Massively Parallelized Interpolated Factored Green Function Method

Christoph Bauinger* and Oscar P. Bruno*

Abstract

This paper presents the first parallel implementation of the novel "Interpolated Factored Green Function" (IFGF) method introduced recently for the accelerated evaluation of discrete integral operators arising in wave scattering and other areas (Bauinger and Bruno, Jour. Computat. Phys., 2021). On the basis of the hierarchical IFGF interpolation strategy, the proposed (hybrid MPI-OpenMP) parallel implementation results in efficient data communication, and it scales up to large numbers of cores—without any hard limitations on the number of cores efficiently employed. Moreover, on any given number of cores, the proposed parallel approach preserves the $\mathcal{O}(N \log N)$ computing cost inherent in the sequential version of the IFGF algorithm. Unlike other approaches, the IFGF method does not utilize the Fast Fourier Transform (FFT), and it is thus better suited for efficient parallelization in distributed-memory computer systems. In particular, the IFGF method relies on a "peer-to-peer" strategy wherein, at every level, field propagation is directly enacted via "exchanges" between "peer" polynomials of constant degree, without data accumulation in large-scale "telephone-central" mathematical constructs such as those used in the Fast Multipole Method (FMM) and pure FFT-based approaches. A variety of numerical results presented in this paper illustrate the character of the proposed parallel algorithm, in particular demonstrating scaling from 1 to all 1,680 cores available in the High Performance Computing cluster used, and for problems of up to 4,096 wavelengths in acoustic size.

Keywords: Parallelization, Green Function, Integral Equations, Acceleration, OpenMP, MPI, Distributed Memory Systems, High Performance Computing

1 Introduction

This paper presents a parallel implementation of the "Interpolated Factored Green Function" (IFGF) method introduced recently for the accelerated evaluation of discrete integral operators arising in wave scattering and other areas [1]. The proposed implementation, which is structured as a hybrid MPI-OpenMP computer program suitable for instantiation in modern high-performance computing systems (HPC), scales up to large numbers of cores without hard limitations on the number of cores efficiently employed, while preserving the linearithmic complexity (namely, $\mathcal{O}(N \log N)$ computing cost) inherent in the sequential IFGF algorithm. The IFGF method accelerates the evaluation of discrete integral operators by relying on a certain factorization of the Green function into two factors, a "centered factor" that is incorporated easily as a common factor in the calculation, and an "analytic factor" which enjoys a property of analyticity up to and including infinity—and which thus motivates the IFGF strategy, namely, evaluation of a discrete integral operators by means of a hierarchical interpolation approach which relies on use of a large number of small and independent interpolation procedures. In particular, the IFGF method does not utilize acceleration elements commonly used by other acceleration methods [2–14] such as the Fast Fourier Transform (FFT), special-function expansions, high-dimensional linear-algebra factorizations,

*Computing and Mathematical Sciences, Caltech, Pasadena, CA 91125, USA

translation operators, equivalent sources, or parabolic scaling; more details in these regards can be found in [1] and below in this section. Roughly speaking, the IFGF method relies on a "peer-to-peer" strategy wherein, at every level, field propagation is directly enacted via "exchanges" between "peer" polynomials of constant degree, without data accumulation in large-scale "telephone-central" mathematical constructs which require a "downward pass" through the box octree inherent in other methods [7], or the surface evaluation of equivalent sources in direct FFT-based methods [10]. (Note that a downward pass was avoided in the low-frequency implementation [15], to increase parallel efficiency at the expense of additional floating point operations, but we are not aware of any high-frequency FMM-based implementations that do not rely on use of a downward pass.) A variety of numerical results presented in this paper illustrate the character of the proposed parallel method, including its favorable weak and strong parallel scaling properties in all cases considered—for problems of up to $4,096$ wavelengths in electrical size, and scaling tests spanning from 1 compute core to all $1,680$ cores available in the HPC cluster used.

The parallelization of accelerated Green function methods has been the subject of a significant literature, which is mostly devoted to tackling a particular difficulty, namely, the "parallelization bottleneck"—which manifests itself under various related guises [16–24], and which almost invariably concerns uses of the hard-to-parallelize [25] FFT algorithm. (Reference [26, Sec. 7], for example, mentions two alternatives to the use of FFTs in the context of the FMM, which, however, it discards as less efficient than an FFT-based procedure.) In the case of the multilevel Fast Multipole Method (FMM), the parallelization bottleneck arises in the evaluation of translation operators associated with the upper part of the octree structure, which leads to low parallel efficiency [2,16,21]. In the "directional" FMM [16] the low efficiency in the upper octree is alleviated as a result of the parabolic scaling utilized; however, the parallelization strategy does suffer from hard limitations in the number of parallel tasks that, in the cases considered in that reference, lead to a "leveling off" of the parallel scaling at 256 or 512 cores [16, Secs. 3.6, 4.2], depending on the geometry under consideration. Reference [17] identifies the part of the FMM relying on FFTs as a parallelization bottleneck which arises from FFT-related "lowest arithmetic intensity" and "bandwidth contention". In references [18, 19], in turn, a hybrid octree storage strategy is used, which stores a complete set of tree nodes for a certain number of "full" levels in each process, and which reduces the communication in the upper octree levels. These articles demonstrate the treatment of problems containing very large numbers of discretization points on up to $2,560$ processes, but they restrict their illustration of the algorithm's parallel efficiency to a limited strong scaling test for a sphere, from 1 process (sequential) to 64 processes, for which an efficiency slightly above 70% is reported in [19]. In contrast to this hybrid octree-storage strategy, reference [20] simultaneously partitions boxes (clusters) and field values representing the radiating and incoming fields of each box. This approach leads to increased efficiency compared to a parallelization purely based on the boxes (clusters), but the communication in the translation step still poses a bottleneck, resulting in 30% parallel efficiency from one (sequential) core to 128 cores. Reference [27], finally, presents scaling results for the parallel BEMFMM implementation of the FMM algorithm, for wave scattering problems on a computer cluster containing 196,608 cores on 6,144 compute nodes. Like the implementations mentioned above, the results in [27] indicate a deterioration of the strong-scaling for growing numbers of cores, as manifested by a flattening of the strong-scaling speedup curves as the numbers of cores increase. Specific comparisons of IFGF and BEMFMM results, including comparisons of runs of the BEMFMM and IFGF software on our in-house computer system, are presented at the end of Section 4.7 and in Section 4.8.

Following a different approach, to avoid the communication bottleneck in the upper multilevel FMM octree entirely, references [21, 22] utilize a single-level Fast Multipole strategy. While this method significantly simplifies the algorithm and minimizes the required communication in a parallel setting, it does give rise to a sub-optimal asymptotic computational cost (e.g. $\mathcal{O}(N^{3/2})$ in [22] or, exploiting the FFT, $\mathcal{O}(N^{4/3}\log^{2/3}N)$ in [21]), and, while resulting in good parallel scaling up to 512 processes in the $\mathcal{O}(N^{3/2})$ algorithm [22], as in the case of [16], the parallel efficiency does level off beyond 512 processes. Direct FFT methods, in turn, present alternatives to the various FMM strategies, including, for example, the Adaptive

Integral Method [11] (AIM) and the sparse-FFT method [10]. Like the single-level FMM algorithms, these FFT methods exhibit sub-optimal algorithmic complexity (of orders $\mathcal{O}(N^{3/2})$ and $\mathcal{O}(N^{4/3})$, respectively), and, owing to their strong reliance on FFTs, they also suffer from reduced parallel efficiency, as shown and discussed for the AIM in e.g. [23, 24]. (A parallel version of the algorithm [10], which has been developed by the authors, has not been published, but we report here that, as may have been expected, the overall parallel efficiency of the method suffers from the typical FFT-related degradation.)

Finally, we consider parallel methods proposed for non-singular [28] and low-frequency [15, 29] problems which, albeit important and interesting, do not incur some of the main challenges associated with the singular and high-frequency kernels considered in this paper. We thus mention the Butterfly Method [28] which is not applicable to singular Green function kernels such as the ones considered here: it provides an acceleration technique for Fourier integral operators. The butterfly method, which is based on linear-algebra constructs instead of the hierarchical interpolation method underlying the IFGF algorithm, incorporates a parallelization strategy that is somewhat reminiscent of the proposed IFGF parallelization approach; its Blue Gene/Q implementation [28] demonstrates results of high quality in terms of parallel scaling to a large number of cores. The parallel FMM method presented in [29], in turn, which is restricted to box geometries and to the Laplace and low-frequency Helmholtz problems, shows scaling up to 299,008 cores on 18,688 nodes. Similarly, the parallel Barnes-Hut tree code [15] for the low-frequency singular problem provides high-quality scaling up to 294,912 cores with up to 2,048,000,000 particles.

The parallel IFGF strategy introduced in this contribution is based on adequate partitioning of the interpolations performed on each level of the underlying octree structure, which facilitates the spatial decomposition of the surface discretization points. As shown in [1], the number of interpolations performed on each level is large and approximately constant (as a function of the octree level). The decomposition and distribution of the interpolation data is based on a total order in the set of spherical cone segments representing the interpolation domains, which is an extension of a domain decomposition based on a space-filling Morton curve to the box-cone data structure inherent in the IFGF approach. The usage of space-filling curves for the representation of octree structures underlying the acceleration methods is not a novel concept [27, 29, 30]. However, the extension of space-filling curves to the box-cone structure of the IFGF method to achieve the desired efficiency has not been reported before. In view of its strong reliance on the IFGF's box-cone structure, the proposed parallelization strategy is therefore not applicable to other acceleration methods such as the FMM. The present parallel IFGF implementation on a 30-node (1,680-core) HPC cluster with Infiniband interconnect, delivers perfect $\mathcal{O}(N \log N)$ performance on all 1,680 cores. And, demonstrating high (albeit imperfect) strong parallel efficiencies, it does not suffer from scaling limitations as the number of processing cores grow.

This paper includes is organized as follows. Section 2 briefly summarizes the description [1] of the IFGF method, and it introduces the required notations and nomenclature. Section 3 then introduces the proposed OpenMP and MPI parallelization strategies for the IFGF method (Sections 3.1 and 3.2, respectively). A variety of numerical results are presented in Section 4. A few concluding comments, including a discussion of known limitations and areas for further improvement, finally, are presented in Section 5.

2 Review of the IFGF Method

As discussed above, the IFGF method provides an accelerated algorithm, requiring $\mathcal{O}(N \log N)$ operations, for the numerical evaluation of discrete integral operators of the form

$$I(x_\ell) \coloneqq \sum_{\substack{m=1 \\ m \neq \ell}}^{N} a_m G(x_\ell, x_m), \quad \ell = 1, \ldots, N, \tag{1}$$

for given points x_ℓ on a surface $\Gamma \subset \mathbb{R}^3$, and for given complex coefficients $a_m \in \mathbb{C}$, where the function $G(x, y)$, defined for $x, y \in \mathbb{R}^3$, denotes a Green function for some partial differential equation, such as the acoustic Green function

$$G(x, y) = \frac{e^{\iota\kappa|x-y|}}{4\pi|x-y|} \tag{2}$$

associated with the Helmholtz equation (ι denotes the imaginary unit and κ the wavenumber) as well as those associated with the Laplace, Stokes, and elasticity equations, among others. In what follows we denote by $\Gamma_N := \{x_1, \ldots, x_N\} \subset \Gamma$ the set of surface discretization points. For definiteness, throughout this paper we restrict attention to the challenging kernel (2), with possibly large values of κ, but other kernels can be treated analogously.

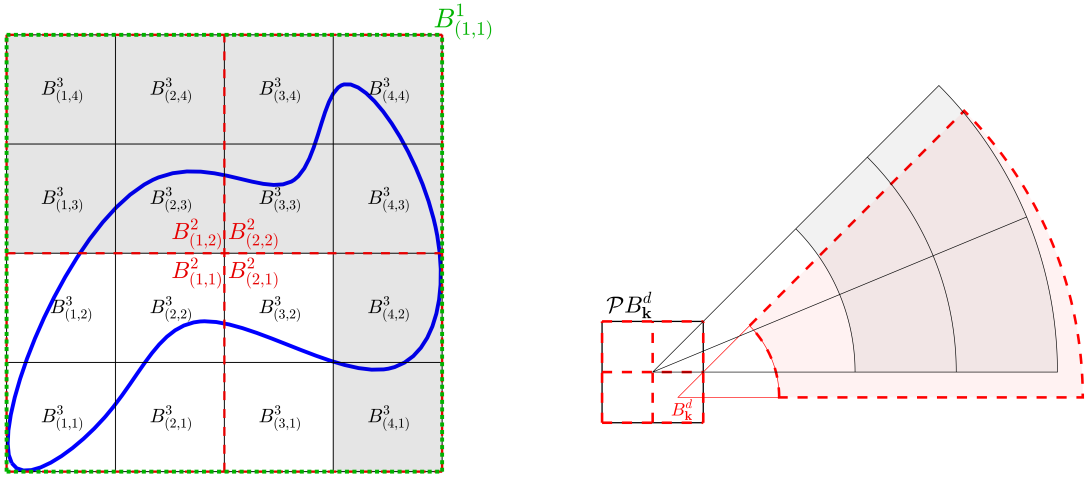
The strategy underlying the IFGF algorithm can be best appreciated by first considering an example of a "restricted" discrete operator

$$I_{\mathrm{R}}(x_p^T) \coloneqq \sum_{q=1}^{M^S} b_q G(x_p^T, x_q^S), \quad q = 1, \ldots, M^T, \tag{3}$$

where $\{x_q^S \ : \ q = 1, \ldots, M^S\} \subset \Gamma_N$ and $\{x_p^T \ : \ p = 1, \ldots, M^T\} \subset \Gamma_N$ are given (mutually disjoint) sets of "source points" and "target points", respectively. The IFGF approach seeks to reduce the cost of evaluation of the restricted operator (3) by relying on interpolation. For example, in order to avoid adding all of the terms in the sum (3) for each target point x_p^T, one might consider to approximate the restricted operator by resorting to evaluation of $I_{\mathrm{R}}(x)$ at a small number of adequately chosen interpolation points x and, subsequently, to evaluate I_{R} at the target points x_p^T via interpolation. A direct interpolation of $I_{\mathrm{R}}(x)$ is not viable, however, on account of the highly oscillatory and/or singular character of the kernels $G(x, y)$ under consideration—which are inherited by the restricted operator I_{R} itself. In order to address these difficulties, the IFGF method interpolates, instead, a modified form of the restricted operator, which is obtained by extraction of a certain common factor from the sum (3), and it considers restricted operators for certain specific sets of source and target points, as discussed in what follows.

The main enabling element in the IFGF method emerges from these considerations: for a group of "neighboring" sources, and a set target points that are adequately "separated" from the set of sources, factorization of the quantity $G(x_p^T, x_0) = \frac{e^{\iota\kappa|x_p^T - x_0|}}{4\pi|x_p^T - x_0|}$ significantly reduces both the oscillations and singularity of the sum, provided, 1) The point x_0 is selected to roughly coincide with some sort of centroid of the set of source points; and 2) The set of target points is adequately separated from the set of source points. For example, reference [1] shows that for a set of source points contained within an axis-aligned cubic box $B(x_0, H)$ of side H and centered at x_0, and for all target points that are at least "one box away" (or, more precisely, outside the concentric box $B(x_0, 3H)$ of side $3H$), the quotient $G(x, x_q^S)/G(x, x_0)$ for each q, and therefore, the complete restricted sum $I_{\mathrm{R}}(x)/G(x, x_0)$, are slowly-oscillatory and non-singular. More precisely, as shown in [1], these functions are analytic up to and including $|x - x_0| = \infty$, and their n-th order derivatives are uniformly bounded by a constant times $(\kappa H)^n$ for all x outside $B(x_0, 3H)$. Thus, with these constraints, the quotient functions considered are "slowly varying", and they may therefore be interpolated, with a fixed error, by means of a number of interpolation points that, for interpolation onto a surface, depends only quadratically on the size κH of the box $B(x_0, H)$. Thus, as shown in detail in [1], for example, if the box size is doubled, the number of interpolation points required to meet a prescribed error tolerance is quadrupled. But, as the box sizes are doubled, the number of boxes covering Γ_N is reduced by a factor of four—so that box-size doubling does not increase the total number of interpolation points used across the surface.

The interpolation procedure is implemented via piecewise Chebyshev interpolants in a certain (s, θ, φ) spherical coordinate system around each box center x_0 (where using the radius $r = |x - x_0|$ and box half-diameter $h = \frac{\sqrt{3}}{2}$, the variable $s = h/r$ is used to exploit the analytic character of the slowly varying



(a) Two-dimensional sketch of a three level ($D = 3$) IFGF domain decomposition with neighbors (in white) and cousin boxes (in gray) for the particular box $B^3_{(2,1)}$. A surrogate scatterer is sketched in blue.

(b) Two-dimensional illustration of cone segments: in red, cone-segments co-centered with the box $B^d_{\mathbf{k}}$, and, in black, cone-segments co-centered with the parent box $\mathcal{P}B^d_{\mathbf{k}}$.

Figure 1: Illustration of the box and cone hierarchical structures used in the IFGF method.

quotients $I_{\mathrm{R}}(x)/G(x, x_0)$). To achieve an overall $\mathcal{O}(N \log N)$ complexity, the interpolation from any given box $B(x_0, H)$ is restricted to certain set "cousin" target points, that are outside $B(x_0, 3H)$, but "not too far" from it. Necessary interpolations to non-cousin points is enacted, recursively, by commingling smaller "child" boxes into larger "parent boxes", and obtaining necessary parent-box interpolation data from the available child-box interpolation data.

In detail, for a given $D \in \mathbb{N}$, the IFGF method is based upon use of a D-level octree hierarchical decomposition of a cube $B^1_{1,1,1}$ containing the discrete surface Γ_N, where each level is determined by a set of axis-aligned boxes $B^d_{\mathbf{k}} \subset \mathbb{R}^3$ (defined as the Cartesian product of three one-dimensional half-open intervals of the form $[a, a + H_d)$ for some $a \in \mathbb{R}$), where $\mathbf{k} \in \mathbb{N}^3$ denotes a multi-index describing the three-dimensional position of the box in the resulting Cartesian grid of boxes, and where d ($1 \leq d \leq D$) denotes the level in the octree. The octree structure of boxes is defined iteratively starting from a single box $B^1_{1,1,1} \supset \Gamma_N$ of side $H_1 \in \mathbb{R}$, $H_1 > 0$, on level $d = 1$. (Note that there is no undue expense incurred for, say, an elongated surface Γ, for which a cubic box could be mostly empty—since, as indicated in what follows, only certain "relevant" child boxes in the box octree are used by the algorithm.) The boxes on consecutive levels $d = 2, \ldots, D$ are defined by means of a partition of each of the level $(d-1)$ boxes into eight equi-sized and disjoint boxes of side $H_d = H_{d-1}/2$ resulting in the level d boxes $B^d_{\mathbf{k}}$ ($\mathbf{k} \in \{1, \ldots, 2^{d-1}\}^3 =: I^d_B$). We note that, in particular, for each d, $1 \leq d \leq D$, we have

$$\Gamma_N = \bigcup_{\mathbf{k} \in I^d_B} (B^d_{\mathbf{k}} \cap \Gamma_N). \tag{4}$$

The two-dimensional equivalent of the resulting hierarchical octree structure for an illustrative three-level configuration ($D = 3$) is depicted in Figure 1a. Clearly, each box $B^d_{\mathbf{k}}$ on level d ($2 \leq d \leq D$) admits a unique $(d-1)$-level parent box $\mathcal{P}B^d_{\mathbf{k}}$ containing $B^d_{\mathbf{k}}$.

To achieve the desired acceleration, the IFGF method only considers interactions between boxes in a certain set $\mathcal{R}_B$ of *relevant boxes*, which are defined as the boxes in the octree structure that contain at

least one surface discretization point:

$$\mathcal{R}_B^d := \{B_{\mathbf{k}}^d \ : \ \Gamma_N \cap B_{\mathbf{k}}^d \neq \emptyset, \mathbf{k} \in I_B^d\}, \quad d = 1, \ldots, D,$$
$$\mathcal{R}_B := \bigcup_{d=1,\ldots,D} \mathcal{R}_B^d.$$

Furthermore, for a given box $B_{\mathbf{k}}^d$ on any level d, the method relies on a number of additional concepts, such as the *set of neighboring boxes* $\mathcal{N}B_{\mathbf{k}}^d$ (namely the set of level-d boxes whose closures have a non-empty intersection with the closure $\overline{B_{\mathbf{k}}^d}$ of $B_{\mathbf{k}}^d$) and the *set of cousin boxes* $\mathcal{M}B_{\mathbf{k}}^d$ (non-neighboring boxes that are children of the parents neighbors), as well as related concepts such as the *set of neighboring points* $\mathcal{U}B_{\mathbf{k}}^d$ and the *set of cousin points* $\mathcal{V}B_{\mathbf{k}}^d$ (which denote the set of surface discretization points within the neighboring boxes and the cousin boxes, respectively):

$$\begin{aligned}
\mathcal{N}B_{\mathbf{k}}^d &:= \{B_{\mathbf{j}}^d \in \mathcal{R}_B : ||\mathbf{j} - \mathbf{k}||_\infty \leq 1\}, \\
\mathcal{M}B_{\mathbf{k}}^d &:= \{B_{\mathbf{j}}^d \in \mathcal{R}_B : B_{\mathbf{j}}^d \notin \mathcal{N}B_{\mathbf{k}}^d \wedge \mathcal{P}B_{\mathbf{j}}^d \in \mathcal{N}\mathcal{P}B_{\mathbf{k}}^d\}, \\
\mathcal{U}B_{\mathbf{k}}^d &:= \left(\bigcup_{B \in \mathcal{N}B_{\mathbf{k}}^d} B \right) \cap \Gamma_N, \\
\mathcal{V}B_{\mathbf{k}}^d &:= \left(\bigcup_{B \in \mathcal{M}B_{\mathbf{k}}^d} B \right) \cap \Gamma_N.
\end{aligned} \tag{5}$$

Figure 1a displays the neighbors and cousins of the box $B_{(2,1)}^3$ in white and gray colors, respectively.

The IFGF algorithm accelerates the evaluation of the operator (1) by evaluating pairwise interactions between cousin boxes on every level d, for $d = D, \ldots, 3$. (Note that the algorithm ends at level $d = 3$—since at $d = 3$, each box is a cousin or a neighbor of all the other boxes, and thus, all remaining surface evaluations are completed at this stage.) The evaluation of these interactions is enacted by means of a simple piecewise interpolation method based on a certain factored form of the Green function in a set of box-centered spherical coordinate systems, with one such spherical-coordinate system centered at each one of the relevant boxes. The use of angular and radial interpolation methods gives rise to so-called *cone segments* $C_{\mathbf{k};\gamma}^d$: one for each box $B_{\mathbf{k}}^d$ and for each multi-index $\gamma \in I_C^d \subset \mathbb{N}^3$ characterizing a conical interpolation domain. A set $\mathcal{X}C_{\mathbf{k};\gamma}^d$ of $P \in \mathbb{N}$ *interpolation points* is used within each cone segment $C_{\mathbf{k};\gamma}^d$, where, letting P_{s} and P_{ang} denote the number of interpolation points in the radial s variable and each one of the angular variables θ and φ, $P = P_{\mathrm{ang}}^2 P_{\mathrm{s}}$ is an arbitrary but fixed number throughout the algorithm. In what follows, cone segments $C_{\mathbf{k};\gamma}^d$ are called *co-centered* with a box $B_{\mathbf{k}}^d$ if and only if the origin of the spherical coordinate system defining the cone segment coincides with the center of the box. Two cone segments are called *co-centered* if they are co-centered with the same box. Note that the sub- and super indices $\mathbf{k}$ and d in the cone-segment notation $C_{\mathbf{k};\gamma}^d$ coincide with the corresponding indices of the co-centered box.

To achieve competitive computation times, a certain factorization of the Green function $G(x, x') = G(x, x_{\mathbf{k}}^d) g_{\mathbf{k}}^d(x, x')$ into a *centered* factor $G(x, x_{\mathbf{k}}^d)$ (centered at the *box-center* $x_{\mathbf{k}}^d$ of the box $B_{\mathbf{k}}^d$) and an *analytic* factor $g_{\mathbf{k}}^d(x, x')$, is used. The field $I(x)$ in (1) can be expressed, for each level d, as the sum, over all multi-indices $\mathbf{k} \in I_B^d$, of fields $I_{\mathbf{k}}^d(x)$ equal to the sum of $G(x, x')$ for all surface discretization points x' within the box $B_{\mathbf{k}}^d$, i.e., for all $x' \in B_{\mathbf{k}}^d \cap \Gamma_N$. Using the aforementioned factorization centered at $x_{\mathbf{k}}^d$ yields

$$I_{\mathbf{k}}^d(x) = \sum_{x' \in B_{\mathbf{k}}^d \cap \Gamma_N} a(x') G(x, x') = G(x, x_{\mathbf{k}}^d) F_{\mathbf{k}}^d(x), \qquad F_{\mathbf{k}}^d(x) := \sum_{x' \in B_{\mathbf{k}}^d \cap \Gamma_N} a(x') g_{\mathbf{k}}^d(x, x'), \tag{6}$$

where $a(x')$ denotes the coefficient a_m in (1) that corresponds to the point $x' \in \Gamma_N$. The IFGF interpolation procedure is then used to evaluate $F_{\mathbf{k}}^d$. The generation of the P coefficients of each one of the degree-P

polynomial interpolants $I_P C^d_{\mathbf{k};\gamma}$, corresponding to interpolation of the field $F^d_{\mathbf{k}}$ (cf. (6)) over the cone segment $C^d_{\mathbf{k};\gamma}$, is achieved on the basis of the field values $F^d_{\mathbf{k}}(\mathcal{X}C^d_{\mathbf{k};\gamma}) := \{F^d_{\mathbf{k}}(x) : x \in \mathcal{X}C^d_{\mathbf{k};\gamma}\}$.

In [1] it is shown that the analytic factor is analytic everywhere in $\mathbb{R}^3 \setminus \mathcal{N}B^d_{\mathbf{k}}$ and up to and including infinity, and it can therefore be interpolated accurately throughout that region on the basis of a small (finite!) number of interpolation points. (It is is easy to check that the same is true for most of the relevant kernels arising in applications.) Since $F^d_{\mathbf{k}}$ equals a linear combination of finitely many analytic-factor functions, it is clear that this function shares the same analytic properties, and it can therefore be interpolated with equal quality and efficiency. The cone segments $C^d_{\mathbf{k};\gamma}$ are defined by means of an iterative procedure similar to the one used in the definition of the boxes $B^d_{\mathbf{k}}$, but in reversed order, starting from $d = D$ and moving upwards the tree to $d = 3$. The set of cone segments that is to be used at a given level d depends strongly on the character of the surface Γ_N, the wavenumber κ and, possibly, the Green function G. A two-dimensional sketch of some illustrative box-centered cone segments for a given box $B^d_{\mathbf{k}}$ and its parent $\mathcal{P}B^d_{\mathbf{k}}$ is provided in Figure 1b.

In order to evaluate the discrete operator (1) in $\mathcal{O}(N \log N)$ operations, the IFGF algorithm uses iterated interpolation, as illustrated in Figure 2, to evaluate the analytic factor at the interpolation points of consecutive levels—thus avoiding the cost of directly evaluating the field $I^d_{\mathbf{k}}(x)$ on levels $(d-1), (d-2), \ldots, 3$, and using instead the interpolation data on level d to generate the necessary interpolation data at on the consecutive level $(d-1)$. It is important to note that, in order to further increase the efficiency and achieve the desired $\mathcal{O}(N \log N)$ complexity, in analogy to the approach used for boxes, the IFGF method only utilizes the set of *relevant cone segments* $\mathcal{R}_C B^d_{\mathbf{k}}$ for each box $B^d_{\mathbf{k}}$, namely, the cone segments that are actually needed for interpolation back to cousin surface discretization points or to relevant cone segments on the parent level. The relevant cone segments $\mathcal{R}_C B^d_{\mathbf{k}}$ are thus defined by

$$\begin{aligned}
&\mathcal{R}_C B^d_{\mathbf{k}} := \emptyset \quad \text{for} \quad d = 1, 2, \\
&\mathcal{R}_C B^d_{\mathbf{k}} := \left\{ C^d_{\mathbf{k};\gamma} \,:\, \gamma \in I^d_C \,,\, C^d_{\mathbf{k};\gamma} \cap \mathcal{V}B^d_{\mathbf{k}} \neq \emptyset \text{ or } C^d_{\mathbf{k};\gamma} \cap \left(\bigcup_{C \in \mathcal{R}_C \mathcal{P} B^d_{\mathbf{k}}} C \right) \neq \emptyset \right\} \quad \text{for} \quad d \geq 3.
\end{aligned} \tag{7}$$

The serial IFGF algorithm, introduced in [1], is summarized in Figure 2 and described in what follows. Starting from the given coefficients in equation (1) at the bottom of Figure 2, the IFGF algorithm first performs "LevelDSingularInteractions" (which, while required for the full evaluation of (1), are not, strictly speaking, a part of the IFGF strategy, and would, in the context of a scattering solver, be substituted by an appropriate local integration scheme; see e.g. [31]). The "LevelDSingularInteractions" stage evaluates the field $I^D_{\mathbf{k}}$ at all surface discretization points x in all the neighbor boxes of each box $B^D_{\mathbf{k}}$ (i.e. at all $x \in \mathcal{U}B^D_{\mathbf{k}}$ for all relevant boxes $B^D_{\mathbf{k}}$). Next, the algorithm performs "LevelDEvaluations", that is, it first evaluates the field $F^D_{\mathbf{k}}(x)$ (see (6)) at every interpolation point in the relevant cone segments co-centered with the box $B^D_{\mathbf{k}}$, and it subsequently generates the necessary level-D interpolants. The IFGF algorithm then proceeds through levels $d = D, \ldots, 3$ by performing, on each level d, 1) Interpolations to cousin surface discretization points in the "Interpolation" stage, as well as, 2) Interpolations to level $d-1$ interpolation points, and subsequent generation of the interpolants on level $d-1$, in the "Propagation" stage, just as in the "LevelDEvaluations" stage, but utilizing the interpolants instead of direct field evaluations.

The corresponding pseudo-code is presented in Algorithm 1. Note that this algorithm does not include evaluations of interactions between neighboring boxes on the lowest level D ("LevelDSingularInteractions" in Figure 2), which would generally be produced by means of a separate algorithm, as mentioned above in this section.

3 Parallel IFGF Method

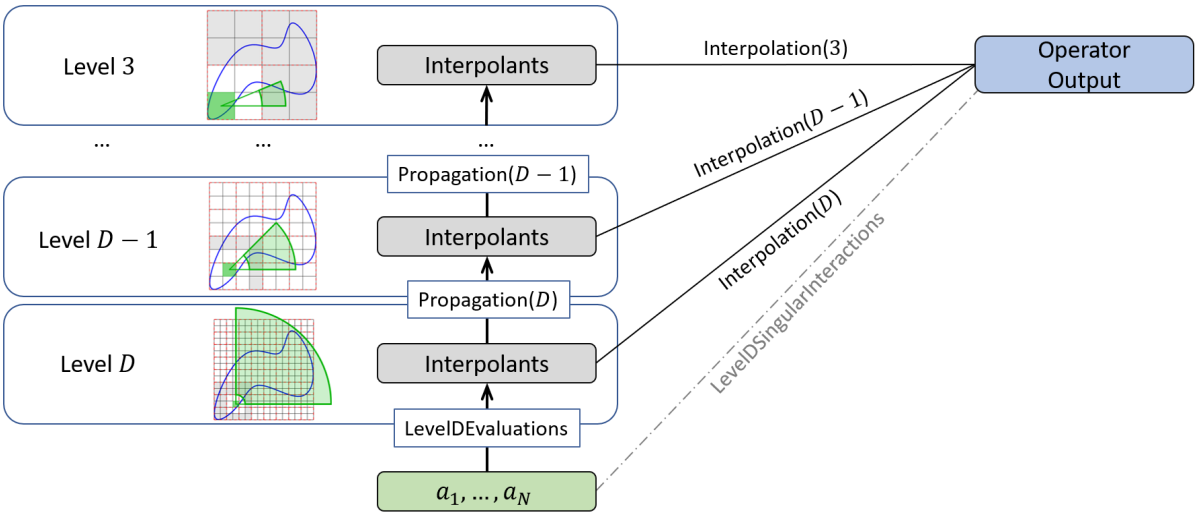


Figure 2: Visual representation of the IFGF algorithm, outlined in Algorithm 1, and also expressed in Algorithm 5 in terms of three fundamental functions called *LevelDEvaluations*, *Propagation* and *Interpolation*. Starting from the given coefficients $a_1, \ldots, a_N$ in equation (1), the *LevelDEvaluations* function generates the first set of interpolants on level D. The *Interpolation* function interpolates to the surface discretization points and the *Propagation* function facilitates the upwards traversal of the octree structure. Although they are not part of the IFGF algorithm, the interactions between level-D neighbor boxes are represented here by the *LevelDSingularInteractions* function. Note that, unlike other acceleration methods such as the FMM, contributions to the operator output are made at every level, and without a requirement of a downward pass over the octree.

The IFGF parallelization scheme proposed in this paper relies on use of a hybrid MPI-OpenMP approach. The OpenMP parallelization method, which is described in Section 3.1, is used to distribute the work assigned to each MPI rank. Hence, in the specific 56-core-per-node hardware implementation demonstrated in this paper, four intra-node MPI ranks are used per compute node, each one of which spawns fourteen OpenMP threads—which, according to our experiments, leads to the best performance achievable without the adverse impact (concerning e.g. code complexity, memory requirements, and communications) entailed in pure MPI parallelism within each node or other intra-node hybrid OpenMP/MPI schemes. Section 4.4 presents results of our investigation of MPI vs OpenMP performance within each computing node, leading to the aforementioned arrangement of intra-node MPI ranks. A general discussion on the performance of hybrid MPI-OpenMP approaches can be found in [32–34].

3.1 IFGF OpenMP parallelization

The proposed strategy proceeds via parallelization of the three independent programming functions that comprise the IFGF method, namely the *LevelDEvaluations* function, the *Interpolation* function and the *Propagation* function, as mentioned in Section 2 and illustrated in Figure 2. The first of these functions, the *LevelDEvaluations* function, which corresponds to the loop in line 2 of Algorithm 1, evaluates, for each relevant level-D box, the field generated by the point sources within the box (given by (6) with $d = D$) at the interpolation points in all relevant cone segments co-centered with the box and generates the required interpolants. The second function, the level-d-dependent *Interpolation* function, which corresponds to line 14 under the loops in lines 11 and 12, and which is represented in Figure 2 by rightward lines connecting various levels to the "Operator Output", performs the necessary interpolations to cousin surface discretization points on level d ($d = 3, \ldots, D$). The third and final programming function, the level-d-dependent

Algorithm 1 IFGF Method

1: \\Direct evaluations on the lowest level.
2: **for** $B_{\mathbf{k}}^D \in \mathcal{R}_B$ **do**
3: **for** $C_{\mathbf{k};\gamma}^D \in \mathcal{R}_C B_{\mathbf{k}}^D$ **do** ▷ Evaluate F at all relevant interpolation points
4: Evaluate and store $F_{\mathbf{k}}^D(\mathcal{X}C_{\mathbf{k};\gamma}^D)$
5: Generate interpolant $I_P C_{\mathbf{k};\gamma}^D$
6: **end for**
7: **end for**
8:
9: \\Interpolation onto surface discretization points and parent interpolation points.
10: **for** $d = D, \ldots, 3$ **do**
11: **for** $B_{\mathbf{k}}^d \in \mathcal{R}_B$ **do**
12: **for** $x \in \mathcal{V}B_{\mathbf{k}}^d$ **do** ▷ Interpolate at cousin surface points
13: Determine $C_{\mathbf{k};\alpha}^d$ s.t. $x \in C_{\mathbf{k};\alpha}^d$
14: Evaluate and add to result $I_P C_{\mathbf{k};\alpha}^d(x) \times G(x, x_{\mathbf{k}}^d)$
15: **end for**
16: **if** $d > 3$ **then** ▷ Evaluate F on parent interpolation points
17: Determine parent $B_{\mathbf{j}}^{d-1} = \mathcal{P}B_{\mathbf{k}}^d$
18: **for** $C_{\mathbf{j};\gamma}^{d-1} \in \mathcal{R}_C B_{\mathbf{j}}^{d-1}$ **do**
19: **for** $x \in \mathcal{X}C_{\mathbf{j};\gamma}^{d-1}$ **do**
20: Determine $C_{\mathbf{k};\alpha}^d$ s.t. $x \in C_{\mathbf{k};\alpha}^d$
21: Evaluate and add $I_P C_{\mathbf{k};\alpha}^d(x) \times G(x, x_{\mathbf{k}}^d)/G(x, x_{\mathbf{j}}^{d-1})$
22: **end for**
23: **end for**
24: **end if**
25: **end for** ▷ Generate interpolants on parent level
26: **for** $B_{\mathbf{j}}^{d-1} \in \mathcal{R}_B$ **do**
27: **for** $C_{\mathbf{j};\gamma}^{d-1} \in \mathcal{R}_C B_{\mathbf{j}}^{d-1}$ **do**
28: Generate interpolant $I_P C_{\mathbf{j};\gamma}^{d-1}$
29: **end for**
30: **end for**
31: **end for**

Propagation function, which corresponds to line 21 under the loops in lines 11, 18, and 19 and is represented in Figure 2 by means of upward pointing arrows targeting the "Interpolant" boxes, interpolates, for each relevant level-d box, to interpolation points in the relevant cone segments co-centered with the parent box on level $(d-1)$ and generates the required interpolants. These three functions are outlined in Algorithms 2, 3, and 4, respectively. Using these functions, the IFGF algorithm (Algorithm 1) may be re-expressed as Algorithm 5. In what follows, we present our strategies for efficient parallelization of each one of these functions separately.

Our approach to efficient parallelization of the *LevelDEvaluations* function is based on changing the viewpoint from iterating through the level-D relevant boxes to iterating through the set $\mathcal{R}_C^D$ of *all relevant cone segments on level* D. Since corresponding sets of level-d relevant cone segments for the wider range $3 \leq d \leq D$ are utilized in the parallelization of the *Propagation* function, we generalize the definition: the set of *all relevant cone segments on level* d is denoted by $\mathcal{R}_C^d$, that is

$$\mathcal{R}_C^d := \bigcup_{\mathbf{k} \in I_B^d : B_{\mathbf{k}}^d \in \mathcal{R}_B} \mathcal{R}_C B_{\mathbf{k}}^d, \quad \text{for } 3 \leq d \leq D. \tag{8}$$

Using (8), a parallel version of Algorithm 2 is presented in Algorithm 6. The aforementioned change in viewpoint corresponds to collapsing the two outermost nested loops in Algorithm 2, effectively increasing

Algorithm 2 LevelDEvaluations

1: **for** $B_{\mathbf{k}}^D \in \mathcal{R}_B$ **do**
2: **for** $C_{\mathbf{k};\gamma}^D \in \mathcal{R}_C B_{\mathbf{k}}^D$ **do**
3: Evaluate and store $F_{\mathbf{k}}^D(\mathcal{X} C_{\mathbf{k};\gamma}^D)$
4: Generate interpolant $I_P C_{\mathbf{k};\gamma}^D$
5: **end for**
6: **end for**

Algorithm 3 Interpolation(d)

1: **for** $B_{\mathbf{k}}^d \in \mathcal{R}_B$ **do**
2: **for** $x \in \mathcal{V} B_{\mathbf{k}}^d$ **do**
3: Determine $C_{\mathbf{k};\alpha}^d$ s.t. $x \in C_{\mathbf{k};\alpha}^d$
4: Evaluate and add to result $I_P C_{\mathbf{k};\alpha}^d(x) \times G(x, x_{\mathbf{k}}^d)$
5: **end for**
6: **end for**

the number of independent tasks and, consequently, the achievable parallelism.

The proposed parallelization of the d-dependent *Propagation* function follows a similar idea as the parallel *LevelDEvaluations* considered above—relying now on iteration over the relevant $(d-1)$ (parent-level) cone segments, which are targets of the interpolation, instead of the relevant level-d boxes emitting the field. In the context of the oscillatory Green functions over two-dimensional surfaces $\Gamma \subset \mathbb{R}^3$ considered in this paper, for example, the IFGF strategy provides an approximately constant number of relevant cone segments on each level d—cf. Section 2 and [1, Sec. 3.3.3]—which the proposed parallelization of the *Propagation* function exploits, resulting in an approximately constant number of independent parallel task on each level. Additionally, the proposed parallel *Propagation* strategy avoids a significant "thread-safety" [35, 36] predicament, that is ubiquitous in the straightforward approach, whereby multiple writes to the same target interpolation point on the parent level take place from different threads. In contrast, the proposed *Propagation* strategy, is by design thread-safe without any additional considerations, since it distributes the targets of the interpolation to the available threads. Note that the practical implementation of this approach requires the algorithm to first determine the relevant box

$$\mathcal{R}_B C_{\mathbf{k};\gamma}^d := B_{\mathbf{k}}^d \qquad \text{s.t.} \quad C_{\mathbf{k};\gamma}^d \in \mathcal{R}_C B_{\mathbf{k}}^d \tag{9}$$

that is co-centered with a given relevant cone segment $C_{\mathbf{k};\gamma}^d$; then to determine the relevant level-$(d+1)$ *child boxes*

$$\mathcal{C} B_{\mathbf{k}}^d := \left\{ B_{\mathbf{j}}^{d+1} \in \mathcal{R}_B : \mathbf{j} \in I_B^{d+1}, \mathcal{P} B_{\mathbf{j}}^{d+1} = B_{\mathbf{k}}^d \right\}, \tag{10}$$

of a given relevant box $B_{\mathbf{k}}^d$ on level d; and, finally, to find all the interpolants $I_P C_{\mathbf{k},\gamma}^d$ on the relevant cone segments (7) co-centered with the child boxes from which the propagation needs to be enacted. Using this notation, the resulting *Parallel Propagation* algorithm is presented in Algorithm 7.

The proposed parallelization strategy for the third and final IFGF programming function, namely, the *Interpolation* function, relies on the strategy used for the *LevelDEvaluations* and *Propagation* functions—which, in the present case, leads to iterating through the surface discretization points that are the target of the interpolation procedure. This approach once again results in an approximately constant number of independent parallel tasks on each level, and it avoids thread-safety difficulties similar to those discussed above in the context of the *Propagation* function. For a concise description of the parallel *Interpolation* function in what follows we denote by

$$\mathcal{M}^d(x) := \left\{ B_{\mathbf{k}}^d \in \mathcal{R}_B : x \in \mathcal{V} B_{\mathbf{k}}^d \right\}, \tag{11}$$

Algorithm 4 Propagation(d)

for $B_{\mathbf{k}}^{d} \in \mathcal{R}_B$ **do**
  Determine parent $B_{\mathbf{j}}^{d-1} = \mathcal{P}B_{\mathbf{k}}^{d}$
  for $C_{\mathbf{j};\gamma}^{d-1} \in \mathcal{R}_C B_{\mathbf{j}}^{d-1}$ **do**
    for $x \in \mathcal{X}C_{\mathbf{j};\gamma}^{d-1}$ **do**
      Determine $C_{\mathbf{k};\alpha}^{d}$ s.t. $x \in C_{\mathbf{k};\alpha}^{d}$
      Evaluate and add $I_P C_{\mathbf{k};\alpha}^{d}(x) \times G(x, x_{\mathbf{k}}^{d})/G(x, x_{\mathbf{j}}^{d-1})$
    end for
  end for
end for
for $B_{\mathbf{j}}^{d-1} \in \mathcal{R}_B$ **do**
  for $C_{\mathbf{j};\gamma}^{d-1} \in \mathcal{R}_C B_{\mathbf{j}}^{d-1}$ **do**
    Generate interpolant $I_P C_{\mathbf{j};\gamma}^{d-1}$
  end for
end for

Algorithm 5 IFGF Method

LevelDEvaluations()

for $d = D, \ldots, 3$ **do**
  Interpolation(d)
  if $d > 3$ **then**
    Propagation(d)
  end if
end for

the set of *cousin boxes of a surface discretization point* $x \in \Gamma_N$ *on level* d, $3 \leq d \leq D$, which extends the concept of cousin boxes of a box, introduced in equation (5). Using the definition (11), the *Parallel Interpolation* function is presented in Algorithm 8.

In summary, the OpenMP parallel functions *Parallel LevelDEvaluations*, *Parallel Propagation* and *Parallel Interpolation* introduced in this section are thread-safe by design, and they provide effective work distribution by relying on iteration over items (relevant cone segments or surface discretization points) that exist in a sufficiently large (and essentially constant) quantities for all levels d, $3 \leq d \leq D$, in the box-octree structure.

3.2 IFGF MPI parallelization

The proposed MPI parallel IFGF algorithm, which enables both data distribution onto the MPI ranks and efficient communication of data between MPI ranks, is described in detail in Sections 3.2.1 through 3.2.2. The approach mirrors the one proposed in Section 3.1 for the corresponding OpenMP interface. In fact, the MPI parallel scheme is based on slight modifications of the OpenMP parallel Algorithms 6, 7, and 8. As indicated by the theoretical discussion in Section 3.2.2, the communication overhead is such that the intrinsic IFGF linearithmic complexity previously demonstrated in [1] for a single core implementation is

Algorithm 6 Parallel LevelDEvaluations

parallel for $C_{\mathbf{k};\gamma}^{D} \in \mathcal{R}_C^{D}$ **do**
  Evaluate and store $F_{\mathbf{k}}^{D}(\mathcal{X}C_{\mathbf{k};\gamma}^{D})$
  Generate interpolant $I_P C_{\mathbf{k};\gamma}^{D}$
end parallel for

Algorithm 7 Parallel Propagation(d)

parallel for $C^{d-1}_{\mathbf{j};\gamma} \in \mathcal{R}^{d-1}_C$ **do**
  for $B^d_{\mathbf{k}} \in \mathcal{C}(\mathcal{R}_B C^{d-1}_{\mathbf{j};\gamma})$ **do**
    for $x \in \mathcal{X}C^{d-1}_{\mathbf{j};\gamma}$ **do**
      Determine $C^d_{\mathbf{k};\alpha}$ s.t. $x \in C^d_{\mathbf{k};\alpha}$
      Evaluate and add $I_P C^d_{\mathbf{k};\alpha}(x) \times G(x, x^d_{\mathbf{k}})/G(x, x^{d-1}_{\mathbf{j}})$
    end for
  end for
  Generate interpolant $I_P C^{d-1}_{\mathbf{j};\gamma}$
end parallel for

Algorithm 8 Parallel Interpolation(d)

parallel for $x \in \Gamma_N$ **do**
  for $B^d_{\mathbf{k}} \in \mathcal{M}^d(x)$ **do**
    Determine $C^d_{\mathbf{k};\gamma}$ s.t. $x \in C^d_{\mathbf{k};\gamma}$
    Evaluate $I_P C^d_{\mathbf{k};\gamma}(x) \times G(x, x^d_{\mathbf{k}})$
  end for
end parallel for

preserved on any fixed number N_c of cores; an illustration of this theoretical result on $N_c = 1,680$ cores is presented in Table 1. Most importantly, as in the OpenMP case (cf. the last paragraph of Section 3.1), for arbitrarily large numbers D of levels, the MPI IFGF algorithm iterates over items (relevant cone segments or surface discretization points) that exist in a sufficiently large (and essentially constant) quantities for all levels d, $3 \leq d \leq D$, in the box-octree structure. The proposed parallelization strategy thus results in an overall MPI-OpenMP IFGF parallel scheme without hard limitations on the achievable parallelism as the number of cores grows.

3.2.1 Problem decomposition and MPI data distribution

The proposed overall MPI data distribution strategy follows directly from consideration of 1) The partition induced on the point-set Γ_N by the leaves of the octree structure, namely, the partition (4) with $d = D$, as well as, 2) Associated partitions of the level-d sets $\{\mathcal{X}C : C \in \mathcal{R}^d_C\}$ of level-d interpolation points. Clearly, for an efficient parallel implementation, the distribution used should balance the amount of work performed by each rank while maintaining a minimal memory footprint per rank, while also minimizing the communication between ranks. A concise description of the method used for data distribution to the MPI ranks is presented in what follows, where we let $N_r \in \mathbb{N}$ and $\rho \in \mathbb{N}$ ($1 \leq \rho \leq N_r$) denote the number of MPI ranks and the index of a specific MPI rank, respectively.

The distribution of the surface discretization points is orchestrated on the basis of an ordering of the set of relevant boxes $\mathcal{R}^d_B$ on each level d, which, in the proposed algorithm, is obtained from a depth-first traversal of the octree structure. This ordering is equivalent to a Morton order of the boxes (as described e.g. in [27, 29, 30, 37] and depicted by the red Σ-looking curve in the left panel of Figure 3) which, as indicated in [37], can be generated quickly from the positions $\mathbf{k} \in I^D_B$ of the level-D boxes $B^D_{\mathbf{k}}$ through a bit-interleaving procedure. At every level d the Morton order introduces a total order $\prec$ on the set of boxes. The surface discretization points Γ_N are then ordered according to the Morton order of their containing level-D box. The resulting overall order has the desirable properties that, on every level d, surface discretization points within any given box are contiguous in memory, and that boxes close in real space are also close in memory. The sorted surface discretization points are distributed to the MPI ranks based on their containing level-D boxes, in such a way that the boxes processed by each rank are an "interval" set of the form $\{B \in \mathcal{R}^D_B \ : \ B^D_{\mathbf{k_1}} \prec B \prec B^D_{\mathbf{k_2}}\}$, for suitable choices of $\mathbf{k_1}, \mathbf{k_2} \in I^D_B$ designed to

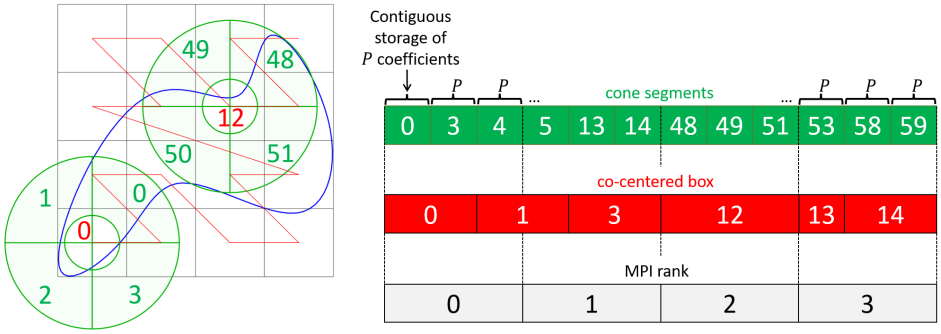


Figure 3: Left panel: Two-dimensional example of an ordering of the cone segments based on the Morton order of the boxes on level $d = 3$ with four cone segments per box. The red line indicates the Morton order of the boxes where the red numbers denote the actual Morton code of the containing box. The green numbers denote the ordering of the cone segments in the proposed Morton-based cone-segment ordering. The blue curve represents the domain boundary Γ. Right panel: Sketch of a possible cone-segment memory layout, demonstrating the equi-distribution of cone segments among ranks, and emphasizing a central element of the proposed parallelization strategy, namely, that co-centered cone segments may be assigned to different MPI ranks. Note that only relevant boxes and cone segments are stored in memory resulting in some numbers in the ordering being skipped.

Algorithm 9 MPI Parallel LevelDEvaluations

1: **parallel for** $C^D_{\mathbf{k};\gamma} \in \mathcal{R}^D_{C,\rho}$ **do**
2: Evaluate and store $F^D_{\mathbf{k}}(\mathcal{X}C^D_{\mathbf{k};\gamma})$
3: Generate interpolant $I_P C^D_{\mathbf{k};\gamma}$
4: **end parallel for**

guarantee that all the boxes on a given rank contain a number of surface discretization points as close as possible to the average value N/N_r.

The set of *surface discretization points stored in the ρ-th MPI rank*, $1 \leq \rho \leq N_r$, is denoted by $\Gamma_{N,\rho}$. By definition, the subsets $\Gamma_{N,\rho}$ of Γ_N are pairwise disjoint and their union over all MPI ranks $\rho = 1, \ldots, N_r$ equals Γ_N. This partition of the set of surface discretization points is used to evenly divide, among all MPI ranks, the work performed in the *Interpolation* function (OpenMP Algorithm 8). The level-D based partition of Γ_N is used to order the discretization points for all subsequent levels ($d = D-1, \ldots, 3$). Thus, the MPI parallel *Interpolation* function results from the straightforward and level-independent modification of Line 1 in Algorithm 8, to read $x \in \Gamma_{N,\rho}$ instead of $x \in \Gamma_N$—as shown in Algorithm 10. Naturally, the values of the discrete operator $I(x_\ell)$ in (1) computed by the ρ-th MPI rank correspond to points $x_\ell \in \Gamma_{N,\rho}$, and they are therefore also stored in the ρ-th MPI rank. In other words, the set of resulting field values $I(x_\ell)$ is partitioned and stored in the MPI ranks according to the partition utilized for the surface discretization points Γ_N.

Algorithm 10 MPI Parallel Interpolation(d)

1: **parallel for** $x \in \Gamma_{N,\rho}$ **do**
2: **for** $B^d_{\mathbf{k}} \in \mathcal{M}^d(x)$ **do**
3: Determine $C^d_{\mathbf{k};\gamma}$ s.t. $x \in C^d_{\mathbf{k};\gamma}$
4: Evaluate $I_P C^d_{\mathbf{k};\gamma}(x) \times G(x, x^d_{\mathbf{k}})$
5: **end for**
6: **end parallel for**

Algorithm 11 MPI Parallel Propagation(d)

1: **parallel for** $C^{d-1}_{\mathbf{j};\gamma} \in \mathcal{R}^{d-1}_{C,\rho}$ **do**
2: **for** $B^d_{\mathbf{k}} \in \mathcal{C}(\mathcal{R}_B C^{d-1}_{\mathbf{j};\gamma})$ **do**
3: **for** $x \in \mathcal{X} C^{d-1}_{\mathbf{j};\gamma}$ **do**
4: Determine $C^d_{\mathbf{k};\alpha}$ s.t. $x \in C^d_{\mathbf{k};\alpha}$
5: Evaluate and add $I_P C^d_{\mathbf{k};\alpha}(x) \times G(x, x^d_{\mathbf{k}})/G(x, x^{d-1}_{\mathbf{j}})$
6: **end for**
7: **end for**
8: Generate interpolant $I_P C^{d-1}_{\mathbf{j};\gamma}$
9: **end parallel for**

The data associated with the level-d relevant cone segments is also distributed to MPI ranks on the basis of a total order—in this case, a total order on the set of level-d cone segments that is based on the Morton order imposed on the level-d boxes, in such a way that co-centered cone segments are close in memory. It should be noted that, for every relevant cone segment $C^d_{\mathbf{k};\gamma} \in R^d_C$, $3 \leq d \leq D$ (see (8)), the set of P coefficients that characterize the polynomial interpolants $I_P C^d_{\mathbf{k};\gamma}$ (Section 2), which approximate the field $F^d_{\mathbf{k}}$ in (6) within the cone segment $C^d_{\mathbf{k};\gamma}$, need to be stored, in appropriately distributed manner, for two consecutive levels. Indeed, for each d, these level-d coefficients are utilized to enable two different interpolation procedures, namely interpolation from level d to interpolation points at the parent-level $(d-1)$ in the *Propagation* function (Line 4 in Algorithm 7), as well as interpolation to the level-d cousin surface discretization points in the *Interpolation* function (Line 3 in Algorithm 8).

The set of level-d relevant cone segments $\mathcal{R}^d_C$ is sorted on the basis of the Morton order induced by the co-centered level-d boxes followed by a suitable sorting of cone segments in each spherical coordinate system—resulting in a total order $\sqsubset$ in the set of all level-d relevant cone segments, as depicted in the left panel of Figure 3. (Each set of co-centered cone segments is ordered using the radial direction first, then elevation and finally azimuth, although any other ordering could be used.) Finally, at each level d ($d = D, \ldots, 3$), approximately equi-sized and pair-wise disjoint intervals of relevant cone segments C of the form $\left\{ C \in \mathcal{R}^d_C \ : \ C^d_{\mathbf{k_1};\gamma_1} \sqsubset C \sqsubset C^d_{\mathbf{k_2};\gamma_2} \right\}$, for some $\mathbf{k_1}, \mathbf{k_2} \in I^d_B$ and $\gamma_1, \gamma_2 \in I^d_C$ (i.e., disjoint intervals of *contiguous* cone segments), are distributed to the MPI ranks, as illustrated in the right panel of Figure 3. Note that the specific assignment of cone segments to MPI ranks is solely determined by the order $\sqsubset$ and the number of MPI ranks and cone segments, and it does not otherwise relate to the underlying box tree. In particular, as suggested in the right panel of Figure 3, co-centered cone segments may be assigned to different MPI ranks—which induces a flexibility that leads to load-balancing of good-quality and, therefore, high parallelization efficiency. As is the case for the relevant boxes, the proposed ordering of the relevant cone segments implies that cone segments which are close in real space (i.e. co-centered with the same box and pointing in the same direction or co-centered with boxes which are close in real space) are also close in memory, and, in particular, are likely to be stored within the same MPI rank. Analogously to the notation introduced above for the distributed surface discretization points, the relevant level-d cone segments assigned to a MPI rank with index ρ, $1 \leq \rho \leq N_r$, are denoted by $\mathcal{R}^d_{C,\rho}$. The MPI-capable algorithm is thus obtained by adjusting the loops in the first lines in Algorithms 6 and 7 to only iterate over the level-d relevant cone segments $\mathcal{R}^d_{C,\rho}$ stored in the current rank ρ, as shown in the MPI parallel Algorithms 9 and 11, instead of iterating over all relevant cone segments on level d.

3.2.2 Data communication

Clearly, for an MPI rank to access data stored in a different rank, explicit communication between the ranks must take place. The proposed solution, which we favor due to the decreased complexity of the implementation it entails, is based on *one-sided* or *remote memory access* (RMA) communication introduced

Algorithm 12 CommunicateInterpolationData(d)

parallel for $x \in \Gamma_{N,\rho}$ **do**
  for $B^d_{\mathbf{k}} \in \mathcal{M}^d(x)$ **do**
    Find $\tilde{\gamma}$ such that $x \in C^d_{\mathbf{k};\tilde{\gamma}} \in \mathcal{R}_C B^d_{\mathbf{k}}$
    Identify the MPI rank ρ on which $I_P C^d_{\mathbf{k};\tilde{\gamma}}$ is stored
    MPI_Get $I_P C^d_{\mathbf{k};\tilde{\gamma}}$ from rank ρ
  end for
end parallel for

Algorithm 13 CommunicatePropagationData(d)

parallel for $C^{d-1}_{\mathbf{j};\gamma} \in \mathcal{R}^{d-1}_{C,\rho}$ **do**
  for $B^d_{\mathbf{k}} \in \mathcal{C}(\mathcal{R}_B C^{d-1}_{\mathbf{j};\gamma})$ **do**
    for $x \in \mathcal{X} C^{d-1}_{\mathbf{j};\gamma}$ **do**
      Find $\tilde{\gamma}$ such that $x \in C^d_{\mathbf{k};\tilde{\gamma}} \in \mathcal{R}_C B^d_{\mathbf{k}}$
      Identify the MPI rank ρ on which $I_P C^d_{\mathbf{k};\tilde{\gamma}}$ is stored
      MPI_Get $I_P C^d_{\mathbf{k};\tilde{\gamma}}$ from rank ρ
    end for
  end for
end parallel for

in MPI-2 [38, Section 5], [39, Section 8]—which utilizes a single MPI_Get or MPI_Put call on the origin rank instead of a coupled MPI_Recv-MPI_Send call (or similar functionalities) involving both the origin and the target rank.

The data any MPI rank may require from other MPI ranks is limited to certain interpolants $I_P C^d_{\mathbf{k};\gamma}$. It is therefore sufficient to store the corresponding coefficients in so-called RMA *windows* (in MPI given by MPI_Win and allocated with e.g. MPI_Win_allocate), which enable the one-sided communication approach. For increased efficiency, the computations and communications are organized among the ranks on the basis of the following two considerations: 1) For each ρ, $1 \leq \rho \leq N_r$, the ρ-th rank asynchronously collects from other ranks all the data (i.e. the coefficients of the interpolants) it requires to perform *Interpolation* or *Propagation* computations assigned to it; and 2) The communications necessary to collect this data are interleaved with the computations in such a way that while the computations by the *Interpolation* function take place, the communication for the next *Propagation* function is performed and vice versa. This approach, which effectively hides the communications behind computations (thus improving the overall performance and parallel efficiency), requires every MPI rank to store all data it obtains from other ranks for one full level-d ($3 \leq d \leq D$) *Interpolation* or *Propagation* step while it continues to store the coefficients it has itself generated—which effectively increases the peak memory per rank requirements slightly (by e.g. 10% or less). The level-d dependent *CommunicateInterpolationData* (resp. *CommunicatePropagationData*) programming function in Algorithm 12 (resp. Algorithm 13) encapsulates the communications performed by each rank to obtain, from other ranks, the polynomial coefficients it needs to enact the necessary level-d interpolation computations (resp. interpolation computations onto level-$(d-1)$ interpolation points) required by the *Interpolation* (resp. *Propagation*) function.

Using the functions 9 through 12, the pseudo-code for the proposed overall MPI-OpenMP IFGF algorithm is given in Algorithm 14. Note that access to RMA windows is usually asynchronous and requires some form of synchronization to ensure the data transfer is finalized before the communicated data is accessed. Moreover, the call to the *CommunicatePropagationData* in Algorithm 14 requires for the *Propagation* function to have completed in all ranks targeted by the communication function.

Algorithm 14 IFGF Method

```
LevelDEvaluations()
CommunicatePropagationData(D)

for d = D, ..., 3 do
    CommunicateInterpolationData(d)
    if d > 3 then
        Propagation(d)
        if d > 4 then
            CommunicatePropagationData(d − 1)
        end if
    end if
    Interpolation(d)
end for
```

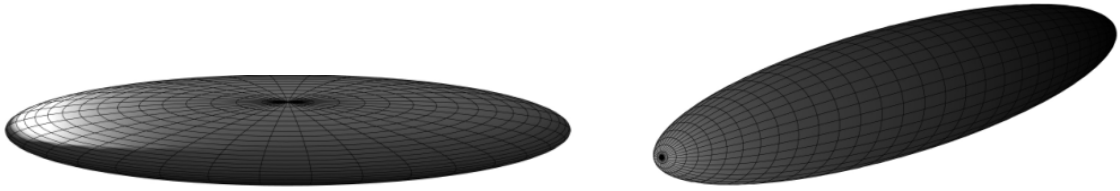


Figure 4: Test geometries. Left: Oblate spheroid $x^2 + y^2 + (z/0.1)^2 = a^2$. Right: Prolate spheroid $x^2 + y^2 + (z/10)^2 = a^2$.

4 Numerical Results

Our numerical examples focus on three simple geometries which coincide with the test cases presented in [1]: a sphere of radius a, the oblate spheroid $x^2+y^2+(z/0.1)^2 = a^2$ and the prolate spheroid $x^2+y^2+(z/10)^2 = a^2$. The latter two geometries are depicted in Figure 4. In what follows the diameter (also referred to as the "size") of a geometry Γ is denoted by

$$A := A(\Gamma) := \max_{x,y\in\Gamma} |x - y|; \tag{12}$$

clearly we have $A = 2a$ in the case of the sphere and the oblate spheroid geometries and $A = 20a$ for the prolate spheroid geometry. These relatively simple geometries present the same kinds of challenges, in the context of the IFGF method, that arise in a wide range of real-world problems, including aircraft, lenses and meta-materials (with a point distribution somewhat similar to that in an oblate spheroid), submarines (prolate spheroid), etc. For example, even though the problem of finding a scattering solution for a submarine is much more challenging than the corresponding problem for a spheroid of the same size, in view of the need for accurate integration of singular kernels and adequate representation of the surface Jacobians, the performance of the IFGF method for the evaluation of the discrete operator (1) for a submarine should not differ significantly from the corresponding performance on a prolate spheroid of a comparable discretization, point distribution and electromagnetic size. Per the discussion in Section 4.4, a strategy based on pinning 4 MPI ranks to each compute node, where each MPI rank spawns 14 OpenMP threads, is utilized in all test cases presented in this paper. Throughout this section, further, numbers $P_s = 3$ and $P_{\text{ang}} = 5$ of interpolation points in the radial and angular variables, respectively, were used in each cone segment; cf. Sections 2 and 5.

In what follows we present IFGF performance data based on various runs for the aforementioned geometries. In detail, Section 4.1 describes the hardware employed for the numerical tests and Section 4.2 introduces the relative L_2 error estimate used. Section 4.3 then details the concepts of strong parallel-efficiency and speedup that are used subsequently, and Section 4.4 describes the observed impact of the

proposed hybrid MPI-OpenMP strategy on performance. Sections 4.5 and 4.6 present the observed communication patterns and linearithmic scaling, respectively. Sections 4.7 and 4.8, finally, present test cases demonstrating the strong parallel efficiency of the algorithm and its performance for large sphere problems, including detailed comparisons with the relevant recent literature.

4.1 Compiler and hardware

The proposed parallel IFGF program was implemented in C++, and the resulting code was compiled with the Intel mpiicpc compiler, version 2021.1, and the Intel MPI library. The following performance-relevant compiler flags were used: -std=c++20, -O3, -ffast-math, -qopt-zmm-usage=high, -no-prec-sqrt, -no-prec-div. All tests were run on our internal *Wavefield* cluster which consists of 30 dual-socket nodes. Each node consists of two Intel Xeon Platinum 8276 processors with 28 cores per processor, for a total of 56 cores per node, and 384 GB of GDDR4 RAM per node. (The Xeon processors we use support hyper-threading, but this capability was not exploited in any of the tests presented in this paper.) The nodes are connected with HDR Infiniband.

4.2 Numerical error estimation

The errors reported in what follows were computed as indicated in [1], that is, utilizing the relative L_2 difference ε_M between the full, non-accelerated evaluation of the field $I(x)$, as stated in (1), and the IFGF-accelerated evaluation $I_{\text{acc}}(x)$ of (1) computed on a randomly chosen subset of M surface discretization points. More precisely, the error is given by

$$\varepsilon_M = \sqrt{\frac{\sum_{i=1}^{M} |I(x_{\sigma(i)}) - I_{\text{acc}}(x_{\sigma(i)})|^2}{\sum_{i=1}^{M} |I(x_{\sigma(i)})|^2}}, \tag{13}$$

where σ is a random permutation and $x_\ell \in \Gamma_N$ denote the surface discretization points. The method used in [1] is suitably extended to the present MPI parallel implementation by using a set of test points x_ℓ that contains a number $M = 1000$ of randomly chosen points on each MPI rank. (In [1] it was shown for sufficiently small examples that an error evaluation on a subset of 1000 points produces an error estimation close to the actual error.) More precisely, 1000 surface discretization points are randomly chosen on each MPI rank from the distinct set of surface discretization points $\Gamma_{N,\rho}$ assigned to MPI rank ρ ($1 \leq \rho \leq N_r$) on the basis of the data distribution strategy introduced in Section 3.2. The final errors are then accumulated resulting in the overall error estimate

$$\varepsilon := \varepsilon_M \quad \text{with} \quad M = 1000 \times N_r. \tag{14}$$

As a result, the error estimates we use depend on the number N_r of MPI ranks, which accounts for the slight variations in the errors reported as number of MPI ranks is varied (cf. Tables 3 and 4). All tests were set up in such a way that an error of approximately 10^{-3} was achieved, although the IFGF method can achieve arbitrarily small errors.

4.3 Strong parallel efficiency concept

Let $T(N_c, N)$ denote the time required by a run of the parallel IFGF algorithm on an N-point discretization Γ_N of a given surface Γ (obtained by means of a fixed discretization scheme) on a total of N_c computing cores. Using this notation, for a given N, the strong parallel efficiency $E^s_{N_c^0,N_c}$ that results as the number

of cores is increased from N_c^0 to N_c is defined as the quotient of the resulting speedup $S_{N_c^0,N_c}$ to the corresponding ideal speedup value $S^{\text{ideal}}_{N_c^0,N_c}$:

$$S^{\text{ideal}}_{N_c^0,N_c} := \frac{N_c}{N_c^0}, \qquad S_{N_c^0,N_c} := \frac{T(N_c^0,\Gamma_N)}{T(N_c,\Gamma_N)}, \qquad E^s_{N_c^0,N_c} := \frac{S_{N_c^0,N_c}}{S^{\text{ideal}}_{N_c^0,N_c}}.$$

Note that the implicit dependence on N and Γ_N is suppressed in the speedup and efficiency notations.

4.4 Impact of single-node OpenMP/MPI pinning selections

In order to optimize code performance we investigated the dependence of the strong parallel efficiency and the memory usage on the number of OpenMP threads per MPI rank for a fixed total of 56 OpenMP threads in a single node. Representative results of these experiments are presented in Figure 5, which correspond to a test case involving 393,216 discretization points on a sphere 16λ in diameter, where $\lambda = \frac{2\pi}{\kappa}$ denotes the wavelength. (A small problem was selected for this example as a stress-test for the parallelization strategy used.) That figure shows that the lowest efficiency results as a single MPI rank spawning all 56 OpenMP threads is utilized, each pinned to one of the 56 physical cores available in the single dual-socket node used. The figure also shows that the use of an increasing number of MPI ranks and a decreasing number of OpenMP threads per rank while maintaining a total of 56 threads, results in varying efficiencies and memory requirements. In particular, the memory requirements increase, albeit sublinearly, with the number of MPI ranks, owing to the grouped communication strategy used—which, as described in Section 3.2.2, requires storage of all the communicated data and which, therefore, leads to increases in the memory requirements as the number of MPI ranks utilized grows. It is therefore desirable to minimize the number of MPI ranks per node while maintaining as high an efficiency as possible. Figure 5 shows that the best efficiency is achieved, while incurring a modest memory usage, when 4 MPI ranks per node are utilized, each one of which spawns 14 OpenMP threads. This selection was therefore adopted and used throughout the numerical examples presented in this paper.

4.5 MPI communication patterns

As detailed in Section 3.2.1, the proposed parallelization strategy and associated data distribution approach are based on ordering of data according to the Morton order and associated Σ curve depicted in Figure 3. This ordering is introduced with the dual intent of minimizing the amount of data communicated between MPI ranks, and localizing the communication that is still required to pairs of ranks containing data corresponding to nearby boxes. Figure 6 displays the resulting communication patterns for a representative test case, wherein, using 1,572,864 discretization points, a problem for a sphere 128λ in diameter was run on 16 ranks (224 cores). The left panel displays the amount of data communicated by rank j to rank k, $0 \le j,k \le 15$, whereas the right panel reports the total amount of data communicated to each rank (in red) and the peak memory usage on each rank (in blue). The results presented in the left panel indicate that a high degree of localization was achieved by the strategy: the majority of the communication takes place between MPI ranks with IDs differing only by 1, although significant amounts of communication can be observed between certain pairs of non-neighboring ranks (e.g. ranks 1 and 8). This is easily understood in terms of the character of the Σ curve depicted in Figure 3, which preserves locality to a significant extent but not perfectly. The right panel of figure 6 shows that the data communicated to each rank through the complete run of the algorithm (including, for the case the $D = 10$ presently considered, the memory communicated in each one of seven levels $d = 3$ to $d = 10$ where memory exchanges take place) is less than 10% of the peak memory requirements of each rank. In particular, since the amount of memory communication is essentially independent on the level d, it follows that approximately 1.5% of the peak memory requirements were communicated on each level.

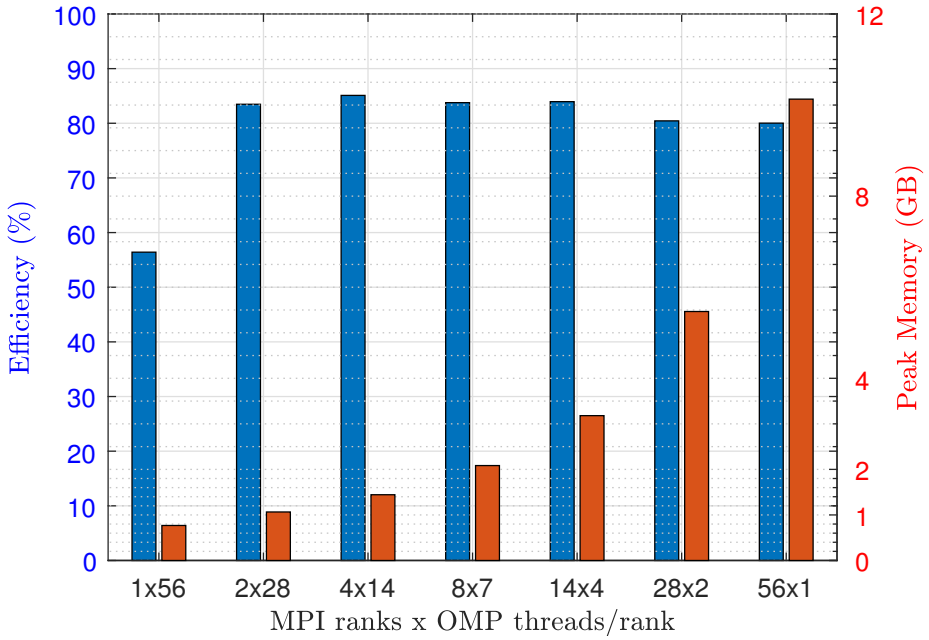


Figure 5: Strong Parallel Efficiency and Peak Memory usage as a function of the number of OpenMP threads per MPI rank for a fixed total of 56 OpenMP threads.

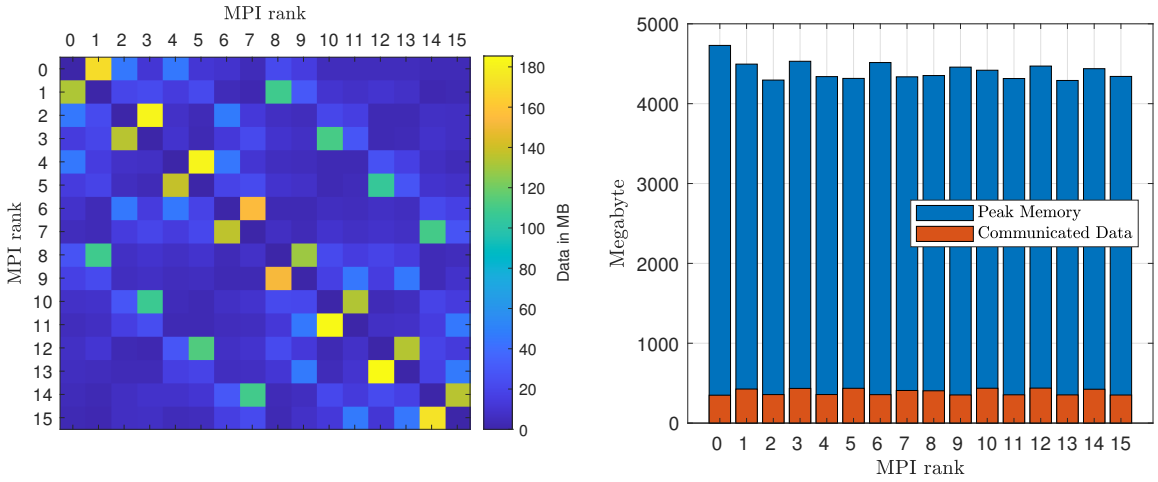


Figure 6: Communication patterns observed for a representative test case run on 16 MPI ranks. Left: amount of data communicated between ranks with IDs j (row) and k (column), $0 \leq j, k \leq 15$. Right: total amount of data communicated to each rank (in red) and the the peak memory usage on each rank (in blue).

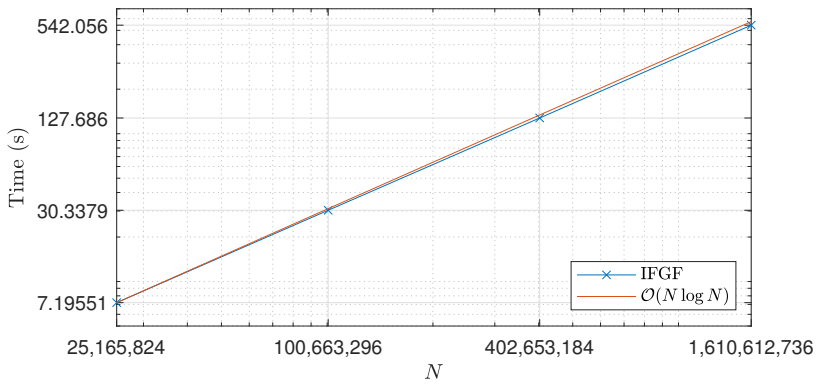


Figure 7: Illustration of the linearithmic complexity of the parallel IFGF method (which had previously been demonstrated [1] for the serial version of the algorithm), for the prolate spheroid geometry, on 30 compute nodes. Various statistics associated with this figure are presented in Table 1 and its caption.

4.6 Linearithmic scaling

Figure 7 presents results of a study of the scaling of the parallel IFGF method for the prolate spheroid geometry on a fixed number of nodes (namely, all 30 nodes available in the computer cluster we use), utilizing four MPI ranks per node (as recommended in Section 4.4), and with parameters resulting in an IFGF error $\varepsilon \approx 2 \times 10^{-2}$ (cf. equation (14)). The study was implemented for values of N ranging from 25,165,824 to 1,610,612,736, and for corresponding diameters ranging from 512λ to $4,096\lambda$. Various statistics associated with this figure are presented in Table 1. The results show that the linearithmic algorithmic complexity and memory requirements of the basic IFGF algorithm are maintained in the parallel setting. In fact, the observed complexity even slightly outperforms the postulated $\mathcal{O}(N \log N)$ within this range of values of N; cf. Table 1 which suggests convergence to exact linearithmic complexity, with a well defined proportionality constant as N grows.

Γ	N	A	N_c	ε	T (s)	Memory	$T/(N \log N)$
Prolate Spheroid	25,165,824	512λ	$1,680$	2×10^{-2}	7.20×10^{0}	83 GB	1.678×10^{-8}
	100,663,296	$1,024\lambda$			3.03×10^{1}	268 GB	1.636×10^{-8}
	402,653,184	$2,048\lambda$			1.28×10^{2}	1022 GB	1.600×10^{-8}
	1,610,612,736	$4,096\lambda$			5.42×10^{2}	4123 GB	1.588×10^{-8}

Table 1: Data underlying the linearithmic scaling test illustrated in Figure 7. In this test the acoustic diameter of the ellipsoid was kept proportional to $\sqrt{N}$, and a discretization density of 5.6 points per wavelength was used. The Memory column presents the sum over all MPI ranks of the peak memory usage per MPI rank. The values in the last column suggests convergence to exact $\mathcal{O}(N \log N)$ scaling.

4.7 Strong parallel efficiency tests

This section concerns the strong parallel efficiency of the proposed parallel IFGF algorithm. In order to properly assess the parallel performance of the code, all the scaling tests were conducted without hyper-threading, and with the Intel®Turbo Boost technology deactivated. (When active, the Turbo Boost technology scales the processors frequency depending on the number of CPU cores in use within each processor.) The observed strong-scaling speedups S_{1,N_c} and associated parallel efficiencies E^s_{1,N_c} ($1 \leq N_c \leq 1,680$) are displayed in the upper and lower panels of Figure 8, respectively, including results for three test cases, namely, a sphere of diameter $A = 128\lambda$ and oblate and prolate spheroids (Figure 4)

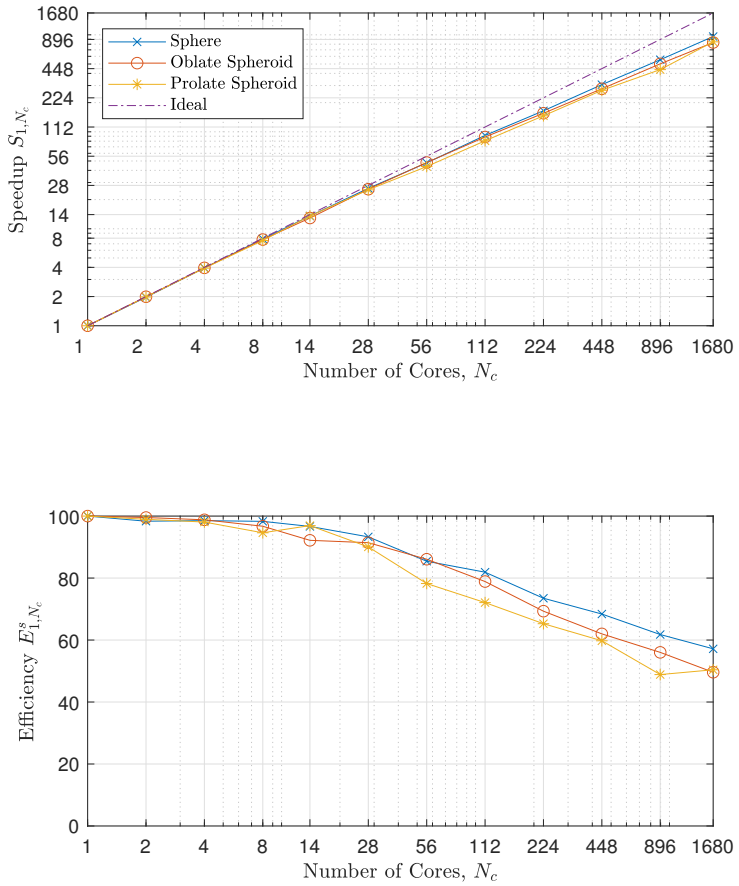


Figure 8: Observed speedup S_{1,N_c} (upper panel) and strong-scaling efficiency $E^s_{N_c^0,N_c}$ (lower panel) versus number of cores N_c in a strong scaling test transitioning from 1 core to 1,680 cores (= 30 compute nodes) for three geometries: a sphere 128 wavelengths in size (blue), an oblate spheroid 128 wavelengths in size (red), and a prolate spheroid 256 wavelengths in size (yellow). The dash-dotted purple line in the upper graph indicates the theoretical perfect speedup.

of large diameters $A = 128\lambda$ and $A = 256\lambda$, respectively. In all test cases considered in this section discretizations at 5.6 points per wavelength were used.

In view of the requirements of the strong-scaling setup, test problems were selected that can be run in a reasonable time on a single core and with the memory available in the corresponding compute node. Clearly, such test problems tend to be too small to admit a perfect distribution onto large numbers of cores. As illustrated in Figure 8, however, in spite of this constraint, scaling is observed in the complete range going from 1 core to $1,680$ cores (30 nodes), and, as discussed in Section 3, no hard limitation on the scaling can be observed. It is reasonable to expect that, unlike other approaches (for which either hard limits arise [16], or which rely on memory duplication [18, 19]), the observed speedup continues to scale with the number of cores, as suggested by Figure 8, up to large numbers of cores.

Γ	N	A	N_c	ε	T (s)	E^s_{1,N_c} (%)	S_{1,N_c}
Sphere	1,572,864	128λ	1	2×10^{-3}	4.29×10^3	100	1.00
			2		2.18×10^3	98	1.97
			4		1.09×10^3	99	3.94
			8		5.46×10^2	98	7.86
			14		3.17×10^2	97	13.53
Oblate Spheroid	1,572,864	128λ	1	5×10^{-4}	1.57×10^3	100	1.00
			2		7.90×10^2	100	1.99
			4		3.98×10^2	99	3.95
			8		2.03×10^2	97	7.74
			14		1.22×10^2	92	12.90
Prolate Spheroid	6,291,456	256λ	1	6×10^{-4}	3.63×10^3	100	1.00
			2		1.83×10^3	99	1.98
			4		9.25×10^2	98	3.92
			8		4.80×10^2	95	7.57
			14		2.67×10^2	97	13.57

Table 2: Strong parallel scaling test of the OpenMP IFGF implementation from $N_c = 1$ to $N_c = 14$ cores in a single node for three different geometries Γ.

Γ	N	A	N_r	N_c	ε	T (s)	E^s_{14,N_c} (%)	S_{14,N_c}
Sphere	1,572,864	128λ	1	14	2×10^{-3}	3.17×10^2	100	1.00
			2	28	2×10^{-3}	1.64×10^2	97	1.93
			4	56	2×10^{-3}	8.98×10^1	88	3.54
Oblate Spheroid	1,572,864	128λ	1	14	5×10^{-4}	1.22×10^2	100	1.00
			2	28	6×10^{-4}	6.14×10^1	99	1.98
			4	56	6×10^{-4}	3.26×10^1	93	3.74
Prolate Spheroid	6,291,456	256λ	1	14	6×10^{-4}	2.67×10^2	100	1.00
			2	28	6×10^{-4}	1.44×10^2	93	1.86
			4	56	5×10^{-4}	8.28×10^1	81	3.23

Table 3: Strong parallel scaling test of the shared-memory MPI implementation on a single node, transitioning from $N_c = 14$ cores to $N_c = 56$ (all cores available in one compute node) by increasing the number N_r of MPI ranks from 1 to 4, for three different geometries Γ.

Γ	N	A	N_c	ε	T (s)	E^s_{56,N_c} (%)	S_{56,N_c}	$E^s_{\frac{N_c}{2},N_c}$
Sphere	1,572,864	128λ	56	2×10^{-3}	8.98×10^1	100	1.00	-
			112	2×10^{-3}	4.68×10^1	96	1.92	96
			224	2×10^{-3}	2.61×10^1	86	3.44	90
			448	2×10^{-3}	1.40×10^1	80	6.40	93
			896	2×10^{-3}	7.76×10^0	72	11.57	90
			1680	2×10^{-3}	4.47×10^0	67	20.08	*93
Oblate Spheroid	1,572,864	128λ	56	6×10^{-4}	3.26×10^1	100	1.00	-
			112	6×10^{-4}	1.78×10^1	92	1.83	92
			224	6×10^{-4}	1.01×10^1	81	3.22	88
			448	6×10^{-4}	5.66×10^0	72	5.76	90
			896	6×10^{-4}	3.13×10^0	65	10.41	90
			1680	6×10^{-4}	1.89×10^0	58	17.29	*89
Prolate Spheroid	6,291,456	256λ	56	4×10^{-4}	8.28×10^1	100	1.00	-
			112	5×10^{-4}	4.50×10^1	92	1.84	92
			224	6×10^{-4}	2.48×10^1	83	3.34	91
			448	6×10^{-4}	1.36×10^1	76	6.11	92
			896	6×10^{-4}	8.29×10^0	62	9.99	82
			1680	6×10^{-4}	4.29×10^0	64	19.33	*103

Table 4: Strong parallel scaling test of the distributed-memory MPI implementation from $N_c = 56$ to $N_c = 1680$ cores (1 to 30 compute nodes) with 4 MPI ranks per node for three different surfaces Γ. In view of the limitation imposed by the total number (1680) of available cores in the hardware used, the data points marked with an asterisk (*) in the last column do not correspond to doubling of core numbers, but show instead the parallel efficiency $E^s_{896,1680}$ with respect to the second to last row for each surface Γ.

Tables 2, 3, and 4 provide details concerning the data displayed in Figure 8: they focus, respectively, on the strong parallel efficiencies achieved by the proposed OpenMP, shared-memory MPI, and distributed-memory MPI parallelization. In detail, these tables display the main two quantifiers of strong parallel performance, namely, the observed strong parallel efficiency $E^s_{N_c^0,N_c}$ and speedup $S_{N_c^0,N_c}$, along with the computing times T, the resulting accuracy ε, the largest diameter A and the numbers of discretization points used. The tables and figure suggest that the IFGF parallel efficiencies are essentially independent of the geometry type. Among these tables greater significance may be attached to Table 4, which demonstrates the scaling of the method under the one hardware element that can truly be significantly increased, namely, the number of compute nodes. According to this table, a strong scaling efficiency of approximately 60% across the complete cluster used can be observed in all cases. The observed efficiency loss can mainly be attributed to the load-imbalance induced by the necessary data partitioning structure and the cost of communication between ranks. In detail, the algorithm's data partitioning strategy is based on an equi-partition of the number of target points of the interpolation procedures associated with both, the interpolation to the surface discretization points (Algorithm 8), and the interpolation points on the parent-level cone segments (Algorithm 7). While this strategy leads to a reasonable memory distribution (as suggested by the right panel in Figure 6), it does not guarantee an equi-partition of the number of operations, as the number of interpolants used on a given target point may vary significantly across the set of target points, resulting in load-imbalance.

The speedups achieved by the proposed parallel strategy appear to outperform those achieved by MPI-parallel implementations of FMM, as can be verified by comparison with results presented in, e.g., [20, 22, 27]. Indeed, [20, Fig. 4, Fig. 5], presents a scaling test case for a problem of spheres of diameters

between 40λ and 120λ, run from 1 to 128 cores, showing, in all cases, parallel efficiencies of the order of 60% for the 128 core runs. According to the value shown in Figure 8 for the sphere 128λ in diameter, in turn, the IFGF method achieves 82% efficiency on 112 cores for this problem, and it remains above 60% efficiency up to 1,680 cores. Reference [22, Fig. 1], on the other hand, utilizes up to 1024 cores for a single-level FMM implementation, showing a parallel efficiency of less than 50% going from eight cores to 1024 cores for a sphere test case of 90λ in size, whereas, according to Figure 8, the IFGF approach achieves 57% efficiency scaling from 1 core to all 1680 cores. Reference [27], in turn, presents scaling results for the parallel BEMFMM implementation of the FMM algorithm; detailed comparisons with results presented in that paper are presented in Section 4.8. To obtain a useful direct comparison with the BEMFMM implementation presented in [27] we additionally utilized the freely available BEMFMM open-source download provided by the authors to performed direct performance comparisons of the IFGF and BEMFMM implementations. (Direct comparisons with data reported in in [27] are presented in Section 4.8.) By necessity, our tests were limited to a test example consisting of a sphere containing approximately 360,000 DOF, which is the largest test case provided with the BEMFMM test code, and we selected a sphere of acoustic diameter of 16λ for this experiment. We run both algorithms in the 30 available nodes in our clusterm each one of which contains 56 computing cores. Our observations are laid out in Table 5. We note that, in particular, that the IFGF algorithm exhibits reasonable scaling even for the extremely small problem considered. Further, the IFGF runs were faster than the BEMFMM runs by factors of 9.3 and 41.6 in the 1-node and 30-node runs, respectively, with an IFGF speedup over 4 times higher than that provided by BEMFMM going from 1 to 30 nodes.

	BEMFMM	IFGF	BEMFMM/IFGF
N	361,224	393,216	
1 Node (s)	14.95	1.60	9.3
30 Nodes (s)	4.99	0.12	41.6
Speedup	3.00	13.33	

Table 5: Computing times, speedups and performance ratios observed in runs of the BEMFMM [27] and IFGF implementations in the 30 Node/1680 core cluster used in the present paper.

4.8 Large sphere tests

Table 6 illustrates the performance of the IFGF method in terms of computing time and memory requirements for a large-sphere problem, $1{,}389\lambda$ in diameter, which was run on the full 30 node, 1,680 core cluster. As indicated in the table, the sphere was discretized using ≈ 2.15 billion degrees of freedom, and, with a memory usage of under 11 TB and with a computing time of under 900 seconds, the algorithm evaluated the discrete integral operator with a relative L^2 near-field error of 6×10^{-3}. This sphere problem corresponds to the 2-meter sphere at $f = 238.086$KHz considered in [27, Table 2]—whose acoustic size indeed equals 2 meters$/\lambda = 2f/c = 2 \cdot 238,086/343 \approx 1,389$.

For reference, [27, Table 2] reports a BEMFMM run configured with 1.0×10^{-3} accuracy for the aforementioned $f = 238,086$KHz sphere test case, showing a near-field solution error of 2×10^{-1} with evaluation of the discrete integral operator in a computing time of ≈ 52 seconds in a computer containing 131,072 cores and with an unspecified amount of memory.

Γ	N	A	N_c	ε	T (s)	Memory
Sphere	2,147,483,648	1,389 λ	1,680	6×10^{-3}	8.77×10^2	10,449 GB

Table 6: Large sphere test case run on thirty 56-core compute nodes (for a total of 1,680 cores), utilizing 120 MPI ranks. The sphere of acoustic size 1,389 λ in diameter (resulting in a resolution of approximately 19 points per wavelength) in this table coincides with largest sphere test case considered in [27].

5 Concluding Remarks

This paper presented a parallel version of the IFGF acceleration method introduced in [1], demonstrating in practice parallel scaling to large core numbers while simultaneously preserving the linearithmic complexity of the sequential IFGF algorithm. The proposed parallelization approach exploits the box-cone octree structure inherent in the IFGF method, resulting in a strategy that, per the theoretical discussion in Section 3.1 and the first paragraph of Section 3.2, natively avoids bottlenecks or hard limits inherent in approaches that orchestrate the parallelization on the basis of octree-box partitioning only. A number of additional questions are left for future work, as briefly summarized in what follows. On one hand, minor modifications to the data-decomposition strategy introduced in Section 3.2.1 could be introduced to not only (approximately) equipartition the surface discretization points and cone segments among MPI ranks, but to also incorporate the number of actual computations and the amount of data required from other MPI ranks in the partitioning scheme. Such an improved data-decomposition design could indeed be obtained by relying on minor adjustments to the cone and box intervals introduced in Section 3.2.1 leading to improved load-balancing, and, thus, improved parallel efficiency. Further, the feasibility of implementations on heterogeneous architectures such as, e.g., computer systems that incorporate general purpose graphical processing units (GPUs), is currently under study, as is the use of CPU vectorization in the performance-critical interpolation stage. In particular, the use of GPUs to accelerate the interpolation processes, which represent the most time consuming part of the IFGF method, appears as a highly promising avenue of inquiry. Concerning mathematical aspects of the algorithm, the implementation presented in this paper is intended for use with relatively low-order interpolation degrees P_{s} and P_{ang}, as in [1], but higher-order counterparts are envisioned. Further, trigonometric coordinate transformations that are required as part of the spherical-coordinates interpolation scheme that is central to the IFGF algorithm, and which entail approximately 50% of the computational cost of the algorithm, have been incorporated by direct evaluation of trigonometric functions whenever needed, presenting an opportunity for further optimization. In any case, while a number of additional optimizations could be pursued, we submit that the current purely memory based load-balancing strategy (presented in Section 3.2.1) achieves an adequate balancing of memory transfers and computing times per MPI rank and leads to an overall favorable parallel scaling.

Acknowledgments

This work was supported by NSF, DARPA and AFOSR through contracts DMS-2109831 and HR00111720035, FA9550-19-1-0173 and FA9550-21-1-0373, and by the NSSEFF Vannevar Bush Fellowship under contract number N00014-16-1-2808.

References

[1] Christoph Bauinger and Oscar Bruno. "Interpolated Factored Green Function" method for accelerated solution of scattering problems. *Journal of Computational Physics*, 430, 01 2021.

[2] Björn Engquist and Lexing Ying. Fast directional multilevel algorithms for oscillatory kernels. *SIAM Journal on Scientific Computing*, 29(4):1710–1737, 2007.

[3] Matthias Messner, Martin Schanz, and Eric Darve. Fast directional multilevel summation for oscillatory kernels based on Chebyshev interpolation. *Journal of Computational Physics*, 231:1175–1196, 2012.

[4] Emmanuel J. Candès, Laurent Demanet, and Lexing Ying. A fast Butterfly algorithm for the computation of Fourier integral operators. *Multiscale Model. Simul.*, 7:1727–1750, 2009.

[5] Eric Michielssen and Amir Boag. A multilevel matrix decomposition algorithm for analyzing scattering from large structures. *IEEE Transactions on Antennas and Propagation*, 44(8):1086–1093, 1996.

[6] V. Rokhlin. Diagonal forms of translation operators for the Helmholtz equation in three dimensions. *Applied and Computational Harmonic Analysis*, 1(1):82 – 93, 1993.

[7] Hongwei Cheng, William Y. Crutchfield, Zydrunas Gimbutas, Leslie F. Greengard, J. Frank Ethridge, Jingfang Huang, Vladimir Rokhlin, Norman Yarvin, and Junsheng Zhao. A wideband Fast Multipole Method for the Helmholtz equation in three dimensions. *Journal of Computational Physics*, 216:300–325, 2006.

[8] Steffen Börm and Jens Melenk. Approximation of the high-frequency Helmholtz kernel by nested directional interpolation. *Numerische Mathematik*, 137(1):1–37, 10 2017.

[9] Steffen Börm. Directional H2-matrix compression for high-frequency problems. *Numerical Linear Algebra with Applications*, 24, 07 2017.

[10] Oscar P. Bruno and Leonid A. Kunyansky. A fast, high-order algorithm for the solution of surface scattering problems: Basic implementation, tests, and applications. *Journal of Computational Physics*, 169:80–110, 2001.

[11] E. Bleszynski, M. Bleszynski, and T. Jaroszewicz. AIM: Adaptive integral method for solving large-scale electromagnetic scattering and radiation problems. *Radio Science*, 31(5):1225–1251, 1996.

[12] Joel R. Phillips and Jacob K. White. A precorrected-FFT method for electrostatic analysis of complicated 3-d structures. *IEEE Transactions on computer-aided design of integrated circuits and systems*, 16(10):1059–1072, 1997.

[13] Lexing Ying, George Biros, Denis Zorin, and M. Harper Langston. A new parallel kernel-independent Fast Multipole Method. In *A New Parallel Kernel-Independent Fast Multipole Method*, 11 2003.

[14] Mario Bebendorf and Sergej Rjasanow. Adaptive low-rank approximation of collocation matrices. *Computing*, 70:1–24, 02 2003.

[15] Mathias Winkel, Robert Speck, Helge Hübner, Lukas Arnold, Rolf Krause, and Paul Gibbon. A massively parallel, multi-disciplinary Barnes–Hut tree code for extreme-scale n-body simulations. *Computer physics communications*, 183(4):880–889, 2012.

[16] Austin R. Benson, Jack Poulson, Kenneth Tran, Björn Engquist, and Lexing Ying. A parallel directional Fast Multipole Method. *SIAM Journal on Scientific Computing*, 36(4):C335–C352, 2014.

[17] A. Chandramowlishwaran, S. Williams, L. Oliker, I. Lashuk, G. Biros, and R. Vuduc. Optimizing and tuning the Fast Multipole Method for state-of-the-art multicore architectures. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–12, 2010.

[18] Rui-Qing Liu, Xiao-Wei Huang, Yu-Lin Du, Ming-Lin Yang, and Xin-Qing Sheng. Massively parallel Discontinuous Galerkin surface integral equation method for solving large-scale electromagnetic scattering problems. *IEEE Transactions on Antennas and Propagation*, 69(9):6122–6127, 2021.

[19] Ming-Lin Yang, Yu-Lin Du, and Xin-Qing Sheng. Solving electromagnetic scattering problems with over 10 billion unknowns with the parallel MLFMA. In *2019 Photonics Electromagnetics Research Symposium - Fall (PIERS - Fall)*, pages 355–360, 2019.

[20] Özgür Ergül and Levent Gurel. A hierarchical partitioning strategy for an efficient parallelization of the Multilevel Fast Multipole Algorithm. *IEEE Transactions on Antennas and Propagation*, 57(6):1740–1750, 2009.

[21] Caleb Waltz, Kubilay Sertel, Michael A. Carr, Brian C. Usner, and John L. Volakis. Massively parallel Fast Multipole Method solutions of large electromagnetic scattering problems. *IEEE Transactions on Antennas and Propagation*, 55(6):1810–1816, 2007.

[22] Luis Landesa, Jose Taboada, Fernando Obelleiro, Jose Rodriguez, José Carlos Gallego, and Andres Gomez. Solution of very large integral-equation problems with single-level FMM. *Microwave and Optical Technology Letters*, 51:2451 – 2453, 10 2009.

[23] Fangzhou Wei and Ali E. Yilmaz. A more scalable and efficient parallelization of the Adaptive Integral Method—Part I: Algorithm. *IEEE Transactions on Antennas and Propagation*, 62(2):714–726, 2014.

[24] Fangzhou Wei and Ali E. Yilmaz. A more scalable and efficient parallelization of the Adaptive Integral Method—Part II: BIOEM Application. *IEEE Transactions on Antennas and Propagation*, 62(2):727–738, 2014.

[25] Miloš Nikolić, Aleksandar Jović, Josip Jakić, Vladimir Slavnić, and Antun Balaž. *An Analysis of FFTW and FFTE Performance*, pages 163–170. Springer International Publishing, Cham, 2014.

[26] Nail A. Gumerov and Ramani Duraiswami. *Fast Multipole Methods for the Helmholtz Equation in Three Dimensions*. Elsevier Science, 2004.

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

[28] Jack Poulson, Laurent Demanet, Nicholas Maxwell, and Lexing Ying. A parallel Butterfly algorithm. *SIAM Journal on Scientific Computing*, 36(1):C49–C65, 2014.

[29] Dhairya Malhotra and George Biros. Algorithm 967: A distributed-memory Fast Multipole Method for volume potentials. *ACM Transactions on Mathematical Software (TOMS)*, 43(2):1–27, 2016.

[30] Michael S. Warren. 2HOT: An improved parallel hashed oct-tree n-body algorithm for cosmological simulation. In *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2013.

[31] Edwin Jimenez, Christoph Bauinger, and Oscar P Bruno. IFGF-accelerated integral equation solvers for acoustic scattering. *arXiv preprint arXiv:2112.06316*, 2021.

[32] Mark Bull, James Enright, Xu Guo, Chris Maynard, and Fiona Reid. Performance evaluation of mixed-mode OpenMP/MPI implementations. *International Journal of Parallel Programming*, 38:396–417, 10 2010.

[33] N. Drosinos and N. Koziris. Performance comparison of pure MPI vs hybrid MPI-OpenMP parallelization models on SMP clusters. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, pages 15–, 2004.

[34] Dana Akhmetova, Roman Iakymchuk, Orjan Ekeberg, and Erwin Laure. Performance study of multithreaded MPI and OpenMP tasking in a large scientific code. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 756–765, 2017.

[35] James Reinders, Ben Ashbaugh, James Brodman, Michael Kinsner, John Pennycook, and Xinmin Tian. *Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL*. Springer Nature, 2021.

[36] P.S. Pacheco. *An Introduction to Parallel Programming*. Morgan Kaufmann, 2011.

[37] M.S. Warren and J.K. Salmon. A parallel hashed oct-tree n-body algorithm. In *Supercomputing '93:Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, pages 12–21, 1993.

[38] Thomas Rauber and Gudula Rünger. *Parallel Programming*. Springer, Berlin, Heidelberg, 2 edition, 2013.

[39] T. Sterling, M. Brodowicz, and M. Anderson. *High Performance Computing: Modern Systems and Practices*. Elsevier Science, 2017.