we discuss some additional issues such as boundary conditions, computing gradients and selecting optimal parameter values. In §2.7, we discuss techniques for reducing memory usage.

2.1. Octree Construction

We partition the domain Ω using an octree $\mathcal{T}$ and approximate f at each leaf octant using Chebyshev polynomials of degree q. Then, at a leaf octant $\mathcal{B}$ (with coordinates mapped to $[-1,1]^3$), we have the following approximation for the density,

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

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 $i+j+k \leq q$. For adaptive octrees, we specify an error tolerance ϵ_{tree} and a maximum octree depth L_{max}. We estimate the truncation error at each leaf octant by computing the absolute sum of the highest order coefficients in the Chebyshev approximation. We subdivide the leaf octants with truncation error larger than ϵ_{tree} and approximate f on each new octant. We refine recursively until the desired accuracy is achieved or we reach the maximum allowed depth L_{max}.

We also represent the volume potential solution u using piecewise Chebyshev polynomials. At each leaf octant $\mathcal{B}$, we compute the following representation for the potential,

$$\hat{u}(x_1, x_2, x_3) = \sum_{i,j,k \geq 0}^{i+j+k \leq q} \beta^{\mathcal{B}}_{i,j,k} T_i(x_1) T_j(x_2) T_k(x_3) \tag{7}$$

Since $u(x) \in H^1$, it is smoother than $f(x) \in L^2$. Based on this observation, we assume that the potential can be represented accurately using the same octree refinement that we used for the density.

2.2. Interaction Operators

To evaluate the potential in (5), we need to compute singular and near-singular integrals over the leaf octants. For a leaf octant $\mathcal{B}$ with Chebyshev approximation of the density $\hat{f}(y) = \sum_{i,j,k} \alpha^{\mathcal{B}}_{i,j,k} T_{i,j,k}(y)$, the potential at a point x is given by,

$$u(x) = \int\limits_{y \in \mathcal{B}} K(x-y) \hat{f}(y) = \sum_{i,j,k} \alpha^{\mathcal{B}}_{i,j,k} \left[\int\limits_{y \in \mathcal{B}} K(x-y) T_{i,j,k}(y) \right] \tag{8}$$

We precompute the integral term in square brackets using the method explained in Appendix A. The potential from any leaf octant $\mathcal{B}$ at a point x (relative to $\mathcal{B}$) can then be evaluated using the sum $u(x) = \sum\limits_{i,j,k} \alpha^{\mathcal{B}}_{i,j,k} I^x_{i,j,k}$.

We precompute these quadratures (once for each level in the octree) so that source-to-multipole (S2L), source-to-target (U-list) and source-to-local (X-list) interactions in the FMM scheme can be represented as matrix-vector products. For S2M interactions, we precompute quadratures to evaluate the potential at each point on the upward-check surface (see [Ying et al. 2004] for definition) of the leaf octant. For X-list interactions, for each possible direction of the target octant, we precompute quadratures to evaluate the potential at the downward-check surface (see [Ying et al. 2004] for definition) of the target octant. Similarly, for U-list interactions, for each possible direction of the target octant, we precompute quadratures to evaluate the potential at the Chebyshev node points in the target octant. Then, we can construct the polynomial approximation of the potential from the values at the Chebyshev node points. We store the composition of these two operations (singular quadrature and polynomial approximation) so that the interactions are represented as translation from Chebyshev coefficients for density $\alpha^{\mathcal{B}_s}_{i,j,k}$ at the source octant $\mathcal{B}_s$, to the Chebyshev coefficients for potential $\beta^{\mathcal{B}_t}_{i,j,k}$ at the target octant $\mathcal{B}_t$.

2.3. 2:1 Balance Constraint

For a general octree, the number of interaction operators that must be precomputed for the volume FMM can be very large. 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.

We now explain the steps in our algorithm for 2:1 balance refinement (Algorithm 1). We loop from the finest to the coarsest level in the octree. In each iteration, we collect the set S of parents of all possible colleagues of non-leaf octants at that level. These are the non-leaf octants which must exist in 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 $\widehat{\mathcal{T}}$. We use "std::set" to implement this algorithm. It allows addition, deletion and searching of octants in $O(logN)$ steps. The complexity of the overall algorithm is $T(N_{oct}) = O(N_{oct} log N_{oct})$ where, N_{oct} is the number of octants.

It is possible to improve this complexity estimate by using alternative data structures. Such as, for a pointer based tree which also maintains a list of colleagues for each octant; addition, deletion and searching of octants in the neighborhood of a given octant requires $\mathcal{O}(1)$ time. Then, we can achieve $\mathcal{O}(N_{oct})$ complexity for the algorithm.

ALGORITHM 1: SEQUENTIALBALANCE

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

$\widehat{\mathcal{T}} \leftarrow \emptyset$, $S \leftarrow \emptyset$;
for $i \leftarrow L_{max}$ to $1$ do
    $N \leftarrow S \cup \{\mathcal{B} : \ \mathcal{B} \in \mathcal{T} \setminus \mathcal{L}(\mathcal{T}), \quad \text{Level}(\mathcal{B}) = i\}$;
    $S \leftarrow \text{Parent}(\text{Colleagues}(N))$;
    $\widehat{\mathcal{T}} \leftarrow \widehat{\mathcal{T}} \cup \text{Children}(N)$;
end
return $\widehat{\mathcal{T}}$;
```

2.4. Summary of Volume FMM

In volume FMM, the source density is represented by a polynomial approximation instead of the discrete sources in classical FMM. Therefore, translations involving the source term (S2M, X,U-list) need to be modified as discussed in §2.2. In addition, we want to represent the final result by a Chebyshev interpolation. We choose the target points to be the Chebyshev node points within each leaf octant and from that we compute the polynomial approximation using L^2 projection. After precomputing all the translation operators: S2M, M2M, L2L, L2T, U,V,W,X-list for each interaction direction and for each level in the octree, the volume FMM can 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 1.
— *Upward-Pass:* For all leaf octants apply S2M translation to construct the multipole expansion. 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 V-list and X-list translations to construct the local expansion of each octant. In top-down order apply the L2L translation to all octants and add the results to their local expansions. For all leaf octants, apply L2T, W-list and U-list translations to construct the final target potential as piecewise Chebyshev interpolation.

2.5. Gradients

To compute gradient of the potential, we differentiate its piecewise polynomial representation obtained from the volume FMM. We present some results using this method in §4.1. The gradient obtained in this way effectively has the degree $q-1$. This also leads to one-sided derivatives at octant boundaries and therefore, this may not be suitable for some applications. An alternative method is to use the gradient of the Green's function to compute U,W-list interactions and local-to-target translation. This has $3\times$ extra cost for near interaction and no extra cost for multipole-to-local interactions.

2.6. Selecting Optimal Parameter Values

We are interested in computing a solution to some accuracy in the least amount of time. To do this we need to select the optimal values of the three parameters: the tolerance for adaptive refinement ϵ_{tree}, the degree of Chebyshev polynomials q and the multipole order m. The parameters ϵ_{tree} and m directly control the accuracy. The value of ϵ_{tree} should be the same as required solution accuracy in L_∞ norm. Similarly, the optimal value for m is determined by the required solution accuracy and can be looked up from the convergence studies in §4.

We select the remaining parameter q to optimize for the solve time. In the following discussion we estimate the optimal value for q for a fixed number of unknowns. The cost of FMM evaluation is given by the number of interactions between the octants, weighted by the cost of each translation. For the range of q and m considered in this work, the cost of U-list and V-list interactions dominate over the cost of other translations. For a uniform octree, the total runtime is estimated by,

$$T_{\text{FMM}} = 9(q+2)^3 N\,\tau_{\text{u}} + 4\text{E}{+}4\,\frac{m^3}{(q+2)^3} N\,\tau_{\text{v}} + \mathcal{O}\left(m^2 N + m^4 q^{-3} N\right) \tag{9}$$

where, N is the number of unknowns, τ_u and τ_v are the inverse FLOP-rates for U-list and V-list interactions respectively. The parameters ϵ_{tree} and m are determined by the desired accuracy of the solution. We choose q to minimize time to solution T_{FMM}. Assuming that the FLOP-rates for U-list and V-list interactions are equal, for a fixed total number of unknowns N, we have $q \approx 4.1\sqrt{m} - 2$.

2.7. Reducing Memory Requirement

Even after enforcing the 2:1 balance constraint, the memory required to store translation operators can be very large. For example, Helmholtz kernel with $q = 14$ and 20 octree levels requires 38GB of memory for storing U-list interaction matrices. We now describe how this memory usage can be reduced.

Scale-invariant Kernels. Several kernel functions are scale-invariant i.e. when the distance between a source and a target point is scaled by α, the interaction between them is scaled by α^γ. For d-dimensional space, the interaction operators computed for the root level 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).

Symmetries. Most kernel functions have rotational symmetry and this allows us to group interaction directions into classes and store one interaction matrix for each interaction class. 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 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. 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 results in rearrangement of the vector elements corresponding to the reordering of points on the equivalent surface. For Chebyshev coefficient data a reflection along an axis results in a change of sign of all the odd order Chebyshev coefficients and swapping axes results in reordering 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. 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.

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, this would correspond to:

$$v_t = v_t + Q_{\alpha_1}^T \cdots Q_{\alpha_n}^T \times M_0 \times P_{\alpha_n} \cdots P_{\alpha_1} v_s \tag{10}$$

The composition of permutation and scaling operators can be precomputed.

3. PARALLEL ALGORITHMS

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 Xeon 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 coprocessors, by concurrently solving different parts of the problem on the coprocessor and the CPU. Then, in §3.1.2 and §3.1.3, we summarize the most important aspects of our work, re-organizing 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. Furthermore, we discuss multithreading and vector intrinsics to extract maximum intra-node performance.

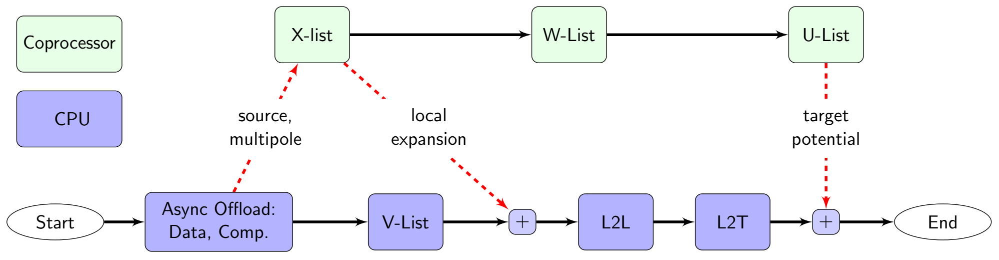


Fig. 2. Asynchronous computation of different interaction types on coprocessor (green) and CPU (blue), and data transfer (dashed arrows) between host and device memory in the downward-pass of FMM. The source density and multipole data is the input and output is the target potential.

3.1.1. Asynchronous Execution on coprocessor. In heterogeneous architectures it is essential that we overlap computation on CPU with computation on coprocessor. In FMM there is a natural decomposition of the downward-pass: we compute U,W,X-list interactions on coprocessor and the rest (V-list, L2L and L2T) are computed on CPU (Figure 2). At the start of the downward-pass, we initiate asynchronously the following operations: transfer source Chebyshev data from host memory to coprocessor, execute X-list interactions on coprocessor, transfer downward-equivalent density (local expansion) from coprocessor to host, transfer upward-equivalent densities (multipole expansion) from host to coprocessor, execute W-list, U-list on coprocessor and transfer target potential from coprocessor to host. These operations are non-blocking, so the CPU can continue its execution; on coprocessor, 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 coprocessor to host and then add the V-list contributions to it. We continue by evaluating L2L and L2T interactions on CPU and then wait for the target Chebyshev potential to complete transferring from coprocessor to CPU before adding the contributions from L2T to the target potential.

3.1.2. U,W,X-List Optimizations. We precompute the translation operators for U,W,X-list translations as matrices. A source octant $\mathcal{B}_s$ interacts with a target octant $\mathcal{B}_t$ through an interaction matrix M_k. The contribution from v_s evaluated through M_k is added to v_t as: $v_t = v_t + M_k v_s$. 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. We combine these matrix-vector products into a single matrix-matrix multiplication as follows,

$$[v_{t_1}, v_{t_2}, \dots v_{t_n}] = [v_{t_1}, v_{t_2}, \dots v_{t_n}] + M_k \, [v_{s_1}, v_{s_2}, \cdots v_{s_n}] \tag{11}$$

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 operation, instead of matrix-vector multiplication (a level-2 BLAS operation) and achieve better performance.

Many interaction matrices can be derived from another matrix through suitable permutation and scaling of its rows and columns, as discussed in §2.7. So, we can now assemble larger matrices and compute all interactions belonging to the same interaction class using a single matrix-matrix multiplication:

$$\left[w_1^1, w_2^1, \cdots w_1^2, \cdots w_n^m\right] = M_k \left[Q_1^T v_{s_1^1}, Q_1^T v_{s_2^1}, \cdots Q_2^T v_{s_1^2}, \cdots Q_m^T v_{s_n^m}\right] \tag{12}$$

$$\left[v_{t_1^1}, v_{t_2^1}, \cdots v_{t_1^2}, \cdots v_{t_n^m}\right] = \left[P_1 w_1^1, P_1 w_2^1, \cdots P_2 w_1^2, \cdots P_m w_n^m\right] \tag{13}$$

where, $(t_i^j, s_i^j) \in I_{k_j}$ and $M_{k_j} = P_j M_k Q_j^T \quad \forall j = 1, \cdots, m$. A similar technique is used [Messner et al. 2012] to optimize M2L interactions in black-box FMM. We have implemented highly optimized kernels for these permutation operations on Phi and GPUs. In addition, when assembling matrices, we try to minimize memory reads and writes by computing all the required permutations together for each vector loaded in cache. Using symmetries significantly improves performance for U,W,X-list interactions on both the CPU and on coprocessor, particularly for small problems.

3.1.3. V-List Optimizations. For a source octant $\mathcal{B}_s$ interacting with a target octant $\mathcal{B}_t$ through multipole-to-local (or V-list) translation operator, 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.

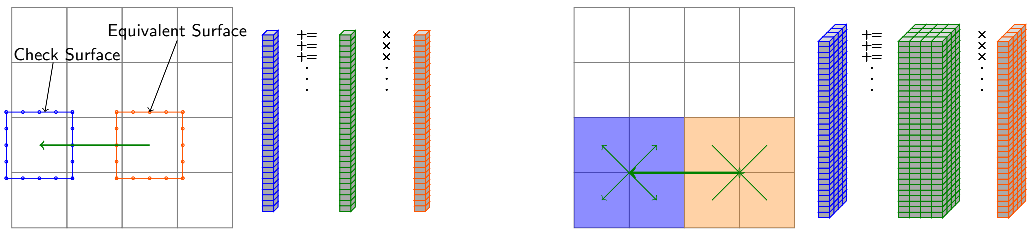


Fig. 3. Left: Interaction list for an octant using the conventional organization of the V-list interaction. Right: Interaction between two sibling groups.

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 3), 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 computation (about 10%), however the increased efficiency justifies the additional computational cost.

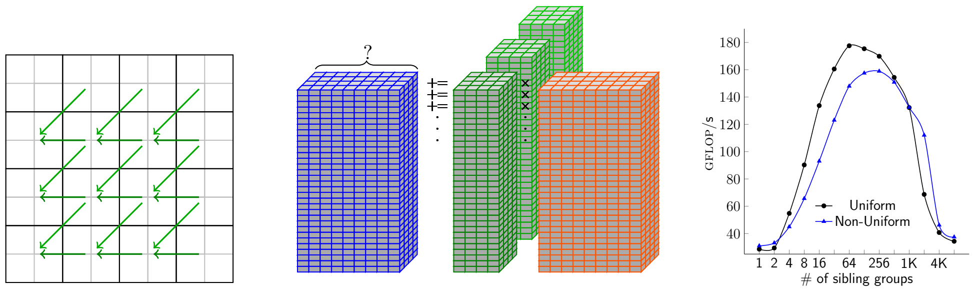


Fig. 4. Combining multiple sibling group interactions and determining the optimal block size for interactions. For the optimal block size, we achieve over 50% of peak performance even for highly adaptive octrees.

Next, we note that as in the case of U,W,X-lists, we can combine several interactions in the same direction (Figure 4) 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 4, 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 180GFLOP/s or about 50% of the theoretical peak. In Table II, 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 memory than what is available in L1 cache and this prevents us from achieving higher performance.

Table II. 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.

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
$\to\infty$	$\to\infty$	$\to\infty$	$\to\infty$	277

The overall algorithm (for one block of data) to compute Hadamard products is described in Algorithm 2. In lines 1-3, we interleave source data for siblings. Next, the outermost loop (line 4) 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 deinterleave target data (lines 11-13) for siblings to get the downward-check potential in Fourier space for each octant.

ALGORITHM 2: 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 $i = 1, \cdots, (2m)^3$
Output: $v_t$ target vectors in Fourier space of length $(2m)^3$

foreach $(s_1, \cdots, s_8) \in$ source sibling octants do                    // interleave data
    for $i \leftarrow 0$ to $(2m)^3 - 1$ do $w_s(8i, \cdots, 8i+7) \leftarrow [\, v_{s_1}(i), \cdots, v_{s_8}(i)\,]$;
    ;
end
for $i \leftarrow 0$ to $(2m)^3 - 1$ do in parallel                          // vector length
    for $k \leftarrow 1$ to $26$ do                                              // directions
        foreach $(s, t)$ pair in direction $k$ do
            $w_t(8i, \cdots, 8i+7) \leftarrow M_k^i \times w_s(8i, \cdots, 8i+7)$;
        end
    end
end
foreach $(t_1, \cdots, t_8) \in$ target sibling octants do                    // deinterleave data
    for $i \leftarrow 0$ to $(2m)^3 - 1$ do $[\, v_{t_1}(i), \cdots, v_{t_8}(i)\,] \leftarrow w_t(8i, \cdots, 8i+7)$;
    ;
end
return $v_t$;
```

Table III. Notation for distributed-memory parallelism.

p	number of processes
p_r	rank of current process
$\mathcal{T}_{p_r}$	local tree of process p_r
$\mathcal{P}_u(\mathcal{B})$	user processes of $\mathcal{B}$
$\text{Send}(S, p_i)$	send S to process i
$\text{Recv}(R, p_i)$	receive R from process i
Complexity Analysis	
t_w	per-word transfer time
t_s	interconnect latency
T_{FMM}	FMM solve time (excluding setup time)

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. To analyze the 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]. In Table III, we have summarized the notation used in this section.

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 p_r is given by the region between M_{p_r} and M_{p_r+1} on the Morton curve. We then proceed with the adaptive refinement. After each level of refinement, we load balance by redistributing the leaf octants equally across processes. The exchange of octant data requires only point-to-point communication with at most eight other processes (although determining which processes must exchange data requires collective communication). For an octree with n_d levels, N_{oct} local octants, the total communication cost is $T(n) = \mathcal{O}\left(t_s n_d \log p + t_w n_d (p \log p + N_{oct} q^3/6)\right)$.

ALGORITHM 3: PARALLELBALANCE

```
Input: $\mathcal{T}_{p_r}$ unbalanced local octree, $L_{max}$ maximum tree depth
Output: $\widehat{\mathcal{T}}_{p_r}$ globally balanced octree

for $i \leftarrow L_{max}$ to $0$ do
    $N_i \leftarrow \{\mathcal{B} : \ \mathcal{B} \in \mathcal{T}_{p_r} \setminus \mathcal{L}(\mathcal{T}_{p_r}), \quad \text{Level}(\mathcal{B}) = i\}$;
    $N_i \leftarrow N_i \cup \text{Parent}(\text{Colleagues}(N_{i+1}))$;
end

$N \leftarrow N_0 \ \cup \ \cdots \ \cup \ N_{L_{max}}$;
$N \leftarrow \text{ParallelSort}(\ N \setminus \text{Ancestors}(N)\ )$;          // HykSort [Sundar et al. 2013]
$N \leftarrow \text{RemoveDuplicates}(N)$;

$\widehat{\mathcal{T}}_{p_r} \leftarrow \text{Children}(N)$;
$\widehat{\mathcal{T}}_{p_r} \leftarrow \text{CompleteOctree}(\widehat{\mathcal{T}}_{p_r})$;          // add missing octants

return $\widehat{\mathcal{T}}$;
```

3.2.2. 2:1 Balance Refinement. For the parallel 2:1 balance algorithm, we start with the distributed linear octree, i.e. a distributed linear array of tree nodes sorted by their Morton ID. The first few lines of Algorithm 3 are similar to the sequential version (Algorithm 1). We generate the set of non-leaf octants N in the local balanced octree, however do not add the leaf octants at this point. The leaf octants are added later and this reduces the communication cost by about $8\times$. Next, we globally sort the set of non-leaf octants N using a variation of the hyperquick sort algorithm that we have developed called HykSort [Sundar et al. 2013]. This is followed by removing duplicate octants, which is trivial given a sorted set of octants. Finally, the leaf octants ($\mathrm{Children}(N)$) are added and the tree is completed by adding missing octants in the Morton ID sequence. The advantage of this 2:1 balance algorithm is that it does not suffer from large load imbalance even for highly adaptive trees. Most other algorithms repartition octants as a post processing step and therefore, during the local refinement process, the load imbalance can potentially be unbounded.

3.2.3. Distributed-Memory FMM. In the upward-pass of FMM, we compute the multipole expansions for each octant. For non-leaf octants which are shared between processes, we need to perform a reduction to sum the contributions from regions owned by different processes.

In the downward-pass of the FMM, we compute interactions between octants. For a distributed octree, the interacting source octants may belong to different processes. Therefore, we build a local essential tree by communicating the ghost octants needed by a process for the downward-pass. Once we have constructed the local essential tree, the downward-pass of the FMM can proceed independently of all other processes.

Compared to [Lashuk et al. 2012], we have decoupled to reduction and broadcast operations. Although this does not change the overall complexity, the resulting algorithms are simpler. This also makes future optimization of the broadcast operation possible, by using point-to-point communication to exchange octants at finer levels in the octree and using the hypercube all-to-all scheme only for coarser octants.

ALGORITHM 4: 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 $p_r \le p_0$ then
        Reduce($S_2$, $R_1$);
        $S_2 \leftarrow R_2$;
    else
        Reduce($S_1$, $R_2$);
        $S_1 \leftarrow R_1$;
    end
end
```

Multipole Reduce. We give the pseudocode for reduction in Algorithm 4. For simplicity, we assume that the processor count p is of the form 2^k. The communication between processes required for the multipole reduction is mapped to a hypercube network topology. Each process identifies the list of octants it shares with other processes. These are the ancestors of either the first or the last leaf octant in the local octree. 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. The 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. At any stage in this process, each process maintains a list of octants which it's region 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 sums 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 time complexity for this algorithm is given by:

$$T(n) = \mathcal{O}(t_s \log p + t_w m^2 \log p L_{max} + m^2 \log p L_{max}) \quad \textbf{(14)}$$

Here, t_s, t_w are the communication latency and the per-word transfer time respectively, p is the number of processes, L_{max} is the maximum depth of the octree.

Multipole Broadcast. We build the local essential tree by sending ghost octants from its owner process to each of its user processes. The pseudocode for the hypercube broadcast is given in Algorithm 5. For simplicity, we have assumed that p is of the form 2^k. For each process p_r, we identify shared local octants Q. We split the processes into two groups ($\{p_1, \cdots, p_1 + 2^i - 1\}$ and $\{p_2, \cdots, p_2 + 2^i - 1\} \ni p_r$) each with 2^i processes. Each process communicates with a process p_0 in the other group $\{p_1, \cdots, p_1 + 2^i - 1\}$ 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 local essential tree.

ALGORITHM 5: 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 $(p - 2^i)$;
    $S \leftarrow \{\mathcal{B} : \mathcal{B} \in Q, \mathcal{P}_u(\mathcal{B}) \cap \{p_1, \cdots, p_1 + 2^i - 1\} \neq \emptyset\}$;
    Send($S$, $p_0$);
    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
return $\widehat{\mathcal{T}}_{p_r} \leftarrow \mathcal{T}_{p_r} \cup Q$
```

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, we get a cost of $\mathcal{O}(t_s \log p + t_w N_s (q^3 + m^2) \log p)$. For our experiments with uniform octrees, the observed complexity appears to agree with this estimate.

4. NUMERICAL EXPERIMENTS

In this section we present results for our parallel volume fast multipole method (PVFMM) library. We 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 software.

Platforms and architectures. For the majority of our experiments we used **TACC's Stampede** system in both strong and weak scaling regimes. Stampede 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 E5-2680 (Sandy Bridge) processors running at 2.7GHz with 2GB/core of memory and a three-level cache. The nodes also feature the Intel Xeon Phi SE10P 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 also ran experiments on **ORNL's Titan**, a Cray XK7 with a total of 18,688 nodes, each consisting of a single 16-core AMD Opteron 6274 (Interlagos) processor, for a total of 299,008 cores. Each node has 32GB of memory. It is also equipped with a Gemini interconnect. Most nodes are also equipped with NVIDIA GPUs but we have not used them in the experiments we report here.

4.1. Convergence Analysis

We conduct experiments to measure errors as a function of various parameters and show that they converge as predicted by the theory. All errors reported in this paper are relative errors. We also report time to solution (T_{FMM}) and CPU cycles per unknown ($cycles/N$) on a single CPU-core and compare the sequential cost as a function of solution accuracy. The time to solution does not include the setup time for precomputing interaction matrices, the tree construction, 2:1 balance refinement and building interaction lists for the octants.

Table IV. Convergence for Laplace kernels with the parameters ϵ_{tree}, q and m tuned for fastest time to solution. T_{FMM} is the evaluation time (excluding setup time and gradient computation). The CPU cycles per unknown increases gradually as we increase the accuracy.

ϵ_{tree}	m	q	N_{leaf}	N	$\|\|e_f\|\|_\infty$	$\|\|e_u\|\|_\infty$	$\|\|e_{\nabla u}\|\|_\infty$	T_{FMM}	$cycles/N$
1.0E-1	2	4	176	6.2E+3	6.9E-3	2.2E-2	1.2E-1	0.003	1.3E+3
4.0E-3	4	6	176	1.5E+4	1.1E-3	6.4E-4	1.0E-2	0.015	2.7E+3
4.2E-4	6	9	176	3.9E+4	5.1E-5	2.2E-5	4.2E-4	0.060	4.2E+3
1.7E-5	8	12	176	8.0E+4	1.3E-6	5.4E-7	1.4E-5	0.207	7.0E+3
1.0E-6	10	14	176	1.2E+5	1.2E-7	6.2E-8	2.0E-6	0.438	9.9E+3

In our first experiment, we solve the Poisson's equation, with free-space boundary conditions 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. In Table IV, we show convergence for the relative output errors $||e_u||_\infty$ and $||e_{\nabla u}||_\infty$. For a given accuracy (determined by m), we choose ϵ_{tree} and q to minimize the total solve time as discussed in §2.6. As we increase accuracy from 2E-2 to 6E-8 the cost per unknown ($cycles/N$) increases by about $8\times$. To obtain $\nabla u(x)$, we compute the output potential $u(x)$ and then differentiate the polynomial representation as discussed in

§2.5. We note that the error $||e_u||_\infty$ is of the same order as $||e_f||_\infty$, however $||e_{\nabla u}||_\infty$ is about an order of magnitude larger. This is expected since numerical differentiation reduces the order of the method. Note that T_{FMM} does not include the time to compute the gradient ∇u, which is computed in a post-processing step.

In Table V, we show results for a highly adaptive octree. We solve a Poisson problem with known analytical solution $u = \exp(-(r/R)^\nu)$ where, $R = 0.3$, $\nu = 40$. We show results for three different accuracies. For each case, we report the time to solution for different values of q. The optimal choice for q as discussed in §2.6 minimizes the cost per unknown, however often a somewhat higher discretization order is favorable since it reduces the number of unknowns N. We observe that for very low accuracies, a low value of q is favorable; however, for higher accuracies a larger value for q gives faster time to solution.

Table V. For a highly adaptive case with Laplace kernel, we show results for different accuracies (determined by the multipole order). We report the relative L^∞ and L^2 norms of the error. For each case, we vary q and choose the optimal value of ϵ_{tree}. We highlight the case with optimal value of q.

ϵ_{tree}	m	q	N_{leaf}	N	$\|\|e_f\|\|_\infty$	$\|\|e_u\|\|_\infty$	$\|\|e_u\|\|_2$	T_{FMM}	$cycles/N$
7.5E-2	2	2	118014	1.2E+6	6.8E-1	1.0E-1	3.5E-2	1.269	2.9E+3
1.1E-1	2	4	10382	3.6E+5	9.0E-1	9.5E-2	3.4E-2	0.171	1.3E+3
5.7E-2	2	6	6504	5.5E+5	5.6E-1	8.7E-2	3.3E-2	0.241	1.2E+3
8.0E-2	2	12	1240	5.6E+5	3.8E-1	6.6E-2	3.3E-2	0.529	2.6E+3
1.8E-3	4	4	356196	1.2E+7	4.4E-2	1.4E-3	3.4E-4	19.78	4.5E+3
1.1E-3	4	6	87816	7.4E+6	4.8E-2	1.7E-3	3.1E-4	6.85	2.5E+3
4.8E-3	4	12	5552	2.5E+6	6.3E-2	1.3E-3	3.6E-4	2.94	3.2E+3
8.5E-6	6	6	783672	6.6E+7	8.6E-4	2.5E-5	2.7E-6	142.4	5.8E+3
1.8E-5	6	12	38592	1.8E+7	1.4E-3	2.3E-5	3.0E-6	26.2	3.9E+3
8.1E-5	6	14	18824	1.3E+7	3.1E-3	3.1E-5	3.2E-6	21.7	4.5E+3

Table VI. We show convergence with tree depth L_{max} for discontinuous input (15) for Laplace kernel. Since the solution is not smooth, specifying an error tolerance will not work, we need to specify both tolerance $\epsilon_{tree} = 1\text{E-}9$ (for the smooth regions of f) and a maximum tree depth L_{max} (for octants that contain the discontinuous part of f). For each case, we select the optimal value of the parameters m and q.

L_{max}	m	q	N_{leaf}	N	$\|\|e_f\|\|_2$	$\|\|e_u\|\|_\infty$	$\|\|e_u\|\|_2$	T_{FMM}	$cycles/N$
4	2	2	120	1.2E+3	4.9E-1	1.1E-1	9.0E-2	0.001	2.3E+3
5	4	4	176	6.2E+3	2.5E-1	3.2E-2	2.2E-2	0.007	3.0E+3
6	4	4	1240	4.3E+4	1.9E-1	6.8E-3	2.1E-3	0.065	4.1E+3
7	4	4	3480	1.2E+5	1.3E-1	1.3E-3	4.3E-4	0.200	4.5E+3
8	6	6	9808	8.2E+5	7.8E-2	4.5E-4	2.4E-4	1.886	6.2E+3
9	6	6	37136	3.1E+6	5.5E-2	1.2E-4	3.5E-5	7.257	6.3E+3
10	6	6	150760	1.3E+7	3.9E-2	1.8E-5	3.0E-6	29.924	6.2E+3

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

$$-\Delta u(x) = \begin{cases} 1.0 & \text{if } |x| < R \\ 0.0 & \text{otherwise} \end{cases} \tag{15}$$

where, $R = 0.1$. We compared our results with the analytical solution for this problem in Table VI. Since, for a discontinuous distribution, it does not make sense to measure $||e_f||_\infty$, we instead report $||e_f||_2$ for the input. Also note that here the refinement of the octree is controlled by both the maximum allowed depth L_{max} and ϵ_{tree}. At the discontinuity, the octants are refined to the depth L_{max} and in the smooth region, the octants are refined until the error is smaller than ϵ_{tree}. We expect that for a discontinuous $f(x)$, the input error $||e_f||_2$ converges as $2^{-L_{max}/2}$ and this is the observed convergence rate in Table VI. We also report the output error norms $||e_u||_\infty$ and $||e_u||_2$, and observe convergence to about 5 digits of accuracy. We choose q to minimize the solve time. For this problem, a relatively lower q results in a faster time to solution as compared to the smooth problems discussed before. This is because for discontinuous functions, a lower discretization order results in fewer unknowns.

We note that in the above experiments the cost per unknown (measured in $cycles/N$) for Poisson problems varies from 1E+3 to 1E+4 for accuracies in the range 1E-2 to 1E-7. We have presented similar results for Stokes and low-frequency Helmholtz in [Malhotra and Biros 2015]. For Stokes and Helmholtz, the cost per unknown is approximately $3\times$ and $2\times$ respectively as compared to the Poisson problem for the same parameter values.

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 and $N_{leaf} = 8^d$ leaf octants.
— *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 performance results on a single node of Stampede, which has 16 CPU-cores and an Intel Xeon Phi coprocessor. In Tables VII and VIII , 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

Table VII. Results with timing and performance in GFLOP/s per core (in parenthesis) for shared-memory strong scaling of V-List for uniform octree with Laplace kernel and $m = 10$. We show the time spent in the Hadamard product stage which we have optimized and discussed in §3.1.3 and the FFT and IFFT computation stage through FFTW library.

cores	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 VIII. Results with timing and performance in GFLOP/s per core (in parenthesis) for shared-memory strong scaling of V-List for non-uniform octree with Laplace kernel and $m = 10$. Even for highly non-uniform octrees and small problems sizes, we achieve very high FLOP-rates. This demonstrates the robustness of our scheme.

cores	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)

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, which was limited by the main-memory bandwidth and attained roughly 16GFLOP/s on one compute node.

Table IX. Results for timing and performance in GFLOP/s (in parenthesis) for downward-pass for Stokes kernel. We report results for low-accuracy case with $m = 2$ and $q = 4$ and high-accuracy case with $m = 8$ and $q = 13$ for different problem sizes on non-uniform octrees. We compare results for different stages of the downward-pass, for CPU only, CPU+Phi and asynchronous runs.

	N_{leaf}	U,W,X-LIST	V-LIST	L2L+L2T	WAIT	ALL
		Low-order ($m = 2,\ \ q = 4$)				
CPU	904	0.005 (100.2)	0.002 (121.4)	0.000 (19.3)	0.000	0.010 (90.7)
CPU+PHI	904	0.046 (12.8)	0.002 (132.5)	0.000 (19.3)	0.000	0.084 (10.4)
ASYNC	904	0.000 (–NA-)	0.003 (107.9)	0.000 (19.3)	0.031	0.044 (19.8)
CPU	9654	0.059 (101.8)	0.027 (159.0)	0.002 (43.9)	0.000	0.090 (116.0)
CPU+PHI	9654	0.069 (87.0)	0.027 (160.2)	0.002 (43.9)	0.000	0.104 (100.3)
ASYNC	9654	0.000 (–NA-)	0.029 (151.9)	0.002 (43.9)	0.043	0.076 (137.6)
		High-order ($m = 8,\ \ q = 13$)				
CPU	904	0.700 (274.2)	0.172 (92.2)	0.045 (94.9)	0.000	0.921 (230.0)
CPU+PHI	904	0.327 (586.4)	0.126 (125.2)	0.031 (137.2)	0.000	0.494 (428.4)
ASYNC	904	0.003 (–NA-)	0.124 (127.2)	0.031 (137.2)	0.173	0.333 (636.5)
CPU	9654	6.144 (305.4)	1.627 (141.7)	0.184 (250.6)	0.000	7.958 (270.6)
CPU+PHI	9654	2.773 (676.7)	1.632 (141.4)	0.189 (243.9)	0.000	4.674 (460.8)
ASYNC	9654	0.003 (–NA-)	1.642 (140.4)	0.190 (242.6)	0.987	2.832 (760.3)

In Table IX we show performance for various stages in the **downward-pass** for the Stokes kernel with a non-uniform octree. We show results for different values of parameters m (order of multipole expansion), q (degree of polynomial approximation) and for different problem sizes (number of leaf octants, N_{leaf}). In 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.

Table X. Results for timing and performance in GFLOP/s (in parenthesis) for downward-pass for Laplace kernel. We compare results for different stages of the downward-pass, for CPU only, CPU+Phi and asynchronous runs. For the high accuracy test case, we also compare with an original version of the code without the V-list optimizations and without symmetry optimizations for U,W,X-lists.

	N_{leaf}	U,W,X-LIST	V-LIST	L2L+L2T	WAIT	ALL
		Low-order ($m = 4,\ \ q = 6$)				
CPU	904	0.005 (97.9)	0.002 (105.3)	0.000 (22.8)	0.000	0.009 (85.1)
CPU+PHI	904	0.040 (12.9)	0.002 (110.3)	0.000 (22.8)	0.000	0.080 (9.6)
ASYNC	904	0.000 (–NA-)	0.003 (72.4)	0.000 (22.8)	0.035	0.060 (12.8)
CPU	9654	0.050 (99.9)	0.026 (133.0)	0.002 (72.6)	0.000	0.080 (108.4)
CPU+PHI	9654	0.064 (78.3)	0.025 (135.7)	0.003 (57.0)	0.000	0.099 (87.5)
ASYNC	9654	0.000 (–NA-)	0.027 (125.7)	0.002 (66.7)	0.035	0.068 (127.1)
		High-order ($m = 10,\ \ q = 13$)				
ORIGINAL	904	0.315 (85.0)	0.214 (14.3)	0.009 (106.0)	0.000	0.570 (55.9)
CPU	904	0.124 (215.0)	0.032 (116.2)	0.006 (153.7)	0.000	0.164 (191.6)
CPU+PHI	904	0.080 (332.1)	0.032 (115.1)	0.007 (146.9)	0.000	0.125 (250.3)
ASYNC	904	0.000 (–NA-)	0.033 (112.0)	0.006 (156.2)	0.040	0.082 (380.5)
ORIGINAL	9654	1.294 (194.1)	2.724 (16.5)	0.069 (153.8)	0.000	4.258 (74.5)
CPU	9654	0.973 (257.5)	0.379 (140.1)	0.046 (228.9)	0.000	1.403 (224.0)
CPU+PHI	9654	0.536 (467.3)	0.380 (139.7)	0.048 (220.8)	0.004	0.999 (314.6)
ASYNC	9654	0.000 (–NA-)	0.387 (137.0)	0.047 (223.1)	0.115	0.558 (563.0)

In Table IX we first show performance for a low-order case with $m = 2$ and $q = 4$. For smaller problem sizes, we observe longer solve time when we use the Phi coprocessor. This is because there is insufficient parallelism to effectively utilize the coprocessor. As we increase the problem size, we observe about 20-40% speedup for the ASYNC case compared to the CPU only case.

For higher accuracy with $m = 8$ and $q = 13$, 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 ($2.8\times$ for large problems) because we are able to keep the Xeon Phi busy.

In Table X, we provide similar results for Laplace kernel. For the high-order case, we also show the performance for the ORIGINAL code i.e. without the optimizations mentioned in this paper. We see speedup of about $3\times$ for CPU version and $7 - 7.6\times$ 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 interactions and in the second we compute without using symmetries. We show that for large problems, there is very 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 in GFLOP/s per node and this is proportional to the parallel scaling efficiency.

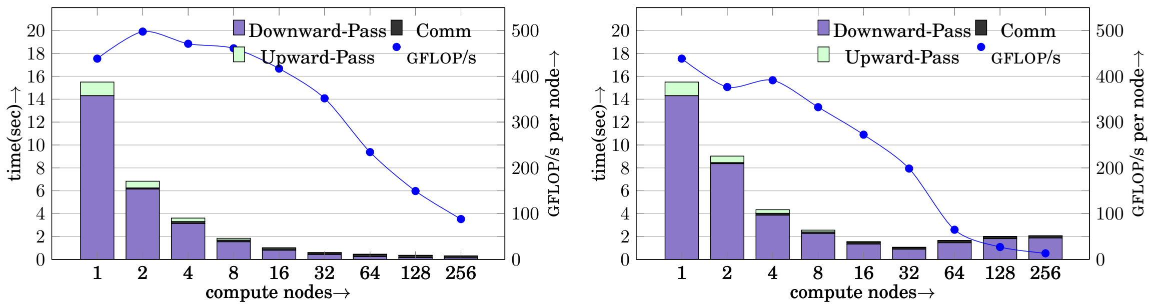
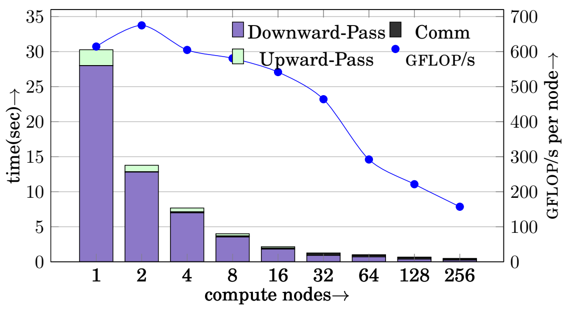


Fig. 5. Strong scaling on Stampede (asynchronous execution on Phi) with and without symmetries for Laplace kernel with $m = 10$, $q = 13$ and non-uniform octree with 200K octants.

In Figure 5 we demonstrate strong scalability for the Laplace kernel. 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 symmetries and therefore, we have lower performance than with two compute node case. We observe that with symmetries, we continue to get nearly 50% efficiency on 64 compute nodes, whereas without symmetries the efficiency has dropped below 15%. Similarly, in Figure 6 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 7 we show results for the low-frequency 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 lose efficiency since the Helmholtz kernel is



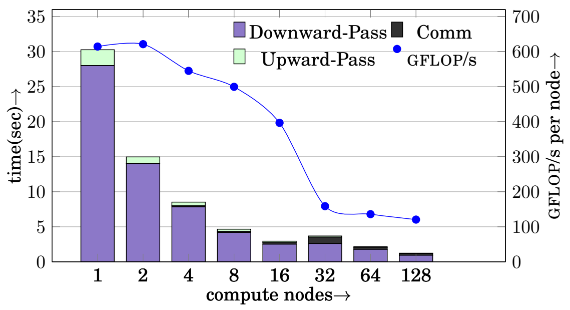
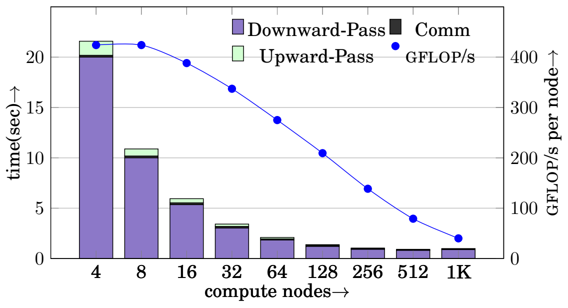


Fig. 6. Strong scaling on Stampede (asynchronous execution on Phi) with and without symmetries for Stokes kernel with $m = 8$, $q = 13$ and non-uniform octree with 82K octants.



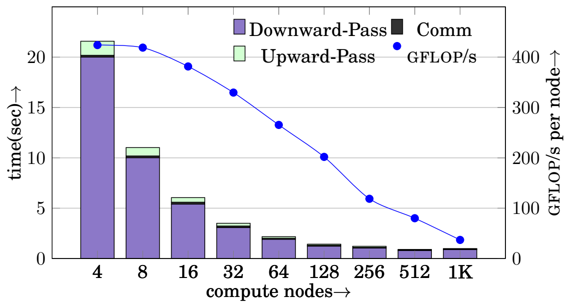
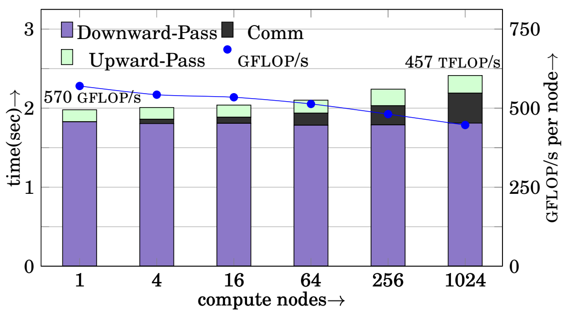


Fig. 7. Strong scaling on Stampede (asynchronous execution on Phi) with and without symmetries for low-frequency Helmholtz kernel with $m = 10$, $q = 14$ and uniform octree with 262K octants.

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.



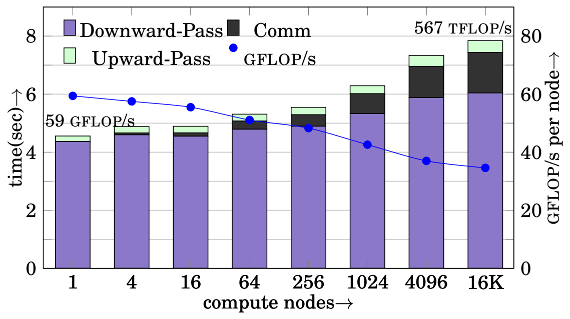


Fig. 8. Weak scaling for highly non-uniform octrees with $m = 10$, $q = 13$. Left: Stampede (asynchronous execution on Phi) with 32K octants per compute node and 23 levels of tree refinement. Right: Titan (without using GPUs) with 8K octants per compute node and 25 levels of refinement.

4.2.3. Weak Scaling. In Figure 8 (left) we demonstrate weak scalability on Stampede for the Laplace kernel. Here, the octree is highly non-uniform with a maximum tree depth of 23 levels. However, we still achieve about 93% efficiency for the upward and downward passes combined. For the overall FMM evaluation, we achieve about 78.4% efficiency with about 457TFLOP/s or 33% of peak theoretical performance. In Figure 8 (right), we present weak scalability results on 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 567TFLOP/s .

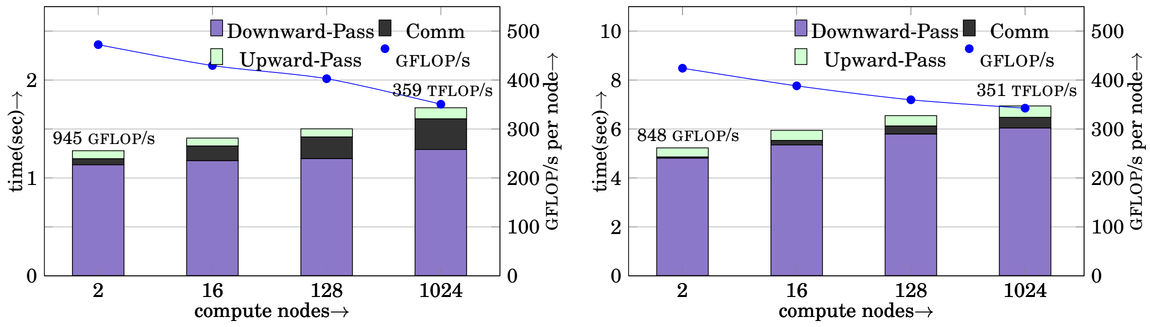


Fig. 9. Weak scaling on Stampede (asynchronous execution on Phi) for Laplace (left) and low-frequency Helmholtz (right) kernels with $m = 10$, $q = 14$ and uniform octrees with grain size of $16K$ octants per compute node.

In Figure 9 we show weak scalability with uniform octrees for Laplace kernel (left) and low-frequency 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 $4\times$ the computation but only $2\times$ the communication compared to the Laplace problem for the same number of octants and therefore communication overhead for Helmholtz is lower. 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 XI compares weak scalability of the original algorithm and the new algorithm presented in §3.2.2. The new algorithm is over 50 times faster for this input case.

Table XI. Weak scaling for 2:1 balance refinement for original algorithm and Algorithms 3.

cores	$N_{leaf}/core$	ORIGINAL	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

4.3. Comparison with Other Methods

In this section we present numerical results to compare the performance of our PVFMM library with other codes.

4.3.1. Comparison with MADNESS. In Table XII, we compare our method with the approach using wavelet decomposition [Harrison et al. 2004; Fann et al. 2007] implemented in the MADNESS library. We solve an analytical Poisson problem with the solution $u = \exp(-(r/R)^\nu)$ to about 5-digits of accuracy in L_2 norm for different values of ν and fixed $R = 0.3$. Increasing ν by $2\times$ increases the number of unknowns by roughly $4\times$. For the case $\nu = 160$, we use four compute nodes since MADNESS required more memory than what was available on a single compute node. We used $q = 10$ and $m = 6$ for PVFMMand 10-th order discretization for MADNESS. We report the time for tree construction, setup and solve. We also report the number of unknowns (N) and cost per unknown ($cycles/N$) for PVFMM. In each case, our PVFMM code is about an order of magnitude faster than MADNESS.

Table XII. Comparison with MADNESS for a Poisson problem with analytical solution $u = \exp(-(r/R)^\nu)$ for different values of ν, solved to about 5-digits of accuracy using 10-th order discretization.

		MADNESS			PVFMM					
ν	cores	$\|\|e_u\|\|_2$	T_{tree}	T_{solve}	N	$\|\|e_u\|\|_2$	T_{tree}	T_{setup}	T_{solve}	$cycles/N$
20	16	1.2E-5	1.8	1.9	1.6E+6	9.1E-6	0.22	0.11	0.19	4.9E+3
40	16	1.3E-5	7.9	6.6	6.7E+6	9.3E-6	0.53	0.56	0.77	5.0E+3
80	16	2.1E-5	33.1	23.2	2.5E+7	1.5E-5	1.74	2.77	3.07	5.2E+3
160	64	1.8E-5	60.2	48.9	1.0E+8	1.6E-5	4.77	3.07	3.14	5.3E+3

4.3.2. Comparison with EXAFMM. While we have limited our discussion to volume FMM, our software also implements particle FMM. The optimized V-list algorithm discussed in §3.1.3 is common to our particle and volume FMM. In Table XIII, we compare our particle code with EXAFMM [Yokota and Barba 2012] for an N-body problem with 1E+5 source and target particles randomly distributed in the domain. We compute the Coulombic potential and force at each target point and report the total time (tree construction, setup and solve time) for different solution accuracies. For extremely low accuracy the two methods have comparable performance; however, for single precision and higher accuracies our KIFMM based scheme is over an order of magnitude faster.

Table XIII. Results for computing Coulombic potential and force for $N = 1E+5$ source and target particles on a single core of Intel Xeon E5-2687W. We optimized parameter values of both codes to minimize the total solve time for the given accuracy.

EXAFMM						PVFMM				
error	N_{crit}	m	θ	T	$cycles/N$	error	N_{crit}	m	T	$cycles/N$
6.1E-2	40	4	0.99	0.28	8.7E+3	1.3E-1	100	2	0.44	1.4E+4
9.9E-3	40	4	0.52	0.89	2.8E+4	8.2E-3	100	4	0.63	2.0E+4
1.2E-3	40	6	0.43	1.50	4.7E+4	2.3E-4	100	6	1.09	3.4E+4
9.1E-6	40	8	0.27	12.32	3.8E+5	4.5E-6	250	8	1.39	4.3E+4
1.1E-6	40	8	0.21	23.77	7.4E+5	3.5E-7	250	10	1.62	5.0E+4
9.2E-8	40	8	0.15	49.67	1.5E+6	7.7E-9	250	12	2.13	6.6E+4
–	–	–	–	–	–	4.1E-12	250	18	4.69	1.5E+5

5. CONCLUSIONS

We have presented a high-order accurate, adaptive and scalable solver for boundary value problems in the unit box. Using this framework we created fast Poisson, Stokes and low-frequency Helmholtz solvers (with either free-space or periodic boundary conditions). We measured its efficiency by looking at FLOP-rates and CPU cycles per unknown for problems with continuous and discontinuous right hand sides.

Besides Stokes, Poisson and Helmholtz many other problems fit in this category with 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.

$$u(r_0) = \int\limits_{\mathcal{B}} K(r - r_0)\, p(r) \tag{16}$$

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 [Duffy 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.

$$u = \int\limits_{0}^{1} \int\limits_{-x}^{x} \int\limits_{-x}^{x} \frac{1}{\sqrt{x^2 + y^2 + z^2}}\, p(x, y, z)\, \mathrm{d}z\, \mathrm{d}y\, \mathrm{d}x \tag{17}$$

We perform a change of variables ($y = u\,x,\ z = v\,x$) and transform the integration domain to a cuboid.

$$u = \int\limits_{0}^{1} \int\limits_{-1}^{1} \int\limits_{-1}^{1} \frac{x}{\sqrt{1^2 + u^2 + v^2}}\, p(x, u\,x, v\,x)\, \mathrm{d}v\, \mathrm{d}u\, \mathrm{d}x \tag{18}$$

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.

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 10). The integral over each of these components is individually evaluated using the technique described above using Duffy transformation.

ACKNOWLEDGMENTS

This material is based upon work supported by AFOSR grants FA9550-12-10484; by NSF grant CCF-1337393; by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Applied Mathematics program under Award Numbers DE-SC0010518 and DE-SC0009286; by NIH grant 10042242; by DARPA grant W911NF-115-2-0121; and by the Technische Universität München—Institute for Advanced Study, funded by the German Excellence Initiative (and the European Union Seventh

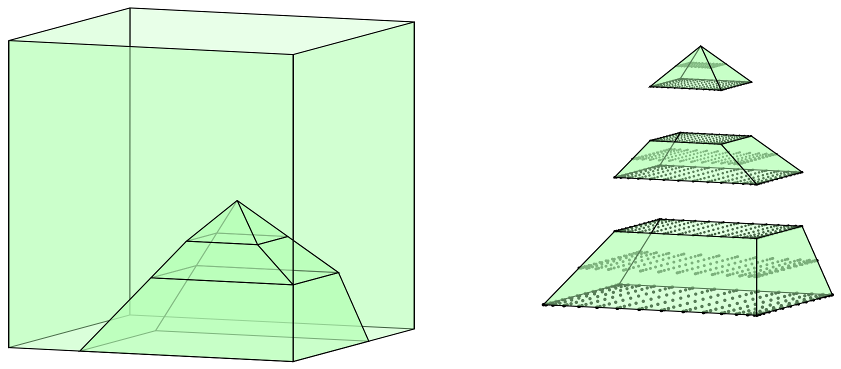


Fig. 10. Intersection of a regular pyramid (apex at r_0) with a cubic octant and decomposition into frustum stack and a smaller pyramid. We also show the node points for the Gauss-quadrature rule.

Framework Programme under grant agreement 291763). Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the AFOSR, DOE, NIH, DARPA, and NSF. Computing time on the Texas Advanced Computing Centers Stampede system was provided by an allocation from TACC and the NSF.

REFERENCES

Paul Bode and Jeremiah P Ostriker. 2008. Tree particle-mesh: an adaptive, efficient, and parallel code for collisionless cosmological simulation. *The Astrophysical Journal Supplement Series* 145, 1 (2008), 1.

Carsten Burstedde, Georg Stadler, Laura Alisic, Lucas C Wilcox, Eh Tan, Michael Gurnis, and Omar Ghattas. 2013. Large-scale adaptive mantle convection simulation. *Geophysical Journal International* 192, 3 (2013), 889–906.

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.

AJ Christlieb, R Krasny, ID Boyd, J Emhoff, and J Verboncoeur. 2005. Grid-Free Plasma Simulations. In *Plasma Science, 2005. ICOPS'05. IEEE Conference Record-Abstracts. IEEE International Conference on*. IEEE, 246–246.

David Colton and Rainer Kress. 1998. *Inverse Acoustic and Electromagnetic Scattering Theory, 2nd Edition*. Springer.

G.H. Cottet and P.D. Koumoutsakos. 2000. *Vortex methods: theory and practice*. Cambridge university press.

Andreas Dedner, Friedemann Kemm, Dietmar Kröner, C-D Munz, Thomas Schnitzer, and Matthias Wesenberg. 2002. Hyperbolic divergence cleaning for the MHD equations. *J. Comput. Phys.* 175, 2 (2002), 645–673.

M. Doi and S.F. Edwards. 1988. *The theory of polymer dynamics*. Vol. 73. Oxford University Press, USA.

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 Duffy. 1982. Quadrature over a pyramid or cube of integrands with a singularity at a vertex. *SIAM journal on Numerical Analysis* 19, 6 (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.

GI Fann, RJ Harrison, G Beylkin, J Jia, R Hartman-Baker, WA Shelton, and S Sugiki. 2007. MADNESS applied to density functional theory in chemistry and nuclear physics. In *Journal of Physics: Conference Series*, Vol. 78. IOP Publishing, 012018.

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

Max D. Gunzburger. 1989. *Finite Element for Viscous Incompressible Flows*. Academic Press.

Max D. Gunzburger and Roy A. Nicolaides. 1993. *Incompressible Computational Fluid Dynamics*. Cambridge University Press.

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.

Robert J Harrison, George I Fann, Takeshi Yanai, Zhengting Gan, and Gregory Beylkin. 2004. Multiresolution quantum chemistry: Basic theory and initial applications. *The Journal of chemical physics* 121, 23 (2004), 11587–11598.

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.

A.V. Kravtsov, A.A. Klypin, and A.M. Khokhlov. 1997. Adaptive refinement tree: a new high-resolution N-body code for cosmological simulations. *The Astrophysical Journal Supplement Series* 111 (1997), 73–94.

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.

Dhairya Malhotra and George Biros. 2015. PVFMM: A Parallel Kernel Independent FMM for Particle and Volume Potentials. *Communications in Computational Physics* 18, 03 (2015), 808–830.

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).

Matthias Messner, Bérenger Bramas, Olivier Coulaud, and Eric Darve. 2012. Optimized M2L kernels for the Chebyshev interpolation based fast multipole method. *arXiv preprint arXiv:1210.7292* (2012).

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.

Costas Pozrikidis. 2001. Interfacial Dynamics for Stokes Flow. *J. Comput. Phys.* 169 (2001), 250–301.

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.

V. Springel, N. Yoshida, and S.D.M. White. 2001. GADGET: a code for collisionless and gasdynamical cosmological simulations. *New Astronomy* 6, 2 (2001), 79–117.

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 Interna-*

tional 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.

Rio Yokota and Lorena A Barba. 2012. A tuned and scalable fast multipole method as a preeminent algorithm for exascale systems. *International Journal of High Performance Computing Applications* 26, 4 (2012), 337–346.

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).

Received August 2014; revised April 2015; accepted February 2016