Towards Exascale Multiparticle Simulations

Srinath Kailasa

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

Department of Mathematics
University College London
December, 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 2 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 3 provides a survey of the fragmented parallel software landscape for fast algorithms. In chapter 4 we consider the trade-offs of implementing our codes in high-level languages via our experience of developing an FMM in Python, which attempted to bridge the gap between a familiar and ergonomic language for researchers and achieving high-performance. Despite its many advantages, including cross-platform support and large numerics ecosystem, we find Python and high-level languages in general to be inadequate for our purposes. In chapter 5 we introduce Rust as our proposed solution for ergonomic and high-performance codes for computational science. Rust is a modern compiled programming language, which we contrast with our experience developing software with Fortran and C++. In chapter 6 we present a demonstrative Rust software output, Rusty Tree, a new Rust-based library for the construction of parallel octrees. Octrees are a foundational data structure for FMMs, as well as other fast algorithms. Finally chapter 7 presents an emerging vector in our research, specifically an overview of a recently developed fast algorithm for matrix inversion for the solution of acoustic scattering problems as described by the Helmholtz equation.

Contents

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 $K(x, y) = \frac{1}{4\pi|x-y|}$ in $\mathbb{R}^3$. 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. Consider the following integral equation formulation,

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

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 *globally data dependent* in the sense that the calculation at each matrix element of the discretised system 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 an 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 with linear algebra techniques such as LU decomposition is $O(N^3)$, where N is the number of unknowns. 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) in $\mathbb{R}^2$.

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

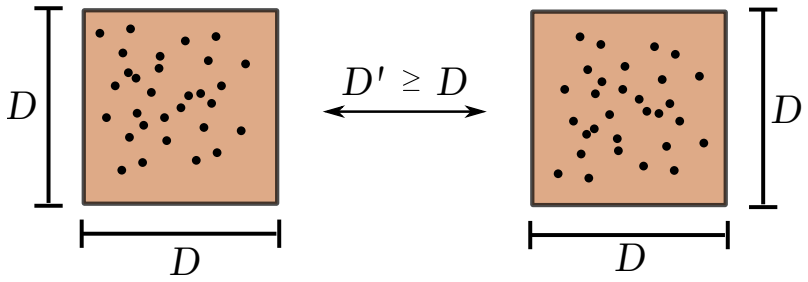


Figure 1.1: Given two boxes in $\mathbb{R}^2$, $\mathcal{B}_1$, $\mathcal{B}_2$, which enclose corresponding degrees of freedom, off diagonal blocks in the linear system matrix $\mathsf{K}_{\mathcal{B}_1\mathcal{B}_2}$ and $\mathsf{K}_{\mathcal{B}_2\mathcal{B}_1}$ are considered low-rank for the FMM when separated by a distance at least equal to their diameter, this is also known as 'strong admissibility'. Figure adapted from [54].

algorithms for matrix inversion display similarly optimal scaling in the best case [54]. Given the wide applicability of boundary integral equations to natural sciences, from acoustics [78, 36] and electrostatics [77] to electromagnetics [21] fluid dynamics [60] and earth science [14], fast algorithms can be seen to have dramatically brought within reach large scale simulations of a wide class of scientific and engineering problems. As a result the FMM has been described as one of the ten most important algorithms of the twentieth century [17]. 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 [4], machine learning [43] and Kalman filtering [45]. 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 [39, 80, 76, 47, 57], or fast matrix inversion [59, 53, 37]. 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 [57] or C++ [47, 80, 76], 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 benchmark (capable of 10^{18} flops) 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

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

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 [47, 80], or take advantage of a heterogenous CPU/GPU environments [80, 8]. In fact for fast inverses there doesn't yet exist an open-source shared memory 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.

From Analytic to Algebraic 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 presentation [33]. Given an N-body evaluation of electrostatic potentials, we let $\{x_i\}_{i=1}^N \in \mathbb{R}^d$ denote the set of locations of charges of strength q_i, where $d = 2$ or $d = 3$. Our task is then to evaluate potentials, ϕ_j for $i = 1, 2, ..., N$. Without loss of generality we take the value of $K(x, x) = 0$. Denoting a square/cube domain containing all points with Ω, we seek to evaluate a matrix vector product of the form,

$$\phi = \mathsf{Kq} \tag{2.1}$$

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 (2.2) where we choose $\mathbb{R}^2$ for simplicity. Here we seek to evaluate the potential induced by the source particles, $\{y_j\}_{j=1}^M$, in Ω_s at the target particles, $\{x_i\}_{i=1}^L$ in Ω_t.

$$\phi_i = \sum_{j=1}^{L} K(x_i, y_j) q_j, \;\; i = 1, 2...M \tag{2.2}$$

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

$$K(x, y) \approx \sum_{p=0}^{P-1} B_p(x) C_p(y), \;\; \text{when } x \in \Omega_t, y \in \Omega_s \tag{2.3}$$

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,

$$\hat{q}_p = \sum_{j \in I_s} C_p(x_j) q_j, \;\; p = 0, 1, 2, ..., P - 1 \tag{2.4}$$

this is a good approximation, as low-rank approximation of K converges rapidly in the far-field. Using this, we evaluate the approximation of the potential at the targets as,

$$\phi_i \approx \sum_{p=1}^{P-1} B_p(x_i)\hat{q}_p \tag{2.5}$$

In doing so we see that we accelerate (2.2) from $O(ML)$ to $O(P(M+L))$. As long as we choose $P \ll M$ and $P \ll L$, we will recover an accelerated matrix vector product. The power of the FMM, and similar fast algorithms, is that we can recover the potential in Ω_t with high accuracy even when P is small.

We deliberately haven't stated how we calculate B_p or C_p. In Greengard and Rokhlin's FMM these took the form of analytical multipole and local expansions of the kernel function [33]. 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,

$$\begin{aligned} K(x,y) = \log(x-y) &= \log((x-c_s)-(y-c_s)) \\ &= \log(x-c_s) + \log(1-\frac{y-c_s}{x-c_s}) \\ &= \log(x-c_s) - \sum_{p=1}^{\infty} \frac{1}{p}\frac{(y-c_s)^p}{(x-c_s)^p} \end{aligned} \tag{2.6}$$

where the series converges for $|y-c_s| < |c-c_s|$. We note (2.6) 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 $\hat{\mathsf{q}}^{\mathsf{s}} = \{\hat{q}_p^s\}_{p=0}^{P-1}$,

$$\begin{cases} \hat{q}_0^s = \sum_{j\in I_s} q_j \\ \hat{q}_p^s = \sum_{j\in I_s} -\frac{1}{p}(x_j-c_s)^p q_j, \;\; p = 1,2,3...,P-1 \end{cases} \tag{2.7}$$

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,

$$\phi(x) = \sum_{j\in I_s} K(x,y)q_j = \log(x-c_s)\hat{q}_0^s + \sum_{p=1}^{\infty} \frac{1}{(x-c_s)^p}\hat{q}_p^s \tag{2.8}$$

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

$$\phi(x) = \sum_{l=1}^{\infty} (x-c_t)^l \hat{\phi}_l^t \tag{2.9}$$

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).

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

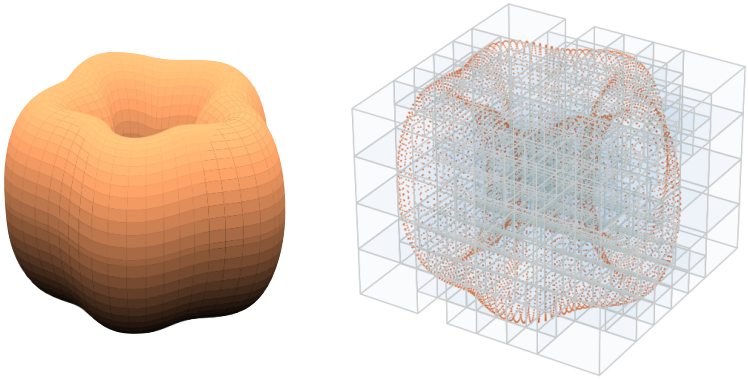


Figure 2.1: An adaptive octree for random point data placed on the surface of a 'wiggly torus' test geometry. The user defines the level of recursion via a threshold for the maximum number of particles in a given node.

$$\hat{\mathsf{q}}^s = \mathsf{T}_s^{P2M} \mathsf{q}(I_s) \tag{2.10}$$

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

$$\hat{\phi}^t = \mathsf{T}_{t,s}^{M2L} \hat{\mathsf{q}}^s \tag{2.11}$$

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

$$\phi^t = \mathsf{T}_t^{L2P} \hat{\phi}^t \tag{2.12}$$

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

We have described how to obtain linear complexity when considering two isolated nodes, however in order to recover this for interactions between *all particles* 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. 2.1). 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 leaf node sizes, and 'balanced' to enforce a maximum size constraint between adjacent leaf nodes [70].

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

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, i.e. the nodes are traversed bottom-up, level-by-level, beginning with the finest nodes. **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, i.e. the nodes are traversed top-down, level-by-level, beginning with the coarsest nodes. **do**
  P2M: compute multipole expansion for the particles they contain.
end for
for each non leaf node B in *postorder* traversal of the tree **do**
  M2M: form a multipole expansion by translating the expansion centre of its children to its centre and summing their multipole expansion coefficients.
end for

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

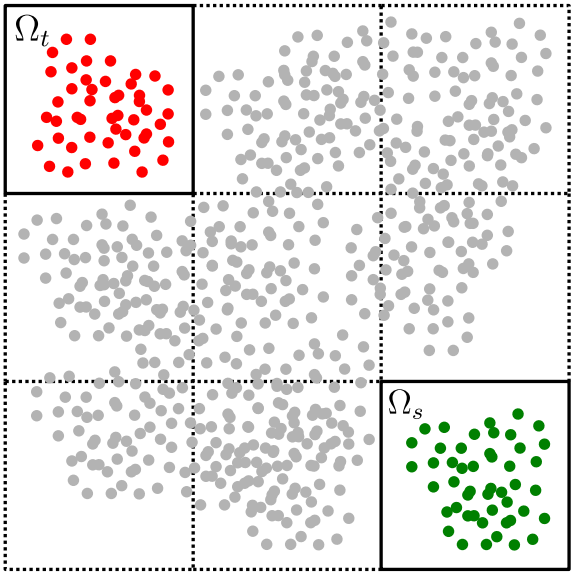


Figure 2.2: Two well separated clusters Ω_t and Ω_s where we can apply a low-rank approximation.

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. Concurrently to the development of the FMM, algebraic analogues were developed. These methods are similarly based on a hierarchical partition of the point data using a recursive tree, and an admissibility condition. The underlying difference being in how they represent field expansions, which are no longer required to be multipole expansions, as well as the translation operators and their representation of the field translations in as an explicit matrix acting on the point charges and their multipole and local expansion coefficients. Once this representation is computed it can be used again allowing for simple extensions to matrix-matrix products. The explicit expression as a matrix also allows for algorithms that can operate on this data structure to approximate the matrix inverse. Examples of algebraic matrix representations include the $\mathcal{H}$ matrix [35], $\mathcal{H}^2$ matrix [10], hierarchically semi-separable (HSS) [15], and hierarchically off-diagonal low-rank (HODLR) [2] matrices.

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. We identify this 'cluster tree' approach as being analogous to the octree used in the analytical FMM. Initially, an binary 'cluster tree' $\mathcal{T}_I$, with a set of nodes T_I, is formed such that

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

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

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

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

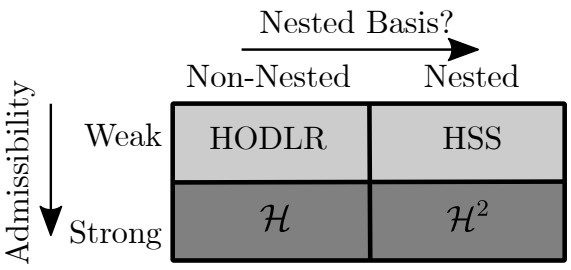


Figure 2.3: Matrix containing common algebraic alternatives to the FMM, adapted from [1].

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 [9]. 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 (2.3). 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 [9].

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.

Similar algebraic approaches, confusingly labelled 'black box' FMMs combine the algorithmic structure of the analytical FMM with alternative field representations that aren't necessarily multipole expansions [44, 25, 49]. The origin of the label 'black box' is due to to the fact that the low rank representations they construct are done so independently of the kernel. We contrast two common black box methods in box (2.4), 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 [76, 39].

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 [34], Stokes [27] and Navier [28], 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 [44] in our previous software [39] 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 [44], as well as the Helmholtz kernel in the low-frequency setting [76].

Computational Considerations for the FMM

In terms of algorithm optimisation significant efforts have gone towards optimising T^{M2L} [51, 25, 44], 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 [51] show that even these 316 operators are permutations of a subset of only 16 unique operators. Permutations corresponding to reflections across various planes (for example $x = 0$, $y = 0$, $z = 0$, etc). We can therefore write a block matrix of *all* unique T^{M2L} for a given level,

$$T^{M2L} = \left[T_1^{M2L}|T_2^{M2L}|...|T_{16}^{M2L}\right] \tag{2.13}$$

where T_i^{M2L} corresponds to the i^{th} *transfer vector* describing a unique T^{M2L}. This block matrix is efficiently compressed using a truncated SVD, which is known to produce an optimal low rank approximation of a rank N matrix, where the new rank, r, is chosen such that $r \ll N$. This is a common optimisation used in both black box [44, 25] and analytical FMMs [32]. 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 black box 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 [31].

Parallelising the KIFMM

The computational complexities of KIFMM operators are defined by a user specified n_{crit}, which is the maximum allowed number of particles in a leaf node, n_e and n_c which are the numbers of quadrature points on the *check* and *equivalent* surfaces respectively (see sec. 3 [44]). The parameters n_e and n_c are quadratically related to the expansion order, i.e. $n_e = 6(p-1)^2 + 2$ [44]. Typical values for n_{crit} used are ~ 100. Notice the depth of the octree is defined by n_{crit}, and hence by the particle distribution.

The near field calculated directly using $P2P$, T^{P2M}, T^{P2L}, T^{M2P}, and T^{L2P} operate independently over the leaf nodes. The T^{M2L} and T^{L2L} operate independently on all nodes at a given level, from level 2 to the leaf level, during the top-down traversal. The T^{M2M} is applied to each node during the bottom-up traversal. All operators, except the T^{M2L}, T^{M2M} and T^{L2L}, rely on the $P2P$.

As explained in algorithm 1, the inputs to the M2L, M2P, P2L and near field operators are defined by 'interaction lists' called the V, W, X and U lists respectively. These interaction lists define the nodes a target node interacts with when an operator is applied to it. We can restrict the size of these interaction lists by demanding that neighboring nodes at the leaf level are at most twice as large as each other [70]. Using this 'balance condition', the V, X, W and U lists in 3D contain at most 189, 19, 148 and 60 nodes, respectively.

The near field operator applies the $P2P$ between the charges contained in the target and the source particles of nodes in its U list, in $O(60 \cdot n_{crit}^2)$. The T^{M2P} applies the $P2P$ between multipole expansion coefficients of source nodes in the target's W list and the charges it contains internally in $O(148 \cdot n_e \cdot n_{crit})$. Similarly, the T^{L2P} applies the $P2P$ between a target's own local expansion coefficients and the charges it contains in $O(n_e \cdot n_{crit})$.

The T^{P2L}, T^{P2M} and T^{M2L} involve creating local and multipole expansions, and rely on a matrix vector product related to the number of source nodes being compressed, which for the T^{P2L} and T^{M2L} are defined by the size of the target node's interaction lists. These matrix vector products have complexities of the form $O(k \cdot n_e^2)$ where $k = |X| = 19$ for the T^{P2L}, $k = |V| = 189$ for the T^{M2L}, and $k = 1$ for the P2M. Additionally, the T^{P2L} and T^{P2M} have to calculate 'check potentials' (see sec. box (2.4)) which require $O(19 \cdot n_{crit} \cdot n_c)$ and $O(n_{crit} \cdot n_c)$ calculations respectively. The T^{M2M} and T^{L2L} operators both involve translating expansions between nodes their eight children, and rely on a matrix vector product of $O(n_e^2)$.

The structure of the FMM's operators expose natural parallelism. The $P2P$ is embarrassingly parallel over each target. As are the T^{M2L}, T^{M2P}, T^{P2L} and near field operators over their interaction lists. The near field, T^{L2P}, T^{M2P}, T^{P2L} and T^{P2M} operators are also embarrassingly parallel over the leaf nodes, as are the T^{M2L}, T^{M2M} and T^{L2L} over the nodes at a given level.

Conclusion

A significant challenge in practical FMM implementations is handling the global data dependency encapsulated in (2.2). 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

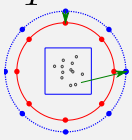
of Dennard scaling[1] from around 2006 and the subsequent multicore revolution the disparity between data movement costs and computation have been even further exacerbated. On modern hardware accelerators are able to achieve $O(1e12)$ flop rates in double precision, but are bandwidth limited to $O(1e11)$ 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 [23], 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 [23]. 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 chapter 6.

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 is an alternative, and has corresponding distributed memory implementations [29]. 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 [79, 30]. Furthermore, a multigrid approach does not allow for a re-use of FMM data structures 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, black box and algebraic FMM variants may have a significant impact on the ultimate implementation, and performance, this is something we explicitly measure in chapter 3. We identify the black box Kernel-Independent FMM of Ying et. al [44], as offering a good compromise between ease of of software design, mathematical simplicity, and problem generality.

[1]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 (2.1) at corresponding evenly spaced points on a 'check surface', displayed in blue.

T^{P2M}

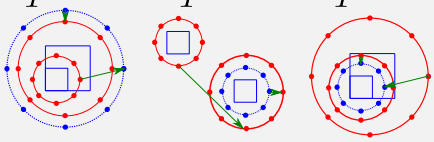


This is matched to N_e 'equivalent charges', q^e, placed evenly on a the (red) equivalent surface at $\{x_i\}_i^{N_e}$. These surfaces are taken to be circles in $\mathbb{R}^2$ and cubes in $\mathbb{R}^3$. They perform a Tikhonov-regularised least squares fit to calculate q^e, which plays the role of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

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

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

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



The KIFMM is designed for non-oscillatory kernels, such as the Laplace kernel of our didactic example, though in practice works well with low-frequency oscillatory problems, with extensions to high frequency problems [24].
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 [25]. An n point Chebyshev approximation to a given function, $g(x)$ with $p_{n-1(x)}$, can be written as,

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

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

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

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

When applied to our generic interpolation for the kernel,

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

which can be substituted into (2.2), recovering linear complexity as long as the number of Chebyshev nodes is small, $n \ll N$.

Figure 2.4: The formulations of the kernel-independent FMM methods, the KIFMM [44] and the bbFMM [25].

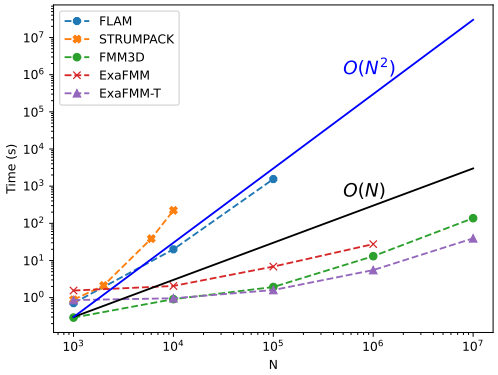
The Software Landscape for Fast Algorithms

Numerous different groups have developed overlapping software for fast algorithms. For FMMs, leading analytical impelementations include the single-node multithreaded, 'FMM3D' and 'FMM2D' [57, 58], and the MPI accelerated 'ExaFMM' [80] packages. The ExaFMM project have also developed a single-node multithreaded kernel-independent FMM, 'ExaFMM-T' [76]. For algebraic methods, the landscape is sparser, with few parallel implementations. The standouts being the MPI accelerated 'STRUMPACK' [31], for HODLR and HSS matrices, the multithreaded single-node 'FLAM' [37], which supports algorithms for $\mathcal{H}^2$ matrices and the single-node multithreaded $\mathcal{H}^2$-lib software [8]. The majority of codes, except FLAM which is written in Matlab, are written in compiled languages, either Fortran, C or C++, with a minority supporting interfaces to higher level languages [76, 57].

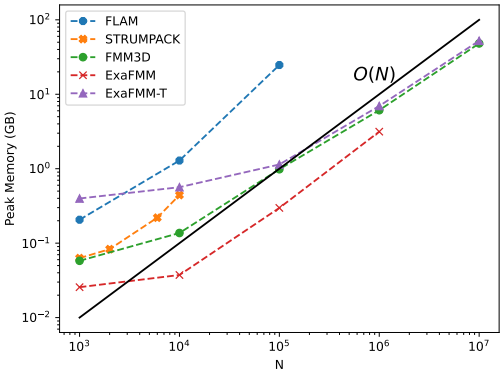
We compare these softwares for the calculation of (2.2) with N randomly distributed particles placed in a unit cube, $[0, 1]^3$, with uniform charge, $q_i = 1$, in figure (3.1) [1]. Experiments are taken on a single-node AMD Ryzen Threadripper 3970X 32 core processor with 250GB of memory. To avoid thread 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 our algebraic software experiments we exclude $\mathcal{H}^2$-lib from the comparison in figure (3.1) due to constraints on time, restricting our study FLAM, which we used to develop our fast direct solver code in chapter 7, and STRUMPACK which is the leading MPI-parallelised code.

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 (3.1a) plots the time to compute (2.2), including the time to create all required data structures. For FLAM this includes the time to factorise and store its explicit hierarchical matrix representation of (2.2) which is a requirement of the software. Strictly speaking, STRUMPACK's HSS and HODLR matrices are not applicable to (2.2), as it is a strongly admissable problem. Furthermore the default running mode of STRUMPACK uses randomised sampling, which repeatedly multiplies a given dense input matrix with a randomly generated matrix in order to estimate its compressed form. This leads to a $O(N^2 r)$ complexity, where r is a maximum rank of a low-rank block based on a user defined tolerance, when calculating the HSS or HODLR representations. We see the effects of this in super-quadratic factorisation times for increasing N. The slow compression can be tuned by a user defined

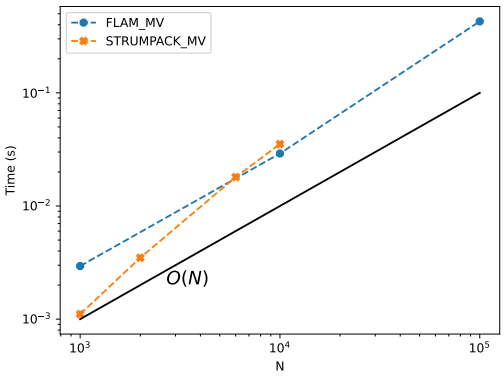
[1] Our experiments are available at `https://github.com/skailasa/phd-thesis/code/`



(a) Total Runtime



(b) Peak Memory Usage



(c) Matrix vector product time for algebraic methods.

Figure 3.1: Comparison of parallel software for fast algorithms, used to calculate (2.2). Simulations were compared for convergence times $\leq 2 \times 10^3$ s that could fit into memory.

'fast' matrix vector product for low-rank sub blocks of the matrix which could use a low-rank decomposition such as a multipole type expansion, or a purely numerical scheme such as an SVD, to compress these sub-blocks, however we evaluate the software as presented. We observe that FLAM and STRUMPACK, run in its default setting, are impractical for computing even moderately sized matrix vector products, i.e. $N \geq O(10^5)$. With runtimes times for factorisation exceeding our imposed limit of 2×10^3 s. Unless a hierarchical matrix representation is expressly required, such as for the development of a fast direct solver, we think it is unlikely that a typical user would attempt to optimise STRUMPACK's matrix vector product, as the FMM software tested beat both algebraic softwares by several orders of magnitude in runtime out of the box. Thus making it considerably faster in terms of development time to use these to develop an iterative solver, or evaluate fast matrix vector products.

As expected from theory, the analytical FMMs, ExaFMM and FMM3D, consume less memory than the black box ExaFMM-T (fig. (3.1b)). 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 [47], as well as using SIMD vectorisation for kernel evaluations [76]. Similarly by stacking T^{M2L} as in (2.13), 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. It is unclear from STRUMPACK's documentation as to whether they use SIMD vectorisations to accelerate kernel evaluations, however they do offer limited GPU offloading but not for algorithms defined on HSS or HODLR matrices. FLAM is multithreaded, however is written for implementation simplicity and algorithm testing and is largely unoptimised for HPC.

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. FLAM, written in Matlab, is easy to install with numerous code examples as well as good documentation. Most softwares, except STRUMPACK which is distributed via the SPACK package manager and FLAM which is a Matlab package, 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 (3.1) it's clear that software choice can have dramatic impact on runtime and memory scaling regardless of algorithmic choice. However software choice is unlikely to be purely guided by performance considerations, as we have seen each example software have different levels of user support in terms of example code, documentation and code comments, and thus have different practical usability. For the Laplace problem of (2.2) all softwares can be used more or less out of the box, however oscillatory Helmholtz kernels are only natively supported by a few [80, 76, 58, 57]. 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 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 are unfortunately limited to single-node systems.

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. 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, the high-level architecture of which is illustrated in figure (3.2). 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 initially explored Python, in conjunction with high-performance 'just in time' (JIT) code generation tool Numba [41], as a base programming model for our infrastructure. We document this through our experience building a Python based KIFMM in chapter 4. Although we evaluate a single code-generation tool, Numba, we believe our experience generalises to other approaches that rely on similar JIT compiled technologies for the acceleration of code written in high-level languages, such as Julia. We found the constraints placed on our software using this approach to be too restrictive for the infrastructure we were planning and therefore pivoted to Rust, a modern compiled language designed explicitly with code portability in mind, as the base language of our infrastructure. We believe that Rust offers a powerful and productive alternative to older compiled languages for the development of scientific software, with similar runtime performance. We explore its beneficial properties in chapter 5. Most important of these is its build system, Cargo, which is specified by a simple TOML file. This makes it easy to deploy Rust codes on most common platforms without the need for complex build CMake style build scripts. With users able to develop software locally, before deploying to a cluster, while being able to expect high-performance 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. Importantly, it is also easy to develop wrappers from Rust to Python using its C foreign function interface [50], making it easy for novice programmers to develop and deploy our solvers on a range of devices, from their laptops to supercomputing clusters. This combined Rust and Python approach combines the usability and portability features we are looking for, commonly associated with interpreted languages, with the performance of more complex compiled languages such as C++.

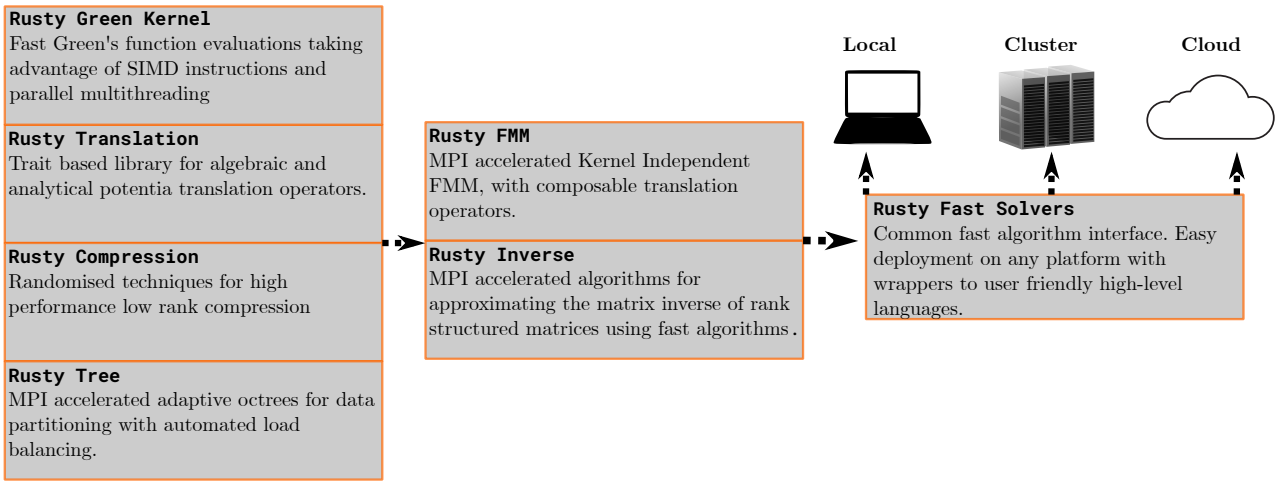


Figure 3.2: Hierarchy of our proposed infrastructure, with key functionality separated into individual libraries we maximise code re-use across projects. Once completed using Rust's code generation infrastructure, users will be able to deploy our software from desktop workstations, to supercomputing clusters and expect scalable high-performance code, while being able to write application code in Python.

Software Study: PyExaFMM, a Python Fast Multipole Method

Python[1][2] 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. 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.

With this in mind we sought to assess the suitability of Python as a base language for fast algorithm software. With stable MPI bindings for multi-node development [20], 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. The Numba compiler For Python was written to targets and optimise code written with NumPy's n-dimensional array, or 'ndarray', data structure, which are homogenously typed containers stored contiguously in memory [41]. Its power comes from the ability to generate multithreaded architecture optimised 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 [39].

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 optimisations such as auto

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

[2]This section is adapted from a paper recently submitted to Computing in Science and Engineering [39]

vectorization, automatically applying them if they are available on the target hardware [42]. 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 [41]. 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 (4.2), (4.3) and (4.1) 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 optimisations, would allow for significantly faster workflows than possible with a compiled language.

Figure (4.3) 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 (4.3) and (4.1) 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 (4.3). This listing shows three implementations of the same logic, a function that generates a random matrix $A \in \mathbb{R}^{100\times100}$ 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 implemen-

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

tations algorithm 1 and algorithm 2 aren't distinguished by the Numba compiler, and both pay a cost to call subroutines defined outside of their function body. However, algorithm 1 pays a small additional (un)boxing cost in order manipulate a globally defined Numba compatible dictionary, in comparison to a locally defined one in algorithm 2. The *nested* function in algorithm 3 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* optimisation, which is picked up by LLVM at compile time.

The runtimes of all three implementations are shown in table (4.1) 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 $A \in \mathbb{R}^{1000\times1000}$ 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 $A \in \mathbb{R}^{1\times1}$ 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.

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

Table 4.1: Testing the effect of inlining and (un)boxing with dense matrix vector products in double precision using implementations from listing (4.3).

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

[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

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 optimise 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' [76], or 'linear' [70] (see chapter 6). 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 neighbours and siblings easy, as one has to just follow pointers[5]. 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 (4.1) 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, T^{P2M}, T^{L2P}, T^{M2P} and T^{P2L} operators, $d-2$ interactions to call the T^{M2L} and T^{L2L} operators, and d interactions for the T^{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

[5]'Siblings' are defined as nodes which share a parent, and 'neighbours' are defined as adjacent nodes which may not share a parent.

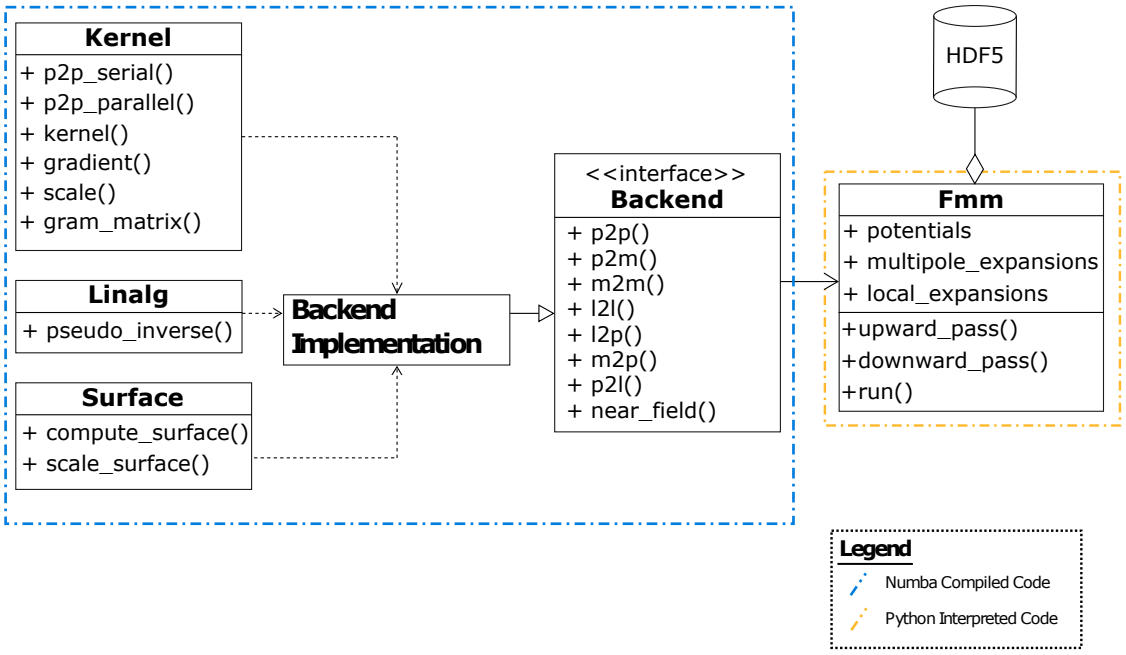


Figure 4.1: Simplified UML model of all PyExaFMM components. Trees and other precomputed quantities are stored in a HDF5 database. The 'Fmm' object acts as the user interface, all other components are modules consisting of methods operating on arrays, adapted from [39].

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.

Multithreading in Numba

Numba enables multithreading via a simple parallel for loop syntax (see listing (4.1)) 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 (4.1). 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 [46]. 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.

Parallising PyExaFMM

The T^{P2M}, T^{P2L}, T^{M2P} and T^{L2P} all rely on the $P2P$ operator, as this computes (2.2) over their respective sources and targets, and are parallelized over their targets the leaf nodes. For the T^{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 T^{L2P}'s targets, increases with the number of calculations in each thread and hence the expansion order p. In an experiment with 32768 leaves, the maximum number of points per leaf, $n_{crit} = 150$, and expansion order $P = 10$, 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, T^{M2L} and T^{M2P} operators. For example, allocating an array large enough to store the maximum possible number of source particle coordinates in double precision for the T^{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 T^{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 T^{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.

The matrices involved in the T^{M2M} and T^{L2L} operators can be precomputed and scaled at each level [76], and their application is parallelized over all nodes at a given level. Multithreading in this way means that we call the P2P, T^{P2M}, T^{M2P}, T^{L2P} and near field operators once during the algorithm, the T^{M2L} and T^{L2L} are called $d - 2$ times, and the T^{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 (4.2) 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 T^{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.

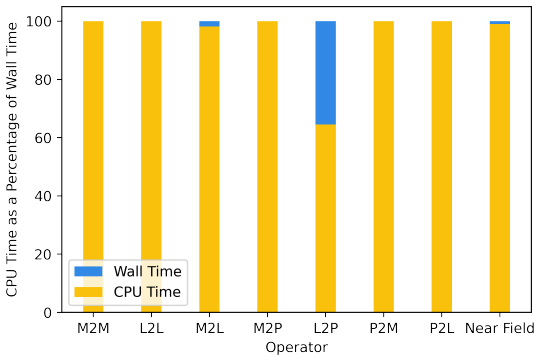


Figure 4.2: CPU time as a percentage of wall time for operators. CPU time is defined as the time in which the algorithm runs pure Numba compiled functions. Wall time is CPU time in addition to the time taken to return control to the Python interpreter, adapted from [39].

Conclusion

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 optimise a Numba implementation, and arguably more than many in its intended target 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 JIT. We believe that this is offered by Rust, which despite being a compiled language, that retains many of these features.

Listing 4.1: 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
```

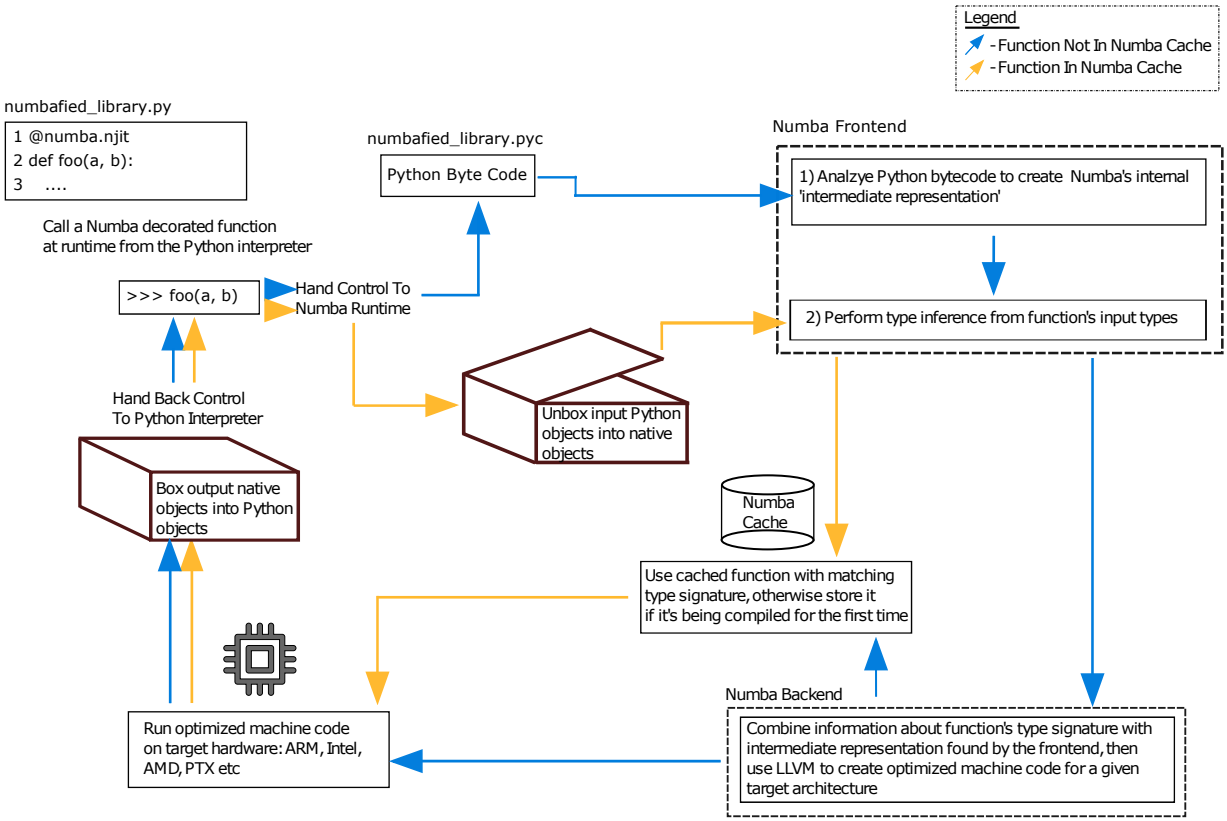


Figure 4.3: Simplified execution path when calling a Numba compiled function from the Python interpreter. The blue path is only taken if the function hasn't been called before. The orange path is taken if a compiled version with the correct type signature already exists in the Numba cache, adapted from [39].

```
    # Numpy is configured to run single
    # threaded.
    for _ in numba.prange(10):
        B = A @ A
```

Listing 4.2: 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
```

Listing 4.3: Three ways of writing a trivial algorithm that passes around a vector, while performing some computations.

```
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
```

Rust, a Modern Programming Language for Scientific Computing

In this chapter we review the attributes of Rust that make it amenable for scientific computing, contrasting the experience of building and writing software with traditionally preferred compiled languages, specifically C, C++ and Fortran which are preferred implementation languages in scientific computing.

As we observed in chapter 4, interpreted languages, despite their portability, great usability benefits and large ecosystems of numerical libraries, are constrained when used to implement high-performance scientific codes. For problems in which this overhead is untenable, compiled languages, such as C, C++ and Fortran are preferred. These languages require greater software engineering expertise, as developers are responsible for allocating memory, installing third party libraries, as well as building their software to target different hardware. These languages in a sense allow a developer to 'do anything', of course only if one knows how to do it. The higher software engineering barrier manifests in a dizzying array of compilers, build systems, package managers, testing and documentation libraries and code organisation techniques. The notable thing about all of these optional choices is that it is relatively unclear which is the *preferred* way of doing things, with novice developers likely to find a development strategy that 'simply works' and stick to it.

Rust stands in contrast to these traditionally preferred compiled languages for high performance scientific computing. Although it is comparably fast, it is relatively *inflexible*. With a strongly preferred way of organising, testing, documenting and deploying software. This leads to significantly more uniform Rust code across projects and a steep, though relatively shorter, learning curve. This uniformity makes Rust code easier to share and port across different operating systems and hardware targets.

Package Managers and Build Systems

For simple programs, it's tempting to use a compiler directly to create an executable, as in listing (5.1). However, as a project's codebase expands with files defined in multiple directories and calls to external libraries often written in other programming languages, this simple one line appeal to a compiler will no longer be sufficient. One could always download their requisite software and install globally over the machine using a system package manager, and attempt to recompile. However, this quickly becomes untenable if one is developing for a range of hardware and operating system targets, or one needs to use different versions of external binaries and libraries. Therefore, software is usually constructed using a 'build system'. This is catch all term that refers to a program that takes source files as input and produces a deployable set of binaries or libraries as an output. Classically, build systems were

based on 'Make' and 'Autotools', these softwares generate 'Makefiles' - which are recipes for constructing a piece of software given a set of source files and external dependencies. These are robust tools, but the onus is squarely with a developer for ensuring that all dependent libraries are visible to Make, and that the external packages are all of compatible versions.

To manage this complexity, builds are often defined using a metabuild system, most commonly CMake. CMake is a scripting language, and as a meta build system it takes a specification of local and third party dependencies and hardware targets, and generates Makefiles. CMake gives developers a great deal of flexibility, it is multi-platform, and language agnostic, however using it directly is not straightforward. Indeed, there is a significant body of literature discussing best practices with CMake [64]. However, CMake is not responsible for downloading and installing third party packages or verifying their relative compatibility, implementing its best practices is again left to users.

Approaches for reliable builds vary amongst projects, ranging from 'low tech' readme driven solutions, in which a user follows a recipe of instructions, to more modern automated solutions. Figure (5.1) provides an overview of the different approaches taken. The lack of a uniform standard means that some projects go as far as to implement a custom build system, such as the Boost library [5].

Keeping on top of packages, which are constantly being iterated upon independently, and may support different features with each release, is nearly impossible to do manually. Furthermore, one may want to support a specific set of packages, and respectively versions, for a given hardware or operating system target, but a different set for a different build of the same software. This has led to the development of 'package managers' which are a catch all term for softwares that can download and install required software for you, and often verify whether version constraints are satisfied too. Package managers can be delineated into 'system package managers', which download operating system specific packages written in any language globally on your machine, and 'language specific package managers' which focus only on packages written in a given target language, but are operating system agnostic from a user's perspective.

In terms of system package managers Linux examples include apt, yum and debian, for MacOS there is Homebrew, and Chocolatey for Windows. These can handle both source installations, in which source code is compiled upon download, as well as binary installation, in which pre-built binaries matching your hardware constraints are installed. Binary installation is often preferred, as it is faster, and often reflects a stable software release. Language specific package managers, such as pip for Python, can handle dependencies written in these languages only. Developing your own packages for system package managers is not developer friendly, official package repositories of Linux package managers are moderated by their respective maintainers, though it is possible to set up a personal repository this is quite a sophisticated approach for simple research outputs. The simple alternative, pursued by up to 26 % of surveyed C++ softwares (see fig. (5.1)), is to simply add third party software as a submodule. Often to ease installation, C++ libraries are shipped as 'header only' libraries, this makes them easy to install, requiring only an *#include*. However by using such techniques a project can quickly grow to contain thousands, to millions of lines of code, much of it in dependencies. Compiling from source has the potential to be a painstakingly slow process, and must be repeated for every combination of hardware and operating system targets one wants to run on.

With the exception of Fortran, which has made recent strides to develop a stan-

Installing and Building Software in C++/Fortran

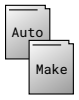
Builds for open source software are often **Readme Driven.** In this common approach developers provide a set of instructions for how to install a project's dependencies and the project itself. Common approaches include:

Dependency Management Methods

1) Simply add all dependencies to source tree of your project. Projects with a large number of dependencies can grow to have millions of lines of code, which can have a drastic effect on compilation times. Large C++ projects for example can take between several minutes to several hours to compile from source

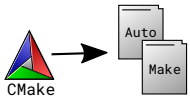
2) Use a system package manager, and source from repositories such as GitHub to install globally. These can then be found by build systems. This makes it difficult to build isolated build environments, and configure builds with different versions, or sets, of dependencies.

Build Methods



1) Developer provided Make and Autotools scripts. lowest barrier to entry but build system will use globally installed dependency libraries, unless alternative provided. Makes multi-platform builds, or those using different software versions challenging.

2) Developer provided CMake scripts, to generate build systems for different environments. Again, reliant on globally installed dependency libraries.



Neither of these methods can check for dependency conflicts, which is left up to the developer to resolve.

Modern Package Management

Modern package managers allow for maximum safety and flexibility. They support multiple operating systems, compilers, build systems, and hardware microarchitetures, and are usually defined by recipes written in a simple scripting language such as Python, and can be used to generate build system scripts such as Makefiles and CMake scripts. The also support dependency tree checking for conflicting requirements, leading to stable builds. Two popular leading tools for HPC in C++ and Fortran are **Spack** and **Conan**.

Spack

+ Supported on Linux and MacOS.
+ Package recipes specified with Python scripts.
+ Over 5000 commonly used packages available.
+ Supports isolated environments, supporting building packages with from specific constraints, and clean global environment.
+ Can target hardware microarchitectures.
+ Growing support for binary package installation, though not available on all hardware targets.
+ Compatible with all major build systems, such as CMake and SCons
- No Windows support.

Conan

+ Supported on Linux, MacOS and Windows
+ Package recipes specified with Python scripts.
+ Supports isolated environments, supporting building packages with from specific constraints, and clean global environment.
+ Support for binary packages, speeding up installation times.
+ Compatible with all major build systems, such as CMake and SCons.

However, we note that even in 2021 modern package managers are still only used in a tiny fraction of all projects. For example Conan, only accounts for 5% of all C++ projects surveyed by JetBrains[1] in comparison to 21% of projects still relying on the system package manager, 26% simply including a dependencies source code as a part of a project's source tree, 23% using the readme driven build of each specific dependency, and 21% installing non-optimised precompiled binaries from the internet. Indeed only a small minority, 22%, used a package manager of any kind, with no solution taking a majority of even this market share. This is in stark contrast to theuniformity of the situation in Rust, in which all projects use Cargo as a build system and package manager and rustc as a compiler.

1. https://www.jetbrains.com/lp/devecosystem-2021/cpp/

Figure 5.1: An overview of building software in other compiled languages.

dardised modern package manager and build system, inspired by Rust's Cargo [26], C and C++ do not have a single officially supported package manager or build system. The resulting landscape is a multitude of package managers [65, 74, 19] and build systems [72, 6, 63] a few of which we have cited here, all of which replicate each others functionality, none of which are universally accepted or implemented across projects nor officially supported by the C++ software foundation.

Listing 5.1: Compiling a simple `source_code.cpp` into a `compiled_binary` file from a terminal.

```
>>> gcc -o compiled_binary source_code.cpp
>>> # Can then run with
>>> ./compiled_binary
```

Recent years have seen the bundling of package managers *with* build systems, resulting in softwares that can simply take source files, and a set of dependencies, and resolve a single binary output for a given hardware and software target. Examples include Conda for Python, and Rust's Cargo system. These modern efforts are able to compile both source and binary packages for all operating systems, and can target a wide range of hardware architectures. Furthermore, the trend has been towards the specification of project dependencies in a single structured text file (see listing (5.2)), which is then handled by the package manager with no further user effort. Efforts have been in this direction for older compiled languages, the most notable examples being Spack and Conan (see fig. (5.1)). Importantly, in the case of Rust, Cargo is officially supported and shipped as a core part of its runtime. In contrast to C, C++ and Fortran, Rust has a *single* officially supported compiler, rustc. By taking global decisions for all software written in Rust, a significant burden is removed from developers of Rust software, similar to the situation in many interpreted languages. Indeed Cargo builds are often executed in a *single line*, eradicating the complex readme driven builds common in other compiled languages. For the project specified by listing (5.2) can be compiled from a terminal with the command: `cargo build`.

Listing 5.2: An example of a simple `Cargo.toml` file for a Rust project, with source as well as binary dependencies.

```
[package]
name = "new_package"
version = "1.1.0"
authors = ["Foo Bar"]
edition = "2018"
description = "A great new package"
license = "BSD-3-Clause"
homepage = "https://github.com/foo_bar/new_package"
repository = "https://github.com/foo_bar/new_package"
keywords = ["numerics"]
categories = ["mathematics", "science"]

[dependencies]
# Build from binary on crates.io
memoffset = "0.6"
rand = "0.8.4"
itertools = "0.10"
vtkio = "0.6"

# Build from source on GitHub
mpi = { git = "https://github.com/rsmpi/rsmpi", branch = "main"}
```

Code Organisation and Quality

Rust introduces many ergonomic features for code organization. The most novel features, which may not be familiar to those coming from other compiled languages, are the concepts of traits, lifetimes and the borrow checker.

Traits

Traits are Rust's system for specifying shared behaviour and building abstraction, constituting a wholesale replacement of object oriented programming, with its inheritance based hierarchies. Traits only enforce *behaviour*, and therefore are strictly less brittle than object orientation which enforces a type. We provide some example syntax in listing (5.3), and contrast it to equivalent object oriented code in listing (5.4). We notice that the object oriented code has a built in hierarchy, which means that adding shared behaviour will effect our `MyType` type, and everything that subsequently depends on it, in contrast to the trait based code in which we can inject new behaviour into `MyType`, its subsequent dependents we only need to know about the traits and their associated interfaces that they themselves rely on. This means that one can focus solely on the behaviour implicit in a given Trait, rather than having to comb through a potentially complex hierarchy of objects and inheritance to understand what a given line of code is doing, making large Rust projects significantly more readable than their object oriented equivalents.

Traits can be seen to specify shared behaviour in a bottom up manner, as opposed to top down object orientation, new features can be injected into existing types. This is useful for scientific programming, as we are usually concerned with data oriented programming. As we saw in chapter 4 exploring data oriented programming in Python, object oriented design obfuscates operations on the data itself behind abstraction.

Listing 5.3: An example of a trait implemented on a custom type.

```
// Custom type declaration, with its own set of attributes
// and methods
pub struct MyType {
    pub attribute String;

    pub fn bar() {
        println!("I'm a default implementation defined on this type.")
    }
};

// Traits specify an interface
pub trait MyTrait {
    fn foo(&self) -> String;
};

pub trait MyOtherTrait {
    fn baz(&self) -> String;
}

// We can implement this interface for our type
impl MyTrait for MyType {
    pub fn foo() {
        format!("I'm a required method for this trait.")
    }
};
```

```
impl MyOtherTrait for MyType {
    pub fn baz() {
        format!("I'm a required method for this trait.")
    }
}

fn main () {
    let mytype = MyType{attribute: "boz"};

    // Use behaviour from MyTrait
    println!(mytype.foo());

    // Use behaviour from MyOtherTrait
    println!(mytype.baz());

    // Use behaviour defined by MyType itself
    println!(mytype.bar());
}
```

Listing 5.4: An example of behaviour in listing (5.3) implemented in an object oriented manner

```
from abc import ABC, abstractmethod

class MyTrait(ABC):
    def __init__(self):
        pass

    @abstractmethod
    def foo(self):
        """A required method for this class"""
        pass

class MyOtherTrait(ABC):
    def __init__(self):
        pass

    @abstractmethod
    def baz(self):
        """A required method for this class"""
        pass

class MyType(MyTrait, MyOtherTrait):
    def __init__(self, attribute):
        self.attribute = attribute

    def bar(self):
        print("I'm a default implementation defined on this type.")

    def bar(self):
        print("I'm a required method for the MyTrait class.")

    def baz(self):
        print("I'm a required method for the MyTrait class.")


def main ():
    mytype = MyType("boz")

    # Use behaviour from MyTrait
    print(mytype.foo())
```

```
    # Use behaviour from MyOtherTrait
    print(mytype.baz())

    # Use behaviour defined by MyType itself
    print(mytype.bar())
```

This isn't to say similar syntax isn't available in other compiled languages. In C++20, 'concepts' were introduced as a trait like mechanism to specify interfaces. However, as with many things in C++, this isn't enforced and it is up to a user to choose to implement their code in this manner. This is borne out in the fact that concepts are *nominally typed*, and therefore don't enforce behaviour as in Rust's *structurally typed* traits. The implication of this being that a type may 'accidentally' implement a concept, if it happens to define its relevant methods. A consequence of this is that a valid C++ program can contain a confusing mix of trait based, and object oriented code, with a a reader then dependent on documentation to understand a software's behaviour, and what has been enforced and where, by the developer.

Lifetimes and the Borrow Checker

Another new Rust concept for developers coming from other compiled languages is the idea of 'lifetimes' and 'ownership'. Every reference in Rust has an associated lifetime, and a singular 'owner'. Which enforce the programming pattern of 'resource allocation is initialisation' (RAII) as a feature of the Rust compiler. The basic rule is that references are owned within a scope, and dropped when out of scope. We provide a basic example in listing (5.5), in a Rust context RAII is better encapsulated by another acronym 'destructors run at exit scope' (DRES).

Listing 5.5: A demonstration of basic rules of borrowing and ownership.

```
fn create_vec() -> Vec<i32> {
    // Create a vector, and pass ownership to caller
    let mut vec = Vec::new();
    let n = 5;
    for i in 1..n {
        vec.push(i)
    }
    vec
}

fn processor_borrows(vec: &Vec<i32>) {
    // Function doesn't take ownership of vector

    for i in vec.iter() {
        println!("i: {}", i);
    }
}

fn  processor_borrows_mut(vec: &mut Vec<i32>) {
    // Function doesn't take ownership of vector,
    // but can mutate it via a mutable reference

    for i in 0..vec.len() {
        vec[i] = vec[i]*2;
    }
}
```

```
fn processor_owns(vec: Vec<i32>) {

    for i in vec.iter() {
        println!("i: {}", i);
    }
}


fn main() {
    let vec = create_vec();
    processor_owns(vec);

    // Calling functions below now will cause an error,
    // as ownership of vec has been passed
    processor_borrows(&vec);
    processor_borrows_mut(&mut vec);
    // to 'processor_owns', resulting in error message:
    // 43 |     processor_borrows(&vec)
    //                           ^^^^ value borrowed here after move

    // The below is valid, as there is still only one mutable reference
    let mut vec2 = create_vec();
    processor_borrows_mut(&mut vec2);
    processor_borrows(&vec2);

    // The following will not work, as we create two mutable references
    let r1 = &mut vec2;
    let r2 = &mut vec2;
    processor_borrows_mut(r1);
    processor_borrows_mut(r2);
    // obtaining the following error:
    // 53 |     let r1 = &mut vec2;
    //    |              ---------- first mutable borrow occurs here
    // 54 |     let r2 = &mut vec2;
    //    |              ^^^^^^^^^^ second mutable borrow occurs here
}
```

Lifetimes are a Rust concept that guarantees that any references to a resource live at least as long as the resource itself. The main aim of this to prevent dangling references, whereby references to unintended data, or de-allocated data, are not present in the compiled binary. This removes a large class of common bugs from Rust software such as dangling pointers, and double free errors. We illustrate this in listing (5.6), which won't compile as the inner scope defines a value `x`, which doesn't survive into the outer scope.

Listing 5.6: A demonstration of lifetimes as a function of scope. This example is adapted from the Cargo book [40].

```
fn main() {
    let r;

    // Lifetimes are defined by scope.
    {
        let x = 5;
        r = &x
    }
    println!("r: {}", r);
    // Results in error
    // 6 |         r = &x;
    // |             ^^ borrowed value does not live long enough
```

```
    // 7|   }
}
```

Rust's compiler enforces ownership and lifetime rules uses a program called the 'borrow checker' to ensure that all references, and lifetimes, are referring to valid memory locations at *compile time* without the need for a runtime garbage collector. This is a huge advantage over other compiled languages, for example memory related bugs have been found to constitute as much as 70 % of all security bugs at Microsoft [52]. From a scientific programming perspective, handling memory additionally bogs down algorithm development, iteration, and ultimately publication, and was one of the major contributing factors in the push towards interpreted languages in recent decades. Again, this is not to say similar features do not exist in other compiled languages. C++ has its notion of 'smart pointers', which enforce RAII principles too, however as with other features in C++, this is a library feature rather than a core part of its compiler, or language specification, making it optional and unenforced.

The benefits of Rust's memory system extend to parallel code too. A wide variety of programming paradigms for parallel computation with threads exist, in scientific software we often use OpenMP, which uses the shared memory paradigm, such that all threads operate on the same data. In this setting one of the most common bugs is a data race condition, whereby multiple threads attempt to write to the same memory location. Rust's ownership system enforces the atomicity of all memory operations, by passing ownership between threads, as only a single mutable reference can exist to a piece of memory Rust is able to prevent data races at compile time. Although atomic operations are available in OpenMP, or other shared memory systems, the benefit of Rust is that we know that our compiled code *cannot* contain this bug, without having to rely on unit tests for expected behaviour.

Code Quality

Rust's runtime includes a test runner, a documentation generator, and a code formatter. As with other Rust features, these are maintained in lock step with the language specification, and with reference to other Rust developments. This imposes universal constraints on all Rust projects, allowing for objectively defined 'good' Rust code, rather than relying on various standards of best practices that vary between projects and organisations.

Rust's Scientific Ecosystem and Foreign Libraries

Despite being a young language, Rust already supports a mature ecosystem of libraries for scientific computing with high-level multithreading support [67], numerical data containers [55], and tools for generating interfaces to Python via its C ABI [50].

Many tools are yet to be ported into native Rust, however high quality bindings exist for core tools such as MPI [66], BLAS and LAPACK [7], which are relatively easy to interface into Rust via its C ABI. The problem with interfacing with tools written in other languages is again related to building software, however Cargo offers tools to build software written in other languages and integrate it with Rust code via the `build` crate, which allows one to leverage existing build systems written for software written in foreign languages. This detracts from the benefits offered by Cargo as a unified package manager and build system, raising similar problems to

those encountered when building software in other compiled languages. However, we observe that this remains a concern of the software's developer, who is responsible for providing build scripts for the operating systems and hardware platforms that they wish to support, and from a downstream user perspective their build process remains the same as with pure Rust packages, where the dependency is defined in their `Cargo.toml` file.

We also note that Rust is missing key tools for scientific computing, such as a code generation for GPUs, as well as for advanced optimised advanced linear algebra routines, especially for sparse matrices. However, both of these applications are an active area of development.

Conclusion

Rust's syntax and program structure are strongly opinionated, its build system emphasises simplicity and portability, and its ecosystem for scientific computing is rapidly developing. A small software team, as is common in academic settings, writing in Rust are effectively able to maintain and deploy Rust projects to a wide variety of platforms, from laptops to supercomputers. Rust's simplified build process results in minimal configuration for users, encouraging the widespread adoption of Rust projects.

Software Study: Rusty Tree, a Rust Based Parallel Octree

In this chapter we present Rusty Tree, an implementation of parallel octrees, distributed with MPI, with a complete Python interface. Octrees are a foundational data structure for fast algorithms in $\mathbb{R}^3$, for which writing software that can be distributed across parallel computing systems is challenging. This is demonstrated by the limited availability of open-source software for parallel octree construction [13, 22]. This is in spite of their ubiquity in scientific computing applications from finite-element methods to many-body algorithms [70]. Rusty Tree is a proof of concept for Rust as a tool for high-performance computational science. We document our experience in developing the software, and use it as a tool to explore the current landscape of scientific computing with Rust. From multithreading, and developing a Python interface, to distributing applications with MPI, writing Rusty Tree involved using many tools from Rust's scientific computing ecosystem.

Parallel Octrees

As mentioned in chapter 4 there are multiple representations of octrees. We choose a linear octree representation, in which we discard interior nodes and store a list of leaves, as in our previous work PyExaFMM [39]. These lie in contrast to pointer based octrees in which each node stores pointers to its parent as well as children, which have a relatively higher storage cost, in addition to a synchronisation and communication overhead for parallel implementations that must keep track of pointers across nodes [73].

Pointer based octrees are often constructed 'top down'. A user specifies a threshold for the maximum number of particles in a leaf node, n_{crit}, and a unit cube encapsulating the problem domain is refined until this is satisfied. After refinement one can optionally 'balance' adjacent leaf nodes such that adjacent nodes obey a relative size constraint. Balancing is useful as it allows us to restrict the number of pre-computed translational operators for fast algorithms. However, translating this logic to a parallel setting is challenging. Constructing octrees 'top down' involve significant data communication. As the point distribution on each processor isn't known a priori, each processor is responsible for constructing its own octree from a random batch of points, reconciling the trees across nodes then requiring a global parallel merge. Furthermore, there are likely to be significant overlaps and duplicates in the nodes of each processor's octree which would result further refinement after the parallel merge in order to satisfy the n_{crit} constraint. Balancing the octree, as well as load-balancing the work assigned to each processor, must be done as post-processing steps.

For these reasons Sundar et al developed 'bottom up' tree construction [70]. 'Bottom up' refers to the fact that octrees are constructed from the leaf level upwards, which allows us to incorporate the initial data distribution into the tree construction. In order to do this they use a space-filling curve that passes through each octant, for which they choose a Morton encoding. Figure (6.1) describes how a Morton encoding can be found for a given octree node, and figure (6.3) describes how we optimise the calculation of Morton encodings in practice, using lookup tables and shared memory parallelism. Morton encoded nodes are referred to as 'Morton keys', and their ordering is enforced by algorithm (3) when they do not share an 'anchor' (see. fig (6.1) for the definition of an anchor). If they do share an anchor, the finer node is defined to be greater. The appendix of [70] describes further properties of Morton encodings that are useful for spatial decompositions. The main relations of a given Morton key are summarised in table (6.1)

Relation	Definition
Siblings(N)	Keys that share a parent with N
Descendants(N)	All descendants of N
Ancestors(N)	All ancestors of N
FinestAncestors(N, M)	Finest shared ancestor of N and M.
FirstChild(N)	The 'first', in Morton order, child of N
LastChild(N)	The 'last', in Morton order, child of N
DeepestFirstDescendent(N)	The 'first', in Morton order, descendant of N at the leaf level
DeepestLastDescendent(N)	The 'last', in Morton order, descendant of N at the leaf level

Table 6.1: Definitions of Morton key relations.

For octree construction, our starting point is a set of distributed point data spread across n_p processors. Our strategy is then as follows:

1. Generate a Morton key for each point at the leaf level corresponding to a user defined octree depth.

2. Perform a parallel sort of these leaf octants using MPI, such that processors with adjacent MPI ranks have adjacent data.

3. *Linearise* leaves on each processor, such that their leaves do not overlap, in a manner that favours smaller leaves over larger ones, as in algorithm (4).

4. *Complete* the space defined by the leaves on each processor, and find the domain occupied by their leaves, see figure (6.2) and algorithm (5). Define the coarsest nodes of this set as *seeds*.

5. Construct a coarse *block tree*, by completing the space between the seeds with the minimum spanning set of nodes, described in algorithm (7). The nodes of this block tree are referred to as *blocks*. This gives us a coarse distributed complete linear octree that is based on the underlying data distribution.

6. Point data is communicated to the processor containing its associated block, and the blocks are refined based on n_{crit}, providing the final linear octree tree.

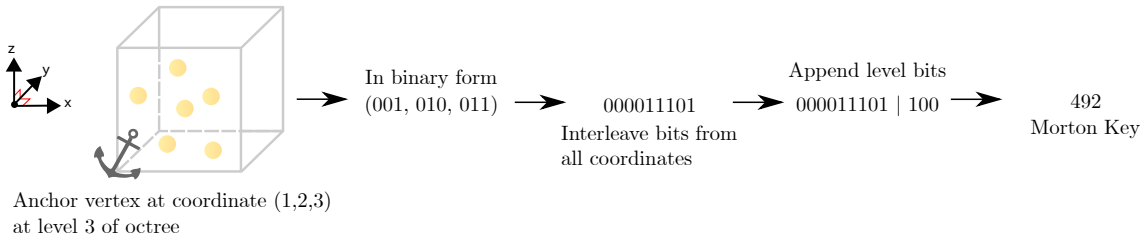


Figure 6.1: How to Morton encode a node in an octree. The node is described by an 'anchor' vertex, and its associated coordinate.

7. Optionally, the tree is *balanced*. This can be done for the local subtree on each processor using algorithm (6). The locally balanced trees can then be sorted in parallel again, and locally linearized as in step (3), to provide a globally balanced tree.

This approach is summarised in algorithm (2). The complexity of this process is bounded by the parallel sorts, which for randomly distributed point data, run in $O(N_{\text{leaf}} \log(N_{\text{leaf}}))$ work and $O(\frac{N_{\text{leaf}}}{n_p} \log(\frac{N_{\text{leaf}}}{n_p}) + n_p \log(n_p))$ time where N_{leaf} is the number of leaves in the final octree, and n_p is the number of processors. This makes the efficiency of the parallel sort a bottleneck in tree construction.

Algorithm 2 Construct Distributed Octree (Parallel)

Input: A distributed list of points L, and a parameter n_{crit} specifying the maximum number of points per octant.
Output: A complete linear octree, B.
Work: $O(n \log n)$, where n=len(L).
Storage: $O(n)$, where n=len(L).
$F \leftarrow$ [Octant(p, MaxDepth), $\forall p \in L$]
ParallelSort(F)
$B \leftarrow$ BlockPartition(F), using algorithm (8)
for $b \in B$ **do**
  if NumberOfPoints(b) $> n_{\text{crit}}$ **then**
    $B \leftarrow B - b +$ Children(b)
  end if
end for

# Optional Balancing over subtrees, f.
if Balance = True **then**
  for $f \in F$ **do**
    Balance(f), using algorithm (6)
  end for
  ParallelSort(F)
  for $f \in F$ **do**
    Linearise(f), using algorithm (4)
  end for
end if

Our strategy extends the original approach presented in [70] in two ways. Firstly, we use HykSort [69], a variant of parallel hypercube sort, which reduces global data communication cost in comparison to a parallel sample sort based on `MPIAllToAllv`.

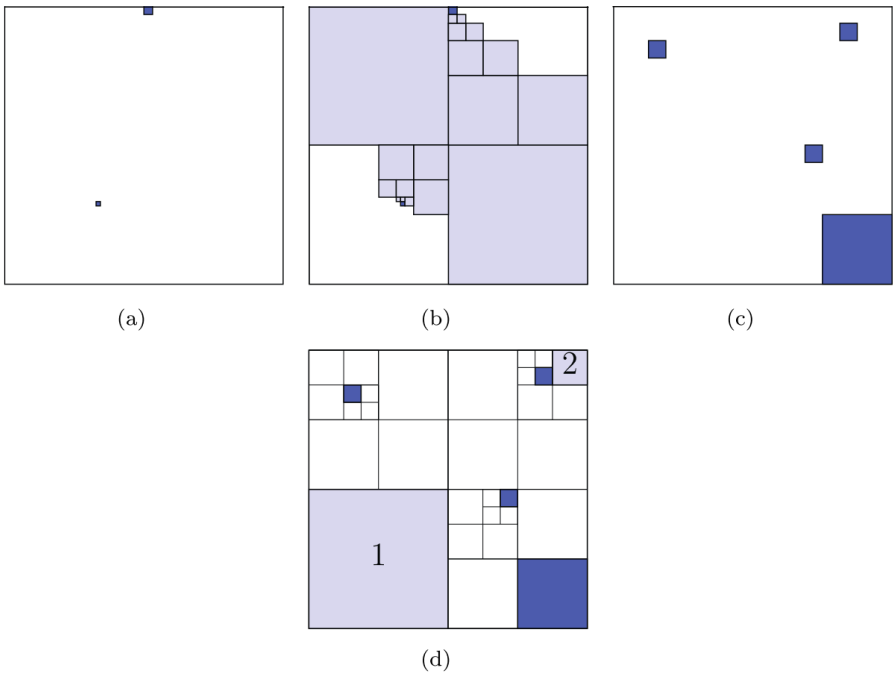


Figure 6.2: Adapted from [70], describing the algorithms in $\mathbb{R}^2$ for clarity. Figure (b) shows the minimal tree constructed using algorithm (5) where the two octants in (a) are used as an input. Figure (c) shows a selection of *seeds*, which are used as an input to algorithm (7), resulting in (d). Figure (d) also shows two octants generated to complete the domain. The first one is the coarsest ancestor of the least input octant which does not overlap with the least octant in the input. The second is the coarsest ancestor of the greatest possible octant, which doesn't overlap with it.

We cover the relative merits of HykSort in the next subsection. Secondly, balancing algorithms in the original implementation are based on a 'ripple propagation scheme'. The 'ripple effect' is the phenomenon that describes how octants can effect immediately adjacent octants if they are split, or combined, which can trigger further changes that must then be propagated across the tree. In distributed trees these changes must be communicated across the network, the communication patterns for which are relatively complex communication [73, 70]. We therefore replace this with an algorithm which, starting with an unbalanced tree, finds the locally balanced tree on each processor using algorithm (6), performs another parallel sort using HykSort, and removes overlaps, favouring smaller octants, using algorithm (4). This guarantees that the balance condition is maintained. This logic will recover a globally balanced tree with a significantly simpler communication pattern. This strategy has been previously been implemented in large scale parallel octree construction [47], and has been shown to obtain similar runtimes for balancing in comparison to ripple propagation based schemes [68].

Load balancing across processors, while maintaining Morton order is challenging. Currently, the data partitioning imposed by the parallel sort imposes an approximate load balance, even for irregular geometries, which we explore in figure (6.6c). However, in future we would like to extend our work to incorporate an explicit load balancing step. Our current idea is based on a modified sample sort, with splitter selection and bucket sizes chosen to reflect the mean load we wish to assign to each processor.

Algorithm 3 Morton Order (Sequential): Find the lesser of wo Morton keys with different anchors.

Input: Two Morton ids, A and B, with different anchors.
Output: R, the lesser Morton key.
$X_i \leftarrow (A_i \bigoplus B_i)$, $i \in x, y, z$
$e \leftarrow \arg\max(\lfloor \log_2(X_i) \rfloor)$
if $A_e < B_e$ **then**
  $R \leftarrow A$
else
  $R \leftarrow B$
end if

Algorithm 4 Remove Overlaps From Sorted List of Octants (Sequential) - `Linearise`. Favour smaller octants over larger overlapping octants.

Input: A sorted list of octants, W.
Output: R, an octree with no overlaps.
Work: $O(n)$, where n=len(W).
Storage: $O(n)$, where n=len(W).
for i $\leftarrow$ 1 to len(W) **do**
  if $W[i] \notin \{\text{Ancestors}(W[i+1]), W[i+1]\}$ **then**
    $R \leftarrow R + W[i]$
  end if
end for
$R \leftarrow R + W[\text{len}(i)]$

Optimised Morton Encoding

Lookup Tables

Consider a given set of indices describing the anchor of an octree node (x, y, z). The Morton encoding can be statically encoded in a lookup table.

For example, (x, y, z) = (4, 55, 132) which in binary is (100, 110111, 10000100). As we have to interleave these bits, we can add zero bits and perform a bitwise `or' operation to find the final Morton key:

$x_{shift} \mid y_{shift} \mid z_{shift} =$
01000000
| 00010010000010010010
| 10000000000000100000000

If x, y, z $\in$ [0, 255], these shifts can be stored in small statically stored lookup tables, and we can encode a Morton key for larger indices by considering their bits byte by byte.

This is a divide and conquer strategy, that has been shown to be faster than on-the-fly implementation using for-loops and bit-shifts[1].

1. https://www.forceflow.be/2013/10/07/morton-encodingdecoding-through-bit-interleaving-implementations/

Shared Memory Parallelism With Rayon

This lookup strategy can be performed in parallel over each anchor being encoded, which is easy to to using the Rayon crate for shared memory parallelism.

Rust defines `iterators', which are a functional programming abstraction to apply a single transformation to a set of data.

For example, consider the calculation of the squares of a vector of numbers, which are then summed. This can be expressed as:

```
let vec: Vec<i32> = vec![0,1,2,3,4,5];

let res = vec.iter()
   .map(|&i| i*i)
   .sum();
```

Rayon's main abstraction is to extend this to a parallel setting with *parallel iterators*:

```
let res = vec.par_iter() // <- only change
   .map(|&i| i*i)
   .sum();
```

Rayon's parallel iterators are an abstraction built on top of its work-stealing based parallel backend. Rayon fully incorporates Rust's multithreading safety features, and therefore guarantees data-race freedom.

Figure 6.3: Optimising Morton encodings using shared memory parallelism and lookup tables.

Algorithm 5 Construct a Minimal Linear Octree Between Two Octants (Sequential) - `CompleteRegion`.

Input: Two octants a and b, where $a > b$ in Morton order.
Output: R, minimal linear octree between a and b.
Work: $O(n \log n)$, where n=len(R).
Storage: $O(n)$, where n=len(R).
for $w \in W$ **do**
  if $a < w < b$ **and** $w \notin \{\text{Ancestors}(b)\}$ **then**
    $R \leftarrow R + w$
  else if $w \notin \{\text{Ancestors}(a), \text{Ancestors}(b)\}$ **then**
    $W \leftarrow W - w + \text{Children}(w)$
  end if
end for
Sort(R)

Algorithm 6 Balance a Local Octree (Sequential) - `Balance`. A 2:1 balancing is enforced, such that adjacent octants are at most twice as large as each other.

Input: A local octree W, on a given node.
Output: R, a 2:1 balanced octree.
Work: $O(n \log n)$, where n=len(R).
Storage: $O(n)$, where n=len(W).
R = Linearize(W)
for $l \leftarrow$ Depth to 1 **do**
  $Q \leftarrow \{x \in W | \text{Level}(x) = l\}$
  for $q \in Q$ **do**
    for $n \in \{\text{Neighbours}(q), q\}$ **do**
      if $n \notin R$ and Parent(n) $\notin R$ **then**
        $R \leftarrow R + \text{Parent}(n)$
        $R \leftarrow R + \text{Siblings}(\text{Parent}(n))$
      end if
    end for
  end for
end for

Algorithm 7 Construct a Complete Linear Octree From a Set of Seed Octants Spread Across Processors (Parallel) - `CompleteOctree`

Input: A distributed sorted list of seeds L.
Output: R, a complete linear octree.
Work: $O(n \log n)$, where n=len(R).
Storage: $O(n)$, where n=len(R).
$L \leftarrow$ Linearise(L), using algorithm (4).
if rank = 0 **then**
  L.push_front(FirstChild(FinestAncestors(DeepestFirstDescendent(root), L[1])))
end if
if rank = $n_p - 1$ **then**
  L.push_back(LastChild(FinestAncestors(DeepestLastDescendent(root), L[len(L)])))
end if
if rank ¿ 0 **then**
  Send(L[1], (rank-1))
end if
if rank ¡ $(n_p - 1)$ **then**
  L.push_back(Receive())
end if
for $i \leftarrow$ 1 to (len(L)-1) **do**
  $A \leftarrow$ CompleteRegion($L[i]$, $L[i+1]$), using algorithm (5)
end for
if rank = $n_p - 1$ **then**
  $R \leftarrow R + L$[L]
end if

Algorithm 8 Partitioning Octants Into Coarse Parallel Blocks (Parallel) - `BlockPartition`.

Input: A distributed list of octants F.
Output: A list of blocks G, F redistributed but the relative order of the octants is preserved.
Work: $O(n)$, where n=len(F).
Storage: $O(n)$, where n=len(F).
$T \leftarrow$ CompleteRegion($F[1]$, $F[\text{len}(F)]$), using algorithm (5)
$C \leftarrow \{x \in T | \forall y \in T, \text{Level}(x) \leq \text{Level}(y)\}$
$G \leftarrow$ CompleteOctree(C), using algorithm (7)
for $g \in G$ **do** weight(g) $\leftarrow$ len($F_{global} \cap \{g, \{\text{Descendents (g)}\}\}$)
end for
$F \leftarrow F_{global} \cap \{g, \{\text{Descendants}(g)\} \forall g \in G\}$

Efficient Parallel Sorts

Designing and implementing an efficient sorting algorithm that can scale to thousands of cores is difficult since it requires irregular data access, communication, and equal load-balance. As first presented by Sundar et al, parallel sorts in the creation of octrees were implemented using a variant of Sample Sort [70]. Briefly, this approach samples elements at each processor to create a set of $n_p - 1$ ordered 'splitters', which are shared across all processors and define a set of n_p buckets. This is followed by a global all-to-all communication call over all n_p processors to assign elements to their corresponding bucket. Finally, a local sort is performed at each bucket to produce a globally sorted array. SampleSort is well understood. However, its performance is quite sensitive to the selection of splitters, which can result in load imbalance. Most importantly, the all-to-all key redistribution scales linearly with the number of tasks and can congest the network. As a result sample sort may scale sub-optimally, especially when the communication volume approaches the available hardware limits [69].

An alternative approach is provided by HykSort [69], which is a generalisation of Quicksort over a hypercube [75] from 2-way splits to k-way splits, with the addition of an optimised algorithm to select splitters. Quicksort, Hyksort and Sample Sort are compared in figure (6.4). Instead of splitting the global array into n_p buckets, Hyksort splits it into $k < n_p$, and recursively sorts for each bucket. We notice that at each recursion step, each task communicates with just k other tasks. Indeed, for $k = 2$ we recover Quicksort over a hypercube. Both Hyksort, and the parallel splitter selection algorithm are provided in appendix (A), alongside complexity comparisons between Hyksort and Sample Sort. We note that Hyksort has a lower asymptotic complexity, with no terms that scale linearly in number of tasks in the MPI communicator, p, unlike Sample Sort.

Rusty Tree

Developed using the algorithms described in the previous sections, Rusty Tree can be found at `https://github.com/rusty-fast-solvers/rusty-tree`. It is consumed either as a Rust crate or a Python package, with binaries distributed via Conda and Rust's crates.io. We have so far tested Rusty Tree on Intel and Arm MacOS as well as various Linux platforms including Cray HPC systems running RHEL, and locally on Ubuntu workstations with AMD hardware. With a fully developed suite of tests,

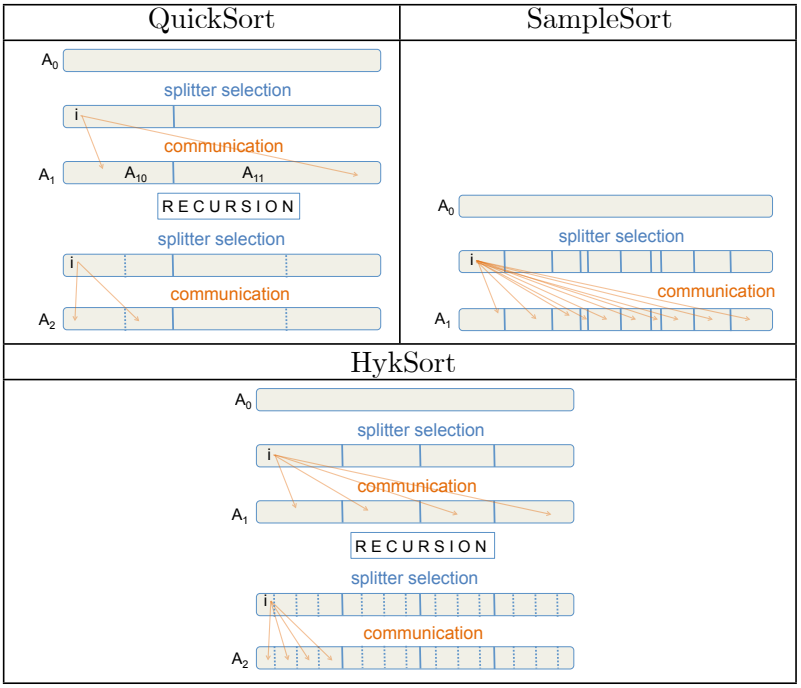


Figure 6.4: Communication pattern of Hyksort algorithm compared to a parallel Sample Sort, as well as a Quicksort over a hypercube, adapted from [69]. We see that HykSort results in a lower communication overhead than Sample Sort.

documentation and Python interfaces, Rusty Tree represents a model example of our final software infrastructure for fast solvers. Example usage in Python mirrors the Rust interface as shown in listings (6.1) and (6.2). Users are able use point data generated at runtime, or provide a HDF5 file for Rusty Tree to consume. Rusty Tree is also capable of outputting a VTK file for visualisation. We follow data oriented principles in Rusty Tree's design, with a shallow abstraction layer that acts as an interface to operations on point data, and encoded Morton indices. An abridged software architecture in UML style is shown in figure (6.5). In terms of lines of code, this translates into just 2578 lines of Rust and 688 lines of Python, including test code, with less than 100 lines of configuration code in terms of Toml and Yaml files specifying the Rust and Python builds.

The Python interfaces are built using Maturin, a tool for building and publishing Rust-based Python packages. It can do this in different ways, including the popular PyO3 [56] and rust-cpython [61] crates both of which offer Rust bindings for Python, allowing one to use Rust libraries as native extension modules as well as interacting with Python code from Rust binaries. However Maturin also supports creating bindings using Rust's C Foreign Function Interface (CFFI), using the `clib` crate to expose Rust types to consumers of the C application binary interface (ABI). We choose the CFFI, as the types we wish to expose into Rust are relatively simple pointers to Rust iterators and the pointer to an MPI communicator created from Python. This also allows for a simple design with minimal external libraries, with our Python interface to be cleanly separated from Rust source code.

Listing 6.1: Using Rusty Tree from its Python package.

```
from mpi4py import MPI
import numpy as np

from rusty_tree.distributed import DistributedTree
```

```
# Setup communicator
comm = MPI.COMM_WORLD

# Cartesian points at this process
points = np.random.rand(100000, 3)

# Generate a balanced tree
balanced = DistributedTree.from_global_points(points, True, comm)

# Generate an unbalanced tree
unbalanced = DistributedTree.from_global_points(points, False, comm)

# Trees map between points and the keys that they correspond to contain
point = balanced.points[0].head # grab a point
print(balanced[point]) # find the key that it maps

# Trees map between keys and the points they contain
key = balanced.keys[0].head # grab a point
print(balanced[key]) # find the points that this key contains

# Trees implement iterator protocol as well as slicing and indexing
# (without copy of underlying Rust data)
# Slice of 10 keys
key_slice = balanced.keys[:10]

# Slice of 10 points
point_slice = balanced.points[:10]

def foo(key):
    """Some key processor function"""
    pass

# Iterating in this way avoids copying data from Rust to Python
for key in key_slice:
    foo(key)

# Copy only performed when printing in Python
print(point_slice)
print(key_slice)
```

Listing 6.2: Using Rusty Tree as a Rust library.

```
use rand::prelude::*;
use rand::SeedableRng;

use mpi::{
    environment::Universe,
    topology::UserCommunicator,
    traits::*
};

use rusty_tree::{
    distributed::DistributedTree,
};


fn main () {
    // Generate a set of randomly distributed points
    let mut range = StdRng::seed_from_u64(0);
    let between = rand::distributions::Uniform::from(0.0..1.0);
    let mut points: Vec<[PointType; 3]> = Vec::new();
```

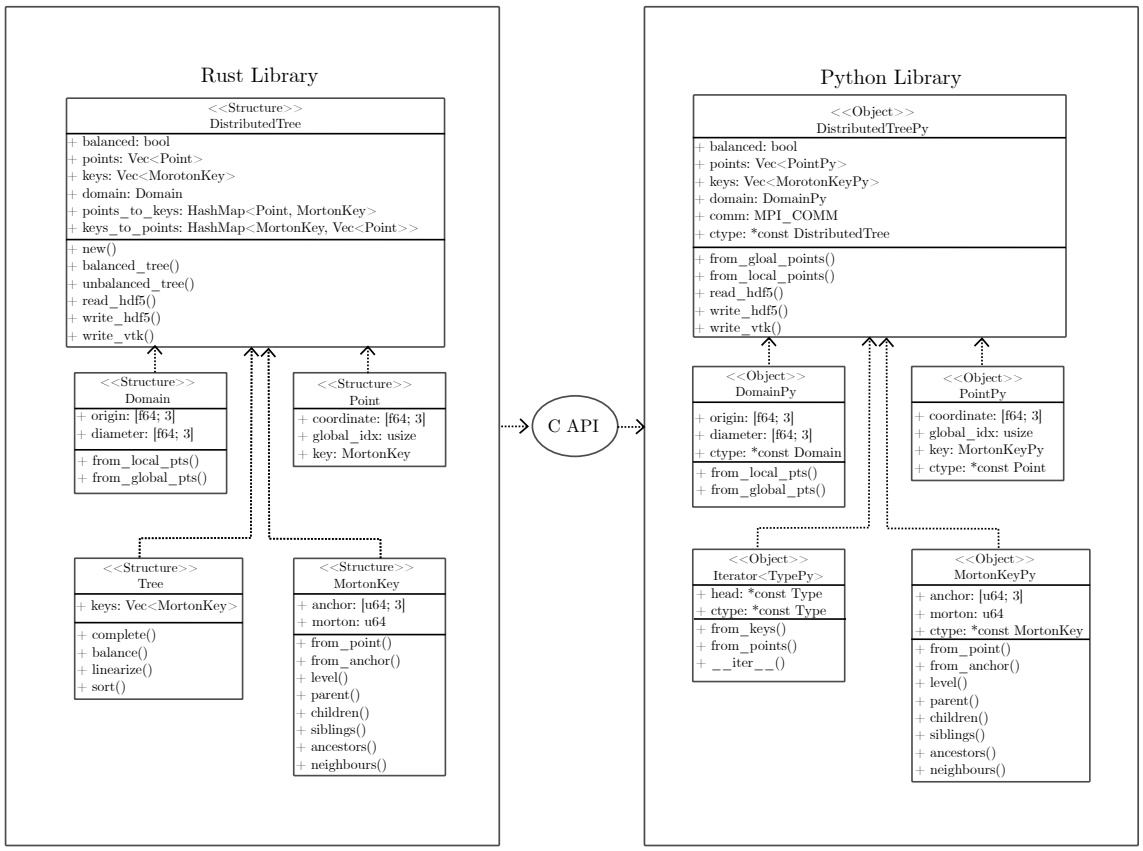


Figure 6.5: We see that the shallow hierarchy inherent in Rusty Tree's design, with a 'Distributed Tree' interface in both Python and Rust which is a wrapper over functionality that acts directly on data. On the Python side we have an additional `Iterator` class, that allows us to wrap Rust iterators into Python iterators.

```
    let npoints = 1000000;

    for _ in 0..npoints {
        points.push([
            between.sample(&mut range),
            between.sample(&mut range),
            between.sample(&mut range),
        ])
    }

    // Setup an MPI environment
    let universe: Universe = mpi::initialize().unwrap();
    let world: UserCommunicator = universe.world();
    let comm = world.duplicate();

    // Unbalanced
    let unbalanced: DistributedTree = DistributedTree::new(&points, false, &comm)

    // Balanced
    let balanced_tree: DistributedTree = DistributedTree::new(&points, true, &comm)

}
```

Using MPI from Rust

MPI, via the rsMPI crate, constitutes the most complex dependency in Rusty Tree. The MPI bindings are built on top of C shim library, as well as automatically generated C/C++ compatible header files using the `cbindgen` crate. The shim library is used to form an equivalence between under-specified identifiers from the MPI standard, that may not be uniform across MPI implementations, and `cbindgen` is used to automatically generate headers for different MPI implementations. This allows rsMPI to expose an interface in Rust which covers most of MPI's core functionality for most MPI implementations, on most operating systems [1]. However, as it is designed to target multiple platforms like all other Rust crates, its build is fragile and requires extra configuration from users who might wish to deploy on unsupported or novel hardware and software targets. Specifically we have found to problems arise on Cray machines, such as ARCHER2, which have custom installations of MPI and do not support the `mpicc` compiler wrapper. However, we note that this is an active area of development, and will be accounted for in a future rsMPI release. As a temporary fix, we currently fork rsMPI with manual configurations to target the HPC systems we wish to run our software on. Consumers of our software will have to follow similar steps if automatic builds do not work, however we note that in the majority of cases this fragility will not be exposed to users, who can continue to who can continue to consume our library using a single line in their project's `Cargo.toml`.

Scaling Experiments

Figure (6.6a) shows the weak scaling performance of Rusty Tree[2] for constructing *balanced octrees*. $5e5$ particles, taken from a random uniform distribution in a unit cube are placed on each processor, up to a total of 16 million particles. The parameter n_{crit} is set to 200. We compare the performance of Rusty Tree with Dendro, a C++ library similarly built on algorithms presented in [70]. Dendro has a limited API in comparison to Rusty tree, with no Python interface, and no ability to query tree nodes for the points they contain and is therefore not usable for developing fast algorithms. However it does act as a showcase of the algorithms presented in [70], allowing for a direct comparison to our own implementation. We observe that both softwares have a similar scaling behaviour for the tested problem sizes, however Rusty Tree is more expensive by a has a fixed constant factor, both in Python and Rust. We observe that this is due to our provision of hashmaps (see fig. (6.5)), that allow users to query tree nodes for the points they contain and vice versa, a functionality not available in Dendro. An experiment confirming this is shown in figure (6.6d).

We observe that Rusty Tree, when called from Python, has a significantly more expensive memory overhead in comparison to the Rust library, displayed in figure (6.6b). Whereas the Rust library, and Dendro, have similar memory profiles. The cause for excess memory overhead in Python is due to the need to assign static lifetimes to all Rust pointers passed to Python, for example for Morton keys and point data, which would be optimised out of a corresponding Rust program at compile time if it was not being used downstream in a program. We observe that

[1] This is an area of active development, and a list of currently supported features can be found here `https://github.com/rsmpi/rsmpi/blob/main/README.md`

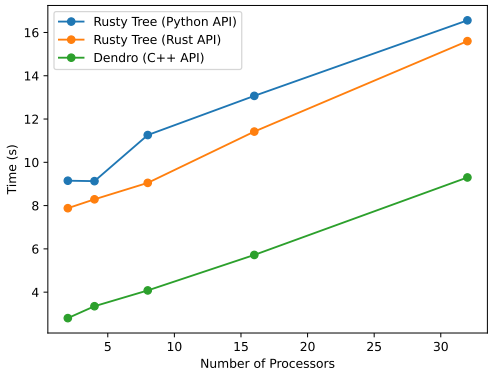
[2] Experiments were taken on an AMD Ryzen Threadripper 3970X 32-Core processor

the runtime cost of using Rusty Tree from Python is moderate, at between 5 and 25 % of total runtime in Rust, for the problem sizes tested.

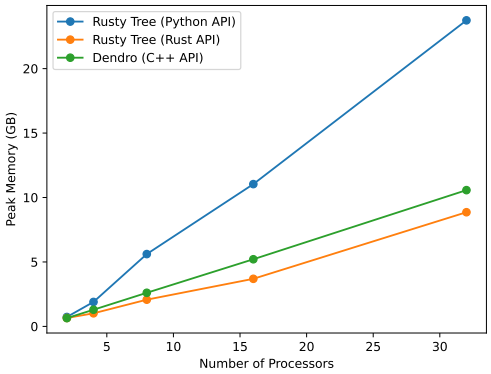
Figure (6.6c) is a plot of the mean difference of the load on each processor with respect to the mean load for a given geometry and problem size, as we scale a problem weakly. The geometries we choose are the surface of a 'wiggly torus', as shown in figure (2.1), and the interior of a unit cube, where we sample $5e5$ points randomly from each domain at each processor. The parameter n_{crit} is again set to 200, and each geometry is tested 5 times for statistics. Comparing the results from both geometries, we observe a qualitatively weak dependence between the problem size and an increasing load imbalance. However, the overlapping nature of the error bars in our experiments, and the fact that an experiment with uniform random data shows similar load imbalance to a significantly less uniform data from the surface of a wiggly torus, make it difficult to conclude a correlation without further investigation. We will need to supplement this investigation with results for larger experiment sizes on large scale HPC systems.

Conclusion

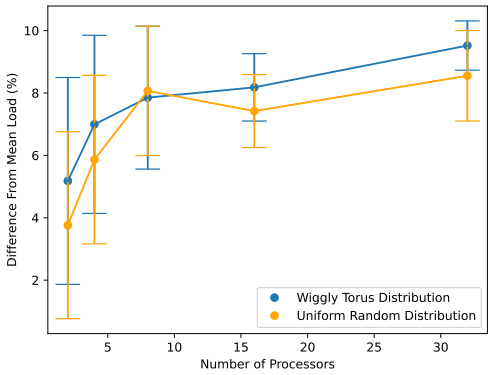
Rusty Tree has high-level Python interfaces, allowing it to be inserted into existing research projects with minimal configuration with a moderate performance hit. It is designed around data and is simple to read, and importantly can be built for a variety of software and hardware environments. It is a demonstration of Rust as a viable alternative to development in Rust as opposed to C++ or Fortran. We believe that this will encourage the widespread adoption of our library, and we hope to publicise this software, as well as our user experience with Rust for high performance scientific computing, in an upcoming paper.



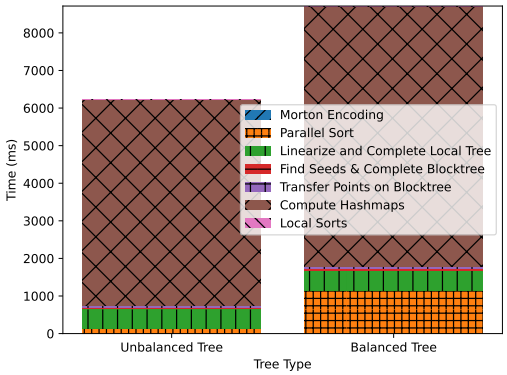
(a) Runtime.



(b) Peak memory



(c) Load balancing



(d) Runtime Breakdown

Figure 6.6: There are $5e5$ points per processor for the weak scaling experiments in (a), with an $n_{\text{crit}} = 200$. Figure (b) shows the peak memory usage of Dendro compared to the Python and Rust interfaces of Rusty Tree. We note that the Python library has a significant overhead. Figure (c) shows the difference in load from the global mean load, as mean over runs, in an experiment that is weakly scaled with the same parameters as in figure (a). Figure (d) shows the breakdown in runtime between the different components of the algorithms for computing balanced and unbalanced trees, experiments are shown for a tree with $2e6$ uniformly randomly distributed points in a unit cube, distributed evenly across 4 processors. We see that the dominant factor in runtime is the creation of the hashmaps.

An $O(N)$ Fast Direct Solver for Strongly Admissable Problems

In this chapter we review a recently developed fast direct solver for the solution of low-medium frequency scattering problems that are described by the Helmholtz equation[1]. Fast direct solvers constitute a fast matrix inversion, in contrast to the fast matrix-vector product of the FMM. Fast direct solvers for strongly admissable oscillatory problems will be a foundational component of our future Rust-based software infrastructure with our key target application being the solution of electromagnetic scattering problems described by the vectorial Maxwell's equations.

In chapter 1 we introduced boundary integral equations (1.2) and their matrix representation (1.3), noting that it could alternatively be represented using hierarchical matrices. As we have seen, these matrix representations (HSS, $\mathcal{H}^2$ etc.) allow us to rapidly *apply* the system matrix (1.3) with linear or near linear time complexity. In combination with modern *iterative* methods, such as GMRES [62], with $O(n_k)$ iterations such that $n_k \ll N$ where N is the number of unknowns in the system, the solution of such problems is brought into practical reach. However, there are situations in which iterative methods fall short. For example for problems which have *multiple* right hand sides we wish to solve for. In such cases, recently developed *fast direct* methods offer a promising alternative. The inversion of a dense linear system by classical direct methods, such LU decomposition, have a complexity of $O(N^3)$. Fast direct methods however take advantage of the low-rank structure implicit in the matrix of (1.2) to find an approximation for the inverse in a similar manner to the FMM. The main trade-off between fast direct methods and the iterative alternative is the relatively greater memory cost in having to pre-compute and store the matrix factorization.

Numerous approaches have been implemented for fast direct solvers for the matrix factorisations described in chapter 2 [48, 2, 3, 38, 54]. Here we introduce the 'Recursive Strong Skeletonisation' (RS-S) algorithm of Minden et. al [54], an $O(N)$ method which incorporates strong admissibility, and thus operates on $\mathcal{H}^2$ matrices, and has good properties for parallelisation. Our main contribution is the extension of this technique to effectively handle acoustic scattering problems described by the Helmholtz equation at low to moderate frequencies where the size of the problem domain is less than a few hundred times the wavelength, building on the exterior Dirichlet case first presented in [71].

[1]This is a part of joint work developed in collaboration with researchers at the Flatiron Institute.

Problem Formulation

We start by deriving the boundary integral equation for the boundary value problem described by the Helmholtz equation and an exterior Dirichlet boundary condition,

$$(\Delta + k^2)u = 0, \text{ in } \mathbb{R} \setminus \Omega \tag{7.1}$$

$$u = f, \text{ on } \Gamma \tag{7.2}$$

$$\lim_{r\to\infty} r\left(\frac{\partial u}{\partial r} - iku\right) = 0 \tag{7.3}$$

The final line above describes a 'radiation condition', which is required to maintain the uniqueness of u. Physically, the solution u corresponds to the spatial distribution of a wave scattered from an object with domain Ω and boundary Γ embedded in $\mathbb{R}^d$ where without loss of generality we will take $d = 3$.

Using the definition of the free space Green's function,

$$\Phi(\mathbf{x}, \mathbf{y}) = \frac{e^{ik|\mathbf{x}-\mathbf{y}|}}{4\pi|\mathbf{x}-\mathbf{y}|} \tag{7.4}$$

And defining,

$$K(\mathbf{x}, \mathbf{y}) = (\mathbf{n}(y) \cdot \nabla_{\mathbf{y}}\Phi(\mathbf{y}, \mathbf{x})) - ik\Phi(\mathbf{x}, \mathbf{y}) \tag{7.5}$$

where $n(\mathbf{y})$ is the outwardly facing unit normal at $y \in \Gamma$, we define the *combined field representation* of u as,

$$u = \int_\Gamma K(\mathbf{x}, \mathbf{y})\sigma(\mathbf{y})da(\mathbf{y}), \ \mathbf{x} \in \mathbb{R}^3 \setminus \Omega \tag{7.6}$$

$$= \int_\Gamma (\mathbf{n}(y) \cdot \nabla_{\mathbf{y}}\Phi(\mathbf{y}, \mathbf{x})) - ik\Phi(\mathbf{x}, \mathbf{y})\sigma(\mathbf{y})da(\mathbf{y}) \tag{7.7}$$

$$= (\mathcal{D} - ik\mathcal{S})\sigma \tag{7.8}$$

where we also define the double layer, $\mathcal{D}$, and single layer, $\mathcal{S}$ potential operators. Representing u in terms of layer potentials gives us a solution of the the PDE (7.1), however the 'surface density' σ is unknown. Armed with a representation we can proceed to form a *boundary integral equation* defined only along Γ in terms of the the unknown σ,

$$(\mathcal{D} - ik\mathcal{S} + \frac{1}{2})\sigma(\mathbf{x}) = f(\mathbf{x}), \ \mathbf{x} \in \Gamma$$

Here we have used the well known *jump relations* which describe a discontinuity in the value of the double layer potential operator as we approach the boundary. Further information about layer potentials, the particular benefits of the combined field representation and the jump relations can be found in reference books such as [18]. For our purposes it's sufficient to recognise that this boundary integral equation motivates our work on fast direct solvers. Writing it in another form,

$$\frac{1}{2}\sigma(\mathbf{x}) + \int_{\Gamma} K(\mathbf{x},\mathbf{y})\sigma(\mathbf{y})da(\mathbf{y}) = f(\mathbf{x}) \tag{7.9}$$

we see that it mirrors the integral equation described in chapter 1, and we proceed to discretise using a suitable method, such as the Galerkin or Nyström methods. Using Nyström we arrive at the following linear system,

$$\frac{1}{2}\sigma_i + \sum_{j=1, j\neq i}^{N} K(\mathbf{x}_i,\mathbf{x}_j)\sigma_j w_{ij} = f(x_i) \tag{7.10}$$

where $\mathbf{x}_i$ and w_{ij} are the quadrature nodes and weights, respectively, and σ_i is the approximation to $\sigma(\mathbf{x}_i)$. Solving this equation for σ, allows one to find the approximation of the general solution of the scattering problem (7.1) in the exterior using (7.6).

An obvious advantage of boundary integral approaches is that they transform an unbounded d dimensional problem into a bounded $d-1$ dimensional problem, however a consequence of this is a dense discrete operator in 7.10. Boundary integral formulations of the form 7.10 also have favourable spectral properties [18], making them significantly more benign mathematical objects than the PDEs they replace.

Recursive Strong Skeletonisation

Consider a domain containing the discretisation points of the boundary Γ, $\mathbf{x}_i \in \Gamma$. We proceed to discretise with an adaptive, 2:1 balanced, octree. The idea of strong skeletonisation is then to use the *interpolative decomposition* (ID) [16], a dense matrix compression algorithm, to globally compress matrices with low-rank structure for the case in which only particular off-diagonal blocks are low-rank. In our problem of interest, (7.10), where the low-rank assumption only applies when two nodes of an octree are strongly admissible with respect to each other, i.e. the far-field interactions via the kernel of the integral equation.

Definition 7.0.1 (Interpolative Decomposition (ID)) *Given a matrix $A_{\mathcal{I}\mathcal{J}} \in \mathbb{C}^{|\mathcal{I}|\times|\mathcal{J}|}$ with rows indexed by $\mathcal{I}$ and columns indexed by $|\mathcal{J}$, an ϵ accurate interpolative decomposition of A is a partitioning of $\mathcal{J}$ into a set of so-called skeleton columns denoted by $\mathcal{S} \subset \mathcal{J}$ and redundant columns $\mathcal{R} = \mathcal{J} \setminus \mathcal{S}$, and the construction of a corresponding interpolation matrix T such that,*

$$\|A_{\mathcal{I}\mathcal{R}} - A_{\mathcal{I}\mathcal{S}}T\| \leq \epsilon \|A_{\mathcal{I}\mathcal{J}}\|$$

Where we take the norms to be defined as the standard spectral norm. The interpretation of the above is that the redundant columns are well approximated by a linear combination of the skeleton columns. We compute the ID using a column-pivoted QR decomposition as in [71], which has a complexity of $O(|\mathcal{I}| \cdot |\mathcal{J}|^2)$.

Now we consider a three by three block matrix, $A \in \mathbb{C}^{N\times N}$, taking a partition of the index set such that $[N] = \mathcal{I} \cup \mathcal{J} \cup \mathcal{K}$ with $[N] = 1, 2, ...N$ such that $A_{\mathcal{I}\mathcal{K}} = 0 = A_{\mathcal{K}\mathcal{I}}$,

$$A = \begin{bmatrix} A_{\mathcal{I}\mathcal{I}} & A_{\mathcal{I}\mathcal{J}} & 0 \\ A_{\mathcal{J}\mathcal{I}} & A_{\mathcal{J}\mathcal{J}} & A_{\mathcal{J}\mathcal{K}} \\ 0 & A_{\mathcal{K}\mathcal{J}} & A_{\mathcal{K}\mathcal{K}} \end{bmatrix}$$

Assuming $A_{\mathcal{I}\mathcal{I}}$ is invertible, and using block Gaussian elimination we can decouple this block from the rest of the matrix as follows,

$$L \cdot A \cdot U = \begin{bmatrix} I & 0 & 0 \\ -A_{\mathcal{J}\mathcal{I}}A_{\mathcal{I}\mathcal{I}}^{-}1 & I & 0 \\ 0 & 0 & I \end{bmatrix} \cdot A \cdot \begin{bmatrix} I - A_{\mathcal{I}\mathcal{I}}^{-}1A_{\mathcal{I}\mathcal{J}} & 0 & \\ 0 & I & 0 \\ 0 & 0 & I \end{bmatrix} = \begin{bmatrix} A_{\mathcal{I}\mathcal{I}} & 0 & 0 \\ 0 & S_{\mathcal{J}\mathcal{J}} & A_{\mathcal{J}\mathcal{K}} \\ 0 & A_{\mathcal{K}\mathcal{J}} & A_{\mathcal{K}\mathcal{K}} \end{bmatrix}$$

where the block $S_{\mathcal{J}\mathcal{J}} = A_{\mathcal{J}\mathcal{J}} - A_{\mathcal{J}\mathcal{I}}A_{\mathcal{I}\mathcal{I}}^{-}1A_{\mathcal{I}\mathcal{J}}$ is the only non-zero block that has been modified, with the second term in $S_{\mathcal{J}\mathcal{J}}$ is known as the Schur complement update.

We now attempt to apply this decoupling to the matrix, A, that arises from our boundary integral formulation. Taking $\mathcal{B}$ as the set of indices of points x_i contained in box B in the octree, $\mathcal{N}$ as the indices of the points in B's near field and $\mathcal{F}$ as the points contained in its far field, as defined by the strong admissibility condition, upon appropriate permutation by a matrix P we arrive at the following block structure for A.

$$P^T AP = \begin{bmatrix} A_{\mathcal{B}\mathcal{B}} & A_{\mathcal{B}\mathcal{N}} & A_{\mathcal{B}\mathcal{F}} \\ A_{\mathcal{N}\mathcal{B}} & A_{\mathcal{N}\mathcal{N}} & A_{\mathcal{N}\mathcal{F}} \\ A_{\mathcal{F}\mathcal{B}} & A_{\mathcal{F}\mathcal{N}} & A_{\mathcal{F}\mathcal{F}} \end{bmatrix} \tag{7.11}$$

We want to numerically compress the far-field interactions, i.e. the blocks corresponding to $A_{\mathcal{B}\mathcal{F}}$ and $A_{\mathcal{F}\mathcal{B}}$ using the ID, as they are considered to be numerically low-rank. To do so, we decouple into a set of redundant points, denoted by indices $\mathcal{R}$ and skeleton points, denoted by indices $\mathcal{S}$, such that (up to a permutation) we have,

$$\begin{bmatrix} A_{\mathcal{F}\mathcal{B}} \\ A_{\mathcal{B}\mathcal{F}}^T \end{bmatrix} = \begin{bmatrix} A_{\mathcal{F}\mathcal{R}} & A_{\mathcal{F}\mathcal{S}} \\ A_{\mathcal{R}\mathcal{F}}^T & A_{\mathcal{R}\mathcal{S}}^T \end{bmatrix} \approx \begin{bmatrix} A_{\mathcal{F}\mathcal{S}} \\ A_{\mathcal{S}\mathcal{F}} \end{bmatrix}^T \cdot \begin{bmatrix} T & I \end{bmatrix}$$

where we've applied the ID as an approximation. We've applied the same interpolation matrix, T, for both blocks making the assumption that the kernel is symmetric. The authors in [71] note that this in practice can also be done for non-symmetric kernels, at the cost of a small increase in the number of skeleton points.

Returning to (7.11), and further splitting $\mathcal{B} = \mathcal{R} \cup \mathcal{S}$ and applying the previous approximation for the far-field blocks, we get

$$P^T AP = \begin{bmatrix} A_{\mathcal{R}\mathcal{R}} & A_{\mathcal{R}\mathcal{S}} & A_{\mathcal{R}\mathcal{N}} & A_{\mathcal{R}\mathcal{F}} \\ A_{\mathcal{S}\mathcal{R}} & A_{\mathcal{S}\mathcal{S}} & A_{\mathcal{S}\mathcal{N}} & A_{\mathcal{S}\mathcal{F}} \\ A_{\mathcal{N}\mathcal{R}} & A_{\mathcal{N}\mathcal{S}} & A_{\mathcal{N}\mathcal{N}} & A_{\mathcal{N}\mathcal{F}} \\ A_{\mathcal{F}\mathcal{R}} & A_{\mathcal{F}\mathcal{S}} & A_{\mathcal{F}\mathcal{N}} & A_{\mathcal{F}\mathcal{F}} \end{bmatrix} = \begin{bmatrix} A_{\mathcal{R}\mathcal{R}} & A_{\mathcal{R}\mathcal{S}} & A_{\mathcal{R}\mathcal{N}} & T^T A_{\mathcal{S}\mathcal{F}} \\ A_{\mathcal{S}\mathcal{R}} & A_{\mathcal{S}\mathcal{S}} & A_{\mathcal{S}\mathcal{N}} & A_{\mathcal{S}\mathcal{F}} \\ A_{\mathcal{N}\mathcal{R}} & A_{\mathcal{N}\mathcal{S}} & A_{\mathcal{N}\mathcal{N}} & A_{\mathcal{N}\mathcal{F}} \\ A_{\mathcal{F}\mathcal{S}}T & A_{\mathcal{F}\mathcal{S}} & A_{\mathcal{F}\mathcal{N}} & A_{\mathcal{F}\mathcal{F}} \end{bmatrix}$$

As the redundant rows and columns of interactions between the points in $\mathcal{R}$ and $\mathcal{F}$ are well approximated by the interactions between $\mathcal{S}$ and $\mathcal{F}$ we can decouple

these points from the rest of the problem as follows. Let E and F be 'elimination' matrices defined on a partition $[N] = \mathcal{R} \cup \mathcal{S} \cup \mathcal{N} \cup \mathcal{F}$ as,

$$E = \begin{bmatrix} I & -T^T & & \\ & I & & \\ & & I & \\ & & & I \end{bmatrix} \text{ and } F = \begin{bmatrix} I & & & \\ -T & I & & \\ & & I & \\ & & & I \end{bmatrix}$$

Then,

$$EP^T APF = \begin{bmatrix} X_{\mathcal{RR}} & X_{\mathcal{RS}} & X_{\mathcal{RN}} & 0 \\ X_{\mathcal{SR}} & A_{\mathcal{SS}} & A_{\mathcal{SN}} & A_{\mathcal{SF}} \\ X_{\mathcal{NR}} & A_{\mathcal{NS}} & A_{\mathcal{NN}} & A_{\mathcal{NF}} \\ 0 & A_{\mathcal{FS}} & A_{\mathcal{FN}} & A_{\mathcal{FF}} \end{bmatrix}$$

where $X_{\mathcal{IJ}}$ indicates blocks which have been updated in comparison to the original matrix. Assuming that $X_{\mathcal{RR}}$ is invertible, we can use it as a pivot block to decouple the redundant degrees of freedon. Let L and U be defined as,

$$L = \begin{bmatrix} I & & & \\ -X_{\mathcal{SR}}X_{\mathcal{RR}}^{-1} & I & & \\ -X_{\mathcal{NR}}X_{\mathcal{RR}}^{-1} & & I & \\ & & & I \end{bmatrix} \text{ and } U = \begin{bmatrix} I & -X_{\mathcal{RR}}^{-1}X_{\mathcal{RS}} & -X_{\mathcal{RR}}^{-1}X_{\mathcal{RN}} & \\ & I & & \\ & & I & \\ & & & I \end{bmatrix}$$

Then

$$\begin{aligned} Z(A;B) &= LEP^T APFU \\ &= \begin{bmatrix} X_{\mathcal{RR}} & 0 & 0 & 0 \\ 0 & X_{\mathcal{SS}} & X_{\mathcal{SN}} & A_{\mathcal{SF}} \\ 0 & X_{\mathcal{NS}} & X_{\mathcal{NN}} & A_{\mathcal{NF}} \\ 0 & A_{\mathcal{FS}} & A_{\mathcal{FN}} & A_{\mathcal{FF}} \end{bmatrix} \end{aligned}$$

This matrix is of the previous decoupled form for a generic matrix. This decoupling is referred to as the *strong skeletonisation* of A with respect to B, with the resulting matrix denoted with $Z(A;B)$. Therefore we can write A in terms of a block LU-style factorisation,

$$A = (PE^{-1}L^{-1})Z(A;B)(U^{-1}F^{-1}P^T) = VZ(A;B)W$$

where V and W are called the skeletonisation operators, which can be stored and used in factored form. Moreover L U E and F are block triangular, with identities in their diagonal and thus are trivial to invert. Giving us a compact representation for skeletonisation,

$$Z(A;B) = V^{-1}AW^{-1}$$

The recursive strong skeletonisation (RS-S) proceeds by sequentially applying strong skeletonisation to each box in an octree that describes the points on arising

from the discretisation of the boundary integral equation. The boxes in the tree are traversed in an upward pass, similar the FMM. After each application of the strong skeletonisation only the skeleton points S associated with each box are retained, referred to as 'active degrees of freedom'. For boxes at coarser levels the active degrees of freedom are constructed from the active degrees of freedom of its children, at which point strong skeletonisation is applied again. The process continues until there are no active degrees of freedom for any box in a given box's far field - which is true by definition at level 1 of the tree. Denoting the skeletonisation operators of box i as V_i and W_i, and assuming the algorithm terminates at box B_M, we denote the permutation matrix P_t which orders the remaining active degrees of freedom in $B'_M s$ domain, denoted with $\mathcal{B}_t$. This results in a final RS-S factorisation of A as,

$$A \approx (V_1 V_2 \dots V_M) P_t D P_t^T (W_M W_{M-1} \dots W_1)$$

where D is a block diagonal matrix given by,

$$D = \begin{bmatrix} X_{\mathcal{R}_1\mathcal{R}_1} & & & \\ & \ddots & & \\ & & X_{\mathcal{R}_M\mathcal{R}_M} & \\ & & & A_{\mathcal{B}_t\mathcal{B}_t} \end{bmatrix}$$

Using this decomposition we can rapidly compute A^{-1}, and hence our 'fast direct solve' with,

$$A^{-1} \approx (W_1^{-1} \dots W_M^{-1}) P_t D^{-1} P_t^T (V_M^{-1} \dots V_1^{-1})$$

The Proxy Trick

The complexity of the fast direct solver described by the RS-S technique is determined by the cost of computing the ID for the far-field interactions of a given box in the octree, B. This will in general contain $O(N)$ points. Therefore computing the ID with respect to far field interactions as we've done above will scale as $O(N^2)$ for each box. The authors of [54] employ the so called 'proxy trick' to reduce this complexity, and recover an $O(N)$ scaling for their fast direct solver. The full complexity analysis of this algorithm is provided in appendix A. Our contribution in this work is the explicit formulation of the proxy trick for a variety of integral equation kernels that appear in scattering problems, with specific application to acoustic scattering problems.

Consider as a setting the boundary integral equation formulation of a Helmholtz scattering problem, the main idea of the proxy trick rests on the principle of representing the far-field particles of a given box B, which may contain $O(N)$ particles, with a set of 'proxy points' contained on a proxy surface that encloses B. This surface is often chosen to be a sphere. By choosing $O(1)$ proxy points, without getting into the details yet of how exactly they are sampled, we are able to obtain the linear complexity we desire.

For a given box B, a proxy surface D and its boundary γ are chosen such that $B \subset D$. The far-field points of B, $\mathcal{F}$ is partitioned such that $\mathcal{F} = \mathcal{Q} \cup \mathcal{P}$, where $\mathcal{Q}$ contains $O(1)$ points. Γ is the boundary of the entire scatterer, and $\tau = \Gamma \cap B$ is

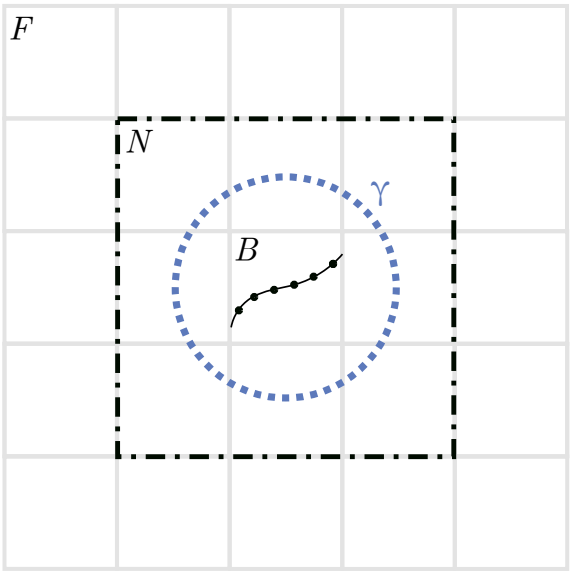


Figure 7.1: Considering the outgoing problem due to charge contained on $\Gamma \cap B$ evaluated in the far-field of B in $\mathbb{R}^2$.

the portion of the scatterer boundary contained in B. The situation is sketched in figure (7.1) in $\mathbb{R}^2$.

We can choose to represent our solution due to the charge in B in $\mathcal{F}$ however we wish. However, our choice will lead to different matrices that we must compress. Generally, we'll end up with a solution matrix of the form $A_{\mathcal{F}B}$ that maps between the charge contained B, ψ_B, and potential, $v_{\mathcal{F}}$, at points in its far-field that can be split up as,

$$v_{\mathcal{F}} = A_{\mathcal{F}B}\psi_B \tag{7.12}$$
$$= B_{\mathcal{F}\gamma}C_{\gamma B}\psi_B \tag{7.13}$$

the subscripts indicate the domains these operators map between, with respect to figure (7.1). We desire a split like this as the far field interaction of B can be expressed as an object involving $C_{\gamma B}$ only.

To see how such a split can be useful consider the fact that $A_{\mathcal{F}B}$ can be written as,

$$A_{\mathcal{F}B} = \begin{bmatrix} A_{\mathcal{Q}B} \\ A_{\mathcal{P}B} \end{bmatrix} = \begin{bmatrix} I & 0 \\ 0 & B_{\mathcal{P}\gamma} \end{bmatrix} \begin{bmatrix} A_{\mathcal{Q}B} \\ C_{\gamma B} \end{bmatrix} \tag{7.14}$$

Our RS-S algorithm relies on a compression of this matrix, however as we noted a direct compression of $A_{\mathcal{F}B}$ is too expensive. However if we can find a decomposition like above, we can apply an interpolative decomposition to the right column in (7.14) which has dimensions $O(1) \times O(n_\gamma)$ by construction where n_γ is the number of proxy points. To prove that this allows us to reconstruct the full matrix after compression. Consider an ID that gives us,

$$\begin{bmatrix} A_{\mathcal{Q}B} \\ C_{\gamma B} \end{bmatrix} = \begin{bmatrix} A_{\mathcal{Q}S} \\ C_{\gamma S} \end{bmatrix} \begin{bmatrix} T_{SR} & 1 \end{bmatrix} \tag{7.15}$$

Where S and R are the skeleton and redundant points respectively. Placing these back into our expression (7.14),

$$A_{\mathcal{F}B} = \begin{bmatrix} I & 0 \\ 0 & B_{\mathcal{P}\gamma} \end{bmatrix} \begin{bmatrix} A_{\mathcal{Q}S} \\ C_{\gamma S} \end{bmatrix} \begin{bmatrix} T_{SR} & 1 \end{bmatrix} \tag{7.16}$$

$$= \begin{bmatrix} A_{\mathcal{Q}S} \\ B_{\mathcal{P}\gamma} C_{\gamma S} \end{bmatrix} \begin{bmatrix} T_{SR} & 1 \end{bmatrix} \tag{7.17}$$

$$= A_{\mathcal{F}S} \begin{bmatrix} T_{SR} & 1 \end{bmatrix} \tag{7.18}$$

Therefore, we see that we can get away with a cheap ID to reconstruct the far-field operator, involving the proxy points rather than the full far field of B. We are thus left with the task of formulating the proxy trick to compress the correct components of the boundary integral kernel.

Below we consider how to apply the proxy trick for different representations of solutions to the Helmholtz equation of the far-field potential due to the charges and dipoles contained in B, known as 'outgoing skeletonisations', as well the inverse operation in which the proxy trick is applied to calculate the potentials within B due to charges in its far-field, known as 'incoming skeletonisations'. We consider the Helmholtz equation here as our prototype as the results can be extended to the time-harmonic Maxwell equations.

Hypersingular - $\mathcal{T}$

Outgoing Skeletonisation

A double-layer potential, due to some unknown density ψ, supported on τ,

$$v(x) = \int_{\Gamma \cap B} \frac{\partial \Phi(x,y)}{\partial n(y)} \psi(y) ds(y) := \mathcal{D}\psi, \;\; x \in \mathbb{R}^m \setminus \tau \tag{7.19}$$

solves the Helmholtz equation everywhere it's valid. Here, $\Phi(x,y)$ is the fundamental solution of the Helmholtz equation. However, its normal derivative evaluated at the target points, known as the hypersingular operator, which we'll need for deriving boundary integral equations for Maxwell problems, does not,

$$\frac{\partial v}{\partial n(x)} = \frac{\partial}{\partial n(x)} \int_{\Gamma \cap B} \frac{\partial \Phi(x,y)}{\partial n(y)} \psi(y) ds(y) := \mathcal{T}\psi, \;\; x \in \Gamma \cap \mathcal{F} \tag{7.20}$$

it's only valid at far-field points, $\Gamma \cap \mathcal{F}$. However, we can separate out the normal part of the derivative,

$$\frac{\partial v}{\partial n(x)} = n(x) \cdot \nabla_x \int_{\Gamma \cap B} \frac{\partial \Phi(x,y)}{\partial n(y)} \psi(y) ds(y) := n \cdot w \tag{7.21}$$

The function

$$w(x) = \nabla_x \int_{\Gamma \cap B} \frac{\partial \Phi(x,y)}{\partial n(y)} \psi(y) ds(y) := \nabla_x \mathcal{D}\psi \tag{7.22}$$

Does satisfy Helmholtz, everywhere, and we'll exploit this fact in a moment. As an aside, we can see that this is true by considering a double layer potential v that is smooth enough to admit,

$$(\Delta + k^2)w = (\Delta + k^2)\nabla_x v = \nabla_x(\Delta + k^2)v = 0 \tag{7.23}$$

where the last equality follows as v satisfies the Helmholtz equation. Therefore w is a solution of the Helmholtz equation. Note that w has three components.

In order to find $C_{\gamma B}$ for this representation, we need to set up an 'associated boundary value problem' for each component of w. The choice of boundary value problem we choose is free, as we only rely on the existence of its solution.

Consider an associated boundary value problem for just a single component of $\tilde{w}$ that satisfies,

$$(\Delta + k^2)\tilde{w} = 0, \;\; x \in \mathbb{R}^m \setminus D \tag{7.24}$$

$$\tilde{w} = w_1(x) \tag{7.25}$$

$$\text{A radiation condition at } \infty \tag{7.26}$$

We choose a combined field representation,

$$\tilde{w} = (\mathcal{D} - ik\mathcal{S})_{\mathcal{F}\gamma}\mu \tag{7.27}$$

where μ is some unknown density supported on the proxy surface γ. Forming the boundary integral equation, and plugging back into the representation for $\tilde{w}$,

$$\tilde{w} = (\mathcal{D} - ik\mathcal{S})_{\mathcal{F}\gamma}(\frac{1}{2}\mathcal{I} + \mathcal{D} - ik\mathcal{S})^{-1}_{\gamma\gamma} w_1 \tag{7.28}$$

$$= (\mathcal{D} - ik\mathcal{S})_{\mathcal{F}\gamma}(\frac{1}{2}\mathcal{I} + \mathcal{D} - ik\mathcal{S})^{-1}_{\gamma\gamma}\nabla_1\mathcal{D}_{\gamma B}\psi_\gamma \tag{7.29}$$

$$\equiv B_{\mathcal{F}\gamma}C_{\gamma B}\psi_\gamma \tag{7.30}$$

where we identify,

$$C_{\gamma B} = \nabla_1\mathcal{D}_{\gamma B} \tag{7.31}$$

This is the matrix we will attempt to compress. Similar analysis follows for the other two components of $w(x)$. Meaning that we end up having to compress $[\nabla_1\mathcal{D}_{\gamma B}, \nabla_2\mathcal{D}_{\gamma B}, \nabla_3\mathcal{D}_{\gamma B}]$ for the outgoing problem. We note that $B_{\mathcal{F}\gamma}$ is never explicitly formed, we just require its existence. When we calculate an approximation of $A_{\mathcal{F}B}$ using (7.16), we only need to know the ID of the $C_{\gamma B}$.

Incoming Skeletonisation

For the incoming skeletonization, were again we're considering the same representation with a hypersingular operator, we observe that we're just looking for,

$$\left[\frac{\partial v}{\partial n(x)}\right]^T_{\mathcal{F}B} \tag{7.32}$$

with the formation of an associated boundary integral equation taking place in much the same way as for the outgoing problem. However, the hypersingular operator is self-adjoint, therefore it leads to the same expressions for $C_{\gamma B}$.

Derivative of the Single Layer - $\mathcal{K}'$

Outgoing Skeletonisation

If we choose to represent our potential with a single-layer potential,

$$u(x) = \int_{\Gamma \cap B} \Phi(x,y)\phi(y)ds(y) := \mathcal{S}\phi, \;\; x \in \mathbb{R}^m \setminus \tau \tag{7.33}$$

and seek a boundary integral equation in terms of its normal derivative at the targets,

$$w(x) = \int_{\Gamma \cap B} \frac{\partial \Phi(x,y)}{\partial n(x)}\phi(y)ds(y) := \mathcal{K}'\phi, \;\; x \in \Gamma \cap \mathcal{F} \tag{7.34}$$

We observe the same problem as in the $\mathcal{T}$ case, where this expression is not a general solution of Helmholtz. We can similarly separate out the normal component and write,

$$\tilde{w}(x) := \int_{\Gamma \cap B} \nabla_x \Phi(x,y)\phi(y)ds(y), \;\; x \in \mathbb{R}^m \setminus \tau \tag{7.35}$$

Using the previous analysis for $\mathcal{T}$, we immediately recognise that the components we must compress are $C_{\gamma B} = \nabla_1 \mathcal{S}_{\gamma B}$, giving us $[\nabla_1 \mathcal{S}_{\gamma B}, \nabla_2 \mathcal{S}_{\gamma B}, \nabla_3 \mathcal{S}_{\gamma B}]$ to compress in total for the outgoing problem.

Incoming Skeletonisation

Noticing that,

$$\left[\frac{\partial u}{\partial n(x)}\right]^T_{\mathcal{F}B} = \int_{\Gamma \cap B} \frac{\partial \Phi(x,y)}{\partial n(y)}\phi(y)ds(y) = \mathcal{D}_{\gamma B}\phi \tag{7.36}$$

already satisfies Helmholtz without any further work, we can simply use it as our Dirichlet data in the associated boundary value problem. The matrix to compress being $C_{\gamma B} = \mathcal{D}_{\gamma B}$.

Single Layer - $\mathcal{S}$

Outgoing Skeletonisation

We now choose to represent our scattered solution with a single-layer operator,

$$u(x) = \int_{\Gamma \cap B} \Phi(x,y)\phi(y)ds(y) := \mathcal{S}\phi, \;\; x \in \mathbb{R}^m \setminus \tau \tag{7.37}$$

This satisfies Helmholtz everywhere. We can now set up an associated exterior boundary value problem as before, and use our single-layer potential as Dirichlet boundary data.

$$(\Delta + k^2)w = 0, \;\; x \in \mathbb{R}^m \setminus D \tag{7.38}$$
$$w = \mathcal{S}\phi, \;\; \text{on } \gamma \tag{7.39}$$
$$\text{Radiation condition at } \infty \tag{7.40}$$

where ϕ is some unknown density supported on τ. As before, we can form a boundary integral equation for this associated problem, and solve, recognising that the matrix to compress $C_{\gamma B} = \mathcal{S}_{\gamma B}$

Incoming Skeletonisation

The single-layer operator is self-adjoint, leading to the same operator to compress. Consider the associated interior boundary value problem,

$$(\Delta + k^2)w = 0 \text{ in } D \tag{7.41}$$
$$w = \mathcal{S}\phi \text{ on } \gamma \tag{7.42}$$

The solution of an interior Helmholtz scattering problem may not be unique, but this doesn't matter for our purposes as our proxy compression formulation doesn't require uniqueness, only existence. Let's seek a solution in the form of a combined-layer potential,

$$w(x) = (\mathcal{D}_\gamma - ik\mathcal{S}_\gamma)[\phi](x) \tag{7.43}$$

where the subscripts make it clear that the density is supported on γ. Forming the boundary integral equation,

$$(-\frac{\mathcal{I}}{2} + \mathcal{D}_\gamma - ik\mathcal{S}_\gamma)[\phi](x) = \mathcal{S}_{\mathcal{F}\gamma}[\phi](x) \tag{7.44}$$

Solving with the representation gives,

$$w = \mathcal{S}_{B\gamma}(-\frac{\mathcal{I}}{2} + \mathcal{D}_\gamma - ik\mathcal{S}_\gamma)^{-1}\mathcal{S}_{\gamma\mathcal{F}}[\phi](x) = C_{B\gamma}B_{\gamma\mathcal{F}} \tag{7.45}$$

where we recognize the matrix to compress as $C_{B\gamma} = \mathcal{S}_{B\gamma}$ in our proxy framework.

Double Layer - $\mathcal{D}$

Outgoing Skeletonisation

A double-layer potential, due to some unknown density ψ, supported on τ,

$$v(x) = \int_{\Gamma \cap B} \frac{\partial \Phi(x,y)}{\partial n(y)} \psi(y) ds(y) := \mathcal{D}\psi, \;\; x \in \mathbb{R}^m \setminus \tau \tag{7.46}$$

solves the Helmholtz equation everywhere it's valid. Therefore it can be used as Dirichlet data for the associated boundary value problem for the outgoing skeletonization. Applying similar analysis to above, we identify the kernel to compress as $C_{\gamma B} = \mathcal{D}_{\gamma B}$.

Incoming Skeletonisation

We notice that the transpose of the double layer operator is,

$$[u]^{T}_{\mathcal{F}B}(x) = \int_{\Gamma \cap B} \frac{\partial \Phi(x,y)}{\partial n(x)} \phi(y) ds(y) = \mathcal{K}'_{\gamma B}\phi \tag{7.47}$$

This in general does not satisfy Helmholtz everywhere, we again separate out the normal component, and as before, recognise that the components to compress are $[\nabla_1 \mathcal{S}_{\gamma B}, \nabla_2 \mathcal{S}_{\gamma B}, \nabla_2 \mathcal{S}_{\gamma B}]$.

Acoustic Sound Hard Scattering

Let's now apply our fast direct solver framework, with proxy compression to some example problems. We begin with acoustic sound-hard scattering, which is a didactic example. Consider a scattered field u^s, that scatters off an object Ω and satisfies the Helmholtz equation in the exterior,

$$(\Delta + k^2)u^s = 0, \quad \text{in } \mathbb{R}^3 \setminus \Omega \tag{7.48}$$

The 'sound hard' boundary condition on the surfae Γ is

$$\frac{\partial u^s}{\partial n} = \frac{\partial u^i}{\partial n}, \quad \text{in } \Gamma \tag{7.49}$$

where u^i is the incident wave. Using the analysis in [12], we write down a 'regularised' representation formula for our solution. This regularisation can be shown to have better spectral properties.

$$u^s = (\mathcal{K}_k \circ \mathcal{S}_K - i\eta \mathcal{S}_k) \tag{7.50}$$

where k and K are complex wave numbers, that may not be the same. We can take the trace of this representation, and its normal derivative at the targets, and find a boundary integral equation for the exterior problem,

$$(\frac{i\eta}{2}\mathcal{I} - i\eta \mathcal{K}'_k + \mathcal{T}_k \circ \mathcal{S}_K)\mu = g \tag{7.51}$$

using the Calderón identity [12],

$$\mathcal{T}_k \circ \mathcal{S}_k = -\frac{1}{4}\mathcal{I} + (K'_k)^2 \tag{7.52}$$

which is true for any k, we arrive at a boundary integral equation,

$$\left(i\eta(\frac{1}{2}\mathcal{I} - \mathcal{K}'_k) - \frac{1}{4}I + (K'_k)^2 \right)\mu = g \tag{7.53}$$

by defining $\theta := \mathcal{K}'_k \mu$, we can write the boundary integral equation as a system,

$$\begin{pmatrix} (\frac{i\eta}{2} - \frac{1}{4})\mathcal{I} - i\eta\mathcal{K}_k' & \mathcal{K}_k' \\ \mathcal{K}_k' & -\mathcal{I} \end{pmatrix} \begin{pmatrix} \mu \\ \theta \end{pmatrix} = \begin{pmatrix} g \\ 0 \end{pmatrix} \tag{7.54}$$

We can then place this system into our fast direct solver framework. Despite not knowing how to compress the system matrix altogether, we do know how to compress each block, as they each correspond to displacements of $\mathcal{K}_k'$.

Consider writing out our block matrix as,

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} \begin{pmatrix} \mu \\ \theta \end{pmatrix} = \begin{pmatrix} g \\ 0 \end{pmatrix} \tag{7.55}$$

and re-writing as,

$$\begin{pmatrix} \begin{pmatrix} A_{11} & B_{11} \\ C_{11} & D_{11} \end{pmatrix} & . & . \\ . & . & \\ . & & \begin{pmatrix} A_{NN} & B_{NN} \\ C_{NN} & D_{NN} \end{pmatrix} \end{pmatrix} \begin{pmatrix} \mu_1 \\ \theta_1 \\ . \\ . \\ \mu_n \\ \theta_n \end{pmatrix} = \begin{pmatrix} g_1 \\ 0 \\ . \\ . \\ g_N \\ 0 \end{pmatrix} \tag{7.56}$$

This system matrix remains numerically low-rank, and therefore can fit into our RS-S framework. Indeed, we now have to compress a system of matrix operators in order to capture its row space via the proxy trick.

Preliminary Numerical Experiments

The paper on which we base our work [71] builds on the original RS-S algorithm [54] by:

1. Implementing generalized Gaussian quadratures for computing near-field interactions [11].
2. Determining a properly sampled discretisation of the proxy surface which depends on the box size in the octree, so as to sufficiently sample oscillatory kernels without oversampling.
3. Handling the construction of the far-field partition into $\mathcal{F} = \mathcal{Q} \cup \mathcal{P}$

We defer to to the original paper for a discussion of these aspects [71]. For our simulations we use these algorithms and quadrature rules as a black box in as implemented by the FMM3D-BIE software [59]. To demonstrate the accuracy of the solver in light of our proxy trick formulation, we suppose that the sound hard condition is generated from a set of 50 test points placed at random locations on the surface of a test geometry, for which we choose the 'wiggly torus' displayed in figure (2.1). The results of our solve are then compared to the potential expected from using a combined field representation at 50 random test locations in the exterior of the geometry.

$$\text{error} = \frac{\sqrt{\sum_{j=1}^{50} |\tilde{u}(t_j) - u(t_j)|^2}}{\sqrt{\int_\Gamma |\sigma(x)|^2 da(x)}}$$

where $\tilde{u}$ is our approximated potential, u is the exact potential calculated from our test points, t_j, and σ is the calculated charge density from the boundary integral equation. The torus is taken to be in a bounding box, each dimension of which is fixed at 1λ where λ is the wavelength of the incident wave, and we fix the error of the ID at $\epsilon = 5 \times 10^{-7}$, and the wavenumber $k = 0.97$. Experiments are taken on a single-node of the Rusty Cluster at the Flatiron Institute, which consists of 240 28-core Intel Broadwell nodes with 512GB of RAM.

p	N_{patch}	N	t_f (s)	t_s (s)	t_q (s)	m_f (GB)	error
4	50	750	0.8	0.03	0.8	0.04	4.7×10^{-3}
4	200	3000	6.6	0.08	1.4	0.6	3.3×10^{-4}
4	450	6750	84.9	0.2	2.5	2.9	5.2×10^{-5}
4	800	12000	324.9	0.6	4.2	7.5	1.6×10^{-5}
4	1250	18750	1138.8	1.1	6.2	15.2	5.8×10^{-6}
4	1800	27000	1839.7	1.7	8.8	25.2	2.4×10^{-6}
4	2450	36750	3322.6	2.4	11.8	41.1	1.2×10^{-6}

Table 7.1: We measure the L^2 error with respect to the density (error), time for solution (t_s), factorisation (t_f), quadrature computation (t_q) and memory required for factorisation (m_f) as a function of quadrature expansion order (p) and number of patches used to discretise the geometry (N_{patch}).

From table (7.1), we observe linear scaling for t_f. However we observe sub-linear scaling for for m_f and t_s, the authors of [71] suggest that this is due to the fixed wavenumber for increasing N. Meaning that the additional degrees of freedom introduced beyond a certain sampling in points per wavelength are more easily compressed. In order to make further observations about the convergence rate of our method more data is required, for higher order quadratures. We note that amount of memory used, even for the modest problem sizes tested, can be quite considerable. With our experiment where the number of quadrature points $N = 36750$ consuming over 40GB of memory. This demonstrates the main trade-off of the increased memory usage of fast direct solvers in comparisons to iterative solvers.

Exposing Parallelism

The RS-S algorithm exposes natural parallelism. Consider figure (7.2) which shows a four-colouring for a non-adaptive octree in $\mathbb{R}^2$ for simplicity, each colour can be independently skeletonised as skeletonisation only updates matrix entries corresponding to a box's near field. The same logic can be applied to the adaptive case, as well as the $\mathbb{R}^3$ case, albeit with more colours. As of writing there are no parallel implementations of the RS-S algorithms available as open-source packages.

Conclusion

We've described a scalar fast direct solver for solving acoustic scattering problems. We've demonstrated the convergence and scaling of our method with some preliminary numerical results, however further numerical testing is required to study the solvers properties. Specifically to understand its convergence behaviour, and scaling behaviour with higher order quadratures. We have only done cursory work on investigating the effect of quadrature choice, as well as the method chosen for sampling

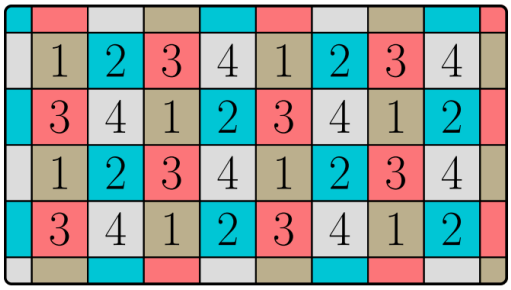


Figure 7.2: A four-colouring of a non-adaptive quadtree in $\mathbb{R}^2$, where the colouring is chosen such that no box shares a colour with its neighbours. All boxes of a given colour can then be skeletonised independently. This figure is adapted from [54].

the proxy surface, using the default generalized Gaussian quadratures and sampling techniques described in [71]. This is another potential vector of research. The extension of this solver, and proxy formulation, to the full vectorial case which would allow us to tackle electromagnetic scattering problems as described by Maxwell's equations is an ongoing area of investigation. The implementation of this in our proposed Rust framework requires the completion of its various sub-components, and is the long-term goal of this PhD project.

Conclusion

In this subsidiary thesis we've presented progress on the development of a new software infrastructure for fast algorithms. We've documented recent outputs towards this goal including foundational software as well as a algorithmic techniques. The main outputs being an investigation into programming languages and environments most suitable for scientific computing, a parallel load balanced octree library designed for high-performance, as well as a proxy-compression based fast direct solver for acoustic Helmholtz scattering problems.

The immediate next steps of this project will be to publish our recent software results on octrees in an appropriate scientific journal. We also intend to continue the developments on fast direct solvers for oscillatory problems, and extend the RS-S framework to electromagnetic scattering problems described by Maxwell's equations. This would constitute a first fast direct solver for electromagnetic scattering problems. In the medium term, we plan to complete the implementation of the parallel FMM software, and the apply this to large scale boundary integral equation problems. This would entail the completion of the majority of our fast solver infrastructure. The final stage of this project will be the extension of our infrastructure to a parallel fast direct solver, and the simulation of large scale electromagnetic scattering problems.

Appendix

Complexity of RS-S Factorisation

This section is adapted from section 3.3.4 of [54]. The RS-S algorithm applies to an arbitrary tree decomposition, i.e. trees may be adaptively refined. To compute complexity bounds they impose some structure to the tree. Specifically, they assume, as standard, a tree with $L = O(\log N)$ levels in d dimensions is given such that each leaf node contains at most a constant number of degrees of freedom independent of N. Letting k_l denote the maximum of $\mathcal{S}_i$, i.e the skeleton degrees of of freedom of the tree's nodes on level l, and assuming $k_l \leq k_{l+1}$ for all l.

Under the above assumptions, and assuming further that a constant number of points is used to discretise the proxy surface Γ, they show that the cost T_f of construction the RS-S factorisation F using the bottom-up recursive procedure described in chapter 7, and the cost T_s of applying F or F^{-1} are given by,

$$T_f = O(N) + \sum_{l-1}^{L-2} O(2^{d(L-l)} k_l^3)$$

$$T_s = O(N) + \sum_{l=1}^{L-2} O(2^{d(L-l)} k_l^2)$$

The memory requirement is $m_f = O(t_s)$. To show this, let $k_0 = 1$ without loss of generality. Note that for the DOF set $\mathcal{B}_i$ corresponding to the points in a box at level l, we have that $|\mathcal{B}_i| = O(k_{l-1})$, the near field degrees of freedom $|\mathcal{N}_i| = O(k_{l-1})$ and the far-field degrees of freedom within the proxy surface $|\mathcal{Q}_i| = O(k_{l-1})$, since for the leaf nodes the number of degrees of freedom is bounded by a constant, and for non-leaf nodes the degrees of freedom are given by aggregating skeleton degrees of freedom of their children at the previous level.

Because of the proxy compression, the cost of the ID for the skeletonisation of a given box was found to be reduced as we only needed the ID on a box of size $O(|\mathcal{Q}_i|) \times O(|\mathcal{B}_i|)$ in chapter 7. Using the complexity result of the ID from chapter 7, we see that the cost of skeletonisation with respect to the degrees of freedom B_i corresponding to a node at level l is $O(k_l^3)$. Finally, at each level l there are at most $2^{d(L-l)}$ boxes, which gives the stated complexity for T_f above using the fact the $2^{dL} = O(N)$. The authors of [54] note that the complexity for T_s can be similarly derived, noting that all block unit-triangular matrices can be trivially inverted.

We note that the initial tree construction cost will still require an algorithm of $O(N \log N)$ at the least, however the authors note that in practice this is usually a negligible cost in comparison to the factorisation or application themselves.

For kernels that we're interested in, i.e. Green's functions arising from elliptic PDEs at low-moderate frequencies, standard multipole expansions can be used to show that the far field blocks have ranks that depend only weakly on N [54]. In RS-S far-field interaction blocks that have received Schur complement updates to some of their entries from earlier skeletonisation steps, therefore multipole expansions don't strictly apply, but Minden et al indicate that similar rank behaviour still applies which they've confirmed with numerical experiments. Proceeding with this this assumption they observe a stronger complexity bound.

For a fixed tolerance ϵ, we have $k_l = O(l^q)$ for some $q > 0$. ie. the skeleton sets grow only as some power of the level index l, and $k_{L-2} = O(\log^q N)$, then we obtain

$$T_f = T_s = m_f = O(N)$$

with differing constants depending on the tolerance ϵ and dimension d.

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

Working in the setting in which we derived the multipole expansion in equation (2.8),

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

Deriving the local expansion centered around the origin, where the bounding box of the targets, Ω_t, is well separated from the source box, Ω_s,

$$\phi(x) = \sum_{l=0}^{\infty} \hat{\phi}_l^t (x - c_t)^l$$

from the multipole expansion relies on the following expressions,

$$\begin{aligned} \log((x - c_t) - c_s) &= \log(-c_s(1 - \frac{x - c_t}{c_s})) \\ &= \log(-c_s) - \sum_{l=1}^{\infty} \frac{1}{l} \left(\frac{x - c_t}{c_s} \right)^l \end{aligned}$$

and,

$$\begin{aligned} ((x - c_t) - c_s)^{-p} &= \left(\frac{-1}{c_s} \right)^p \left(\frac{1}{1 - \frac{x - c_t}{c_s}} \right)^p \\ &= \left(\frac{-1}{c_s} \right)^p \sum_{l=0}^{\infty} \binom{l+p-1}{p-1} \left(\frac{x - c_t}{c_s} \right)^l \end{aligned}$$

Substituting these expressions into (A.1), translated to be centred on Ω_t

$$\phi(x) = \log((x - c_t) - c_s)\hat{q}_0^s + \sum_{p=1}^{\infty} \frac{1}{((x - c_t) - c_s)^p}\hat{q}_p^s$$
$$= \log(-c_s)\hat{q}_0^s - \left(\sum_{l=1}^{\infty} \frac{1}{l}\left(\frac{x - c_t}{c_s}\right)^l\right)\hat{q}_0^s + \sum_{p=1}^{\infty}\left(\frac{-1}{c_s}\right)^p \sum_{l=0}^{\infty}\binom{l+p-1}{p-1}\left(\frac{x - c_t}{c_s}\right)^l \hat{q}_p^s$$

Identifying the local expansion coefficients as,

$$\hat{\phi}_0^t = \hat{q}_0^s \log(-c_s) + \sum_{p=1}^{\infty} \frac{\hat{q}_p^s}{c_s^p}(-1)^p$$

and,

$$\hat{\phi}_l^t = \frac{-\hat{q}_0^s}{l c_s^l} + \frac{1}{c_s^l}\sum_{p=1}^{\infty} \frac{\hat{q}_p^s}{c_s^p}\binom{l+p-1}{p-1}(-1)^p$$

HykSort

The parallel splitter selection and HykSort algorithms are provided below. In terms of complexity analysis, we adapt the analysis provided in section 3.4 of [69]. The main costs of SampleSort is sorting the splitters and the MPI collectives for data reshuffling. This can lead to a load imbalance and network congestion, represented by a constant c below,

$$T_{ss} = t_c c\frac{N}{p}\log\frac{N}{p} + (t_s + t_w o)\log^2 p + t_w c\frac{N}{p}$$

Where t_c is the intranode memory slowness (1/RAM bandwidth), t_s interconnect latency, t_w is the interconnect slowness (1/bandwidth), p is the number of MPI tasks in *comm*, and N is the total number of keys in an input array A, of length N.

The parallel splitter selection algorithm for determining k splitters uses MPI collectives, `All_Gather`() and `All_Reduce`(). The main cost is in determining the local ranks of the samples using a binary search. The number of iterations η depends on the input distribution, the required tolerance N_ϵ/N and the parameter β. The expected value of η varies as $\log(\epsilon)/\log(\beta)$ and β is chosen experimentally to minimise the running time, leading to a complexity of,

$$T_{ps} = \eta t_c \beta k \log\frac{N}{p} + \eta(t_s + t_w \beta k)\log p$$

HykSort relies on a specialised `All_to_all_kway`() collective, we defer to the original paper for details. It uses only point to point communications with staged message sends and receives, allowing HykSort to minimise network congestion. It has $\log p/\log k$ stages with $O(N/p)$ data transfer and k messages for each task in every stage. This leads to a complexity of,

$$T_{a2a} = \left(t_s k + t_w \frac{N}{p}\right) \frac{\log p}{\log k}$$

Finally, HykSort has the same communication pattern as `All_to_all_kway`(). In addition it relies on the parallel splitter selection algorithm to determine splitters. The main computational cost is the initial local sort, and merging k arrays during each iteration.

$$T_{Hk} = t_c \frac{N}{p} \log \frac{N}{p} + \left(t_c \frac{N}{p} + T_{ps}\right) \frac{\log p}{\log k} + T_{a2a} \tag{A.2}$$

Unlike SampleSort, the complexity of HykSort doesn't involve any $O(p)$ terms. This is the term that can lead to network congestion for higher core counts.

Algorithm 9 Parallel Select

Input: A_r - array to be sorted (local to each process), n - number of elements in A_r, N - total number of elements, $R[0, ..., k-1]$ - expected global ranks, N_ϵ - global rank tolerance, $\beta \in [20, 40]$,
Output: $S \subset A$ - global splitters, where A is the global array to be sorted, with approximate global ranks $R[0, ..., k-1]$
$R^{\text{start}} \leftarrow [0, ..., 0]$ - Start range of sampling splitters
$R^{\text{end}} \leftarrow [n, ..., n]$ - End range of sampling splitters
$n_s \leftarrow [\beta/p, ..., \beta/p]$ - Number of local samples, each splitters
$N_{\text{err}} \leftarrow N_\epsilon + 1$
while $N_{\text{err}} > N_\epsilon$ **do**
  $Q' \leftarrow A_r[\texttt{rand}(n_s, (R^{\text{start}}, R^{\text{end}}))]$
  $Q \leftarrow \texttt{Sort}(\texttt{All\_Gather}(\hat{Q}'))$
  $R^{loc} \leftarrow \texttt{Rank}(Q, A_r)$
  $R^{glb} \leftarrow \texttt{All\_Reduce}(R^{loc})$
  $I[i] \leftarrow \text{argmin}_j |R^{glb} - R[I]|$
  $N_{err} \leftarrow \max |R^{glb} - RI|$
  $R^{\text{start}} \leftarrow R^{loc}[I-1]$
  $R^{\text{end}} \leftarrow R^{loc}[I+1]$
  $n_s \leftarrow \beta \frac{R^{\text{end}} - R^{\text{start}}}{R^{glb}[I+1] - R^{glb}[I-1]}$
end while
return $S \leftarrow Q[I]$

Algorithm 10 HykSort

Input: A_r - array to be sorted (local to each process), *comm* - MPI communicator, p - number of processes, p_r - rank of current task in *comm*
Output: globally sorted array B.
while $p > 1$, Iters: $O(\log p / \log k)$ **do**
  $N \leftarrow$ `MPI_AllReduce`$(|B|, comm)$
  $s \leftarrow$ `ParallelSelect`$(B, \{iN/k; i = 1, ..., k-1\})$
  $d_{i+1} \leftarrow$ `Rank`(s_i, B), $\forall i$
  $[d_0, d_k] \leftarrow [0, n]$
  $color \leftarrow \lfloor kp_r/p \rfloor$
  parfor $i \in 0, ..., k-1$ **do**
    $p_{recv} \leftarrow m((color - i) \text{mod} k) + (p_r \text{mod} m)$
    $R_i \leftarrow$ `MPI_Irecv`$(p_{recv}, comm)$
  end parfor
  for $i \in 0, ..., k-1$ **do**
    $p_{recv} \leftarrow m((color - i) \text{mod} k) + p_r \text{mod} m$
    $p_{send} \leftarrow m((color + i) \text{mod} k) + p_r \text{mod} m$
    $j \leftarrow 2$
    while $i > 0$ and $i \text{mod} j = 0$ **do**
      $R_{i-j} \leftarrow$ `merge`$(R_{i-j}, R_{i-j/2})$
      $j \leftarrow 2j$
    end while
    `MPI_WaitRecv`(p_{recv})
  end for
  `MPI_WaitAll`()
  $B \leftarrow$ `merge`$(R_0, R_{k/2})$
  $comm \leftarrow$ `MPI_Comm_splitt`(color, comm)
  $p_r \leftarrow$ `MPI_Comm_rank`$(comm)$
end while
return B

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 and Eric Darve. "The inverse fast multipole method". In: *arXiv preprint arXiv:1407.1572* (2014).

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

[5] *B2: makes it easy to build C++ projects, everywhere*. 2022. URL: `https://github.com/boostorg/build`.

[6] *Bazel - a fast, scalable, multi-language and extensible build system*. Version 5.3.2. 2022. URL: `https://github.com/bazelbuild/bazel`.

[7] *BLAS and LAPACK for Rust*. 2022. URL: `https://github.com/blas-lapack-rs`.

[8] Steffen Boerm et al. *H2Lib: A software for hierarchical and H2 matrices*. Version 3.0.0. URL: `https://github.com/H2Lib/H2Lib`.

[9] 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.

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

[11] James Bremer and Zydrunas Gimbutas. "On the numerical evaluation of the singular integrals of scattering theory". In: *Journal of Computational Physics* 251 (2013), pp. 327–343.

[12] Oscar Bruno, Tim Elling, and Catalin Turc. "Regularized integral equations and fast high-order solvers for sound-hard acoustic scattering problems". In: *International Journal for Numerical Methods in Engineering* 91.10 (2012), pp. 1045–1072. ISSN: 00295981. DOI: `10.1002/NME.4302`.

[13] Carsten Burstedde, Lucas C. Wilcox, and Omar Ghattas. "`p4est`: Scalable Algorithms for Parallel Adaptive Mesh Refinement on Forests of Octrees". In: *SIAM Journal on Scientific Computing* 33.3 (2011), pp. 1103–1133. DOI: `10.1137/100791634`.

[14] 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.

[15] 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.

[16] Hongwei Cheng et al. "On the compression of low rank matrices". In: *SIAM Journal on Scientific Computing* 26.4 (2005), pp. 1389–1404.

[17] Barry A Cipra. "The best of the 20th century: Editors name top 10 algorithms". In: *SIAM news* 33.4 (2000), pp. 1–2.

[18] David Colton and Rainer Kress. *Integral equation methods in scattering theory*. SIAM, 2013.

[19] *Conan - The open-source C/C++ package manager*. Version 1.53.0. 2022. URL: `https://github.com/conan-io/conan`.

[20] 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.

[21] 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.

[22] *Dendro - Distributed adaptive octree construction*. Version 2.0. 2020. URL: `https://github.com/paralab/Dendro-5.01`.

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

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

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

[26] *Fortran Package Manager (fpm)*. Version 0.7.0. 2022. URL: `https://github.com/fortran-lang/fpm`.

[27] 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.

[28] 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.

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

[30] 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.

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

[32] 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.

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

[34] 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.

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

[36] 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.

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

[38] Kenneth L Ho and Leslie Greengard. "A fast direct solver for structured linear systems by recursive skeletonization". In: *SIAM Journal on Scientific Computing* 34.5 (2012), A2507–A2532.

[39] 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).

[40] Steve Klabnik and Carol Nichols. *The Rust Programming Language (Covers Rust 2018)*. No Starch Press, 2019.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] Per-Gunnar Martinsson and Vladimir Rokhlin. "A fast direct solver for boundary integral equations in two dimensions". In: *Journal of Computational Physics* 205.1 (2005), pp. 1–23.

[49] 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.

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

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

[52] *Microsoft: 70 percent of all security bugs are memory safety issues*. `https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/`. Accessed: 2022-10-31.

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

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

[55] *ndarray: an N-dimensional array with array views, multidimensional slicing, and efficient operations*. Version 0.15.6. 2022. URL: `https://github.com/rust-ndarray/ndarray`.

[56] *PyO3: Rust bindings for the Python interpreter*. Version 0.17.3. Nov. 1, 2022. URL: `https://github.com/PyO3/pyo3`.

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

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

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

[60] 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.

[61] *Rust-CPython: Rust-Python bindings*. Version 0.7.1. Oct. 25, 2022. URL: `https://github.com/dgrunwald/rust-cpython`.

[62] Youcef Saad and Martin H Schultz. "GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems". In: *SIAM Journal on scientific and statistical computing* 7.3 (1986), pp. 856–869.

[63] *SCons - a software construction tool*. Version 4.4.0. 2022. URL: `https://github.com/SCons/scons`.

[64] Craig Scott. *Professional CMake: A Practical Guide*. 2018.

[65] *Spack - A flexible package manager that supports multiple versions, configurations, platforms, and compilers*. Version 0.18.1. 2022. URL: `https://github.com/spack/spack`.

[66] Benedikt Steinbusch and Andrew Gaspar et al. *RSMPI: MPI bindings for Rust*. Version 0.5.4. 2018. URL: `https://github.com/rsmpi/rsmpi`.

[67] Josh Stone and Niko Matsakis et. al. *Rayon: A data parallelism library for Rust*. Version 1.5.3. 2022. URL: `https://github.com/rayon-rs/rayon`.

[68] Hansol Suh and Tobin Isaac. "Evaluation of a minimally synchronous algorithm for 2: 1 octree balance". In: *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2020, pp. 1–12.

[69] Hari Sundar, Dhairya Malhotra, and George Biros. "Hyksort: a new variant of hypercube quicksort on distributed memory architectures". In: *Proceedings of the 27th international ACM conference on international conference on supercomputing*. 2013, pp. 293–302.

[70] 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.

[71] Daria Sushnikova et al. "FMM-LU: A fast direct solver for multiscale boundary integral equations in three dimensions". In: *arXiv preprint arXiv:2201.07325* (2022).

[72] *The Meson Build System*. Version 0.63.3. 2022. URL: `https://github.com/mesonbuild/meson`.

[73] Tiankai Tu, David R O'Hallaron, and Omar Ghattas. "Scalable parallel octree meshing for terascale applications". In: *SC'05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*. IEEE. 2005, pp. 4–4.

[74] *VCPKG - C++ Library Manager for Windows, Linux, and MacOS*. Version 2022.10.19. 2022. URL: `https://github.com/microsoft/vcpkg`.

[75] Bruce Wagar. "Hyperquicksort: A fast sorting algorithm for hypercubes". In: *Hypercube Multiprocessors* 1987 (1987), pp. 292–299.

[76] 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.

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

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

[79] 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.

[80] 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`.