Towards Exascale Multiparticle Simulations

Srinath Kailasa

A thesis submitted in partial fulfillment of the requirements
for the degree Master of Philosophy

Department of Mathematics
University College London
October, 2022

Declaration

I, Srinath Kailasa, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the thesis.

Abstract

The past three decades have seen the emergence of so called 'fast algorithms' that are able to optimally apply and invert dense matrices that exhibit a special low-rank structure in their off-diagonal elements. Such matrices arise in numerous areas of science and engineering, for example in the linear system matrices of boundary integral formulations of problems from acoustics and electromagnetics to fluid dynamics, geomechanics and even seismology. In the best case matrices can be stored, applied and inverted in $O(N)$, in contrast to $O(N^2)$ for storage and application, and $O(N^3)$ for inversion when computed naively.

The unification of software for the forward and inverse application of these operators in a single set of open-source libraries optimised for distributed computing environments is lacking, and is the central concern of this research project. We propose the creation of a unified solver infrastructure that can demonstrates good weak scaling from local workstations to upcoming exascale machines. Developing high-performance implementations of fast algorithms is challenging due to highly-technical nature of their underlying mathematical machinery, further complicated by the diversity of software and hardware environments in which research code is expected to run.

This subsidiary thesis presents current progress towards this goal. Chapter (1) introduces the Fast Multipole Method (FMM), the prototypical fast algorithm for $O(N)$ matrix vector products, and discusses implementation strategies in the context of high-performance software implementations. Chapter (2) provides a survey of the fragmented software landscape for fast algorithms, before proceeding with a case study of a Python implementation of an FMM, which attempted to bridge the gap between a familiar and ergonomic language for researchers and achieving high-performance. The remainder of the chapter introduces Rust, our proposed solution for ergonomic and high-performance codes for computational science, and it concludes with an overview of a software output: Rusty Tree, a new Rust-based library for the construction of parallel octrees, a foundational datastructure for FMMs, as well as other fast algorithms. Chapter (3) introduces vectors for future research, specifically an introduction to fast algorithms and software for matrix inversion, and the potential pitfalls we will face in their implementation for performance, as well as an overview of a proposed investigation into the optimal mathematical implementation of field translations - a crucial component of a performant FMM. We conclude with a look ahead towards a key target application for our software, the solution of electromagnetic scattering problems described by Maxwell's equations that can demonstrate performance at exascale.

Contents

1 Introduction **1**
1.1 Motivation 1
1.2 From Analytic to Algebraic Hierarchical Fast Multipole Methods . . . 3

2 Designing Software for Fast Algorithms **12**
2.1 The Software Landscape 12
2.2 Case Study: PyExaFMM, a Python Fast Multipole Method 14
2.3 Rust for High Performance Computing 15
2.4 Case Study: RustyTree, a Rust based Parallel Octree 15

3 Looking Ahead **16**
3.1 Fast Direct Solvers on Distributed Memory Systems 16
3.2 Optimal Translation Operators for Fast Algorithms 16
3.3 Target Application: Maxwell Scattering 16

4 Conclusion **17**

A Appendix **18**
A.1 Deriving Local Expansion Coefficents from Multipole Expansion in $\mathbb{R}^2$ 18

Bibliography **19**

Introduction

1.1 Motivation

The motivation behind the development of the original fast multipole method (FMM), was the calculation of potentials in N-body problems,

$$\phi_j = \sum_{i=1}^{N} K(x_i, x_j) q_i \tag{1.1}$$

Consider electrostatics, or gravitation, where q_i is a point charge or mass, and the kernels are of the form $K(x, y) = log|x - y|$ in $\mathbb{R}^2$, or $K(x, y) = \frac{1}{|x-y|}$ in $\mathbb{R}^{\not\vdash}$. Similar sums appear in the discretised form of boundary integral equation (BIE) formulations for elliptic partial differential equations (PDEs), which are the example that motivates our research. Generically, an integral equation formulation can be written as,

$$a(x)u(x) + b(x) \int_{\Omega} K(x, y)c(y)u(y)dy = f(x), \;\; x \in \Omega \subset \mathbb{R}^d \tag{1.2}$$

where the dimension $d = 2$ or 3. The functions $a(x)$, $b(x)$ and $c(y)$ are given and linked to the parameters of a problem, $K(x, y)$ is some known kernel function and $f(x)$ is a known right hand side, $K(x, y)$ is associated with the PDE - either its Green's function, or the derivative. This is a very general formulation, and includes common problems such as the Laplace and Helmholtz equations. Upon discretisation with an appropriate method, for example the Nyström or Galerkin methods, we obtain a linear system of the form,

$$\mathsf{K}u = f \tag{1.3}$$

The key feature of this linear system is that K is *dense*, with non-zero off-diagonal elements. Such problems are also *globally data dependent*, in the sense that the calculation at each matrix element of the discretized system in the depends on all other elements. This density made numerical methods based on boundary integral equations prohibitively expensive prior to the discovery of so called 'fast algorithms', of which the FMM is the prototypical example. The naive computational complexity of storing a dense matrix, or calculating its matrix vector product is $O(N^2)$, and the complexity of finding its inverse is $O(N^3)$ with linear algebra techniques such as LU decomposition on Gaussian elimination, where N is the number unknowns.

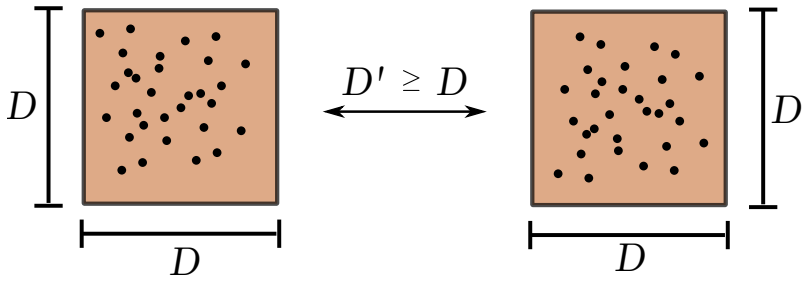


Figure 1.1: Given two boxes in $\mathbb{R}^2$, $\mathcal{B}_1$, $\mathcal{B}_2$, which enclose corresponding degrees of freedom, off diagonal blocks in the linear system matrix $\mathsf{K}_{\mathcal{B}_1\mathcal{B}_2}$ and $\mathsf{K}_{\mathcal{B}_2\mathcal{B}_1}$ are considered low-rank for the FMM when separated by a distance at least equal to their diameter, this is also known as 'strong admissibility'. Figure adapted from [31].

The critical insight behind the FMM, and other fast algorithms, is that one can compress physically distant interactions by utilising the rapid decay behaviour of the problem's kernel. A compressed 'low rank' representation can be sought in this situation, displayed in figure (1.1) for $\mathbb{R}^2$.

Using the FMM the best case the matrix vector product described by (1.1) and (1.3) can be computed in just $O(N)$ flops and stored with $O(N)$ memory. Fast algorithm based matrix inversion techniques display similarly optimal scaling in the best case. Given the wide applicability of boundary integral equations to natural sciences, from acoustics [39, 21] and electrostatics [38] to electromagnetics [8] fluid dynamics [35] and earth science [6]. Fast algorithms can be seen to have dramatically brought within reach large scale simulations of a wide class of scientific and engineering problems. The applications of FMMs aren't restricted to BIEs, as (1.1) shares its form with the kernel summations often found in statistical applications, the FMM has found uses in and computational statistics [3], machine learning [24] and Kalman filtering [26]. The uniting feature of these applications is their global data dependency.

Recent decades have seen the development of numerous mathematical techniques for the computation of fast algorithms. However given their broad applicability across various fields of science and engineering, this has not been met with a commensurate development of black box open-source software solutions which are easy to use and deploy by non-experts. This is not to say that there is an absence of research software for the FMM [23, 41, 37, 27, 32], or fast matrix inversion [34, 30, 22]. However, the software landscape is heavily fragmented, codes often arising out of a software or mathematical investigation with infrequent maintenance or development post-publication. Few attempts have been made to re-use data structures, or application programming interfaces (APIs) between projects, and source code is often poorly documented leading to little to no interoperability between projects. Furthermore, as codes are often written in compiled languages such as Fortran [32] or C++ [27, 41, 37], there is a relatively high software engineering barrier entry for community contributions, further discouraging widespread adoption amongst non-specialist academics and industry practitioners. Additionally, significant domain specific expertise in numerical analysis is required by users to discern the subtle differences between fast algorithm implementations, or indeed to write one independently.

Computer hardware and architectures continue to advance concurrently with advances in numerical algorithms. Recently, the exascale (capable of 10^{18} flops)

benchmark was achieved by Oak Ridge National Labs' Frontier machine[1]. With 9,472 AMD 64 core Trento nodes with a total of 606,208 compute cores, alongside 37,888 Radeon Instinct GPUs with a total of 8,335,360 cores, programming fast algorithms with their inbuilt global data dependency is challenging at a software level due to the communication bottlenecks imposed by the necessary all to all communications. Furthermore, the dense matrix operations required by fast algorithms require delicate tuning to fully take advantage of memory hierarchies on each node. Currently there exist very few open-source fast algorithm implementations that are capable of being deployed on parallel machines [27, 41], or take advantage of a heterogenous CPU/GPU environments [41]. In fact for fast inverses there doesn't yet exist an open-source parallel implementation. Furthermore, developers must using existing codes must employ careful consideration in order to successfully compile the software in each new hardware environment they encounter, from desktop workstations to supercomputing clusters.

Resultantly, researchers who may want to write application code that takes advantage of fast algorithms as a black box without the necessary software or numerical analysis expertise to implement their own have few choices, and fewer still in a distributed computing setting. Identifying this as a significant barrier to entry for the adoption of fast algorithms in the wider community, we propose a new unified framework for fast algorithms, beginning with an implementation of a parallel FMM, which we introduce in the following section, designed for modern large scale supercomputing clusters. We emphasise our focus on ergonomic and malleable code, such that our code is easy to edit and deploy on a multitude of architectures while still achieving good scaling. With a key target application being the simulation of exascale boundary integral problems for electromagnetics, specified by Maxwell's equations.

1.2 From Analytic to Algebraic Hierarchical Fast Multipole Methods

The FMM as originally presented has since been extended into a broad class of algorithms with differing implications for practical implementations. We consider the problem in its most generic form by returning to the matrix vector product (1.1).

We consider a non-oscillatory problem with a Laplace kernel from electrostatics to introduce the ideas behind the FMM, as in the original formulation [18]. Given an N-body evaluation of electrostatic potentials, we let $\{x_i\}_{i=1}^{N} \in \mathbb{R}^d$ denote the set of locations of charges of strength q_i, where $d = 2$ or $d = 3$. Our task is then to evaluate potentials, ϕ_j for $i = 1, 2, ..., N$. We can without loss of generality take the value of $K(x, x) = 0$. Denoting our square domain containing all points with Ω, we seek a matrix vector product of the form,

$$\phi = \mathsf{K}\mathsf{q} \tag{1.4}$$

where $\phi \in \mathbb{C}^N$, $\mathsf{q} \in \mathbb{C}^N$ and $\mathsf{K} \in \mathbb{C}^{N \times N}$. The idea is to compress the kernel interactions defined by $K(x, y)$ when x and y are distant. Consider the situation in figure (1.3) where we choose $\mathbb{R}^2$ for simplicity. Here we seek to evaluate the potential induced by the source particles, $\{y_j\}_{j=1}^{M}$, in Ω_s at the target particles, $\{x_i\}_{i=1}^{L}$ in Ω_t.

[1] https://www.olcf.ornl.gov/frontier/

$$\phi_i = \sum_{j=1}^{L} K(x_i, y_j) q_j, \;\; i = 1, 2...M \tag{1.5}$$

As the sources and targets are physically distant, we can apply a low-rank approximation for the kernel as a sum of tensor products,

$$K(x, y) \approx \sum_{p=0}^{P-1} B_p(x) C_p(y), \;\; \text{when } x \in \Omega_t, y \in \Omega_s \tag{1.6}$$

where P is called the 'expansion order', or 'interaction rank'. We introduce the index sets I_s and I_t which list the points inside Ω_s and Ω_t respectively, and consider a generic approximation by tensor products where,

$$\hat{q}_p = \sum_{j \in I_s} C_p(x_j) q_j, \;\; p = 0, 1, 2, ..., P-1 \tag{1.7}$$

this is valid as K is smooth in the far field. Using this, we evaluate the approximation of the potential at the targets as,

$$\phi_i \approx \sum_{p=1}^{P-1} B_p(x_i) \hat{q}_p \tag{1.8}$$

In doing so we see that we accelerate (1.5) from $O(ML)$ to $O(P(M+L))$. As long as we choose $P \ll M$ and $P \ll L$, we will recover an accelerated matrix vector product. The power of the FMM, and similar fast algorithms, is that we can recover the potential in Ω_t with high accuracy even when P is small. We deliberately haven't stated how we calculate B_p or C_p. In Greengard and Rokhlin's FMM these took the form of analytical multipole and local expansions of the kernel function [18].

To demonstrate this we derive an expansion in the $\mathbb{R}^2$ case, taking c_s and c_t as the centres of Ω_s and Ω_t respectively,

$$\begin{aligned} K(x, y) = \log(x - y) &= \log((x - c_s) - (y - c_s)) \\ &= \log(x - c_s) + \log(1 - \frac{y - c_s}{x - c_s}) \\ &= \log(x - c_s) - \sum_{p=1}^{\infty} \frac{1}{p} \frac{(y - c_s)^p}{(x - c_s)^p} \end{aligned} \tag{1.9}$$

where the series converges for $|y - c_s| < |c - c_s|$. We note (1.9) is exactly of the form required with $C_p(y) = -\frac{1}{p}(y - c_s)^p$ and $B_p(x) = (x - c_s)^{-p}$. We define a 'multipole expansion' of the charges in Ω_s as a vector $\hat{\mathbf{q}}^{\mathbf{s}} = \{\hat{q}_p^s\}_{p=0}^{P-1}$,

$$\begin{cases} \hat{q}_0^s = \sum_{j \in I_s} q_j \\ \hat{q}_p^s = \sum_{j \in I_s} -\frac{1}{p}(x_j - c_s)^p q_j, \;\; p = 1, 2, 3..., P-1 \end{cases} \tag{1.10}$$

The multipole expansion is a representation of the charges in Ω_s and can be truncated to any required precision. We can use the multipole expansion in place of a direct calculation with the particles in Ω_s. As the potential in Ω_t can be written as,

$$\phi(x) = \sum_{j \in I_s} K(x, y) q_j = \log(x - c_s)\hat{q}_0^s + \sum_{p=1}^{\infty} \frac{1}{(x - c_s)^p} \hat{q}_p^\sigma \tag{1.11}$$

Greengard and Rokhlin also define a local expansion centered on Ω_t, that represents the potential due to the sources in Ω_s.

$$\phi(x) = \sum_{p=1}^{\infty} (x - c_t)^p \hat{\phi}_p^t \tag{1.12}$$

with a simple computation to derive the local expansion coefficients $\{\hat{\phi}_p^t\}_{p=0}^{\infty}$ from $\{\hat{q}_p^s\}_{p=0}^{P-1}$ (see app. A.1).

For our purposes it's useful to write the multipole expansion in linear algebraic terms as a linear map between vectors,

$$\hat{\mathsf{q}}^s = \mathsf{T}_s^{P2M} \mathsf{q}(I_s) \tag{1.13}$$

where T_s^{P2M} is a $P \times N_s$ matrix, analogously for the local expansion coefficients we can write,

$$\hat{\phi}^t = \mathsf{T}_{t,s}^{M2L} \hat{\mathsf{q}}^s \tag{1.14}$$

where $\mathsf{T}_{t,s}^{M2L}$ is a $P \times P$ matrix, and the calculation of the final potentials as,

$$\phi^t = \mathsf{T}_t^{L2P} \hat{\phi}^t \tag{1.15}$$

where T_t^{L2P} is a $N_t \times P$ matrix. Here we denote each *translation* operator, T^{X2Y}, with a label read as 'X to Y' where L stands for local, M for multipole and P for particle. Written in this form, we observe that one could use a different method to approximate the translation operators than explicit kernel expansions to recover our approach's algorithmic complexity, and this is indeed the main difference between different implementations of the FMM.

We have described how to obtain linear complexity when considering two isolated nodes, however in order to recover this for interactions between *all particles* with we rely on a hierarchical partitioning of Ω using a data structure from computer science called a *quadtree* in $\mathbb{R}^2$ or an *octree* in R^3. The defining feature of these data structures is a recursive partition of a bounding box drawn over the region of interest (see fig. 1.2). This 'root node' is subdivided into four equal parts in $\mathbb{R}^2$ and eight equal parts in $\mathbb{R}^3$. These 'child nodes' turn are recursively subdivided until a user defined threshold is reached based on the maximum number of points per leaf node. These trees can be 'adaptive' by allowing for non-uniform node sizes, and 'balanced' to enforce a maximum size constraint between adjacent nodes [36].

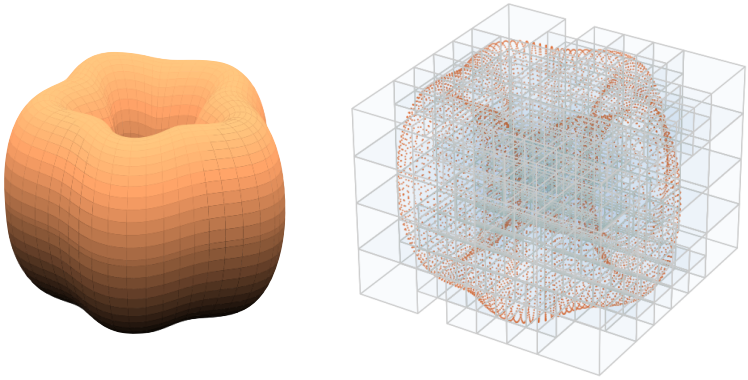


Figure 1.2: An adaptive octree for random point data placed on the surface of a 'wiggly torus' test geometry. The user defines the level of recursion via a threshold for the maximum number of particles in a given node.

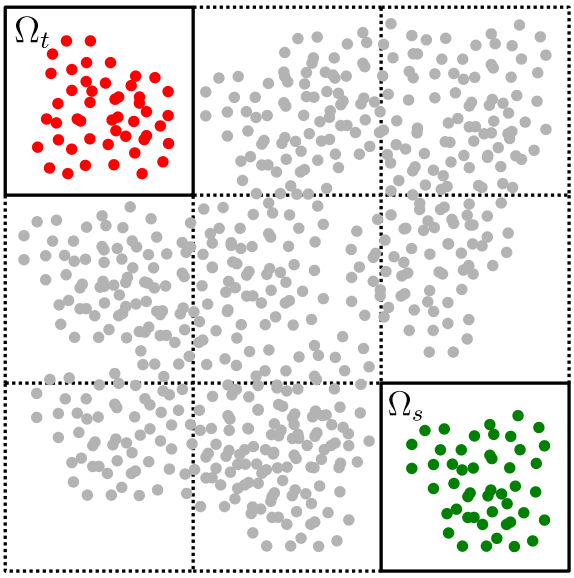


Figure 1.3: Two well separated clusters Ω_t and Ω_s where we can apply a low-rank approximation.

In addition to the T^{P2M}, T^{M2L} and T^{L2P} the FMM also require operators that can translate the expansion centre of a multipole or local expansion, T^{L2L}, T^{M2M}, an operator that can form a local expansion from a set of points T^{P2L}, and apply a multipole approximation to a set of points, T^{M2P}, finally we need define a $P2P$ operator as short hand for direct kernel evaluations. Algorithm (1) provides a brief sketch of the full FMM algorithm.

In its original analytical form the applicability of the FMM is limited by the requirement for an explicit multipole and local expansions, as well as a restriction to matrix vector products. This lead to the development of algebraic variants that operate on the matrix represented by (1.4) directly, without the need for geometrical considerations via an octree. Examples include, the $\mathcal{H}$ matrix [20], $\mathcal{H}^2$ matrix [5], hierarchically semi-separable (HSS) [7], and hierarchically off-diagonal low-rank (HODLR) [2] matrices.

These methods all represent the system matrix in (1.4) using a stored hierarchical matrix factorisation. Once this factorisation is computed, it can be used again - allowing for simple extensions to matrix-matrix products. Furthermore, an explicit

Algorithm 1 Adaptive Fast Multipole Method: FMM literature distinguishes between types of relationships between neighbouring nodes with the concept of *interaction lists*. There are four such lists for a given node B, called V, U, W and X. For a leaf node B, the U list contains B itself and leaf nodes adjacent to B. and the W list consists of the descendants of B's neighbours whose parents are adjacent to B. For non-leaf nodes, the V list is the set of children of the neighbours of the parent of B which are not adjacent to B, and the X list consists of all nodes A such that B is in their W lists. The non-adaptive algorithm is similar, however the W and X lists are empty.

N is the total number of points
s is the maximum number of points in a leaf node.

Step 1: Tree construction
for each node B in *preorder* traversal of tree **do**
subdivide B if it contains more than s points.
end for
for each node B in *preorder* traversal of tree **do**
construct *interaction lists*, U, V, X, W
end for

Step 2: Upward Pass
for each leaf node B in *postorder* traversal of the tree **do**
P2M: compute multipole expansion for the particles they contain.
end for
for each non leaf node B in *postorder* traversal of the tree **do**
M2M: form a multipole expansion by translating the expansion centre of its children to its centre and summing their multipole expansion coefficients.
end for

Step 3: Downward Pass
for each non-root node B in *preorder* traversal of the tree **do**
M2L: translate multipole expansions of nodes in B's V list to a local expansion at B.
P2L: translate the charges of particles in B's X to the local expansion at B.
L2L: translate B's local expansion to its children by translating its expansion centre to the centre of its children, and assigning the same coefficients.
end for
for each leaf node B in *preorder* traversal of the tree **do**
P2P: directly compute the local interactions using the kernel between the particles in B and its U list.
L2P: translate local expansions for nodes in B's W list to the particles in B.
M2P: translate the multipole expansions for nodes in B's W list to the particles in B.
end for

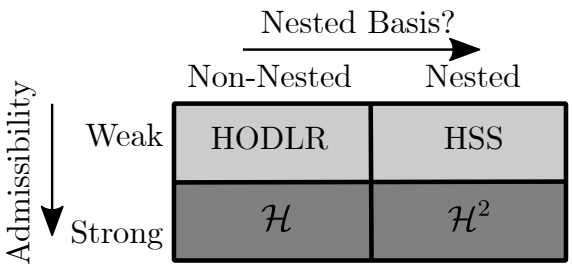


Figure 1.4: Matrix containing common algebraic alternatives to the FMM, adapted from [1].

matrix forms also allows for optimal matrix inversion algorithms.

Consider an index set I, corresponding to the indices of all points $\{x_i\}_{i=1}^N$ in a given discretisation. The general approach of these methods is to partition I in such a way that we can exploit the low-rank interactions between distantly separated clusters of particles. Initially, an n-ary 'cluster tree' $\mathcal{T}_I$, with a set of nodes T_I, is formed such that

1. $T_I \subseteq \mathcal{P}(I) \setminus \{\emptyset\}$, meaning each node of $\mathcal{T}_I$ is a subset of the index set I, here $\mathcal{P}(I)$ denotes the power set of I.

2. I is the root of $\mathcal{T}_I$

3. The number of indices in a leaf node $\tau \in T_I$ is such that $|\tau| \leq C_{\text{leaf}}$ where C_{leaf} is a small constant.

4. Non leaf nodes τ have n child nodes $\{\tau\}_i^n$, and is formed of their disjoint union $\tau = \cup_{i=1}^n \tau_i$

Cluster trees may in general be n-ary, however common implementations are as binary trees. Using a cluster tree, one forms a 'block tree', $\mathcal{T}_{I\times I}$. Each node, or 'block', of a block tree, $N(\tau \times \sigma)$, corresponds to the clusters represented by indices $\tau \subseteq I$ and $\sigma \subseteq I$ respectively. The $\mathcal{H}$ and $\mathcal{H}^2$ representations are 'strongly admissible', meaning that well separated blocks as in figure (1.1) can be compressed. In contrast, the HSS and HODLR approaches are 'weakly admissible', and also consider adjacent clusters to be compressible. Admissibility is calculated by forming a bounding box around clusters, and checking their separation along each dimension [4]. Furthermore, as in the FMM which creates parent multipole expansions for a given node from that of its children, and vice versa for local expansions, an analagous approach is taken by $\mathcal{H}^2$ and HSS matrices, referred to as 'nested bases'. These different approaches are succinctly visualised in figure (1.4).

Expressed in this way the matrix implicit in the FMM (alg. (1)), can be seen to be a member of the $\mathcal{H}^2$ class of matrices. Therefore the algorithms developed for algebraic approaches for matrix vector products similarly recover optimal $O(N)$ scaling in the best case, with the additional benefit of easy extension to matrix matrix products, and matrix inversion in optimal complexity [4].

From a computational perspective, the trade-off between these analytical and algebraic approaches for the FMM are best expressed in terms of the ratio of compute (flops) to memory (bytes), or 'arithmetic intensity'. The analytical FMM has a high arithmetic intensity, due to its matrix free nature. However, using explicit $\mathcal{H}^2$ representations requires one to compute and store the full hierarchical matrix in

memory, resulting in increased memory movement costs, both vertically on a single node, and horizontally in a distributed memory setting.

A third 'semi-analytical' method, combining the best of the purely analytical and algebraic methods are known as 'black box', or 'kernel independent', FMMs [25, 11, 28]. These methods mimic the algorithmic structure of the analytical FMM, with its matrix free nature, however the low rank representations they construct are done so independently of the kernel, hence the name 'black box'. We contrast two common black box methods in box (1.5), the underlying difference being in their representation of translation operators. Black box methods are preferable from a software standpoint for three reasons:

1. Generic software interfaces can be built for a variety of kernels, unlike for analytical FMMs, as demonstrated by [37, 23].

2. They have reduced storage requirements in comparison to purely algebraic approaches.

3. the majority of computations are simple matrix vector products that are readily expressed with BLAS L2 operations, and therefore maximise cache efficiency

We emphasise that it is possible to extend the analytical FMM to more general kernels, such as the modified Laplacian [19], Stokes [12] and Navier [13], however software implementations are cumbersome, requiring the optimised calculation of special functions such as spherical harmonics, and are difficult to generalise for different problem settings. We have preferentially implemented the Kernel-Independent FMM of Ying et. al [25] in our previous software [23] as it has a particularly simple mathematical structure, and extends to a variety of kernels for elliptic partial differential equations of interest. Specifically, the Laplace, Stokes and Navier kernels alongside their modified variants [25], as well as the Helmholtz kernel in the low-frequency setting [37].

In terms of algorithm optimisation significant efforts have gone towards optimising T^{M2L} [29, 11, 25], which is the most time consuming translation operator. It is computed for each non-leaf node between level one and the leaf level of a tree, it therefore grows as $O(N)$. In $\mathbb{R}^3$, the V list for a node, containing its parent's neighbors' children that are non-adjacent to the node, can be up to $|V| = 6^3 - 3^3 = 189$. However if a kernel exhibits translational invariance, such that

$$K(x, y) = K(x - y)$$

which is the case for the kernels mentioned above, we recognise that there are just $7^3 - 3^3 = 316$ unique T^{M2L} operators per level. However, Messner et. al [29] show that even these 316 operators are permutations of a subset of only 16 unique operators. Permutations corresponding to reflections across various planes (for example $x = 0$, $y = 0$, $z = 0$, etc). We can therefore write a block matrix of *all* unique T^{M2L} for a given level,

$$T^{M2L} = \left[T_1^{M2L}|T_2^{M2L}|...|T_{16}^{M2L}\right] \tag{1.16}$$

where T_i^{M2L} corresponds to the i^{th} *transfer vector* describing a unique T^{M2L}. This block matrix is efficiently compressed using a truncated SVD, which is known

to produce an optimal low rank approximation of a rank N matrix, where the new rank, r, is chosen such that $r \ll N$. This is a common optimisation used in both black box [25, 11] and analytical FMMs [17]. By precomputing and storing a compressed T^{M2L} for each level, the T^{M2L} operator for each node can be formed as a single BLAS level 2 operation at runtime by looking up the relevant compressed operators corresponding to its V list. The T^{M2M}, T^{L2L} can be similarly stored on a level by level basis, reducing the pre-computation time for operators to $O(d)$, where d is the depth of an quad/octree. For homogenous kernels, for which,

$$K(\alpha x, \alpha y) = \alpha^m K(x, y)$$

pre-computations can performed for a single level, and scaled to different levels, reducing pre-computation complexity to $O(1)$. Such optimisations further separate runtime performance between implementations of algebraic and semi-analytical methods, as the transfer operators T^{M2L} correspond to duplicate blocks in an algebraic $\mathcal{H}^2$ which are redundantly computed and stored in implementations [16].

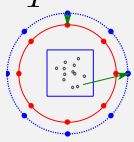
A significant challenge in practical FMM implementations is handling the global data dependency encapsulated in (1.5). Historical single-core architectures have always suffered from the Von Neumann bottleneck, whereby data is loaded from main memory slower than the processor is able to absorb it. However, with the breakdown of Dennard scaling[2] from around 2006 and the subsequent multicore revolution, the disparity between data movement costs and computation have been even further exacerbated. On modern hardware accelerators are able to achieve $O(1e12)$ flop rates in double precision, but are bandwidth limited to $O(1e11)$ bytes/second, meaning that an order of magnitude more operations have to performed for every byte retrieved from memory, to remain efficient. Dongarra et. al [9], go as far as to suggest that the dominant role of communication costs on emerging hardware, especially in the context of exascale architectures, mean that complexity analysis of distributed algorithms based on flop counts are redundant entirely. Indeed, in a distributed memory setting global reductions are often the largest performance limiting factor [9]. The data dependency of the FMM is expressed in the quad/octree, with significant global reductions required for translation operators. It is therefore critical to implement highly optimised algorithms for tree construction in a distributed setting, however we defer a discussion on this until section 2.4.

We conclude by mentioning that the FMM and its variants are not the only technique available to accelerate (1.1), for problems in which only uniform resolution is required the FFT, which has corresponding distributed memory implementations [14]. However, we are commonly interested in multiscale problems in which neighbouring nodes can be of differing sizes. In this setting, multigrid methods have shown slower convergence in comparison to the FMM [40, 15]. Furthermore, a multigrid approach does not allow for a re-use of FMM data structures (sec. 3.1) for other fast algorithms we are interested in implementing, and aim to present as a unified framework with maximum code re-use. We observe that the choice between analytical, semi-analytical and algebraic FMM variants may have a significant impact on the ultimate implementation, and performance, this is something we explicitly measure in section 2.1. We conclude that the semi-analytical Kernel-Independent FMM of Ying et. al [25], offers a good compromise between ease of of software design, mathematical simplicity, and problem generality.

[2]Dennard Scaling is the insight that the power density of transistors appeared to remain constant as they grew smaller.

Kernel-Independent FMM (KIFMM)
The KIFMM approximates the multipole expansion for a given leaf node containing particles $\{y_j\}_j^N$ with charges $\{q_j\}_j^N$ by evaluating their potential directly (1.4) at corresponding evenly spaced points on a 'check surface', displayed in blue.

T^{P2M}

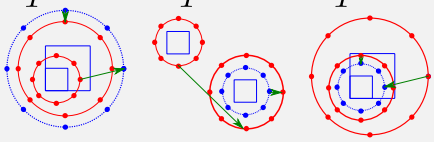


This is matched to N_e 'equivalent charges', q^e, placed evenly on a the (red) equivalent surface at $\{x_i\}_i^{N_e}$. These surfaces are taken to be circles in $\mathbb{R}^2$ and cubes in $\mathbb{R}^3$. They perform a Tikhonov-regularised least squares fit to calculate q^e, which plays the role of the *multipole expansion.*

$$\mathsf{q}^e = (\alpha I + \mathsf{K}^*\mathsf{K})^{-1}\mathsf{K}\mathsf{q}^{\text{node}}$$

where q^{node} are the charges in the leaf node. Using a similar framework, involving equivalent and check surfaces, the T^{M2M}, T^{M2L}, T^{L2L} and T^{L2P} operators are also calculated with Tikhonov-regularised least squares fitting, and are sketched below.

T^{M2M} T^{M2L} T^{L2L}



The KIFMM is designed for non-oscillatory kernels, such as the Laplace kernel of our didactic example, though in practice works well with low-frequency oscillatory problems, with extensions to high frequency problems [10].
The least-squares fit can be pre-computed for each given geometry, and re-used. Additionally, as the KIFMM decomposes into a series of matrix vector products, it is well suited to software implementations.

Black-Box FMM (bbFMM)
An n point interpolation scheme for the kernel is constructed sequentially over each variable,

$$K(x,y) \approx \sum_{l=1}^{n} K(x_l, y) w_l(x)$$

$$K(x,y) \approx \sum_{l=1}^{n}\sum_{m=1}^{n} K(x_l, x_y) w_l(x) w_m(x)$$

with coefficients $w_l(x)$ and $w_m(x)$. The bbFMM uses first kind Chebyshev polynomials, T_n to interpolate the kernel, defined as,

$$T_n(x) = \cos(n\theta), \text{ where } x = \cos(\theta)$$

over the closed interval $x \in [-1, 1]$. The roots, $\{\bar{x}_m\}$, known as Chebyshev nodes are defined as,

$$\bar{x}_m = \cos(\theta_m) = \cos(\frac{(2m-1\pi)}{2n})$$

This is a well known, stable, uniformly convergent, interpolation scheme, that doesn't suffer from Runge's phenomenon [11]. An n point Chebyshev approximation to a given function, $g(x)$ with $p_{n-1(x)}$, can be written as,

$$p_{n-1}(x) = \sum_{k=1}^{n-1} c_k T_k(x)$$

where, $c_k = \begin{cases} \frac{2}{n}\sum_{l=1}^{n} g(\bar{x}_l) T_k(\bar{x}_l), & \text{if } k > 0 \\ \frac{1}{n}\sum_{l=1}^{n} g(\bar{x}_l), & \text{if } k = 0 \end{cases}$
we recognise,

$$p_{n-1}(x) = \sum_{l=1}^{n} g(\bar{x}_l) S_n(\bar{x}_l, x)$$

$$\text{with, } S_n(x,y) = \frac{1}{n} + \frac{2}{n}\sum_{k=1}^{n-1} T_k(x) T_k(y)$$

When applied to our generic interpolation for the kernel,

$$K(x,y) \approx \sum_{l=1}^{n}\sum_{m=1}^{n} K(\bar{x}_l, \bar{y}_m) S_n(\bar{x}_l, x) S_n(\bar{y_m}, y)$$

which can be substituted into (1.5), recovering linear complexity as long as the number of Chebyshev nodes is small, $n \ll N$.

Figure 1.5: The formulations of the kernel-independent FMM methods, the KIFMM [25] and the bbFMM [11].

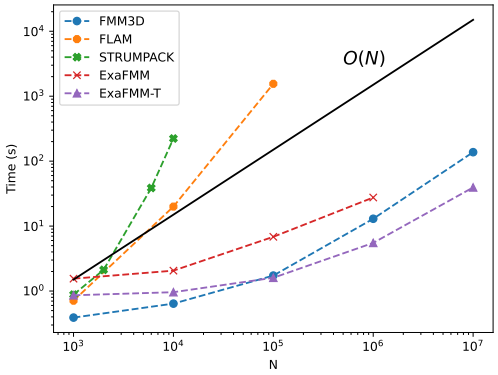
Designing Software for Fast Algorithms

2.1 The Software Landscape

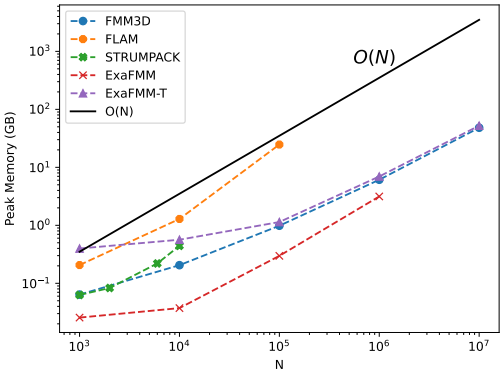
The software landscape for fast algorithms is fragmented. Development is split amongst a plurality of groups each operating siloed infrastructure. There are no parallel open-source softwares for the fast construction and use of strongly admissible $\mathcal{H}$ and $\mathcal{H}^2$ matrices for shared-memory systems. However, the 'STRUMPACK' [16] package does offer an MPI accelerated HODLR and HSS matrix package. For $\mathcal{H}$ and $\mathcal{H}^2$ matrices, Ho et. al provide 'FLAM' [22], a multithreaded single node implementation of an alternative factorization based on 'recursive skeletonisation'. We defer a discussion of recursive skeletonisation until section 3.1, for our current purposes we can assume it's another hierarchical format for $\mathcal{H}^2$ which allows for $O(N)$ matrix vector products and inversions. In STRUMPACK, we use a HSS matrix approximation, as noted in [40] this isn't strictly appropriate for (1.5) as it's a strongly admissable problem. Regardless of this, STRUMPACK offers an understanding of the performance of a highly optimised parallel algebraic solver. For FMMs the situation is better, with a greater number of open source projects in active development. The 'Fast Algorithms' project[1] have developed multithreaded single-node analytical FMMs in $\mathbb{R}^2$and $\mathbb{R}^3$ [33, 32]. Another set of leading implementations are provided by the ExaFMM project[2], with the MPI accelerated analytical, 'ExaFMM' [41], and the multithreaded single-node kernel-independent FMM, 'ExaFMM-T' [37]. All the above software, except FLAM, are written in either Fortran or C++. FLAM, written in Matlab, is not designed for high performance. However, it remains the only publicly available multithreaded implementation of fast algorithms for $\mathcal{H}^2$ matrices, and we include it for comparison.

We compare these softwares for the calculation of (1.5) with N randomly distributed particles placed in a unit cube, $[0,1]^3$, with uniform charge, $q_i = 1$, in figure (2.1). Experiments are taken on a single-node AMD Ryzen Threadripper 3970X 32 core Processor, with 250GB of memory. To avoid thread oversubscription in STRUMPACK and ExaFMM, both designed for multi-node systems, the maximum number of OpenMP threads is restricted to one. In comparing these softwares we aimed to emulate the workflow of a typical user evaluating the differences between software packages, and therefore restricted our study to runs which converged in less than 2×10^3 s. Figure (2.1a) plots the time to compute (1.5), including the time to create all required data structures. For the algebraic softwares, FLAM and STRUMPACK, this includes the time to factorise and store an explicit ma-

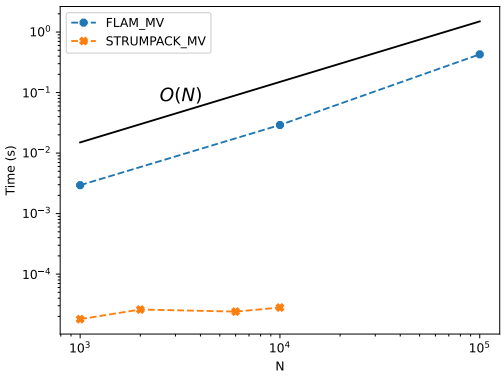
[1] https://github.com/fastalgorithms
[2] https://github.com/exafmm



(a) Runtime



(b) Peak memory



(c) Matrix vector product time for algebraic methods.

Figure 2.1: Comparison of parallel software for fast algorithms, used to caclulate (1.5). Simulations were compared for convergence times $\leq 2 \times 10^3$ s.

trix representation of (1.5). The factorization scales as $O(N^2)$ for both softwares, which lies in contrast with the linear scaling of their matrix vector products once factorised (fig. (2.1c))s. We note that for STRUMPACK it is difficult to recover a linear scaling for the matrix vector product in our experiments as convergence times for moderate problem sizes, $N \geq 10^5$, exceeded 2×10^3 s. The algebraic software currently available in the open source is completely impractical for computing even moderately sized matrix vector products, i.e. $N = O(10^6)$. One might argue that their expensive factorization cost is worth it if one proceeds to compute (1.5) multiple times, or if one wants to quickly find an inverse. However, the FMM softwares tested beat both algebraic softwares by several orders of magnitude, making it much more tenable to apply them multiple times if required, and to find matrix inverses via an iterative Krylov type methods.

The FMM softwares also appear to exhibit markedly different behaviours from each other, which are difficult to explain with purely theoretical considerations. Each is run with an expansion order $P = 10$, as expected from theory the analytical FMMs, ExaFMM and FMM3D, consume less memory than the kernel-independent ExaFMM-T (fig. (2.1b)). However, for larger problem sizes this difference is marginal between FMM3D and ExaFMM-T. For the largest problem size tested, $N = 10^7$, ExaFMM-T is roughly 3.5 times faster than FMM3D with a similar memory footprint. Strangely the analytical ExaFMM fails to converge for $N = 10^7$, as its memory requirements exceed 250 GB. Each software implements numerous tech-

niques for acceleration, however these don't necessarily overlap. ExaFMM-T for example uses optimized algorithms for the calculation of inverse square roots [27], as well as using SIMD vectorisation with SSE/AVX and AVX-512 compatibility. Similarly by stacking T^{M2L} as in (1.16), they optimise cache-reuse [37]. However, FMM3D and ExaFMM are presented opaquely to users, with little openly available documentation of their optimisations.

Another point of difference between the softwares is in their usability, defined by the ease with which one can install software in different environments, and learn about its API. This important work is often dismissed as mere software engineering, but it is as crucial for the dissemination and uptake of scientific software as raw performance. Of the softwares tested, FMM3D and ExaFMM-T standout with publicly available websites documenting their installation and APIs, code examples, as well as bindings to higher level languages such as Python, Julia and Matlab. This lies in contrast to ExaFMM, which is poorly documented, and clearly still an 'alpha' project, not ready for wide scale distribution. STRUMPACK, though relatively well documented, does not support wrappers to higher-level languages. Written in C++, its documentation is largely a description of its object hierarchy, with few examples documenting real world use cases.

- large number of projects, not clear how they map from theory.

- unclear whether differences are implementational or due to underlying algorithm

- publicly available literature doesn't make it clear which software can do what problem.

- many/most don't document their optimisations

- often no bindings to other languages.

- no high performance library for algebraic methods, that's good and extends to $\mathcal{H}/\mathcal{H}$ matrices

- No overlap between software projects has lead to redundant development of high performance optimisations.

- Building codes is not all easy. STRUMPACK, distribution via spack, but otherwise large list of custom dependencies. Traditional C++ dependency hell to get set up. Other software isn't designed for high-performance, and distributed with custom instructions to build from source. SPACK packages, need to look into how they are made and how easy they are to distribute.

- JIT compilers, and the rise of performant development in higher level languages. Can we develop an ergonomic FMM? source code all in Python? ... Next chapter.

- If not Python then what? C++ projects have a documented history of overcomplicating data structures. We want it to be as usable as possible, with few complex hierarchies, and designed for data oriented design (vis object oriented design).

2.2 Case Study: PyExaFMM, a Python Fast Multipole Method

- Summarise pyexafmm paper, and what it hoped to discover - Can we use JIT compilers to build cse applications? Answer, probably not.

- Can largely copy from Numba section of paper, its pitfalls, as well as the section on PyExaFMM's API and parallelization strategies.

- Conclude with idea that an alternative is necessary, but going back to C++ isn't the right option.

- List what we ideally want from a language for scientific computing. Speed is one thing, but we also want maintainability, easy testing, building on different environments, Rust...

2.3 Rust for High Performance Computing

- Summary of Rust's core features for computational science.
- cargo, code organisation features, traits system, python interfacing.
- Rebuke common misconceptions: safety (bounds checking), lack of appropriate libraries for numerical data.
- Highlight what actually is missing, e.g. rust-native tools (linear algebra, MPI etc) - and what's being done about it.
- When is using rust appropriate? When is it not?
- Why are we turning to rust for this project?

2.4 Case Study: RustyTree, a Rust based Parallel Octree

- Case study for Rusty tree on different HPC environments and architectures.
- Explain criticality of tree for FMM, where the global reductions are comms bottlenecks in the FMM.
- introduce tree algorithms (parallel sorting, parallel tree construction, load balancing) as well as idea and reasoning behind morton encoding. How do we do this fast?
- The novelty isn't the fact that it's a parallel octree, of which there are many, it's that it's one that you can use easily from Python, and deploy to different HPC environments and architectures, with a simple API.
- Scaling test on a large HPC system (Myriad/Archer2) scale to a very large tree (O(1e9)) points, use as a Python package.
- Talk about the ease of writing a Python interface, and how this interoperability works. Talk about rsMPI project, it's important as this is an example that makes installation harder than it needs to be as it's a C shim - and that this is an example of a (relative) pitfall as an early adopter.
- Contrast with existing tree libraries, their performance on different architectures, and how easy they are to install and edit. i.e. contrast how malleable they are.
- Conclude with vindication of Rust as a HPC language, due to its positive features.

Looking Ahead

- Towards a fully distributed fast solver infrastructure
    - Explain the context of the project, and how we plan to achieve its goals.

3.1 Fast Direct Solvers on Distributed Memory Systems

- Introduce the logic behind fast direct solvers via a short literature survey of the most popular methods.
    - Introduce RS-S and skeletonization based approaches, why these are good (proxy compression, can re-use octree data structure, work with moderate frequency oscillatory problems, straightforward to parallelize)
    - Introduce current state of the art work with Manas on proxy compression for Helmholtz problems.
    - Conclude with future plans for fast direct solver using our Galerkin discretized BIE.

3.2 Optimal Translation Operators for Fast Algorithms

- Translation operators, what are they, and what are the different approaches currently used.
    - What are the trade-offs of different approaches?
    - Can I write some quick software for the quick comparison of translation operators - maybe in Python, on top of RustyTree? This would allow me to get some graphs to compare between approaches. If this is too much work, I will have to just compare the approaches in words.

3.3 Target Application: Maxwell Scattering

- Very brief summary of the Maxwell scattering problem, how we will form the BIE, the representation formulae we'll use, and how the integral operator will be discretised.
    - Overview of what kind of problems this would help us solve?

Conclusion

- Short monograph summarising near term (translation operators, algebraic fmm) and longer term (inverse library) goals. Talk about recent achievements and results, to demonstrate that the goals are achievable in the time remaining.

Appendix

A.1 Deriving Local Expansion Coefficents from Multipole Expansion in $\mathbb{R}^2$

Bibliography

[1] Sivaram Ambikasaran. *Fast algorithms for dense numerical linear algebra and applications*. Stanford University, 2013.

[2] Sivaram Ambikasaran and Eric Darve. "An O(Nlog N) Fast Direct Solver for Partial Hierarchically Semi-Separable Matrices". In: *Journal of Scientific Computing* 57.3 (2013), pp. 477–501.

[3] Sivaram Ambikasaran et al. "Large-scale stochastic linear inversion using hierarchical matrices". In: *Computational Geosciences* 17.6 (2013), pp. 913–927.

[4] Steffen Börm, Lars Grasedyck, and Wolfgang Hackbusch. "Introduction to hierarchical matrices with applications". In: *Engineering analysis with boundary elements* 27.5 (2003), pp. 405–422.

[5] Steffen Börm and Wolfgang Hackbusch. "A short overview of H2-matrices". In: *Proceedings in applied mathematics and mechanics* 2.1 (2003), pp. 33–36.

[6] Stéphanie Chaillat, Marc Bonnet, and Jean-François Semblat. "A multi-level fast multipole BEM for 3-D elastodynamics in the frequency domain". In: *Computer Methods in Applied Mechanics and Engineering* 197.49-50 (2008), pp. 4233–4249.

[7] Shiv Chandrasekaran et al. "A fast solver for HSS representations via sparse matrices". In: *SIAM Journal on Matrix Analysis and Applications* 29.1 (2007), pp. 67–81.

[8] Eric Darve and Pascal Havé. "A fast multipole method for Maxwell equations stable at all frequencies". In: *Philosophical Transactions of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences* 362.1816 (2004), pp. 603–628.

[9] Jack Dongarra et al. "With extreme computing, the rules have changed". In: *Computing in Science & Engineering* 19.3 (2017), pp. 52–62.

[10] Björn Engquist and Lexing Ying. "Fast directional multilevel algorithms for oscillatory kernels". In: *SIAM Journal on Scientific Computing* 29.4 (2007), pp. 1710–1737.

[11] William Fong and Eric Darve. "The black-box fast multipole method". In: *Journal of Computational Physics* 228.23 (2009), pp. 8712–8725.

[12] Yuhong Fu and Gregory J Rodin. "Fast solution method for three-dimensional Stokesian many-particle problems". In: *Communications in Numerical Methods in Engineering* 16.2 (2000), pp. 145–149.

[13] Yuhong Fu et al. "A fast solution method for three-dimensional many-particle problems of linear elasticity". In: *International Journal for Numerical Methods in Engineering* 42.7 (1998), pp. 1215–1229.

[14] Amir Gholami et al. “AccFFT: A library for distributed-memory FFT on CPU and GPU architectures”. In: *arXiv preprint arXiv:1506.07933* (2015).

[15] Amir Gholami et al. “FFT, FMM, or multigrid? A comparative study of state-of-the-art Poisson solvers for uniform and nonuniform grids in the unit cube”. In: *SIAM Journal on Scientific Computing* 38.3 (2016), pp. C280–C306.

[16] Pieter Ghysels et al. “STRUMPACK: Scalable Preconditioning using Low-Rank Approximations and Random Sampling”. In: ().

[17] Zydrunas Gimbutas and Vladimir Rokhlin. “A generalized fast multipole method for nonoscillatory kernels”. In: *SIAM Journal on Scientific Computing* 24.3 (2003), pp. 796–817.

[18] Leslie Greengard and Vladimir Rokhlin. “A fast algorithm for particle simulations”. In: *Journal of computational physics* 73.2 (1987), pp. 325–348.

[19] Leslie F Greengard and Jingfang Huang. “A new version of the fast multipole method for screened Coulomb interactions in three dimensions”. In: *Journal of Computational Physics* 180.2 (2002), pp. 642–658.

[20] Wolfgang Hackbusch. “A sparse matrix arithmetic based on H-matrices. part i: Introduction to H-matrices”. In: *Computing* 62.2 (1999), pp. 89–108.

[21] Sijia Hao, Per-Gunnar Martinsson, and Patrick Young. “An efficient and highly accurate solver for multi-body acoustic scattering problems involving rotationally symmetric scatterers”. In: *Computers & Mathematics with Applications* 69.4 (2015), pp. 304–318.

[22] Kenneth L Ho. “FLAM: Fast linear algebra in MATLAB-Algorithms for hierarchical matrices”. In: *Journal of Open Source Software* 5.51 (2020), p. 1906.

[23] Srinath Kailasa et al. “PyExaFMM: an exercise in designing high-performance softawre with Python and Numba”. In: *To Appear in Computing in Science and Engineering* 24.4 (2022).

[24] Dongryeol Lee, Richard Vuduc, and Alexander G Gray. “A distributed kernel summation framework for general-dimension machine learning”. In: *Proceedings of the 2012 SIAM International Conference on Data Mining*. SIAM. 2012, pp. 391–402.

[25] Denis Zorin Lexing Ying George Biros. “A kernel-independent adaptive fast multipole algorithm in two and three dimensions”. In: *Journal of Computational Physics* 196.2 (2004), pp. 591–626. DOI: `http://dx.doi.org/10.1016/j.jcp.2003.11.021`.

[26] Judith Yue Li et al. “A Kalman filter powered by-matrices for quasi-continuous data assimilation problems”. In: *Water Resources Research* 50.5 (2014), pp. 3734–3749.

[27] Dhairya Malhotra and George Biros. “PVFMM: A parallel kernel independent FMM for particle and volume potentials”. In: *Communications in Computational Physics* 18.3 (2015), pp. 808–830.

[28] Per-Gunnar Martinsson and Vladimir Rokhlin. “An accelerated kernel-independent fast multipole method in one dimension”. In: *SIAM Journal on Scientific Computing* 29.3 (2007), pp. 1160–1178.

[29] Matthias Messner et al. “Optimized M2L kernels for the Chebyshev interpolation based fast multipole method”. In: *arXiv preprint arXiv:1210.7292* (2012).

[30] Victor Minden. *strong-skel*. Version 0.1.0. Dec. 15, 2018. URL: https://github.com/victorminden/strongskel.

[31] Victor Minden et al. "A recursive skeletonization factorization based on strong admissibility". In: *Multiscale Modeling & Simulation* 15.2 (2017), pp. 768–796.

[32] Manas Rachh et al. *FMM3D*. Version 1.0.1. June 29, 2022. URL: https://github.com/flatironinstitute/FMM3D.

[33] Manas Rachh et al. *FMM3D*. Version 1.0.1. June 29, 2022. URL: https://github.com/flatironinstitute/FMM3D.

[34] Manas Rachh et al. *FMM3DBIE*. Version 0.1.0. Oct. 5, 2022. URL: https://github.com/fastalgorithms/fmm3dbie.

[35] Abtin Rahimian et al. "Petascale direct numerical simulation of blood flow on 200k cores and heterogeneous architectures". In: *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2010, pp. 1–11.

[36] Hari Sundar, Rahul S Sampath, and George Biros. "Bottom-up construction and 2: 1 balance refinement of linear octrees in parallel". In: *SIAM Journal on Scientific Computing* 30.5 (2008), pp. 2675–2708.

[37] Tingyu Wang, Rio Yokota, and Lorena A Barba. "ExaFMM: a high-performance fast multipole method library with C++ and Python interfaces". In: *Journal of Open Source Software* 6.61 (2021), p. 3145.

[38] Tingyu Wang et al. "High-productivity, high-performance workflow for virus-scale electrostatic simulations with Bempp-Exafmm". In: *arXiv preprint arXiv:2103.01048* (2021).

[39] William R Wolf and Sanjiva K Lele. "Aeroacoustic integrals accelerated by fast multipole method". In: *AIAA journal* 49.7 (2011), pp. 1466–1477.

[40] Rio Yokota, Huda Ibeid, and David Keyes. "Fast multipole method as a matrix-free hierarchical low-rank approximation". In: *International Workshop on Eigenvalue Problems: Algorithms, Software and Applications in Petascale Computing*. Springer. 2015, pp. 267–286.

[41] Yokota, Rio and Wang, Tingu and Zhang, Chen Wu and Barba, Lorena A. *ExaFMM*. Version 0.1.0. Oct. 5, 2022. URL: https://github.com/exafmm/exafmm.