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 **8**
2.1 The Software Landscape 8
2.2 Case Study: PyExaFMM, a Python Fast Multipole Method 8
2.3 Rust for High Performance Computing 8
2.4 Case Study: RustyTree, a Rust based Parallel Octree 9

3 Looking Ahead **10**
3.1 Fast Direct Solvers on Distributed Memory Systems 10
3.2 Optimal Translation Operators for Fast Algorithms 10
3.3 Target Application: Maxwell Scattering 10

4 Conclusion **11**

A Appendix **12**
A.1 Deriving Local Expansion Coefficents from Multipole Expansion in $\mathbb{R}^2$ 12

Bibliography **13**

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.

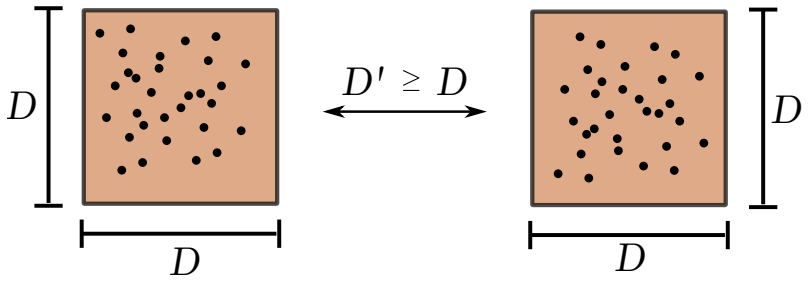

Figure 1.1: Given two boxes in $\mathbb{R}^2$, $\mathcal{B}_1$, $\mathcal{B}_2$, which enclose corresponding degrees of freedom, off diagonal blocks in the linear system matrix $\mathsf{K}_{\mathcal{B}_1\mathcal{B}_2}$ and $\mathsf{K}_{\mathcal{B}_2\mathcal{B}_1}$ are considered low-rank for the FMM when separated by a distance at least equal to their diameter, this is also known as 'strong admissibility'. Figure adapted from [12].

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 [19, 5] and electrostatics [18] to electromagnetics [3] fluid dynamics [15] and earth science [2]. 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 [1], machine learning [8] and Kalman filtering [9]. 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 [7, 20, 17, 10, 13], or fast matrix inversion [14, 11, 6]. 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 [13] or C++ [10, 20, 17], 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 [10, 20], or take advantage of a heterogenous CPU/GPU environments [20]. 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). Consider an N body evaluation of electrostatic potentials, which motivated the development of the original FMM of Greengard and Rokhlin [4]. 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 [FIGURE OF SINGLE LEVEL SITUATION] where we choose $\mathbb{R}^2$ for simplicity. Here we seek to evaluate the potential induced by the 'source particles, ${y_j}^M$, in Ω_s at the target particles, ${x_i}^L$ 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 $O(M + L)$ to $O(P(M + L))$. As long as we choose $P << M$ and $P << L$, 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 [4].

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 $\hat{\mathsf{q}}^{\mathsf{s}} = \{\hat{q}_p^s\}_{p=0}^{P-1}$,

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

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,

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

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, with a label read ''X to Y' where L stands for local, M for multipole and P for particle. Written in this form, we observe that we 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 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 [16].

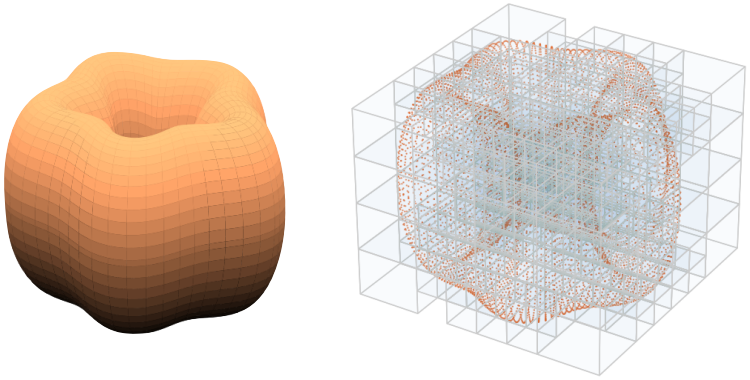

Figure 1.2: An adaptive octree for random point data placed on the surface of a 'wiggly torus' test geometry. The user defines the level of recursion via a threshold for the maximum number of particles in a given node.

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 FMM then consists of three steps: tree construction, a preorder 'bottom up' traveral, and a postorder 'top down' traversal, made up of eight operators. P2M, M2L and L2P have been introduced, however we also require P2L, M2L, L2L, L2P, M2P and P2P, applied once to each applicable node. The operators define interactions between a given 'target' node, and potentially multiple 'source' nodes from the tree. They are read as 'X to Y', where 'P' stands for particle(s), 'M' for multipole expansion and 'L' for local expansion. The non-adaptive case is similar, except the W and X lists are now empty.

Algorithm 1 Fast Multipole Method

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 and summing the expansion coefficients of the multipole expansions of its children.
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.
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

Designing Software for Fast Algorithms

- Monograph on the complexities involved in designing software that is performant & usable for the majority of researchers who may not be software experts.

- Get more examples and data on the difficulties faced by researchers for research software.

- Explain how the software goal of this research is to design software that can scale from a laptop to the latest supercomputing cluster.

1 page

2.1 The Software Landscape

- Current software projects, what they focus on, what their pitfalls are.

- Emphasise lack of integrated approach, and relatively few examples of open-source codes that are easy to build/deploy - i.e. aren't special research codes created to demonstrate a result.

- Why are we experimenting with Python and Rust? Where does the need for this come from, what has been done in the past?

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.

- Give an overview of the constraints on program design.

- 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, Python...

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.

2.4 Case Study: RustyTree, a Rust based Parallel Octree

- Case study for Rusty tree on different HPC environments and architectures.

- briefly introduce algorithms (parallel sorting, tree construction).

- The novelty isn't the fact that it's a parallel octree, it's that it's one that you can use easily from Python, and deploy to different HPC environments and architectures.

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 libraries, their performance on different architectures, and how easy they are to install and edit - how malleable are they?

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

[2] 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.

[3] 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.

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

[5] 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.

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

[7] 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).

[8] 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.

[9] 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.

[10] 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.

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

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

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

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

[15] 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.

[16] 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.

[17] 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.

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

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

[20] 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.