# 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

# 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 $\mathbb{R}^{\nVDash}$. 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 $\mathsf{K}$ 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.

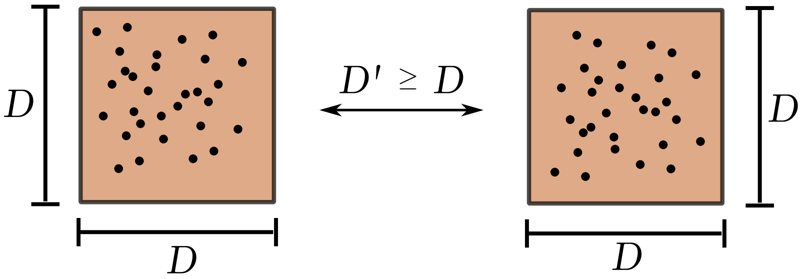


Figure 1.1: Given two boxes in $\mathbb{R}^2$, $\mathcal{B}_1$, $\mathcal{B}_2$, which enclose corresponding degrees of freedom, off diagonal blocks in the linear system matrix $\mathsf{K}_{\mathcal{B}_1\mathcal{B}_2}$ and $\mathsf{K}_{\mathcal{B}_2\mathcal{B}_1}$ are considered low-rank for the FMM when separated by a distance at least equal to their diameter, this is also known as 'strong admissibility'. Figure adapted from [36].

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 [44, 22] and electrostatics [43] to electromagnetics [9] fluid dynamics [40] 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 [27] and Kalman filtering [29]. 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 [24, 46, 42, 31, 37], or fast matrix inversion [39, 35, 23]. 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 [37] or C++ [31, 46, 42], 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 [31, 46], or take advantage of a heterogenous CPU/GPU environments [46]. 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 [19]. 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, $\phi_j$ 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 $\Omega$, 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 $\Omega_s$ at the target particles, $\{x_i\}_{i=1}^L$ in $\Omega_t$.

[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 $\Omega_s$ and $\Omega_t$ 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 $\Omega_t$ 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 [19].

To demonstrate this we derive an expansion in the $\mathbb{R}^2$ case, taking $c_s$ and $c_t$ as the centres of $\Omega_s$ and $\Omega_t$ 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 $\Omega_s$ 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 $\Omega_s$ and can be truncated to any required precision. We can use the multipole expansion in place of a direct calculation with the particles in $\Omega_s$. As the potential in $\Omega_t$ 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 $\Omega_t$, that represents the potential due to the sources in $\Omega_s$.

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

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 $\mathsf{T}_s^{P2M}$ 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 $\mathsf{T}_t^{L2P}$ is a $N_t \times P$ matrix. Here we denote each *translation* operator, $\mathsf{T}^{X2Y}$, 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 $\Omega$ 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 [41].

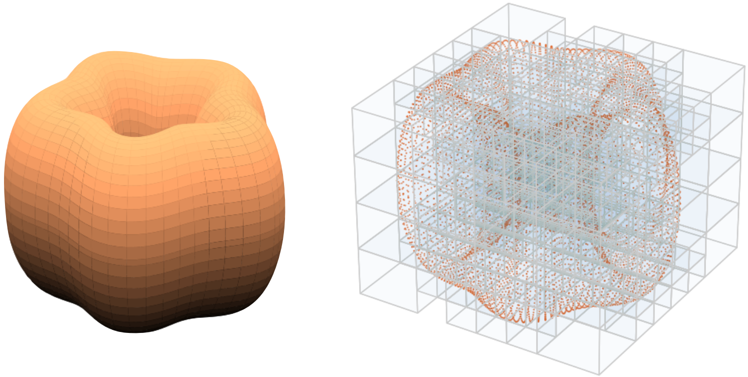


Figure 1.2: An adaptive octree for random point data placed on the surface of a 'wiggly torus' test geometry. The user defines the level of recursion via a threshold for the maximum number of particles in a given node.

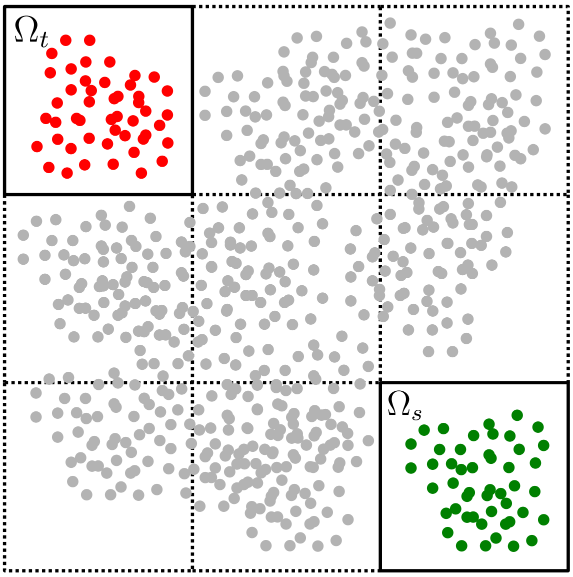


Figure 1.3: Two well separated clusters $\Omega_t$ and $\Omega_s$ where we can apply a low-rank approximation.

In addition to the $\mathsf{T}^{P2M}$, $\mathsf{T}^{M2L}$ and $\mathsf{T}^{L2P}$ the FMM also require operators that can translate the expansion centre of a multipole or local expansion, $\mathsf{T}^{L2L}$, $\mathsf{T}^{M2M}$, an operator that can form a local expansion from a set of points $\mathsf{T}^{P2L}$, and apply a multipole approximation to a set of points, $\mathsf{T}^{M2P}$, 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 [21], $\mathcal{H}^2$ 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

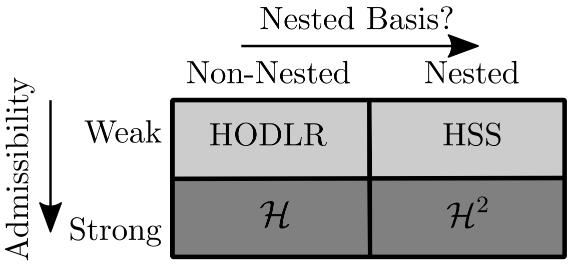
**Algorithm 1 Adaptive Fast Multipole Method**: FMM literature distinguishes between types of relationships between neighbouring nodes with the concept of *interaction lists*. There are four such lists for a given node $B$, called $V$, $U$, $W$ and $X$. For a leaf node $B$, the $U$ list contains $B$ 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
```



| | Non-Nested | Nested |
|---|---|---|
| Weak | HODLR | HSS |
| Strong | $\mathcal{H}$ | $\mathcal{H}^2$ |

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 $C_{\text{leaf}}$ is a small constant.

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

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 [28, 12, 32]. 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 [42, 24].
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 [20], Stokes [13] and Navier [14], 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 [28] in our previous software [24] 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 [28], as well as the Helmholtz kernel in the low-frequency setting [42].

In terms of algorithm optimisation significant efforts have gone towards optimising $T^{M2L}$ [34, 12, 28], 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 [34] 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,

$$T^{M2L} = \left[T^{M2L}_1 | T^{M2L}_2 | ... | T^{M2L}_{16}\right] \tag{1.16}$$

where $T^{M2L}_i$ corresponds to the $i^{\text{th}}$ *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 [28, 12] and analytical FMMs [18]. 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 [17].

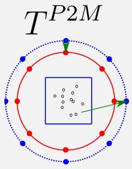
A significant challenge in practical FMM implementations is handling the global data dependency encapsulated in (1.5). Historical single-core architectures have always suffered from the Von Neumann bottleneck, whereby data is loaded from main memory slower than the processor is able to absorb it. However, 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 [10], 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 [10]. 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 [15]. 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 [45, 16]. 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 [28], 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)**

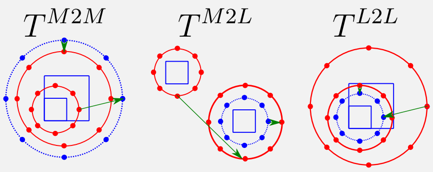
The KIFMM approximates the multipole expansion for a given leaf node containing particles $\{y_j\}_j^N$ with charges $\{q_j\}_j^N$ by evaluating their potential directly (1.4) at corresponding evenly spaced points on a 'check surface', displayed in blue.



This is matched to $N_e$ 'equivalent charges', $\mathsf{q}^e$, 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 $\mathsf{q}^e$, 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 $\mathsf{q}^{\text{node}}$ 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.



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 [11].

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 [12]. An $n$ point Chebyshev approximation to a given function, $g(x)$ with $p_{n-1(x)}$, can be written as,

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

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

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 [28] and the bbFMM [12].

# Designing Software for Fast Algorithms

## 2.1 The Software Landscape

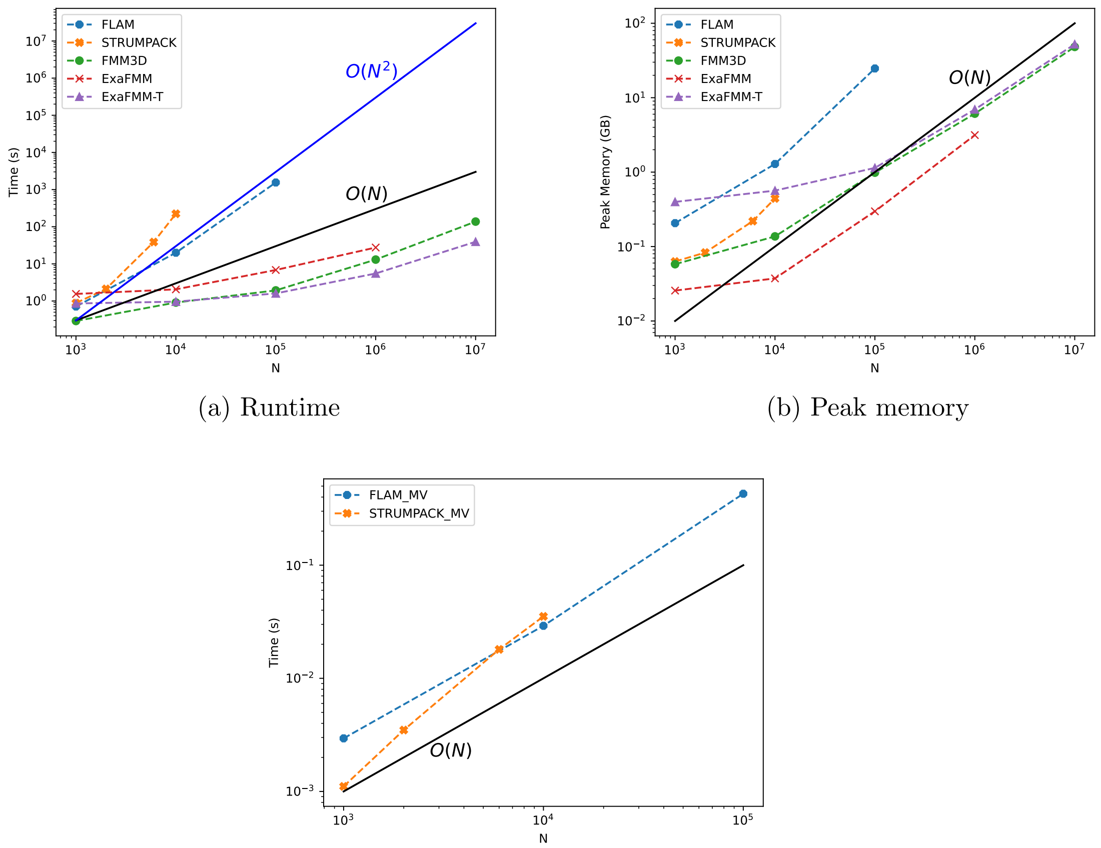
Numerous different groups have developed overlapping software for fast algorithms. For FMMs, leading analytical impelementations include the single-node multithreaded, 'FMM3D' and 'FMM2D' [37, 38], and the MPI accelerated 'ExaFMM' [46] packages. The ExaFMM project have also developed a single-node multithreaded kernel-independent FMM, 'ExaFMM-T' [42]. For algebraic methods, the landscape is sparser, with few parallel implementations. The standouts being the MPI accelerated 'STRUMPACK' [17], for HODLR and HSS matrices, as well as the multithreaded single-node 'FLAM' [23], which supports algorithms for $\mathcal{H}^2$ matrices. The majority of codes, except FLAM, are written in either Fortran or C++, with a few supporting interfaces to higher level languages such as Python and Matlab [42, 37].

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) [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. Each FMM software is run with an expansion order $P = 10$, and the algebraic software's parameters are adjusted to match this level of accuracy.

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 30 minutes, $\approx 2 \times 10^3$ 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 matrix representation of (1.5). Strictly speaking, STRUMPACK's HSS and HODLR matrices are not applicable to (1.5), as it is a strongly admissable problem. We see the effects of this in super-quadratic factorisation times for increasing $N$. Additionally, FLAM, written in Matlab, is not designed for high performance. However, these remain the only well supported, publicly available, parallel implementations of algebraic fast algorithms and we include them for comparison. As expected, the factorization scales super-linearly for both softwares, which lies in contrast with their relatively fast matrix vector products once factorised (fig. (2.1c)).

We observe that currently available open source algebraic software is impractical for computing even moderately sized matrix vector products, i.e. $N \geq O(10^5)$.

[1] Our experiments are available at `https://github.com/skailasa/phd-thesis/code/ch_2/`



(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 that could fit into memory.

With runtimes times for factorisation exceeding our imposed limit of $2 \times 10^3$ s. 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.

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. Counter intuitively, the analytical ExaFMM fails to converge in our experiments for $N = 10^7$, as its memory requirements exceed the available 250 GB.

Theoretical considerations aside, each software's performance is practically determined by the computational and algorithmic performance optimisations they take. ExaFMM-T for example uses optimized algorithms for the calculation of inverse square roots [31], as well as using SIMD vectorisation for kernel evaluations [42]. Similarly by stacking $T^{M2L}$ as in (1.16), they optimise cache-reuse. Similar SIMD vectorisations are implemented by FMM3D and ExaFMM, however their remaining optimisations are presented opaquely, making it difficult for users to evaluate their relative merits.

Usability, defined in terms of quality of API documentation, available descriptions of mathematical and software optimisations, quality of software engineering in terms of self-documenting and well commented code, as well as ease of installation, is as critical to the dissemination of scientific software as raw performance. It's unlikely that a user will incorporate a performant piece of software into their work if it's not usable. In this respect FMM3D and ExaFMM-T standout, with well documented APIs, code examples, as well as wrappers to call functionality from higher level languages. 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. ExaFMM on the other hand is poorly documented, with few tests or comments, and no wrappers to high-level languages. Most softwares, except STRUMPACK which is distributed via the SPACK package manager, have traditional source builds based on CMake and Make. Local installation can therefore be challenging when building on non-traditional HPC operating systems such as Windows and MacOS.

From the experiments in figure (2.1) it's clear that software choice can have dramatic impact on runtime and memory scaling regardless of algorithmic choice. For the Laplace problem of (1.5) all softwares can be used more or less out of the box, however oscillatory Helmholtz kernels are only natively supported by a few [46, 42, 38, 37]. The most complete functionality, with the addition of Stokes and Maxwell kernels, are only supported by FMM2D and FMM3D. From a non-expert user's perspective, merely understanding the theory behind each algorithm isn't enough to expect a guaranteed performance, as demonstrated by the poor scaling of algebraic softwares, as well as the different memory requirements of the analytical FMMs. In light of this, users would likely be tempted to pick a 'median' option, which guarantees good performance, scalability, easy installation, and usability via a high-level language wrapper and support for multiple problems. This makes FMM2D and FMM3D the standout options, however as they're developed as a high-performance Fortran libraries, they are relatively unmalleable, with a steep learning curve for potential contributors and limited to single-node systems.

In conclusion, we identify a distinct lack of software for fast algorithms that fulfills our usability criteria and can also scale to multi-node systems. Presently available software is not composable, with differing APIs and re-implementations of high-performance kernels as well as data structures such as quad and octrees. Algebraic methods lack an optimised software for strongly admissable problems, with FLAM designed as a research code for prototyping purposes. Wrappers for high-level languages are not uniformly available, and builds can be challenging in non-Linux environments.

We aim to solve this with our proposed software infrastructure (fig (2.2)). By offering a set of composable libraries with a unified API for FMMs and algebraic methods for constructing matrix inverses, we aim to maximally re-use highly-optimised code, such as for SIMD vectorised kernel evaluations, and quad and octrees that can scale across multiple processors. We propose Rust as the language for our this infrastructure. We initially explored high-performance code generation tools for Python as a bridge between an ergonomic language for researchers and developers with high-performance. This is documented in section 2.2 via our attempt to build a Python FMM. Due to the constraints of development in high-level languages, and not wishing to use C/C++ or Fortran, we pivoted to Rust. Rust is a young low-level

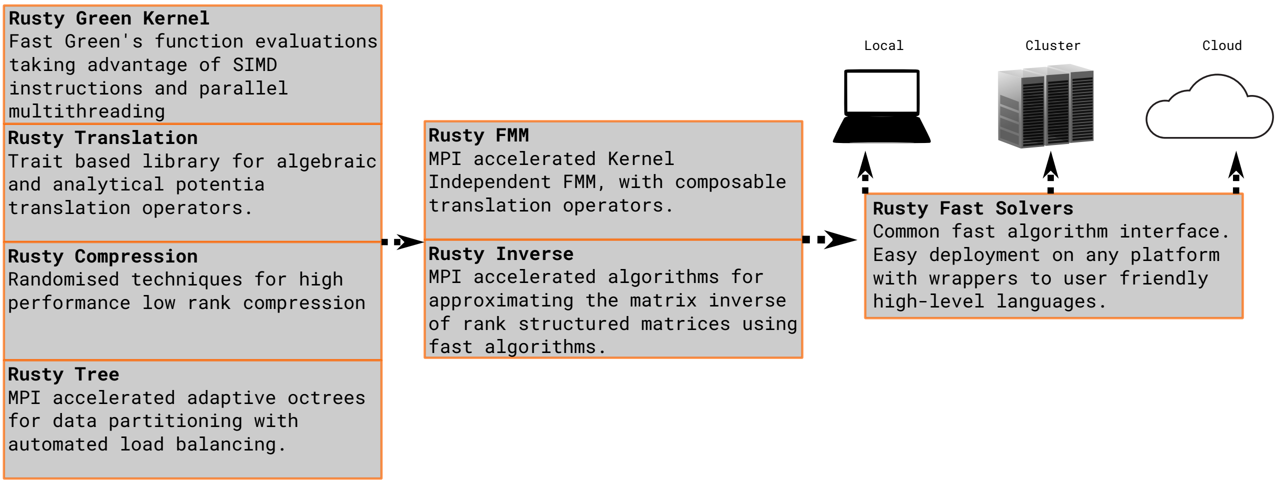

Rusty Green Kernel
Fast Green's function evaluations taking advantage of SIMD instructions and parallel multithreading
Rusty Translation
Trait based library for algebraic and analytical potentia translation operators.
Rusty Compression
Randomised techniques for high performance low rank compression
Rusty Tree
MPI accelerated adaptive octrees for data partitioning with automated load balancing.
Rusty FMM
MPI accelerated Kernel Independent FMM, with composable translation operators.
Rusty Inverse
MPI accelerated algorithms for approximating the matrix inverse of rank structured matrices using fast algorithms.
Local
Cluster
Cloud
Rusty Fast Solvers
Common fast algorithm interface. Easy deployment on any platform with wrappers to user friendly high-level languages.

Figure 2.2: Hierarchy of our proposed infrastructure, with key functionality separated into individual libraries we maximise code re-use across projects. Once completed using Rust's code generation infrastructure, users will be able to deploy our software from desktop workstations, to supercomputing clusters and expect scalable high-performance code, while being able to write application code in Python.

systems programming language, with significant industry support, which has many features that improve the usability of scientific software. Most importantly its build system, Cargo, is specified by a simple TOML file, making it easy to deploy Rust codes on most common platforms without the need for complex build CMake style build scripts. This will make it easy for users to develop software locally, before deploying to a super computing cluster, while still being able to expect good performance. Rust's syntax inherits from both functional and imperative paradigms, and its trait system wholly replaces object orientation, making the description of shared behaviour, and data oriented programming, significantly more readable. However, importantly, it is also easy to develop wrappers from Rust to Python using its C foreign function interface [33], allowing us to expose our infrastructure to novice programmers. We document the prominent features of Rust that make it useful for scientific computing in section 2.3, before exploring the performance of a completed Rust based library for MPI accelerated octrees in section 2.4.

## 2.2 Case Study: PyExaFMM, a Python Fast Multipole Method

Python[2], is designed for memory safety and programmer productivity. Its simplicity allows Computational Scientists to spend more time exploring their science, and less time being confused by software quirks, memory errors, and the nightmare of incompatible dependencies, which conspire to drain productivity in lower level languages.

Python fulfills many of the key usability criteria for scientific software. Cross platform builds are trivial with open source build systems such as Conda, and its simple syntax and large scientific computing ecosystem of numerical libraries allows for rapid dissemination amongst the wider community.

[2] We use 'Python' to refer to CPython, the popular C language implementation of Python, which is dominant in computational science.

With this in mind we sought to assess the suitability of Python as a base language for fast algorithm software. With stable MPI bindings [8] for multi-node development, Python's major pitfall is its restriction to run in a single thread due a software construction called the 'Global Interpreter Lock' (GIL). Libraries for high performance computational science have traditionally bypassed the issue of the GIL by using Python's C interface to call extensions built in C or other compiled languages which can be multithreaded or compiled to target special hardware features, thus enabling hierarchical parallelism.

Recently 'just in time' (JIT), compilers have emerged as a technique for compiling high-level languages to fast machine code. The idea being that a user is able to rapidly iterate on their algorithm, with the compiler taking responsibility to deliver performance. For Python, the Numba compiler [25] was written to targets and optimize code written with NumPy's $n$-dimensional array, or 'ndarray', data structure, which are homogenously typed containers stored contiguously in memory. Its power comes from the ability to generate multithreaded architecture optimized compiled code while *only writing Python.* The promise of Numba is the ability to develop applications with speed that can rival C++ or Fortran, while retaining the simplicity and productivity of working in Python. We tested Numba's suitability by developing 'PyExaFMM', an implementation of the three-dimensional Kernel Independent FMM [24].

## Numba

Numba is a compiler built with LLVM, a framework for building custom compilers, to target a subset of Python code that uses ndarrays. LLVM provides an API for generating machine code for different hardware architectures such as CPUs and GPUs and is also able to analyze code for hardware level optimizations such as auto vectorization, automatically applying them if they are available on the target hardware [26]. Additionally, LLVM generated code may be multithreaded - bypassing the issue of the GIL. Furthermore Numba is able to use the metadata provided by ndarrays describing their dimensionality, type and layout to generate code that takes advantage of the hierarchical caches available in modern CPUs [25]. Altogether, this allows code generated by Numba to run significantly faster than ordinary Python code, and often be competitive with code generated from compiled languages such as C++ or Fortran.

From a programmer perspective using Numba, at least naively, doesn't involve a significant rewrite. Python functions are simply marked for compilation with a special decorator, see listings (2.3), (2.1) and (2.2) for example syntax. This encapsulates the appeal of Numba. The ability to generate high-performance code for different hardware targets from Python, and letting Numba worry about how to perform optimizations, would allow for significantly faster workflows than possible with a compiled language.

Figure (2.5) illustrates the program execution path when a Numba decorated function is called from the Python interpreter. We see that Numba doesn't replace the Python interpreter. If a marked function is called at runtime, program execution is handed to Numba's runtime which compiles the function on the fly with a type signature matching the input arguments. This is the origin of the term 'just in time' [JIT] to describe such compilers.

The Numba runtime interacts with the Python interpreter dynamically, and control over program execution is passed back and forth between the two. There is a cost to this interaction from having to 'unbox' Python objects into types compatible with the compiled machine code, and 'box' the outputs of the compiled functions back into Python compatible objects. This process doesn't involve re-allocating memory, however pointers to memory locations have to be converted and placed in a type compatible with either Numba compiled code or Python.

## Numba's Pitfalls

Since its first release Numba has been extended to compile most functionality from the NumPy library, as well as the majority of Python's basic features and standard library modules[3]. However, if Numba isn't able to find a suitable Numba type for each Python type in a decorated function, or it sees a Python feature it doesn't yet support, it runs in 'object mode', handling all unknown quantities as generic Python objects. To ensure a seamless experience Numba does this without reporting it to the user, unless explicitly marked to run in 'no Python' mode (see listing (2.1) and (2.2) for example syntax). However, object mode is often no faster than vanilla Python, putting the burden on the programmer to understand when and where Numba works. As Numba influences the way Python is written it's more akin to a programming framework rather than just a compiler.

An example of Numba's framework-like behavior arises when implementing algorithms that share data, and have multiple logical steps as in listing (2.1). This listing shows three implementations of the same logic, a function that generates a random matrix $A \in \mathbb{R}^{100\times 100}$ and multiplies it with itself, and also writes an input vector $v \in \mathbb{R}^{100}$ to a Numba dictionary. Data of this size is chosen to reflect the typical amount of computation in a task parallelized by PyExaFMM. The implementations algorithm1 and algorithm2 aren't distinguished by the Numba compiler, and both pay a cost to call subroutines defined outside of their function body. However, algorithm1 pays a small additional (un)boxing cost in order manipulate a globally defined Numba compatible dictionary, in comparison to a locally defined one in algorithm2. The *nested* function in algorithm3 differs from the other two implementation, by defining its sub-routines within its function body, rather than calling externally defined functions. This is an example of an *inlining* optimization, which is picked up by LLVM at compile time.

The runtimes of all three implementations are shown in table (2.1) for three contrasting problem sizes, and shows how inlining can have a significant impact on runtime for this algorithm [4]. This example is designed to illustrate how small changes to writing style can impact the performance of an algorithm written with Numba. We emphasize that the speedup obtained from inlining is dependent on the size of the data being operated on as well as the program logic. Other factors such as memory latency for large data, or the passing of execution control between Python and Numba, with small data are may become more significant. Indeed the experiment with $A \in \mathbb{R}^{1000\times 1000}$ and $v \in \mathbb{R}^{1000}$, we observe that inlining is still the dominant factor in performance difference and is even more prominent than with the smaller dataset. With $A \in \mathbb{R}^{1\times 1}$ and $v \in \mathbb{R}^{1}$, we see that the instantiation of the result dictionary from within a Numba function is now a significant part of total runtime.

[3] A full list of supported features for the current release can be found at: https://numba.pydata.org/numbadoc/dev/reference/pysupported.html

[4] All experiments in this work were taken on an AMD Ryzen Threadripper 3970X 32-Core processor running Python 3.8.5 and Numba 0.53.0

| Algorithm | Matrix Dimension | Time ($\mu$s) |
|---|---|---|
| 1 | $\mathbb{R}^{1\times1}$ | $1.74 \pm 0.01$ |
| 1 | $\mathbb{R}^{100\times100}$ | $308 \pm 1$ |
| 1 | $\mathbb{R}^{1000\times1000}$ | $27100 \pm 200$ |
| 2 | $\mathbb{R}^{1\times1}$ | $2.94 \pm 0.01$ |
| 2 | $\mathbb{R}^{100\times100}$ | $306 \pm 1$ |
| 2 | $\mathbb{R}^{1000\times1000}$ | $27100 \pm 200$ |
| 3 | $\mathbb{R}^{1\times1}$ | $2.61 \pm 0.01$ |
| 3 | $\mathbb{R}^{100\times100}$ | $2.64 \pm 0.07$ |
| 3 | $\mathbb{R}^{1000\times1000}$ | $2.84 \pm 0.07$ |

Table 2.1: Testing the effect of inlining and (un)boxing with dense matrix vector products in double precision using implementations from listing (2.1).

Nested functions have the tendency to grow long in performant Numba code, in order to minimize the number of interactions between Numba and Python. However this makes them more difficult to unit test. Indeed, performant Numba code can look decidedly un-Pythonic. Numba encourages fewer user created objects, performance critical sections written in terms of loops over simple array based data structures, and potentially long nested functions.

Furthermore, not every supported feature from Python behaves in a way an ordinary Python programmer would expect, which has an impact on program design. An example of this arises when using Python dictionaries, which are central to Python, but are only partially supported by Numba. As they are untyped, and can have any Python objects as members, they don't neatly fit into a Numba compatible type. Programmers can declare a Numba compatible 'typed dictionary', where the keys and values are constrained to Numba compatible types, and pass it to a Numba decorated function at low cost. However, using a Numba dictionary from the Python interpreter is *always slower* than an ordinary Python dictionary due to the (un)boxing cost when getting and setting any item.

Therefore, though Numba is advertised as an easy way of injecting performance into your program via a simple decorator, it can be seen to have its own learning curve. Achieving performance requires a programmer to be familiar with the internals of its implementation and potential discrepancies that arise when translating between Python and the LLVM generated code, which may lead to significant alterations in the design of algorithms and data structures.

## Data Oriented Design

Data oriented design is about writing code that operates on data structures with simple memory layouts, such as arrays, in order to optimally take advantage of modern hardware features. The idea being that it is easier for programmers to optimize for cache locality and parallelization if the data structures are easier to map to the hardware. This contrasts with object oriented design, where although code is organized around data the focus is on user created types or 'objects', where memory layout is obfuscated by the potential complexity of an object, which can contain multiple attributes of different types. This makes it harder to write code that takes advantage of cache locality. Numba's focus on ndarrays strongly encourages data oriented design principles, which are reflected in the design of PyExaFMM's octrees as well as its API.

Octrees can either be 'pointer based' [42], or 'linear' [41] (see sec 2.4). A pointer based octree uses objects to represent each node, with fields for a unique id, contained particles, associated expansion coefficients, potentials, and pointers to their parent and sibling nodes. This makes searching for neighbors and siblings easy, as one has to just follow pointers. The linear octree implemented by PyExaFMM represents nodes by a unique id stored in a 1D vector, all other data such as expansion coefficients, particle data, and calculated potentials, are also stored in 1D vectors. Data is looked up by creating indices to tie a node's unique id to the associated data. This is an example of how Numba can affect design decisions, and make software more complex, despite the data structures being simpler.

Figure (2.3) illustrates PyExaFMM's design. There is only a single Python object, 'Fmm', which acts as the API. It initializes ndarrays for expansion coefficients and calculated potentials, and its methods interface with Numba compiled functions for the FMM operators and their associated data manipulation functions. When sharing data we prefer nested functions, however we keep the operator implementations separate from each other, which allows us to unit test them individually. This means that we must have at least one interaction between Numba and the Python interpreter to call the near field, P2M, L2P, M2P and P2L operators, $d-2$ interactions to call the M2L and L2L operators, and $d$ interactions for the M2M operator where $d$ is the depth of the octree. The most performant implementation would be a single Numba routine that interacts with Python just once, however this would sacrifice other principles of clean software engineering such as modularity, and unit testing.

This structure has strong parallels with software designs that arise from traditional methods of achieving performance with Python by interfacing with a compiled language such as C or Fortran. The benefit of writing in Numba is that we can continue to write in Python. Though as seen above, performant Numba code may only be superficially Pythonic through its shared syntax.

Numba enables multithreading via a simple parallel for loop syntax (see listing (2.2)) reminiscent of OpenMP. Internally Numba can use either OpenMP or Intel TBB to generate multithreaded code. We choose OpenMP for PyExaFMM, as it's more suited to functions in which each thread has an approximately similar workload. The threading library can be set via the NUMBA_THREADING_LAYER environment variable.

Numerical libraries such as NumPy and SciPy implement many of their mathematical operations using multithreaded compiled libraries internally, such as OpenBLAS or IntelMKL. Numba compiled versions of these operations retains this internal multithreading. This leads to *nested parallelism* when combined with a multithreaded region declared with Numba, as in listing (2.2). This is where a parallel region calls a function with another parallel region inside it. Threads created by the two functions are not coordinated by Numba, and this leads to *oversubscription*, where the number of active threads exceeds the CPU's hardware capacity. Many threads are left idle while the CPU is forced to jump between threads operating on different data. This leads to broken cache locality, and hanging threads threads, stuck waiting for others to finish [30]. We avoid this in PyExaFMM by explicitly setting NumPy operations to be single threaded, via the environment variable OMP_NUM_THREADS=1, before starting our program. This ensures that the only threads created are those explicitly declared using Numba.

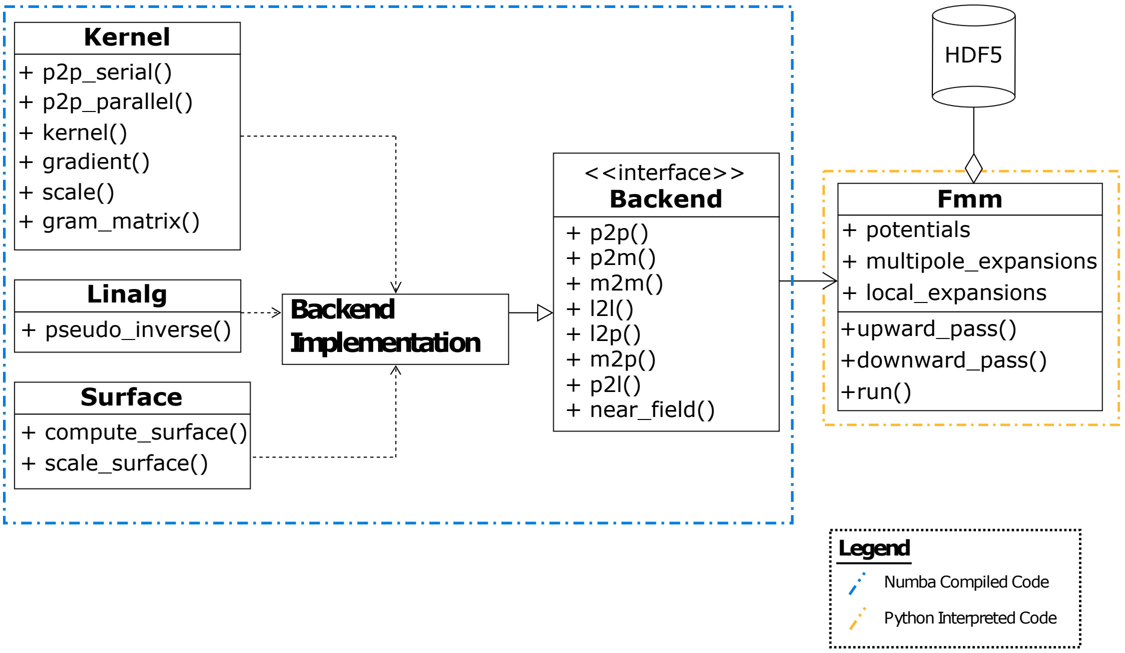


Figure 2.3: Simplified UML model of all PyExaFMM components. Trees and other precomputed quantities are stored in a HDF5 database. The ‘Fmm’ object acts as the user interface, all other components are modules consisting of methods operating on arrays, adapted from [24].

## Parallising the KIFMM

The P2P, P2M, P2L, M2P, L2P and near field operators all rely on the P2P operator, as this computes (1.5) over their respective sources and targets, and are parallelized over their targets, the leaf nodes. For the L2P operator we encourage cache locality for the P2P step, and keep the data structures passed to Numba as simple as possible, by allocating 1D vectors for the source positions, target positions and the source expansion coefficients, such that all the data required to apply an operator to single target node is adjacent in memory. By storing a vector of ‘index pointers’, that bookend the data corresponding to each target in these 1D vectors, we can form parallel for loops over each target to compute the P2P that encourages cache-locality in the CPU. In order to to this, we have to first iterate through the target nodes, and lookup the associated data to fill the cache local vectors.

The speedup achieved with this strategy, in comparison to a naive parallel iteration over the L2P’s targets, increases with the number of calculations in each thread and hence the expansion order $p$. In an experiment with 32768 leaves, $n_{crit} = 150$ and $p = 10$, our strategy is 13 % faster. This corresponds to a realistic FMM problem with approximately $1e6$ randomly distributed particles.

Due to their large interaction lists, the previous strategy is too expensive in terms of memory for the near field, M2L and M2P operators. For example, allocating an array large enough to store the maximum possible number of source particle coordinates in double precision for the M2P operator; with $|W| = 148$ and $n_{crit} = 150$, requires $\sim$ 17GB, and a runtime cost for memory allocations that exceeds the computation time. Instead, for the M2L we perform a parallel loop over the target nodes at each given level, and over the leaf nodes for the M2P and near field, looking up the relevant data from the linear tree as needed. The P2L interaction list of each target is at most 19 nodes, and the P2M must also calculate a check potential, cancelling out any speedup from cache locality for these operators.

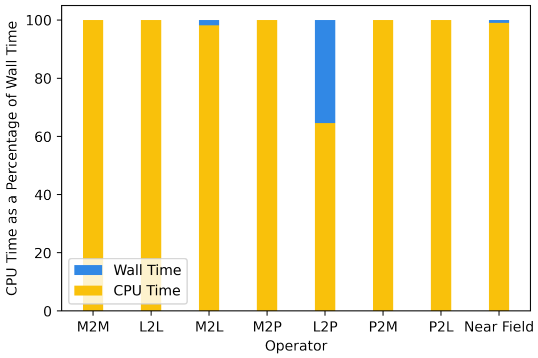


Figure 2.4: CPU time as a percentage of wall time for operators. CPU time is defined as the time in which the algorithm runs pure Numba compiled functions. Wall time is CPU time in addition to the time taken to return control to the Python interpreter, adapted from [24].

The matrices involved in the M2M and L2L operators can be precomputed and scaled at each level [42], and their application is parallelized over all nodes at a given level. Multithreading in this way means that we call the P2P, P2M, M2P, L2P and near field operators once during the algorithm, the M2L and L2L are called $d-2$ times, and the M2M is called $d$ times, where $d$ is the depth of the octree. This is the minimum number of calls while keeping the operator implementations separate for unit testing.

Figure (2.4) compares the time spent within each Numba-compiled operator ('CPU time') to the total runtime ('wall time') of each operator. The results are computed over five trials over 32768 leaves, with $n_{crit} = 150$ and $p = 6$, for a random distribution of $1e6$ charges distributed on the surface of a sphere representing a typical FMM problem. The mean size of the interaction lists are $|U| = 11$, $|V| = 42$, $|X| = 3$, $|W| = 3$, and the entire algorithm is computed in $5.95 \pm 0.02s$, with an additional $9.00 \pm 0.01s$ for operator pre-computations for a given dataset, which is unachievable in ordinary single-threaded interpreted Python.

The wall time includes the time to (un)box data, organize inputs for Numba compiled functions, and pass control between Numba and Python. Except for the L2P which has a different parallelization strategy that requires significant data organization that must take place within the GIL restricted Python interpreter, the runtime costs are less than 5 % of each operator's total wall time, implying that we are nearly always running multithreaded code and utilizing all available CPU cores.

## If Not Python, Then What?

Achieving optimal multithreaded performance with Numba requires careful consideration of the algorithm being accelerated, details of Numba's backend implementation, as well as a design that suits Numba's data oriented framework. The pitfalls illustrated above show how a user must potentially adapt their code significantly in order to achieve the best performance. Altogether, Numba accelerated code may look decidedly un-Pythonic despite using Python syntax. The complexities involved when using Numba to implement a non-trivial algorithm contrast with its advertisement as a simple way of injecting performance into Python code by applying a decorator. Significant software development expertise is needed in order to optimize a Numba implementation, and arguably more than many in its intended target

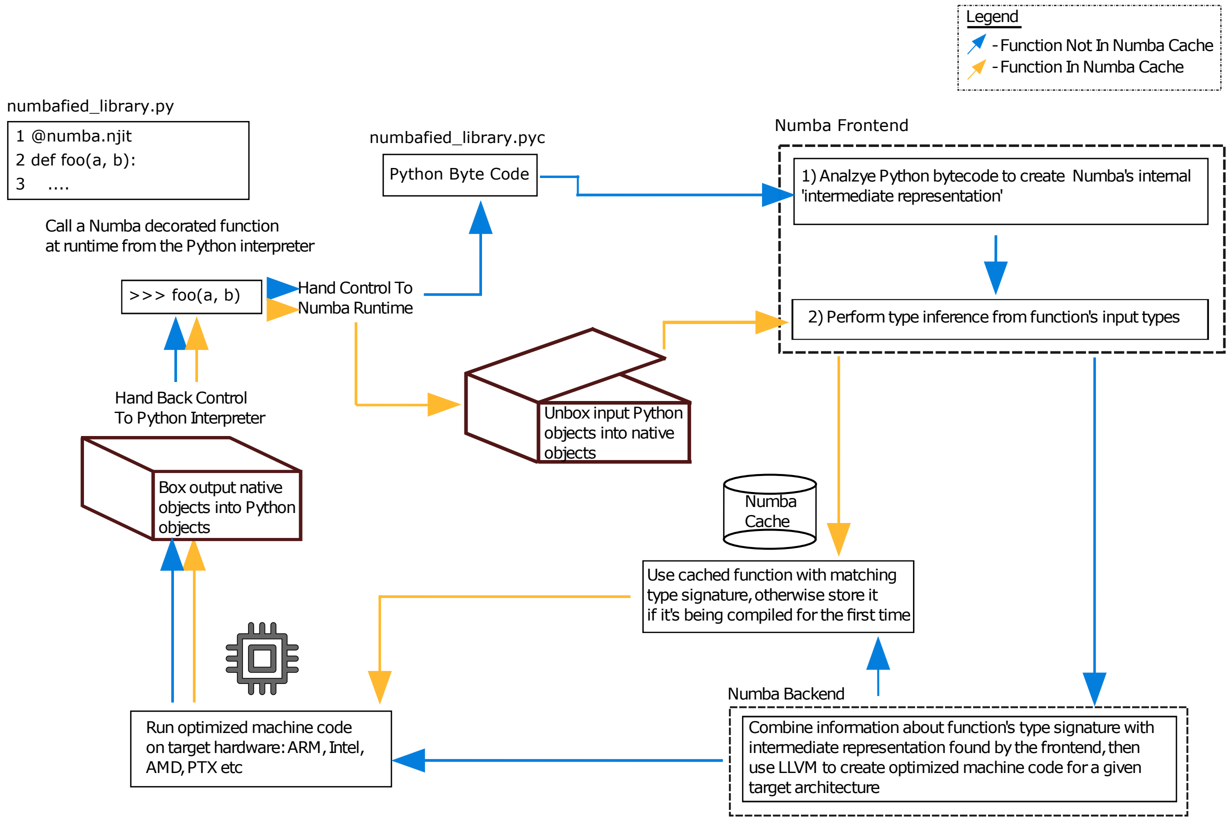

Legend
- Function Not In Numba Cache
- Function In Numba Cache
numbafied_library.py
1 @numba.njit
2 def foo(a, b):
3 ....
numbafied_library.pyc
Python Byte Code
Numba Frontend
1) Analzye Python bytecode to create Numba's internal 'intermediate representation'
2) Perform type inference from function's input types
Call a Numba decorated function at runtime from the Python interpreter
>>> foo(a, b)
Hand Control To Numba Runtime
Hand Back Control To Python Interpreter
Unbox input Python objects into native objects
Box output native objects into Python objects
Numba Cache
Use cached function with matching type signature, otherwise store it if it's being compiled for the first time
Numba Backend
Combine information about function's type signature with intermediate representation found by the frontend, then use LLVM to create optimized machine code for a given target architecture
Run optimized machine code on target hardware: ARM, Intel, AMD, PTX etc

Figure 2.5: Simplified execution path when calling a Numba compiled function from the Python interpreter. The blue path is only taken if the function hasn't been called before. The orange path is taken if a compiled version with the correct type signature already exists in the Numba cache, adapted from [24].

audience can be expected to possess.

Despite this, Numba is a remarkable tool. Projects which value Python's expressiveness, simple cross platform builds, as well as large open source ecosystem, and only contain a small number of isolated performance bottlenecks would benefit the most from a Numba implementation. Indeed, by writing only in Python our project size is kept minimal with the entire project running to just 4901 lines of code. Furthermore, we are able to deploy PyExaFMM cross platform trivially with Conda and distribute our software in popular Python channels.

Resultantly, we wish to find a middle way, that would retain the usability of a higher level language, with the performance benefits of writing in a lower level language, filling the gap from the handoff between a high-level language interface and the programming constraints, as well as the handoff performance hit of a just in time compiler. We believe that this is offered by Rust, a low-level systems programming language, that retains many of these features.

Listing 2.1: Three ways of writing a trivial algorithm that passes around a vector, while performing some computations.

```python
import numpy as np
import numba
import numba.core
import numba.typed

# Initialize in Python interpreter
data = numba.typed.Dict.empty(
    key_type=numba.core.types.unicode_type,
    value_type=numba.core.types.float64[:]
)

data['v'] = np.ones(100)

# Functions marked with 'njit' rather than
# 'jit' decorator, to force Numba to run in
# no Python mode, dissallowing the compilation
# of Python code it is not specialized for.

# Subroutine 1
@numba.njit
def step_1(data):
    # An example allocation & calculation
    a = np.random.rand(100, 100)
    b = a @ a
    data['step_1'] = data['v']

# Subroutine 2
@numba.njit
def step_2(data):
    # An example allocation & calculation
    A = np.random.rand(100, 100)
    B = A @ A
    data['step_2'] =  data['step_1']

@numba.njit
def algorithm1(data):
    # Pays the un(boxing) cost to exchange
    # data with interpreter
    step_1(data); step_2(data)

@numba.njit
def algorithm2():
    # Only pays the boxing cost to return a
    # Python type.
    data = dict()
    data['v'] = np.ones(100)
    step_1(data); step_2(data)
    return data
```

```
@numba.njit
def algorithm3():
    # Numba interprets subroutines as a
    # part of a **single** function body.
    data = dict()
    data['v'] = np.ones(100)

    def step_1(data):
        A = np.random.rand(100, 100)
        B = A @ A
        data['step_1'] = data['v']

    def step_2(data):
        A = np.random.rand(100, 100)
        B = A @ A
        data['step_2'] =  data['step_1']

    step_1(data); step_2(data)
    return data
```

Listing 2.2: An example of parallel multithreading.

```
import numba
import numpy as np

@numba.njit(cache=True, parallel=True)
def multithreading(A):
    # consider an A as an nxn matrix
    # This is a situation that can lead
    # to thread oversubscription unless
    # Numpy is configured to run single
    # threaded.
    for _ in numba.prange(10):
        B = A @ A
```

## 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.

Listing 2.3: An example of using Numba in a Python function operating on ndarrays.

```
import numba
import numpy as np

# optionally the decorator can take
# the option nopython=True, which
# disallows Numba from running in
# object mode
@numba.jit
def loop_fusion(A):
    """
    An example of loop fusion, an
    optimization that Numba is able
    to perform on a user's behalf.
    When it recognizes that they are
    operating similarly on a single
    data structure
    """
    for i in range(10):
        A[i] += 1

    for i in range(10):
        A[i] *= 5

    return A
```

- 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] Lisandro Dalcin and Yao-Lung L Fang. "mpi4py: Status update after 12 years of development". In: *Computing in Science & Engineering* 23.4 (2021), pp. 47–54.

[9] 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.

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

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

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

[13] 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.

[14] 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.

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

[16] 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.

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

[18] 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.

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

[20] 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.

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

[22] 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.

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

[24] 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).

[25] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. “Numba: A llvm-based python jit compiler”. In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. 2015, pp. 1–6.

[26] Chris Lattner and Vikram Adve. “LLVM: A compilation framework for lifelong program analysis & transformation”. In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE. 2004, pp. 75–86.

[27] 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.

[28] 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`.

[29] 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.

[30] Anton Malakhov. "Composable Multi-Threading for Python Libraries". In: *Proceedings of the 15th Python in Science Conference* (2016), pp. 15–19. DOI: 10.25080/majora-629e541a-002. URL: https://youtu.be/kfQcWez2URE.

[31] 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.

[32] 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.

[33] *Maturin*. Version 0.13.0. 2022. URL: https://github.com/PyO3/maturin.

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

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

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

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

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

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

[40] 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.

[41] 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.

[42] 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.

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

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

[45] 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.

[46] 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.