Implementing Fast Multipole Methods with High-Level Interpreted Languages

Srinath Kailasa

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

Department of Physics
University College London
August 31, 2020

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.

Acknowledgements

This thesis has been the product of two very long and concurrently short years at UCL. Returning to education whilst extricating myself from industry presented a set of challenges which I wasn't entirely expecting, but thrilled to have overcome. I'd like to thank my Mum for keeping me on the right path, and without whose support this thesis would probably never have happened. Furthermore, I'd like to thank my supervisor Timo Betcke, whose door was always open, I couldn't really have asked for a much better thesis supervisor; and with whom, Master's in hand, I'm greatly looking forward to starting my PhD with in the autumn.

Abstract

The Fast Multipole Method (FMM) is a numerical method to accelerate the solution of the N-Body problem, which appears in numerous contexts in science and engineering, for example in solving for gravitational or electrostatic potentials in multi-particle systems. It does so by approximating the Green's function of the system with analytic infinite series expansions (the origin of the 'multipole' in its name), and coalescing the effect of distinct distant sources together so as to greatly reduce the number of computations. The analytic expansions of the original Fast Multipole Method depend on the Green's function of the system in question (Helmholtz, Laplace etc.), and in practice a new implementation must be written for a given system.

The Kernel-Independent Fast Multipole Method (KIFMM), first presented by Ying et al. [17], is a similar approach that replaces the analytic series expansions with a continuous distribution of so called 'equivalent density' supported at discrete points on a box enclosing a set of particles. These equivalent densities are found by matching the potential they generate to those generated by the original sources at another surface in the far field. Usefully, this approach doesn't require multipole expansions of the Green's functions of a system, and therefore can be programmed in an agnostic way, hence the origin of it's name. The KIFMM is compatible with a broad class of elliptic Green's functions, and is therefore extremely useful for studying a broad range of problems with a single software implementation.

This thesis presents a well tested and extensible Python implementation of the KIFMM for simulations in three dimensions, PyExaFMM, with investigations made into both mathematical and computational techniques for the acceleration of the software, using the N-Body electrostatic problem as model on which to test the implementation. Python is chosen as it has emerged as a standard for scientific and data intensive computing in recent years, with a huge increase in adoption, and a well supported ecosystem of libraries and tools available for accelerating numerical codes. The wider context of this thesis is an ongoing collaboration with the ExaFMM Project [24] to produce a Python implementation of the KIFMM that sacrifices as little performance as possible. This thesis introduces the relevant theoretical background, before proceeding to discuss the strategies used in the practical implementation of the software. Key bottlenecks in the implementation are examined and addressed, with the speed and accuracy of the software benchmarked with the Laplace kernel for three dimensional electrostatic problems. Finally, future avenues of investigation and software development are discussed, in the context of the wider literature.

Contents

Introduction

1.1 Overview of the Analytic FMM

1.1.1 Motivation

The paradigmatic problem of Fast Multipole Methods (**FMM**) [1] is the so called N-Body problem. This classic problem refers to the calculation of the pairwise interactions between N particles over a potentially long-range, for example in gravitational or electrostatic problems. The straightforward calculation can be written in the form of the following sum,

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

Where $i, j \in [1, N]$ and $K(x, y)$ is called the Green's function, or equivalently a 'kernel function', where one is generally concerned with coordinates of particles in an $n = 2$ or 3 dimensional Hilbert space taking $x_i \in \mathbb{R}^n$. Additionally, each summand is weighted by w_i. For solving for electrostatic potential in three dimensions, which is used as the model problem throughout this thesis, this goes to,

$$\Phi(x_j) = \sum_{i=1}^{N} q_i K(x_i, x_j) \tag{1.2}$$

where q_i refers to a charge density with the kernel function,

$$K(x, y) = \frac{1}{4\pi\epsilon_0} \frac{1}{|x - y|} \tag{1.3}$$

the constant ϵ_0 is the permittivity of free space. For practical purposes, we define $K(x, x) = 0$. It's easy to see how a naive direct application of this equation over N particles results in an algorithm of $O(N^2)$ complexity, therefore it's only practicable for systems of moderate size, whereas in realistic systems, one may be interested in interactions involving 10^6 to 10^8 particles.

This chapter introduces the analytic FMM, the kernel-independent version, which is the main focus of this thesis, is presented later. Though substantially different in implementation, the analytic FMM will provide the opportunity to exposit many of the key ideas behind all FMM-based algorithms, and provides a good starting point for understanding and developing upon these algorithms. First presented by Greengard [11], the analytic FMM represented a sea change for N-Body simulation. By trading off computations for error, it manages to achieve an asymptotic complexity

[1] The first usage of a technical term or abbreviation listed in the glossary is highlighted throughout the text for ease of reference.

of just $O(N)$. Additionally, it comes equipped with rigorous error bounds, making fast and accurate massive N-Body simulations feasible on available computing hardware. It's success has been such that it is regarded as one of the key developments in numerical algorithms in the twentieth century [5].

The original analytic FMM solves the electrostatic problem in two and three dimensions, this is equivalently known as the Poisson problem, represented by the differential equation,

$$\nabla^2 \phi = f \tag{1.4}$$

Where ϕ is some scalar potential to be determined, and f is a scalar source term which is usually known. For electrostatics the corresponding formulation can be derived from Gauss' law as [14],

$$\nabla^2 \phi = -\frac{q}{\epsilon_0} \tag{1.5}$$

where ϕ is the electrostatic potential, q is the charge density and ϵ_0 is the permittivity of free space. It can therefore be seen that the FMM is actually solving the Poisson problem by reformulating it as an integral equation [7]. The ubiquity of problems of the form (1.1) in computational science has lead to diverse applications of the FMM. For example, in the modeling the electrostatic interactions of charged particles in complex biological molecules at biologically relevant length scales [3]. The extension of FMMs to Helmholtz equations [25], has lead to even more applications, such as in seismic and acoustic scattering [16]. Though as the focus of this thesis is on solving the Poisson model problem for electrostatics, this is mentioned only for completeness.

The key insight that leads to the FMM's asymptotic complexity is the idea that if the field created by a distribution of charge (or mass) density is approximated to be relatively smooth in the **far field**, then it should be possible to apply some form of compression for the evaluation of contribution to local potentials due to particles in the far field. The FMM performs this compression by encoding the field contributions of particles in the far field using a multipole expansion.

For simple kernel functions and charge distributions, such as the model problem of this thesis, one can easily derive the expression for this multipole expansion by finding an series expansion of the system's Green's function. In order to generalise the discussion, an arbitrary continuous distribution of charge is considered as shown in figure 1.1, for which the potential is evaluated at some other evaluation point outside of the distribution. This can be written as follows [14],

$$\Phi(\mathbf{r}) = \frac{1}{4\pi\epsilon_0} \int \frac{1}{d} \rho(\mathbf{r}') d\tau' \tag{1.6}$$

where $\rho(\mathbf{r}')$ is a charge density, and the other symbols take their meanings from figure (1.1).

From law of the cosines,

$$d^2 = r^2 + (r')^2 - 2rr' \cos\alpha = r^2 \left[1 + \left(\frac{r'}{r} \right)^2 - 2 \left(\frac{r'}{r} \right) \cos\alpha \right] \tag{1.7}$$

$$d = r\sqrt{1+\epsilon}, \tag{1.8}$$

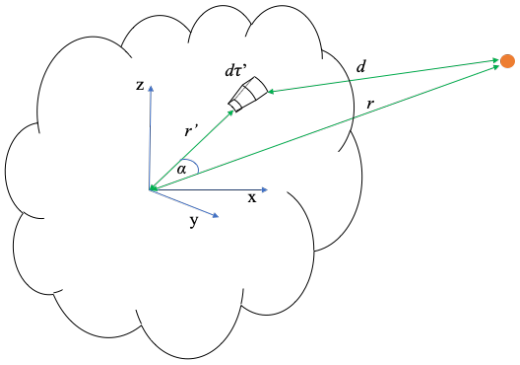

Figure 1.1: An arbitrary charge distribution, with an orange point to mark a point where the potential is being evaluated. Here, $\mathbf{r}$ is the the vector between the centre of the multipole expansion and the evaluation point, $\mathbf{r'}$ is the vector between the centre of expansion and a given volume element $d\tau'$, and d is a vector between the volume element $d\tau'$ and the evaluation point.

where,

$$\epsilon \equiv \left(\frac{r'}{r}\right)\left(\frac{r'}{r} - 2\cos\alpha\right) \tag{1.9}$$

As ϵ is small far away from charge distribution one can expand $1/d$ binomially,

$$\frac{1}{d} = \frac{1}{r}(1+\epsilon)^{-1/2} = \frac{1}{r}\left(1 - \frac{1}{2}\epsilon + \frac{3}{8}\epsilon^2 - ...\right) \tag{1.10}$$

$$\frac{1}{d} = \frac{1}{r}\sum_{n=0}^{\infty}\left(\frac{r'}{r}\right)^n P_n(\cos\alpha), \tag{1.11}$$

where $P_n(\cos\alpha)$ are Legendre polynomials. Using this, the exact multipole expansion for this charge distribution is,

$$\Phi(\mathbf{r}) = \frac{1}{4\pi\epsilon_0}\sum_{n=0}^{\infty}\frac{1}{r^{n+1}}\int (r')^n P_n(\cos\alpha)\rho(\mathbf{r'})d\tau' \tag{1.12}$$

If instead one considers a charge distribution composed of N discrete charges q_i at positions r_i, this goes to,

$$\Phi(\mathbf{r}) = \frac{1}{4\pi\epsilon_0}\sum_{i=1}^{N}\sum_{n=0}^{\infty}\frac{(r_i)^n q_i}{r^{n+1}}P_n(\cos\alpha) \tag{1.13}$$

Using the addition theorem for Legendre polynomials [12],

$$P_n(\cos\gamma) = \sum_{m=-n}^{n} Y_n^{-m}(\alpha,\beta)Y_n^m(\theta,\phi), \tag{1.14}$$

where the Legendre polynomial is written in terms of spherical harmonics, where (r,θ,ϕ) and (ρ,α,β) define two spherical coordinates, and γ is the angle subtended between them. The multipole expansion goes to,

$$\Phi(\mathbf{r}) = \frac{1}{4\pi\epsilon_0} \sum_{i=1}^{N} \sum_{n=0}^{\infty} \frac{(r_i)^n q_i}{r^{n+1}} P_n(\cos\alpha) \tag{1.15}$$

$$= \frac{1}{4\pi\epsilon_0} \sum_{i=1}^{N} \sum_{n=0}^{\infty} \sum_{m=-n}^{n} \frac{(r_i)^n q_i Y_n^{-m}(\alpha_i, \beta_i)}{r^{n+1}} Y_n^m(\theta, \phi) \tag{1.16}$$

$$= \frac{1}{4\pi\epsilon_0} \sum_{n=0}^{\infty} \sum_{m=-n}^{n} M_n^m \cdot \frac{Y_n^m(\theta, \phi)}{r^{n+1}}, \tag{1.17}$$

where,

$$M_n^m = \sum_{i=1}^{N} (r_i)^n q_i Y_n^{-m}(\alpha_i, \beta_i) \tag{1.18}$$

This is an exact expansion, and it converges for $\frac{r_i}{r} < 1$. This convergence condition means that estimating the potential at a given evaluation point using the multipole expansion is only possible in the far-field, the boundary of which is often tuned empirically for different systems as it's user defined. If instead the expansion is taken centered at the evaluation point, one can rewrite as the multipole expansion as a 'local' expansion,

$$\Phi(\mathbf{r}) = \frac{1}{4\pi\epsilon_0} \sum_{i=1}^{N} \sum_{n=0}^{\infty} \sum_{m=-n}^{n} \frac{(r)^n q_i Y_n^{-m}(\alpha_i, \beta_i)}{(r_i)^{n+1}} Y_n^m(\theta, \phi) \tag{1.19}$$

$$= \frac{1}{4\pi\epsilon_0} \sum_{n=0}^{\infty} \sum_{m=-n}^{n} L_n^m \cdot Y_n^m(\theta, \phi) \cdot r^n, \tag{1.20}$$

where,

$$L_n^m = \sum_{i=1}^{N} \frac{q_i Y_n^{-m}(\alpha_i, \beta_i)}{(r_i)^{n+1}}, \tag{1.21}$$

which converges when $\frac{r}{r_i} < 1$. The region of convergence for both types of expansions are shown in figure (1.2).

The key point to note is that the multipole and local expansions are exact, and can be truncated as required to ensure that the asymptotic complexity of evaluating a multipole or local expansion at an evaluation point is bounded by $O(N)$. A rigorous error analysis of the analytic FMM is outside the scope of this thesis, and we defer to the literature for exact expressions for these truncation errors [11], Furthermore, there exist exact operations for shifting the center of these expansions, as well as for translating multipole expansion coefficients into equivalent local expansion coefficients. These are crucial in providing the improvements to asymptotic complexity[2].

[2]Expressions for these shift operators for the three dimensional Laplace kernel considered as our model problem are provided in Appendix A.2.

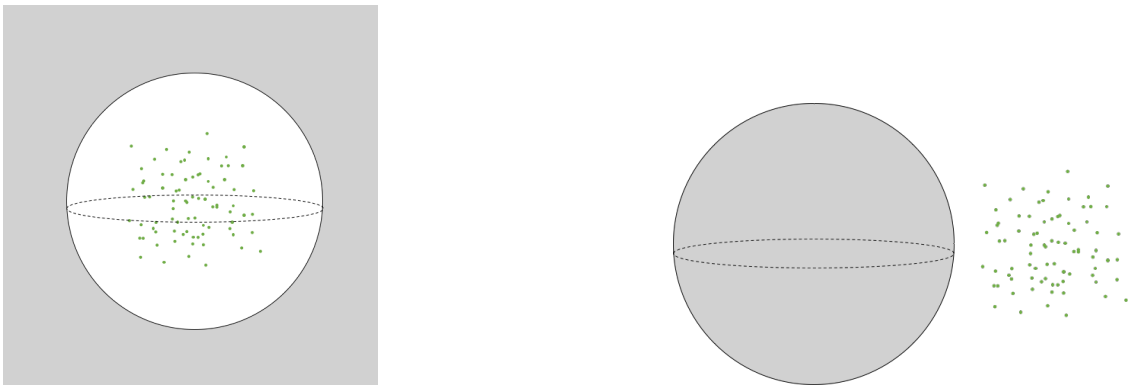

Figure 1.2: (A) A multipole expansion centered on charge distribution. (B) A local expansion, centered around a point of evaluation. Regions in which the expansions converge are shaded in grey. For the multipole expansion this is the entire domain outside of the region for $r > r_i$, $\forall i \in [1, N]$, and for the local expansion this is the region for which $r < r_i$, $\forall i \in [1, N]$

1.1.2 Algorithm Structure & Analysis

The convergence condition of the multipole expansion prohibits the compression of charges from particles in the **near field**. Therefore the FMM makes use of a tree structure in a recursive algorithm, this structure is known as an Octree in three dimensions and a Quadtree in two dimensions[3]. This structure hierarchically partitions space such that each level, l, of the tree is equally partioned into $(2^d)^l$ boxes[4] over the domain of the tree, where d is the dimension, i.e. $d = 3$ in three dimensions. If one were to simply traverse the tree from the coarsest, or 'root', level to the finest, or 'leaf', level and find the multipole expansion of source particles in each box of each level, one could then evaluate these multipole expansions at each particle to solve the N-Body problem. As there are $O(\log(N))$ boxes in the tree, and N particles, this results in a $O(N \log(N))$ asymptotic complexity.

However the FMM reduces computational complexity further by making use of local expansions. Using the exact expressions to shift the multipole expansion coefficients[5] M_n^m to local expansion coefficients L_n^m one can approximate the the interaction between two boxes in the tree. Roughly speaking, because of the hierarchical nature of the tree, each box only needs to consider the interaction with a constant number of neighbouring boxes. Because the number of boxes is $O(N)$, the FMM is bound by an $O(N)$ asymptotic complexity [16].

Using the above analysis, one can then describe the FMM algorithm in terms of two basic steps,

1. ***Upward Pass***: The tree is traversed **post-order**. Beginning at the leaf boxes, a multipole expansion is computed for each box due to the **source**

[3]The usage of trees naturally leads to biological adjectives to describe their structure, for example the coarsest level of a tree is referred to as the 'root' level, and the finest level as the 'leaf' level. Confusingly these are often combined with familial adjectives to describe the relationship between tree nodes, for example parent and child nodes to describe the relationship between a given node and the nodes that occupy the same domain a level deeper in the tree. Each node has exactly one parent, except the root node which has no parents.

[4]Regardless of the spatial dimension, partitions of a domain are invariably referred to as boxes.

[5]Expressions for these shift operators for the three dimensional Laplace kernel considered as our model problem are provided in Appendix A.2.

particles it contains. This is also referred to as the particle-to-multipole operation, or **P2M**. Then as one moves up the tree hierarchy, the multipole expansions of a each box's parent box is computed by shifting the expansion centers of the multipole expansion of a given child box, to the center of the parent box, in a multipole-to-multipole operation or **M2M**, and summing together all the coefficients. Following the upward pass, one obtains the multipole expansion for each box containing source particles at all levels of the hierarchical tree.

2. ***Downward Pass***: The tree is now traversed in **pre-order**, and the local expansion of each box is computed. This local expansion is the sum of two parts: (1) the local expansion of the parent box of a given box, if it exists, which is a compression of the potential due to boxes non-adjacent to a given box's parent. The parent box's local expansion centre is shifted to the center of the child box, this is also known as the local-to-local operation or **L2L**. (2) The multipole expansion of boxes which are the children of the **near neighbours** of a given box's parent but are not adjacent to the box itself. Such 'source' boxes are described as being in the **interaction list** of a given box. These multipole expansions for each source box in a given box's interaction list are translated into local expansions centered at the given box, this is also known as the multipole-to-local operation, or **M2L**. Notice that the M2L operation is only available from $l = 2$ of the tree, as in coarser levels the interaction list for all boxes is empty. The coefficients found from (1) and (2) are summed for each box. Operations (1) and (2) are repeated for each box until the leaf level. At this point, the local expansion of each leaf box is evaluated at all the **target particles** it contains. This local-to-particle, or **L2P**, operation encodes all the far field interaction of the target particles in this leaf box. This is then combined with a near field interaction, due to the source particles in the leaf box, as well as in the near neighbours, which are computed directly. As the tree is refined to the point where the leaf levels contain only a small constant number of particles, this final direct computation is of low cost.

With this specification, a more detailed analysis of the algorithm is possible, though we defer to [11] for a rigorous discussion. Firstly, as mentioned above, the tree must be refined such that the leaf boxes contain only a small constant number of particles, κ. The level of refinement n is therefore approximately taken to be $n \approx \log(N)$, where N is the number of source particles in the tree. Beginning with upward pass, at the leaf level each particle contributes to one multipole expansion. If this expansion is truncated to contain p multipole terms, the P2M operation has a complexity of $O(Np^2)$, which can be seen from (1.17), as well as the fact that the nature of a hierarchical tree means that are $O(N)$ boxes at the leaf level. The shift operators[6] M2L, L2L and M2M require p^4 operations with this truncation, so the computation of all of these are bounded by[7] $O(Np^4)$. Finally, evaluating the p^{th} degree local expansions at each target particle in the L2P operation, is bounded by $O(Np^2)$. The choice for level of refinement n, leads to $O(\kappa N)$ complexity for direct calculations at the leaf level. The whole algorithm is therefore bounded by $O(N)$. In practice, the number of expansion terms p is chosen for a prescribed relative error ϵ,

[6]Expressions for these shift operators for the three dimensional Laplace kernel considered as our model problem are provided in Appendix A.2

[7]A more precise bound depends on the size of a given box's interaction list.

using [8] $p = \log_c \epsilon$. The algorithm is illustrated in figure (1.3) in the two dimensional case, which is direct analogue of the three dimensional case which is the focus of this thesis, and a full pseudo-code specification is provided in Appendix A.1.

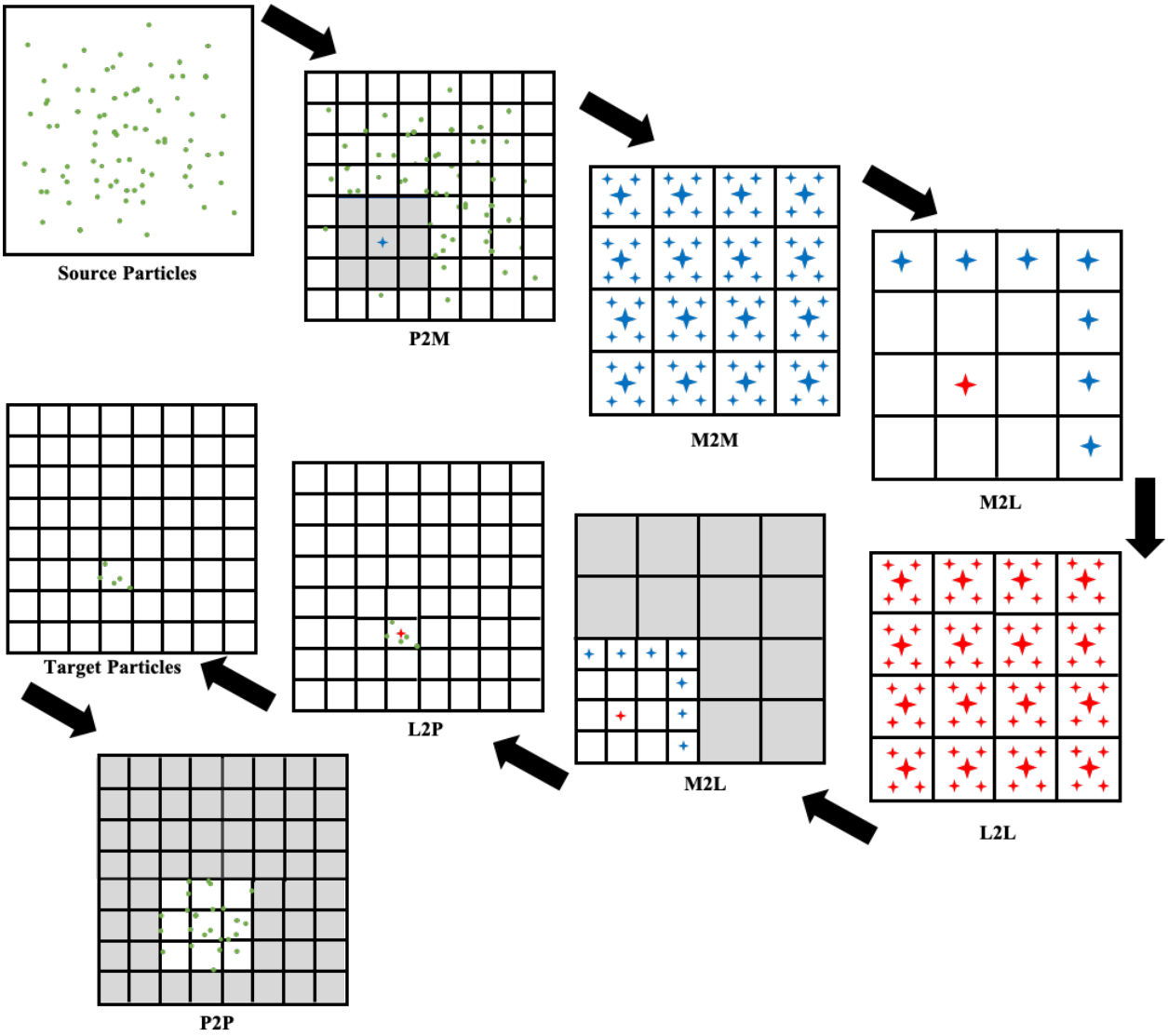

Figure 1.3: The main FMM loop in two dimensions, encapsulating the upward and downward pass. The same set of particles are used for the sources and targets, and are shown in green. Local expansions are illustrated with red stars, and multipole expansions are illustrated with blue stars. For the P2M step the grey region indicates the near field, where this multipole expansion does not converge. For the M2L and P2P steps the grey region indicates box interactions already compressed and available via the translation or direct usage of local expansions. The larger and smaller stars in the M2M and L2L steps correspond to the parent and child expansions respectively.

[8] There are optimal choices for c, with the authors of [17] specifying $c = \frac{4-\sqrt{3}}{\sqrt{3}}$ for three dimensional problems.

1.1.3 Summary

The main practical challenge in implementing software to solve the analytic FMM is the requirement of kernel-specific code to calculate the expansion coefficients of (1.17) and (1.20). For example, multiple different implementations already exist for Poison problems alone [13, 9]. In software engineering terms this is inconvenient for the purposes of studying the applications of the FMM in multiple different problem settings, as it leads to the requirement to implement problem-specific solvers for each kernel one may encounter. This leads to a productivity overhead in either creating a single, extensible library, which by definition will be complex to design. Or to otherwise maintain multiple problem-specific libraries.

The algorithm thus far described referred to as the analytic FMM is more correctly called the non-adaptive analytic FMM. The non-adaptivity refers to the assumption that all boxes at the leaf level are refined to the same degree, however as mentioned in the above analysis of the algorithm, the degree of refinement is chosen such that the number of particles in a leaf box is *constant*. Therefore, If the distribution of the particles of interest is not uniform over the computational domain, one may be needlessly refining the boxes in some regions which may even be empty. Therefore, although the asymptotic complexity of the FMM is $O(N)$, it is clear that practical implementations will suffer unless care is taken to use efficient vectorised data structures for the creation of an appropriate hierarchical tree as required by the algorithm.

Additionally, there is significant scope for multiple levels of parallelism in practical implementations. For example, the computations for the local and multipole coefficients as well as the application of M2L, L2L and M2M operators at a given level, are candidates for an implementation of **task-level parallelism**. In addition to parallelising of each operator application as a task, there is scope for implementing **data-level parallelism** to find the expansion coefficients. For example, Yokota and Barba [16] demonstrate how the calculation of the P2P, and M2L operators can be transferred to **GPU**s using **CUDA**. The M2L, and P2P operators represent the largest computational bottlenecks due to the number of such interactions in the FMM algorithm, therefore are a priority for acceleration. For example, in three dimensions each target box has potentially up to 189 source boxes in its interaction list with which to compute the M2L operation, and for any particularly deep tree there will be roughly $O(N)$ leaf boxes, for which interactions are calculated directly. This leads to both of these operations dominating the run-time of any FMM implementation.

In summary, the implementation of the analytical FMM is complicated by the fact that it is problem specific - which will also apply to any parallel optimisation code. This in itself provides the main motivation for developing an implementation that does not rely on explicit kernel expansions. Furthermore, the desire for developer productivity, at the expense of hyper-optimised implementations, is realised in this thesis by making use of Python, a **high level interpreted language**, for our software implementation.

1.2 Overview of the Kernel-Independent FMM

1.2.1 Motivation

First introduced by Ying et. al [17], the kernel-independent FMM (**KIFMM**) provides an algorithm that maintains the basic recursive structure and $O(N)$ asymptotic complexity of the analytic FMM, but without the requirement for the implementation of analytic expansions of the kernel function for each kernel. Instead the method relies only on kernel function evaluations. This allows software implementations to be written in an easily extensible manner for different kernels. The main difference to the analytic FMM of Section 1.1, lies in the the way that source and target densities are represented, and how the M2M, L2L and M2L operators are computed.

1.2.2 Algorithm Structure & Analysis

For the KIFMM presented in [17] the far field, $\mathcal{F}^B$, and near field, $\mathcal{N}^B$, have precise specifications. For a given box B centered at $\mathbf{c}$ with sides of length $2r$, $\mathcal{N}^B$ is a box centered at $\mathbf{c}$ with sides of length $6r$. The far field is then defined as $\mathbb{R}^d/\mathcal{N}^B$, where d is spatial dimension. Here, B is in the near field. Consider the potential in the far field $\mathcal{F}^B$, generated by a set of source particles, described by **source densities** $\{\phi_i,\ i \in I_s^B\}$ where I_s^B is the set of indices for the source particles in box B[9]. Specifying the indices for the source particles specifically to make it clear that they may be distinct from the target particles. These source particles can be equivalently described with an **upward equivalent density** distribution $\phi^{B,u}$ supported at discrete points on an **upward equivalent surface** $\mathbf{y}^{B,u}$ that encloses the set of source particles. The KIFMM relies on the assumption that the potential produced by the equivalent densities is smooth, which is guaranteed in the case that $\mathbf{y}^{B,u}$ does not overlap with the far-field $\mathcal{F}^B$ [17], furthermore the requirement that $\mathbf{y}^{B,u}$ must enclose all particles in B leads to the requirement that it must also not overlap with B. For second-order linear elliptic **PDE**s, for which the KIFMM is defined, and of which equation (1.4) is an example, the solution for the potential in the far field, which can be seen as an exterior Dirichlet problem, is guaranteed to be unique [17]. Therefore, the potentials induced by the source particles and the equivalent densities satisfy are guaranteed to be equivalent in the far field $\mathcal{F}^B$ if they coincide at the boundary of the far field $\mathcal{F}^B$, or anywhere between the boundary of the far field and the upward equivalent surface. This boundary is referred to as the **upward check surface**, $\mathbf{x}^{B,u}$, and the entire scheme is illustrated in figure (1.4A). The equality of the potentials from the source points and the equivalent density can be stated mathematically as follows,

$$\int_{\mathbf{y}^{B,u}} K(\mathbf{x},\mathbf{y})\phi^{B,u}d\mathbf{y} = \sum_{i\in I_s^B} K(\mathbf{x},\mathbf{y})\phi_i = q^{B,u} \text{ for any } \mathbf{x} \in \mathbf{x}^{B,u} \tag{1.22}$$

where the integral form of (1.1) is used for the summation of the contribution from the equivalent densities, and $q^{B,u}$ is referred to as the **upward check potential**, with the other symbols taking their previous definitions. One can define a very similar scheme for the case in which the source densities are in $\mathcal{F}^B$, as a potential

[9] This notation matches that used in [17] in order for ease of reference.

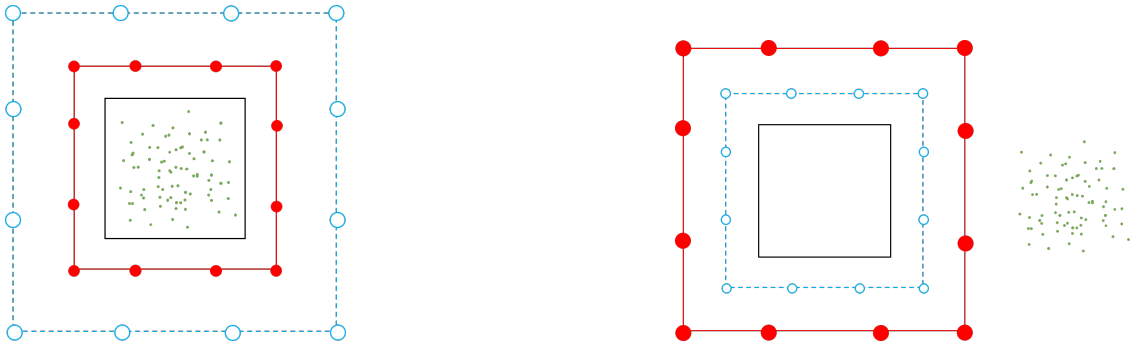

Figure 1.4: Cross section of three dimensional cubic (A) upward and (B) downward, equivalent and check surfaces. Source points are denoted by green circles. Red solid lines denote equivalent surfaces, and blue checked lines denote check surfaces. The black solid line defines the box B. This figure is adapted directly from [17].

induced by a **downward equivalent density** $\phi^{B,d}$ supported at discrete points on a **downward equivalent surface** $\mathbf{y}^{B,d}$. This surface needs to be located between the boundary of $\mathcal{F}^B$ and B, and again as the solution for interior Dirichlet style problems for the types of PDEs considered in this thesis is also unique [17], one can equate the potentials generated by the source points with that generated by the equivalent densities at some surface between $\mathbf{y}^{B,d}$ and B. This surface is called the **downward check surface**, $\mathbf{x}^{B,d}$, and a corresponding mathematical statement of the equality of the potentials can be written as follows,

$$\int_{\mathbf{y}^{B,d}} K(\mathbf{x},\mathbf{y})\phi^{B,d}d\mathbf{y} = \sum_{i\in I_s^{\mathcal{F}B}} K(\mathbf{x},\mathbf{y})\phi_i = q^{B,d} \text{ for any } \mathbf{x}\in\mathbf{x}^{B,d} \tag{1.23}$$

where $I_s^{\mathcal{F}B}$ represents the indices of source points in the far field of B, and $q^{B,d}$ is the **downward check potential**.

The equation (1.22) is an Fredholm integral equation, of the first kind, and it's clear that its solution $\phi^{B,u}$ can be seen to be equivalent to the multipole expansion for the source particles contained in B. Therefore, (1.22) can be seen to be correspond to the P2M operation. Similarly, the solution of (1.23) can be seen to correspond to a particle-to-local, or P2L operation. Crucially, this method of solving a set of linear equations to find equivalents of the multipole and local expansions does not depend on finding a series expansion of a kernel function, and just on its evaluation.

Using this language of equivalent densities and check surfaces, one is also able to write operations equivalent to the M2M, L2L and M2L operations. For a box A and it's parent box B, the M2M operation can be written as,

$$\int_{\mathbf{y}^{A,u}} K(\mathbf{x},\mathbf{y})\phi^{A,u}d\mathbf{y} = \int_{\mathbf{y}^{B,u}} K(\mathbf{x},\mathbf{y})\phi^{B,u}d\mathbf{y}, \text{ for any } \mathbf{x}\in\mathbf{x}^{B,u} \tag{1.24}$$

Once $\phi^{A,u}$ has been calculated for boxes at the leaf level of the tree, one can apply (1.24) to evaluate the multipole expansions for all the boxes containing source particles in the tree in a manner equivalent to the upward pass of Section 1.1. The M2M operation is illustrated in figure (1.5A).

For the downward pass, the M2L operation, between a box B and a box A in its interaction list can be written as,

$$\int_{\mathbf{y}^{A,u}} K(\mathbf{x},\mathbf{y})\phi^{A,u}d\mathbf{y} = \int_{\mathbf{y}^{B,d}} K(\mathbf{x},\mathbf{y})\phi^{B,d}d\mathbf{y}, \text{ for any } \mathbf{x} \in \mathbf{x}^{B,d} \tag{1.25}$$

This is illustrated in figure (1.6). Once the local expansions have been computed starting at level 2 of the tree, one can perform the L2L operation to transfer the local expansion of a parent box to its children, for a box A and it's child B, which can be written as,

$$\int_{\mathbf{y}^{A,d}} K(\mathbf{x},\mathbf{y})\phi^{A,d}d\mathbf{y} = \int_{\mathbf{y}^{B,d}} K(\mathbf{x},\mathbf{y})\phi^{B,d}d\mathbf{y}, \text{ for any } \mathbf{x} \in \mathbf{x}^{B,d} \tag{1.26}$$

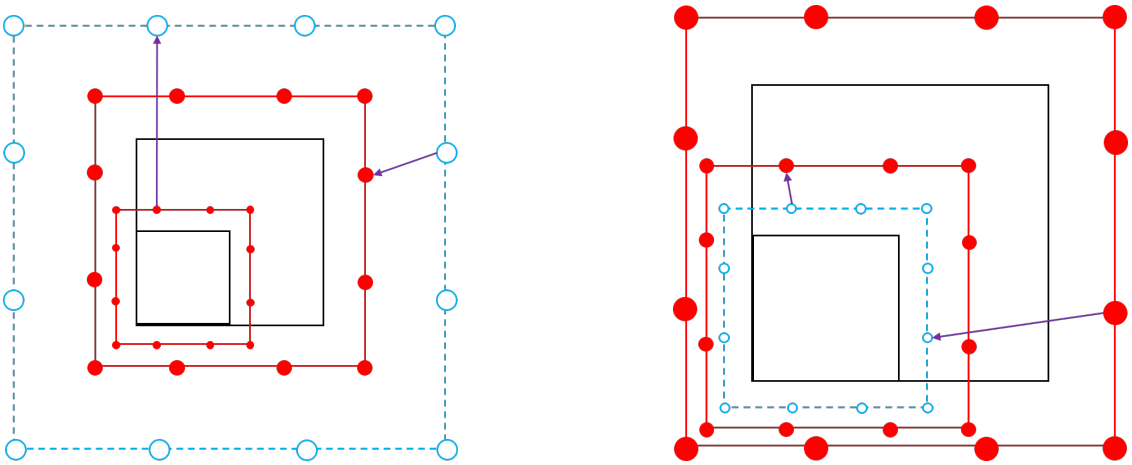

Figure 1.5: Cross section of three dimensional cubic surfaces. The (A) M2M operation and (B) L2L operation. Red solid lines denote equivalent surfaces, and blue checked lines denote check surfaces. The black solid line defines the box B. This figure is adapted directly from [17].

The M2M, L2L and M2L operations are illustrated using cubic surfaces in figure (1.5) The reason for using cubic check and equivalent surfaces are for ease of integration in the case where the discretisation points chosen are from a regular Cartesian grid.

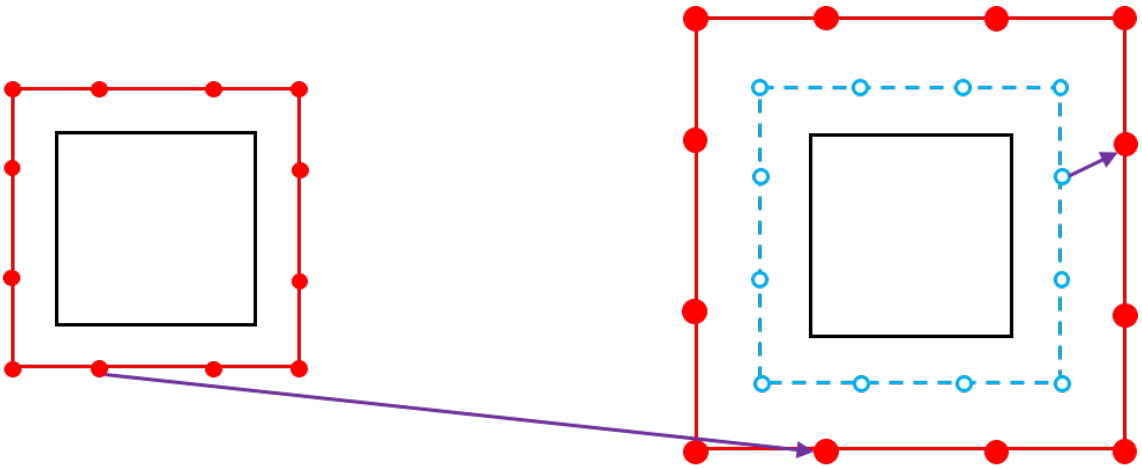

Figure 1.6: Cross section of three dimensional cubic surfaces for the M2L operation. Red solid lines denote equivalent surfaces, and blue checked lines denote check surfaces. The black solid line defines the box B.

In principle, it is now possible to replace the kernel function expansions with the appropriate solutions of the integral equations (1.22), (1.24), (1.25) and (1.26). However all of the above equations are ill-conditioned, as they are ill-posed and in general infinite dimensional problems, and therefore require appropriate regularisation in order to solve. Consider the following general statement of the first-kind Fredholm equation that each of the above operations requires,

$$K\phi = q \tag{1.27}$$

This expression reflects the fact that in practical implementations these continuous integrals are discretised as matrix-vector products. Here K refers to a 'kernel matrix' or operator matrix, which can be found via numerical quadrature, ϕ is a vector of equivalent density to be found, and q is a known vector containing the check potential. The specifics of the surfaces used in PyExaFMM is discussed in greater detail in Chapter 2, Section 2.3. One can use Tikhonov regularisation to write the solution,

$$\phi = (\alpha I + K^* K)^{-1} K^* q \tag{1.28}$$

where α is an appropriate regularisation parameter, and I is the identity matrix. This equation, a second-kind Fredholm equation, can be solved in numerous ways. The authors of [17] make use of a Nyström method, and indicate the possibility of using either Galerkin or Collocation methods. For ease of implementation, a simpler approach is used instead to approximate the solution using a pseudoinverse estimated from a Singular Value Decomposition, or **SVD**. This approach is adapted from current C++ implementations of the KIFMM: ExaFMM-t, as well as PVFMM [19, 24].

Consider the SVD of a matrix A, with m rows and n columns,

$$A = U\Sigma V^* \tag{1.29}$$

where as usual U is the left singular matrix, V is the right singular matrix and Σ is a diagonal matrix whose elements are the singular values of A. We can write an approximate pseudoinverse of A as [27],

$$A^{-1} = V\Sigma^{-1}U^* \tag{1.30}$$

here, A^{-1} has n rows and m columns, and Σ^{-1} is formed by taking the reciprocal of all the diagonal elements. Furthermore, to ensure numerical stability for this reciprocal calculation, one can choose to filter out components of V and U^* if their singular values are smaller than a specified tolerance. PyExaFMM calculates the pseudoinverse of an operator matrix K, by first applying an SVD decomposition using (1.29) and then inverting the diagonal matrix Σ using Tikhonov regularisation similar to (1.28), and finally by applying (1.30). The justification of the choices for the regularisation parameter α and the tolerance for the singular values are discussed in detail in Chapter 2, Section 2.3. In terms of the asymptotic complexity of the SVD, as the singular values and vectors are iterated through once in a straightforward manner to find the pseudoinverse, the complexity of this operation is at most $O(N)$, where N is the number of singular values of A.

The KIFMM algorithm shares almost all of its algorithmic steps with the analytic FMM, the main novelties are the least-squares solves (1.28) to compute the required expansions of each box at each step of the algorithm. In turn these solves require the computation of the check and equivalent surfaces involved, as well as the

numerical quadrature over the equivalent/source densities to compute the required check potentials. However, the the matrix to be inverted in the M2M operation as well as the L2L operation are very similar, which can be seen from figure (1.5). In fact for the kernel function of the model problem of three dimensional electrostatics with a Laplace kernel (1.2), if the upward check surface is chosen to coincide with the downward equivalent surface, and the upward equivalent surface is chosen to coincide with the downward check surface, the matrix to be inverted for the M2M operation is the transpose of the matrix to be inverted for the L2L operation, except for a scaling factor dependent on the level of the Octree the surfaces are created in. This scaling factor is easy to identify for the model problem, kernel matrices evaluated at two adjacent Octree levels, l, are scaled as $K^{l+1} = 2K^l$. Similarly, the required matrix inverse for the L2L and M2L operations exactly coincide except for a scaling factor. Therefore, practical implementations of (1.2) need to compute at most one matrix inversion which can be cached. This can then be scaled and applied as required at each step of the algorithm. Furthermore, for the Laplace kernel, the scaling factors are present on both sides of the equations (1.24) and (1.26), cancelling each other out. This means that the M2M and L2L operations can also be calculated between just one parent box and its respective child boxes, and cached for later use.

The calculation of the pseudoinverse is bound by a term proportional to the number of singular values of the kernel matrix, which in turn is dependent on the number of quadrature points chosen for the check and equivalent surfaces. Therefore as long as this number of quadrature points is kept small, the pseudoinverse does not effect the asymptotic complexity of the KIFMM. The remainder of the M2M, M2L and L2L operations are bound by the complexity of computing a matrix vector product between the calculated inverse matrix and the check potentials, which is resultantly also not impactful on the algorithm's asymptotic complexity as long as the requirement on the number of quadrature points is satisfied. As the remaining algorithmic analysis from Section 1.1 remains the same, we can see approximately how the $O(N)$ complexity is maintained for the KIFMM.

1.2.3 Summary

The different approach of the KIFMM leads to different implementation challenges in comparison to the FMM. The key difficulty arises from the instability in computing the matrix inverse of the kernel matrix between the check and equivalent matrices. As mentioned, choices for the regularisation parameter α as well as the tolerance for singular values taken in the pseudoinverse are discussed with this in mind in Chapter 2, Section 2.3.

The ability to cache the M2M and L2L operators, after calculation for a single parent box and its respective child boxes, leaves the main bottleneck to precomputing all the required operators as the matrix-vector products required to compute the M2L operators. The original authors of the KIFMM accelerate the calculation of the M2L operation using a Fast Fourier Transform, or **FFT** [17]. To see why this can be done, consider the surfaces describing the M2L operation illustrated in figure (1.6), if the downward check surface of the target box, and the upward equivalent surface are chosen to be equivalent and lie on a regular Cartesian grid, then the component of the M2L operation that evaluates the action of a kernel function between these two surfaces can be considered as a convolution between the kernel function and the equivalent density. Mathematically, if the kernel function

$K(x, y)$ depends only the difference between x and y as it does for these choices of surfaces for the M2L operation, the left hand side of (1.25) goes to,

$$\int_{\mathbf{y}^{A,u}} K(\mathbf{x}-\mathbf{y})\phi^{A,u}d\mathbf{y} = q^{B,d} \text{ for any } \mathbf{x} \in \mathbf{x}^{B,d} \tag{1.31}$$

Padding the empty nodes with zeros allows us to apply the Fourier convolution theorem, and solve for equivalent density as,

$$\phi^{A,u}(\mathbf{y}) = \mathcal{F}^{-1}\left[\frac{\mathcal{F}(q^{B,d}(\mathbf{x}))}{\mathcal{F}(K(\mathbf{x}))}\right] \tag{1.32}$$

The currently available major KIFMM implementations use this to accelerate the calculation of the M2L operator including ExaFMM-T and PVFMM [19, 24]. Though reasoned in more detail in Chapter 2, Section 2.3, we state for the present that the expansion order p is by definition proportional to the number of quadrature points used to discretise the surface of the check and equivalent surfaces. Specifically, PyeExaFMM follows the example of ExaFMM-T and PVFMM, taking the relationship between p and number of quadrature points, n_q, to be,

$$n_q = 6(p-1)^2 + 2 \tag{1.33}$$

Meaning that each surface has $O(p^2)$ points, and therefore the M2L operation (1.25) is of $O(p^4)$ between a given target box and source box. This FFT based method to find equivalent densities, accelerates the M2L calculation between a given target box and a given source box to $O(p^3 \log(p))$ [17]. In PyExaFMM we use an alternative acceleration scheme, based on low rank SVD approximations, discussed further in Chapter 2, Section 2.4. Therefore we defer to the literature for further details on FFT based acceleration methods [17, 19].

The KIFMM presents many of the same opportunities for parallel implementation as the FMM. In terms of task-level parallelism, PyExaFMM implements parallel processing, via Python's native multiprocessing tools, for the calculation of the M2L operations. As these calculations involve dense matrix-vector products, they are also ideal candidates to be transferred to a GPU for rapid parallel evaluation in the future. PVFMM implements task-level parallelism for the calculation of the M2M, L2L, and M2L operators with **MPI** as well as an interface with CUDA for processing the dense matrix-vector products. ExaFMM-T makes use of **OpenMP** for a similar purpose.

In summary, the key benefits of the KIFMM lie in the ability to write an implementation that is compatible with a wide class of kernel functions, without evaluating the expansion coefficients. This greatly reduces the complexity of the software design to support multiple applications of the FMM across different problem settings. Furthermore, the formulation of the key FMM operations as dense matrix-vector products makes it easy to map portions of the KIFMM to dedicated parallel hardware such as GPUs. The fact that KIFMM logic is not dependent on the form of the kernel function expansion, and just on kernel function evaluations, makes it trivial to separate the concerns of a software implementation, and write optimisation libraries for the KIFMM independently of the main loop (fig. 1.3) itself. The impact on software design and architecture is signficant, and is explored further in Chapter 2 Section 2.5. This design is the key benefit of PyExaFMM in comparison to existing KIFMM implementations, as PyExaFMM is already sacrificing a degree of performance via the choice of implementation language, we are able to implement

some effective software engineering principles to ease the burden on a user in terms of debugging and expanding upon PyExaFMM.

1.3 Thesis Objectives and Structural Overview

The major contribution of the work underlying this thesis, is a well-tested and extensible Python implementation of the KIFMM algorithm, PyExaFMM. The KIFMM algorithm is chosen for its relative simplicity in comparison to the FMM, and the ability to trivially extend an implementation to study new kernels, and apply the FMM in new problem settings. The decision to use Python as the implementation language arises from the desire to produce an KIFMM implementation that sacrifices some computational performance for developer productivity, via the usage of object oriented design principles that ensure future extensibility, as well as testability.

The remainder of this thesis develops on the mathematical background introduced in Chapter 1. Chapter 2, granularly examines the mathematical and software techniques taken to optimise the implementation of PyExaFMM. Specifically, the introduction of key bottlenecks in Section 2.1, the efficient construction of trees in Section 2.2, the implementation of multiprocessing to distribute and cache the computation of the KIFMM operators in Section 2.3, the low-rank SVD based approximations applied to compress the M2L operators in Section 2.4, and finally the software design choices that were influenced by the desire to make PyExaFMM extensibile and testable, while leaving room to optimise performance in Section 2.5. Chapter 3 begins by benchmarking the performance of PyExaFMM in Section 3.1, on the model problem, in addition to providing some quantitative analysis for the speedup offered by optimisations already implemented by PyExaFMM. Chapter 3 then proceeds to examine optimum parameters for the low-rank SVD approximation of the M2L operation in Section 3.2. We conclude with a discussion of the wider research context of PyExaFMM, as well as avenues for future study in Chapter 4.

Strategy for Practical Implementation

2.1 Bottleneck Overview

Some of the key bottlenecks in implementing the FMM and KIFMM, and potential optimisation strategies in the computation of these algorithms, have already been discussed. This chapter adds detail to the approaches used by PyExaFMM to tackle the issues raised. Namely, the approach to multiprocessing and caching the M2M, L2L and M2L operations, and the acceleration of the M2L calculation via a low-rank approximation using an SVD. The other significant bottlenecks and implementation issues addressed thus far in the implementation of PyExaFMM are the efficient construction of tree data structures, and software design considerations.

PyExaFMM's design philosophy is based on the twin goals of testability and extensibility. The continuous nature of the results of numerical codes makes debugging uniquely challenging, as the nature of the bug could be a numerical error, an error in implementation of algorithm logic, or an error due to the misuse of a programming language or framework. Testability for PyExaFMM refers to the attempt to create stable interfaces around algorithmic logic, return values, and data input and output. Furthermore, as the current implementation represents in many ways the first iteration of the software, we adopt object-oriented design principles to ensure that new optimisations and extensions are easy to implement in the future. Using Python, an **interpreted** and **Object Oriented Language**, also results in a tradeoff in terms of computational performance for developer productivity - which further impacts on design preferences, and pushes PyExaFMM to prefer a small number of simple objects, without relying too much on inheritance as many optimisation tools rely on access to underlying data containers.

As with all optimised numeric codes, PyExaFMM relies on an efficient vectorised data structures, specifically an efficient representation of the tree data structure used in the main FMM loop. This is done via an Morton encoding, which has the effect of translating multidimensional data to one dimension whilst preserving the data's locality. This allows for an efficient vector representation of box centers, at each level of a tree, and for the application of optimised numeric libraries.

2.2 Efficient Tree Implementations

Spatial decompositions of a set of d dimensional coordinates into a single dimension is widely applied across computational science. From forming the basis of implementations of finite-element methods, to algorithms for mesh refinement, and

importantly in our case, N-body simulations [26]. Spatial decompositions utilise space-filling curves, these infinite-curves are fractal, and by increasing their 'order' can be made to 'fill', or describe, a d dimensional space uniquely simply by their position along the length of the curve [4]. This concept is illustrated in figure (2.1) for the Morton encoding, or a 'Z-order' encoding. This figure illustrates the fractal nature of a Morton encoded line, with two 'orders' of line encoding plotted over one another. We see how by using a higher order encoding, we are able to to describe each position in the domain of the box with a greater degree of precision, by using just the length along the encoded line as a coordinate. This variant of space-filling curves is used by PyExaFMM as it better preserves the proximality of data in the encoded coordinates than most other encodings, meaning that distance between data points in physical space is preserved to an extent by the encoding. In N-Body problems that use tree data structures, the encoded line is chosen to pass through the centre of each box in the tree, at each level. Usefully, Morton encodings do not have the requirement to be continuous as illustrated by figure (2.1), making them relatively simple to implement.

As the line is chosen to pass through the center of each box of the quadtree, the position along it can be efficiently represented as a pair of 'binary coordinates' corresponding to the index of the box the line is passing through, as the dimensions of the box at each level are pre-defined for a given octree. For example, the bottom-left box at the coarser level in figure (2.1) will have a binary coordinate of,

$$\underbrace{0}_{x}\underbrace{0}_{y}, \tag{2.1}$$

where x and y correspond to the index of the box along the x and y directions at this level. This encoding can be made unique for each box in each level by adding a displacement corresponding to the number of boxes so far encoded at all coarser levels. For example, the first encoded box, which in figure (2.1) corresponds to the binary coordinate 00 will have a displacement of 0. This leaves this first box with a Morton encoded position, or **key**, of 0 in base ten. Similarly, the encoding for the bottom-left box at the finer level in figure (2.1) will have binary coordinates of,

$$\underbrace{00}_{x}\underbrace{00}_{y}, \tag{2.2}$$

Four bits are needed, two each for the x and y coordinate, as there are 16 boxes at this level. The displacement due to the number of boxes encoded at previous levels is 4 [1], therefore it's Morton key in base ten will be 4. This concept is easily extended to three dimensions, and is illustrated in figure (2.2) at level 1 of an octree. Here the binary coordinates simply contain three bits, for the xyz box indices at his level in the octree, and the displacement mechanism is entirely analogous with the two dimensional case.

The binary coordinates of Morton encoded octrees make it easy to perform simple bitwise shift and comparison operations, to assign source and target particles to the Morton key that encodes the box they lie in for each level of a given octree, as well as traverse the levels of the octree. Specifically, PyExaFMM makes use of a *linear* Morton encoding, meaning that it assigns all particles to the keys of boxes they occupy at a maximum depth which has been refined to the pre-defined leaf level of the desired octree. It does so by assuming that the leaf level of the octree

[1] As is common in computer science, we begin indexing at 0, rather than 1.

describing the domain of interest is partitioned into leaf boxes of equal size, and checking the relative coordinates of each particle, with respect to the coordinates defining the leaf box vertices. Furthermore, this operation only provides a set of keys at the leaf-level which are non-empty, allowing PyExaFMM to ignore the remainder of the space covered by the domain of the octree. Once this linear representation has been found, and the non-empty boxes containing source and target particles identified, it's easy to traverse the octree to find which ancestor boxes particles lie in by simply subtracting the appropriate offset due to the box's position along the entire Morton encoded line, and adding a new offset at at level of the ancestor box. The inverse operation can be done to provide the set of keys corresponding to the descendent boxes for a given box. In terms of the FMM and KIFMM algorithms, the above operations can be extended trivially in simple combination to pre-compute interaction lists for each box, containing only source boxes which aren't empty, as well as to find near neighbours of a given box at a given level of the octree.

The power of encoding the domain of the octree lie in the optimisation opportunities that arise. PyExaFMM works entirely in terms of encoded Morton keys, using them to lookup the correct precomputed operators, and store computational results against. The ability to store an Morton encoding in Numpy vectors makes it possible to use Numba's **JIT** functionality to precompile the bitwise operations, which are used heavily in PyExaFMM's implementation of the KIFMM main loop, mainly to traverse the octree. Properties of JIT compilation are elaborated on in Section 2.5, and the speedup offered by just-in-time compilation is benchmarked in Chapter 3, Section 3.1. Most significantly, this implementation detail represents the first step towards optimising tree construction, with the field of optimising highly-parallel tree constructions for octrees potentially containing 10^9 or more octree boxes, being an active area of research [26, 19]. For example, the authors of PVFMM, one of the major C++ KIFMM implementations, evenly distribute the load of encoding particles for particularly deep octrees across multiple processes using MPI [19]. For highly non-uniform data distributions the non-adaptive implementation of PyExaFMM currently will wastefully refine regions of the octree with a low density of particles, major open source KIFMM implementations thus implement adaptive tree refinement, with different sized leaf boxes in different regions of the domain decsribed by the octree [17, 19, 24].

2.3 Operator Caching and Multiprocessing

Practical implementations of the KIFMM rely on the creation of efficient surfaces used in the calculation of kernel matrices, as well as a stable method for computing the inverse of these kernel matrices. The former problem naturally leads to the implementation of caching codes, used to store, re-use, and scale operator matrices as appropriate. Whereas the latter requires the implementation of a stable method to solve the system of linear equations for each KIFMM operator. As mentioned in Chapter 1, Section 1.2, PyExaFMM uses a pseudoinverse computed via an SVD to solve this linear system. This section provides the implementation details used to address both of these issues.

PyExaFMM uses cubic surfaces for all check and equivalent surfaces discretised to the same degree, this follows the precedent set by the original authors [17], and allows for the application of simple quadrature rules to calculate the various operator integrals such as those in equations (1.24), (1.25) and (1.26). In fact, as the

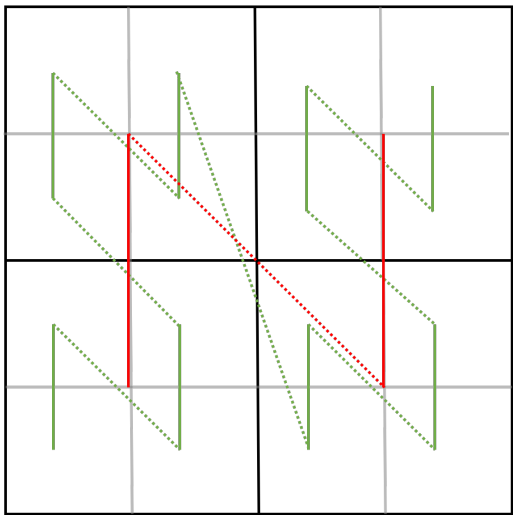

Figure 2.1: Two orders of Morton encoded lines in two dimensions for a quadtree across two adjacent levels. The red line encodes the center of the boxes at the higher level, and and the green line encodes the corresponding points a level deeper. The dashed line indicates the order in which displacements are calculated in the encoding where the line is discontinuous.

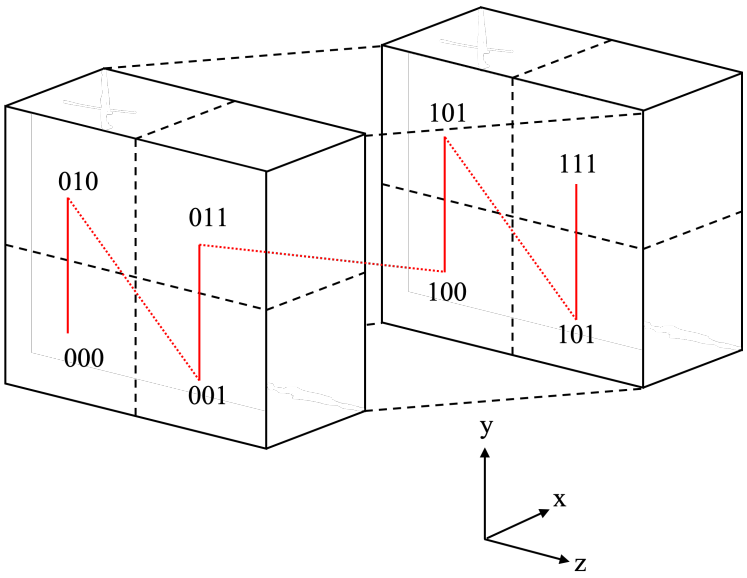

Figure 2.2: Morton encoding at level 1 of domain described by an non-adaptive octree. The two 'layers' of the octree at this level have been broken apart in order to illustrate the encoded line. The Morton keys are written as binary coordinates in each box.

number of quadrature points is generally chosen to be relatively low, PyExaFMM uses a direct computation to compute the required integrals. Specifically, an order p expansion results in,

$$n_q = 6(p-1)^2 + 2 \tag{2.3}$$

where n_q is the number of quadrature points on a given surface. This equation can be justified from the enforcement of quadrature points placed on the corners of the cube, as well as being placed evenly along each edge of the cube. Giving p quadrature points along a given cube edge. Similarly the interior of each face of the

cube is discretised to have quadrature points that form a uniform Cartesian grid with the points at the edges and corners. Therefore, by over-counting the nodes on the edge the 6 sides of the cube lead to $6p^2$ quadrature points, however each corner point is over-counted three times, and the quadrature points in the interior of a edge are over-counted twice. This leads to $6p^2 - 12(p-2) - 16$, which is equivalent to (2.3), however computed with more operations. Therefore even for an order $p = 9$ expansion, which can be employed to compute potentials to 9 digits of accuracy with respect to a direct computation [19], there are just 386 quadrature points on a given check or equivalent surface.

The operators associated with the KIFMM, described by equations (1.24), (1.25) and (1.26) are of the general form,

$$K_A \phi^A = K_B \phi^B \tag{2.4}$$

where K_A and K_B are integral operators describing the action of the kernel between two surfaces describing their given box, and can be discretised into matrices by performing numerical quadrature but storing the element-wise results as matrix elements; and ϕ^A and ϕ^B are equivalent densities, one of which is generally unknown and sought. This can be reformulated as,

$$\phi^A = (K_A)^{-1} K_B \phi^B \tag{2.5}$$

in the case ϕ^B is known and ϕ^A is sought. For the remainder of this thesis, the quantity $(K_A)^{-1}K_B$ will be referred to as an **operator matrix**, for example the 'M2L operator matrix', and so forth. This is a useful quantity to cache, in comparison to just storing the inverse and the kernel matrices separately, as we can just apply the M2L operator matrix to the known source density in order to calculate the unknown source density in a single matrix-vector product.

As described in Chapter 1, Section 1.2, for the model problem (1.2) one can take advantage of the similarities between the nature of the calculation of (2.5) for the M2M and L2L operator matrices, if the check and equivalent surfaces in both cases are chosen correctly. Specifically, if the upward check surface is chosen to coincide with the downward equivalent surface, and the upward equivalent surface is chosen to coincide with the downward check surface, the matrix K_A in for both cases are the transpose of of one another, with the addition of a scaling factor to account for the fact that K_A describes the surfaces at the child level for a the L2L operation, versus at the parent level for the M2M operation. Furthermore as each of these operations is only relative between a parent box and its respective children, one can just compute all M2L and L2L operator matrices for a given parent and its respective children and re-use them at any level of an octree. PyExaFMM therefore computes, and caches, the M2M and L2L operators and saves them alongside the corresponding child Morton key of the child box involved in their calcualtion, this can be looked up as needed during the KIFMM main loop.

For the M2L matrices, an alternative approach is taken, PyExaFMM computes and caches all M2L operator matrices for all source boxes in a given target box's interaction list for all target boxes at all levels. Presently, this computation is distributed across the available processors of multicore machine with Python's native multiprocessing utilities. However, dense matrix-vector product operations, such as those required to find the M2L operator matrices, are good candidates for offloading to GPUs. Modern GPUs consist of up to $O(10^3)$ processor cores at the higher end, in comparison to $O(10)$ on an average desktop work station's **CPU** [16]. Roughly

speaking, this number of cores allows for massive task level parallelism, and works especially well for single-instruction multiple-data (SIMD) type tasks, in which a single instruction is passed to each process alongside a small amount of data on which it is required to operate. A dense matrix vector product is therefore an ideal candidate for GPU acceleration, due to the independence of the calculation between each row of the matrix and the vector, each of which can be broken up again into independent element-wise products. As there are up to 189 M2L operator matrices for a given target box, corresponding to the potential length of a target box's interaction list in three dimensions [17], this problem can be reformulated as a linear system involving the concatenated matrix vector products of M2L operator matrices with their respective source equivalent densities,

$$[A_1|A_2|...|A_I] \begin{bmatrix} \phi_1 \\ \phi_2 \\ ... \\ \phi_I \end{bmatrix} = \begin{bmatrix} q_1 \\ q_2 \\ ... \\ q_I \end{bmatrix} \tag{2.6}$$

where $\{A_i|i \in [1, 2, ..., I]\}$, are the M2L operator matrices for a given source and target box pair and I is the size of a given target box's interaction list. Additionally, $\{\phi_i|i \in [1, 2, ..., I]\}$ and $\{q_i|i \in [1, 2, ..., I]\}$ are the source box's equivalent density and the corresponding check potential vectors respectively. The concatenated operator matrix for this system can be compressed, and still provide accurate results. This is the focus of Section 2.4, where it's shown how PyExaFMM uses a low-rank estimate of this matrix using an SVD. This compression has the potential to offer a large saving in terms of the size of the cached M2L operator, and the time complexity of its application on source densities. Therefore, the decision to compute the M2L operators for all target boxes is taken with the potential for future GPU acceleration and compression techniques in mind.

The values chosen for the size of the for the size of the upward/downward check and equivalent matrices are taken with reference to [19], as the authors empirically find them to achieve good results. Specifically, the radius[2], of the upward check surface and downward equivalent surface of a given box is set to be 2.95 times the radius of the box, and the radius of upward equivalent surface and downward check surface are set to be 1.05 times the radius of the given box. This parameter choice is common to both major C++ implementations of of the KIFMM [24, 19].

As mentioned, the required operator matrices are loaded from disk at run-time, ready to be used by PyExaFMM. Currently, objects corresponding to cached M2L matrices are simply serialised to be loaded in their entirety at run time by PyExaFMM. However, the translation of the cache to HDF5 based storage is an area of future extension, and has been implemented for the storage of the M2M and L2L operator matrices. HDF5 offers an interface for loading slices of large datasets that may be stored on disk, allowing for rapid read times for subsets of a dataset, even if the underlying dataset is of the order of gigabytes [28]. Furthermore, HDF5 allows for the hierarchical storage of numeric arrays, this amounts to creating a database like structure on disk which can store numeric arrays of different dimensions.

The benchmark figures for the potential speedup offered by the above optimisations implemented in PyExaFMM, namely multiprocessing for the M2L operators and the usage of HDF5 in comparison to object serialisation for large datasets are given in Chapter 3, Section 3.1.

[2]It's conventional in FMM literature to refer to the half-side length of a box as the 'radius'

As discussed in Chapter 1, Section 1.2, the calculation of (2.5) is ill-conditioned, therefore PyExaFMM uses a Tikhonov regularisation scheme in combination with an SVD to to calculate the inverse of the operator matrices, which is adapted from the implementation of ExaFMM-T and PVFMM [19, 24].

Consider the SVD of a given operator matrix,

$$K_A = U\Sigma V^*, \tag{2.7}$$

with a pseudoinverse of

$$K_A^{-1} = V\Sigma^{-1}U^*, \tag{2.8}$$

the inversion of the diagonal matrix of singular values can be regularised as,

$$(\alpha I + \Sigma^*\Sigma)^{-1}\Sigma^* =: \Sigma_{\text{reg}}^{-1}, \tag{2.9}$$

giving,

$$K_A^{-1} = V\Sigma_{\text{reg}}^{-1}U^* \tag{2.10}$$

For numerical stability, singular vectors corresponding to 'large' diagonal terms in $(\alpha I + \Sigma^*\Sigma)^{-1}$ are filtered out, if they exceed a specified tolerance. This tolerance value is set using a number close to machine precision, and PyExaFMM uses a value of tolerance $= 1/(4 \cdot \text{EPS} \cdot \max(\alpha I + \Sigma^*\Sigma))$, where EPS is machine precision, and $\max(\alpha I + \Sigma^*\Sigma)$ is the largest entry in $(\alpha I + \Sigma^*\Sigma)$. This has been determined empirically to provide stable results. The regularisation parameter α is also determined empirically, and a value proportional to the largest singular value of K_A, $\alpha = 0.075 \cdot \max(\Sigma)$ is found to provide stable results.[3]

2.4 Low-Rank Matrix Approximations Using SVD

As mentioned in Section 2.3, PyExaFMM compresses a concatenated version of the M2L operator matrices using an SVD. The utility of using an SVD, in comparison to say an eigenvalue decomposition, lies in the fact that it is defined on all matrices, real and complex [27]. From equation (2.6), the concatenated M2L operators can be written as,

$$A = [A_1|A_2|...|A_I]\,, \tag{2.11}$$

where, as before, $\{A_i|i \in [1, 2, ..., I]\}$ are the M2L operator matrices for a given source and target box pair and I is the size of a given target box's interaction list, can be written as a single matrix A, where generally $A \in \mathbb{C}^{T,I\cdot S}$. Here T refers to the number of quadrature points on a given target box's surface, and S refers to the number of quadrature points on all the source boxes in its interaction list's surfaces. For non-square matrices, it is always the case that either the rows or columns (whichever is greater in number), are linearly dependent. Therefore,

$$\text{rank}(A) \leq \min(T, I \cdot S), \tag{2.12}$$

[3]The regularisation parameter α used by the authors of [19] is $\alpha \approx 1 \times 10^{-9}$, which was provided through direct email correspondence. However, they note that this is an empirical value, reflecting the expansion order of the surfaces they used in their simulations.

using this fact, it's therefore possible to write a lower-rank approximation of A. However, an even lower rank approximation is possible using an SVD,

$$A = U\Sigma V^*, \tag{2.13}$$

this can also be rephrased as a weighted sum of rank one matrices, or put simply, written in terms of the products of the left and right singular vectors [27],

$$A = \sum_{j=1}^{r} \sigma_j u_j v_j^*, \tag{2.14}$$

where $r = \text{rank}(A)$, $\{\sigma_j | j \in [1, ..., r]\}$ are the singular values and $\{u_j | j \in [1, ..., r]\}$ and $\{v_j | j \in [1, ..., r]\}$ are the left and right singular vectors respectively.

However, noting that singular values are generally arranged in weakly increasing order in terms of magnitude such that,

$$\sigma_1 \geq \sigma_2 \geq ... \geq \sigma_{r-1} \geq \sigma_r, \tag{2.15}$$

one can say that the τ^{th} partial sum where $\tau \leq r$ captures as much 'energy' of A as possible,

$$A_\tau = \sum_{j=1}^{\tau} \sigma_j u_j v_j^*, \tag{2.16}$$

here the 'energy' of an operator is defined in terms of either a 2-norm or the Frobenius norm as developed in [27]. In fact it can be shown that for any τ with $0 \leq \tau \leq r$, if $\tau = \min\{T, I \cdot S\}$ and we define $\sigma_{\tau+1} = 0$ then,

$$||A - A_\tau||_2 = \sigma_{\tau+1} = 0, \tag{2.17}$$

meaning that A_τ is the best approximation of A, by a matrix of a lower rank [27]. However, one can also cut off the sum (2.16), if the energy of the approximated A_τ is within some acceptable tolerance,

$$||A - A_\tau||_2 \leq \text{tol}, \tag{2.18}$$

such that if the sum is cut off at the k^{th} value, leaving the approximation with $\text{rank}(A_\tau) = k$, where $\sigma_k > \text{tol}$ and $\sigma_{k+1} \leq \text{tol}$. The SVD of the approximation A_τ can then be written as,

$$A_\tau = U_k \Sigma_k V_k^*, \tag{2.19}$$

where U_k and V_k have k rows and columns respectively. As $k < r$, this method is known as a low-rank SVD approximation. The application of (2.19) represents a complexity saving in comparison to applying A directly. This can be seen by considering the full SVD of a generic matrix $B \in \mathbb{C}^{m,n}$,

$$\overset{\leftarrow n \rightarrow}{\underset{m}{\uparrow\downarrow}\begin{pmatrix} \\ B \\ \\ \end{pmatrix}} = \overset{\leftarrow m \rightarrow}{\underset{m}{\uparrow\downarrow}\begin{pmatrix} \\ U \\ \\ \end{pmatrix}} \overset{\leftarrow n \rightarrow}{\underset{m}{\uparrow\downarrow}\begin{pmatrix} \\ \Sigma \\ \\ \end{pmatrix}} \overset{\leftarrow n \rightarrow}{\underset{n}{\uparrow\downarrow}\begin{pmatrix} \\ V^* \\ \\ \end{pmatrix}} \tag{2.20}$$

where the dimension of each matrix in the SVD decomposition has been illustrated. For an approximated matrix B_τ, with $\text{rank}(B_\tau) = k$, we find by applying (2.16),

$$\begin{array}{c} \leftarrow n \rightarrow \\ \uparrow \\ m \\ \downarrow \end{array}\!\!\begin{pmatrix} \\ B_\tau \\ \\ \end{pmatrix} = \begin{array}{c} \leftarrow k \rightarrow \\ \uparrow \\ m \\ \downarrow \end{array}\!\!\begin{pmatrix} \\ U \\ \\ \end{pmatrix} \begin{array}{c} \leftarrow k \rightarrow \\ \uparrow \\ k \\ \downarrow \end{array}\!\!\begin{pmatrix} \\ \Sigma \\ \\ \end{pmatrix} \begin{array}{c} \leftarrow n \rightarrow \\ \uparrow \\ k \\ \downarrow \end{array}\!\!\begin{pmatrix} \\ V^* \\ \\ \end{pmatrix} \tag{2.21}$$

If we were to apply the SVD decomposition of B to a given vector $x \in \mathbb{C}^n$, this would result in an asymptotic complexity of $O(m^2+n^2)$ from (2.20). Compared with the application of the SVD decomposition of B_τ, which results in an asymptotic complexity of $O(k(m+n))$. As long as $k < \min(m, n)$, this represents a complexity saving and therefore faster matrix-vector products with the approximation B_τ. This internal dimension k which represents the amount of compression is often referred to as the target rank. Optimum choices for k for compressing the concatenated M2L operator matrix, A, are explored in Chapter 3 Section 3.2.

Currently, PyExaFMM computes the SVD using the provided function from the SciPy module, which itself calls an optimised **LAPACK** function. However, this implementation is optimised to reduce the number of **FLOPS**, rather than take advantage of advances in modern **heterogenous** multicore machines, and distributed memory programming paradigms which are more often limited by the communication and data transfer overhead between processes, than by number of operations [20]. Halko et. al introduce new 'randomised' approaches for generating low-rank approximations of matrices, which can be optimised for modern heterogenous and distributed computing environments as they consist of processes which can be easily parallelised [20].

The low-rank approximation implemented in PyExaFMM via a truncated SVD (2.19) is costly to compute for large matrices. This is because a full SVD decomposition must be computed, of which all but the first k terms are ignored. We roughly derive the randomised method in application to the SVD below, however guide the reader to the literature for further detail [8, 20]. Consider again the matrix $B \in \mathbb{C}^{m,n}$. Randomised methods consist of two logical steps to find low-rank approximations of B: Step one, construct a low-dimensional subspace that approximates the column space of B, i.e. find an orthonormal matrix $Q \in \mathbb{C}^{m,k}$ such that,

$$B \approx QQ^*B \tag{2.22}$$

where k takes the same meaning as before as the target rank of the compressed matrix. Step two, form a smaller matrix defined $C := Q^*B \in \mathbb{C}^{k,n}$, by which B is restricted to a lower dimensional space spanned by the basis Q.

Step one is 'randomised' by drawing k random vectors $\{\omega_i | \omega_i \in \mathbb{R}^m\}_{i=1}^k$, from a distribution, for example the standard normal distribution, and finding the resulting projections due to B, $y_i = B\omega_i$. In matrix form we can write,

$$Y := B\Omega \tag{2.23}$$

where Ω is a matrix with columns formed formed from ω_i. Probability theory guarantees a high-chance of each y_i being linearly independent [8]. This step provides the opportunity to implement task-level parallelism by distributing the

matrix-vector products over multiple processor cores [20]. The resultant matrix, Y, can be orthonormalised via a QR decomposition to find,

$$Y =: QR \tag{2.24}$$

where Q is the orthonormal basis we desire, and R as usual is an upper triangular matrix. This definition of Q satisfies (2.22). Step two is now computed as,

$$C := Q^* B \tag{2.25}$$

which provides the compressed matrix $C \in \mathbb{C}^{k,n}$. We can now place the SVD into the randomised framework above. Consider a compressed matrix $C \in \mathbb{C}^{k,n}$ obtained via the two steps detailed above. An ordinary deterministic implementation can be used to calculate the the SVD of C cheaply in comparison to the full SVD of B as long as $k \ll n$, such that

$$C = \tilde{U}\Sigma V^* \tag{2.26}$$

here we cheaply obtain the first k right singular vectors of B, from $V \in \mathbb{C}^{n,k}$, as well as the first k singular values $\Sigma \in \mathbb{R}^{k,k}$. To find the entire SVD of B, including the left singular vectors, we notice that,

$$B \approx QQ^*B = QC = Q\tilde{U}\Sigma V^* := U\Sigma V^* \tag{2.27}$$

where we combine (2.22) and (2.25) with the results of the SVD of C (2.26), and define the left singular vectors as $U = Q\tilde{U}$. In terms of PyExaFMM, randomised methods for low-rank approximations of the M2L operator matrices are attractive candidates for future implementation as they represent the state of the art for accelerating the computation of an SVD. As noted in Chapter 1 Section 1.2, PyExaFMM's approach differs to other major KIFMM implementations [19, 24]. The differing approach will therefore make an interesting point of comparison. For the concatenated M2L matrix, the application of the low rank SVD scheme over all source boxes in a given target box's interaction list has an asymptotic complexity of $O(k(T + I \cdot S))$, as S and T are set to be equivalent by PyExaFMM this may be written as $O(k(p^2 + I \cdot p^2))$, where p is the order of the multipole and local expansion where we have used (2.3). This compares favourably with the FFT scheme described in Chapter 1 Section 1.2 as long as $k \ll p^2$, which resulted in an asymptotic complexity of $O(I(p^3 \log(p)))$, or as I is bounded by a constant for three dimensional simulations, $O(p^3 \log(p))$. Furthermore, an open source **apache software foundation** lead software implementation for a randomised SVD based on these methods, designed to take advantage of instruction-level-parallelism, is readily available [1].

2.5 Software Design

Designing easily extensible and testable numerical codes is inherently challenging, the complexity of numerical algorithms and the continuous nature of their results make it hard to separate the logic from application specific implementation details. As PyExaFMM is implemented in Python, an interpreted and inherently Object Oriented Language, we are able to use some common and powerful object oriented design principles to ensure extensibility and testability. However, as most parallel optimisation software relies on access to data containers themselves, rather than an

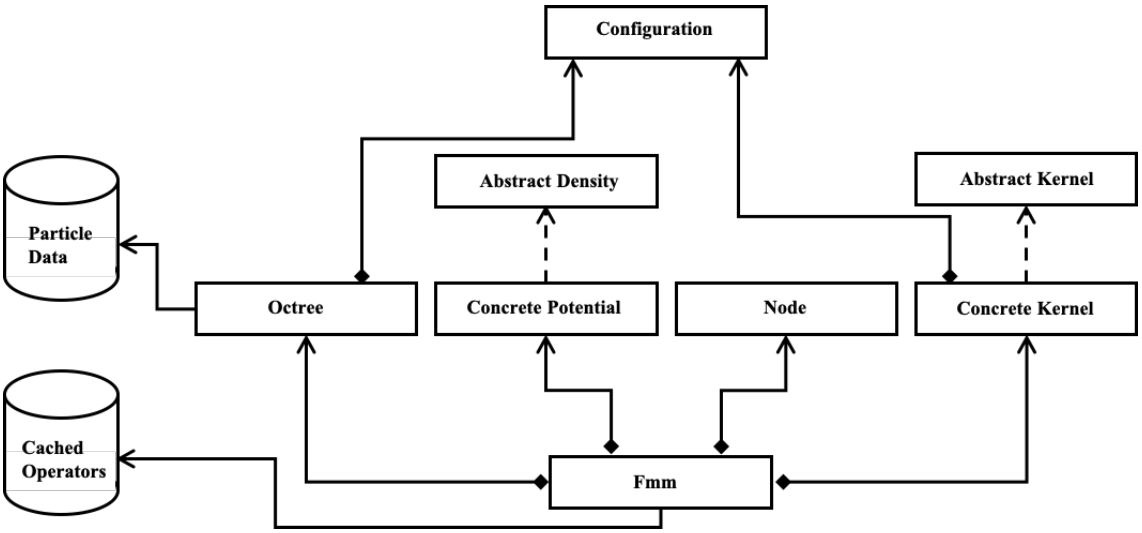

Figure 2.3: Object hierarchy of PyExaFMM. Objects are illustrated with square boxes, and data (either on disk, or in a cache) by a cylinder using standard software engineering convention [10]. Solid lines indicate a dependent relationship, whereas dashed lines indicate an inheritance relationship. A diamond arrow head indicates an object instantiation, a pointed arrow head indicates the direction of a dependency.

abstraction layer between a data object and a client object, the number of object abstractions must also be kept to a minimum.

Concerns about the overhead of using object oriented design principles, in comparison to directly manipulating primitives, such as those available in compiled languages such as C or C++, are misplaced for two reasons. Firstly, the implementation of Python objects is syntactic sugar. Methods and data for a given object are written into a simple dictionary structure at run time, with the 'class' syntax common to other object oriented languages such as C++ or Java, offered as it is familiar to programmers coming from such languages. Furthermore, all 'primitives' such as integers and strings in Python are actually objects by design, being quite different in their construction from C or C++ where the programmer is given the power to allocate memory for true primitives themselves. In Python the overhead therefore comes from the interpreter, which transforms the source code into byte code which is then compiled. This additional interpreter layer is the source of Pythonic overhead, rather than the object abstraction in itself. Therefore, once the decision has been made to use Python, there isn't a significant overhead from using object oriented design principles, with the added benefit of increasing programmer productivity, and increasing testability [23]. Secondly, the vast Python ecosystem of optimised libraries for numerical computation allow for the manipulation and allocation of primitives in memory in a manner closer to a compiled language. Specifically, PyExaFMM implements all of its containers with NumPy, which offers an interface for the allocation and access of numerical data with C-like efficiency, due to the underlying subroutines being written in C. Additionally, PyExaFMM uses just-in-time (JIT) via the Numba library on numeric subroutines implemented with NumPy containers. Just-in-time compilation refers to a system which analyses the byte-code generated by an interpreter for repetitive operations which would benefit from compilation and caching, therefore combining the speed benefits of compiled languages, with the flexibility of interpreted languages.

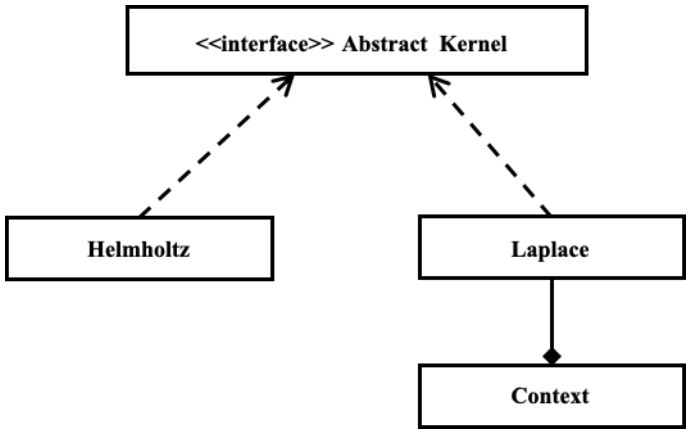

Figure 2.4: Strategy design pattern implemented for the Kernel class. Solid lines indicate a dependent, relationship dashed lines indicate an inheritance relationship. A diamond arrow head indicates an object instantiation, a pointed arrow head indicates the direction of a dependency.

The main object hierarchy used in PyExaFMM is shown in figure (2.3). The first major design principle followed is dependency inversion, whereby components are arranged in a hierarchy such that downstream components, whether they be objects or entire modules, are not dependent on upstream components for any of their logic. Instead, they are coupled together via abstractions in the form of known interfaces [10]. PyExaFMM is architected around the main 'Fmm' object, which is placed downstream, at the top of the object hierarchy. This object contains methods that implement the logic for the upward and downward pass steps of the FMM loop, as well as access methods for the source and target particle data, and the results of the expansion at each box, for each level of the associated octree. In PyExaFMM, a configuration object, specified via a **JSON** file, is instantiated at runtime. This is used to instantiate an Fmm object with its upstream dependencies. Namely, an Octree object is configured, which creates a linear octree as well as the JIT compiled bitwise methods for tree traversal, from user specified source and target particle data. Additionally, a user specified kernel choice from the configuration file is used to instantiate a kernel object. This kernel object is defined using the strategy design pattern [10], illustrated in figure (2.4). This pattern allows for concrete implementations of kernel objects to be separated from the context in which they're used, allowing the user - in this case the Fmm object - to access kernel functions via a uniform interface. Kernel objects also allow for the encapsulation of kernel specific information such as their scaling properties at different levels of an octree. This allows PyExaFMM to be trivially extended to work with new kernel functions, as the main loop relies only on the interface enforced by the abstract kernel object. Furthermore, the precomputed operator matrices are loaded from a cache, and injected into the Fmm object. Crucially, this means that the Fmm object can remain ignorant of the implementation details of how these operators were computed.

The main point to note is that a loose hierarchy of objects allows for the logic of the main FMM loop to be separated from the code it depends on, for example for the creation and partitioning of an octree. Safe interactions between objects are further enforced by interfaces for data containers. Specifically the Abstract Density and Node classes define interfaces for access to concrete Potential Density objects (used to store the potential computed at a target particle as a result of the

FMM algorithm), and the multipole or local expansions for a given box in an octree respectively, as well as their underlying Numpy containers, in a uniform manner. These types of objects are often referred to as return objects in object-oriented design as they enforce the format of the result of a computation, here they help offer guarantees to their client, the main Fmm object, about the dimensions and data types of their respective containers.

The second major design principle followed by PyExaFMM is the separation of concerns, whereby implementation details of distinct components of a program are separated by modules, with known interfaces [10]. Specifically, PyExaFMM separates into independent scripts: the code for operator caching discussed in Section 2.3, and the code for SVD based compression of the M2L operator matrices discussed in Section 2.4. These scripts are accessible via the command-line, through a custom command-line interface, invoked with the command prefix 'ci', created as a part of PyExaFMM package. We defer to the PyExaFMM documentation for more information about the command-line interface, however a basic workflow would start at the command line, creating the M2M L2L and M2L operator matrices, and compressing the M2L matrices,

```
Last login: <Day> <Month> <Date> 00:00:00 on console
user@Workstation ~ % ci compute-operators
user@Workstation ~ % ci compress-m2l
```

These commands are again configured using the JSON configuration file which specifies the parameters used in a simulation, such as the order the multipole and local expansions. Following the execution of the above, PyExaFMM is automatically configured with the required operator and particle data, and simulations can be run from within a Python interpreter or as a script using the following syntax,

```
from fmm.fmm import Fmm

# Instantiate FMM object
fmm = Fmm(config_filename='my_custom_config.json')

# Run Upward Pass
fmm.upward_pass()

# Run Downward Pass
fmm.downward_pass()

# Get Results at Target Points
fmm.result_data
```

At runtime, the instantiation of the Fmm object is preceded by the injection of dependencies from beneath it in the object hierarchy. However, this complexity is hidden from the user, who only specifies simulation parameters and source and target particle data locations via a JSON configuration file, and interacts with simulation outputs via the Fmm object's interface.

The utility of separating functionality to do with operator computation, and compression into separate scripts is that it allows for iterative optimisation without impacting on the main logic, or design, of PyExaFMM. For example, the main Fmm object is in general ignorant of whether multiprocessing is being used in operator

precomputations, or what the regularisation parameter was taken to be for computing pseudoinverses of kernel operator matrices via the SVD has been taken to be, or other specific optimisations outside of the main loop's logic. This is as it should be, as if the Fmm object contained too many implementation details it would obfuscate the logic, making the code harder to debug or write unit tests for. This encapsulates the idea of the separation of concerns.

PyExaFMM takes this further by creating modules for specific operations called by these scripts; specifically it offers its own multiprocessing and SVD utility modules, which offer wrappers around common interfaces from underlying standard Python functionality, specialised for usage in PyExaFMM. This offers an advantage in that their implementation can also be changed, without effecting the logic of the operator computation and compression scripts. For example, the planned extension to randomised SVD methods discussed in Section 2.4 can be separately implemented in PyExaFMM's SVD utility module while maintaining the wrapper's interface, allowing for the new functionality to propagate into the operator compression script without a change of the script's logic. Separation of concerns as a design principle naturally leads to extensible code [10].

Experiments & Results

All experiments are computed on a 2.4 GHz Quad-Core Intel Core i5 processor. Data is stored on an Apple SM0512G SSD hard drive, with benchmarked mean read bandwidths of 1,225 Mb/s and mean write bandwidths of 948 Mb/s [2]. The memory card is a 8GB LPDDR3 with a speed of 2133 MHz. All results are computed in double precision, unless otherwise specified.

3.1 Benchmarking

The major usage of JIT compilation by PyExaFMM is in the construction of the Octree, including the calculation of the interaction lists for target boxes, and to accelerate functions for its traversal. The impact of JIT compilation on tree construction is illustrated in figures (3.1) and (3.2), where octrees constructed with Numpy and Numba together are compared to octrees constructed with just Numpy. Each tree is constructed to a maximum depth of 5 levels over a unit box centered at the origin. The construction time is measured against the numbers of particles N placed in the tree, which are taken to be both the sources and the targets. The construction is repeated five times on each problem size N for statistics, though in both figure (3.1) and (3.2) the error bars, taken from the standard deviations, are generally too small to see. Figure (3.2) illustrates how the multiplicative speedup offered by JIT compilation via Numba has an optimimum problem size N, after which the speedups provided diminish.

Table (3.1) provides timing comparisons for all the octree traversal functions used in PyExaFMM when using Numba and Numpy, or just Numpy alone. JIT compilation relies on a preliminary code analysis, to find optimisations in the interpreted byte code, this can lead to relatively longer runtimes when a JIT compiled function is first invoked, in comparison to subsequent invocations. All the calculations in table (3.1) were run for a Morton key of 100 placed in a unit box centered at the origin, and were repeated one hundred times for statistics. We observe that the preliminary analysis step for some JIT compiled function is so expensive that the mean time for the JIT compiled function is significantly longer than using a function with Numpy alone. Further investigation is needed to understand exactly why the JIT preliminary analysis takes so much longer for some Numpy based functions than others.

The loading and saving speeds of files into/from memory, as a function of file size, using HDF5 in comparison to Python's native object serialisation library, Pickle, are illustrated in figures (3.3) and (3.4) respectively. Each trial was repeated five times with each file size for statistics. The multiplicative speedups for loading and saving using HDF5 are illustrated in figures (3.5) and (3.6) respectively.

We observe that the speedup offered by HDF5 for loading and saving data is

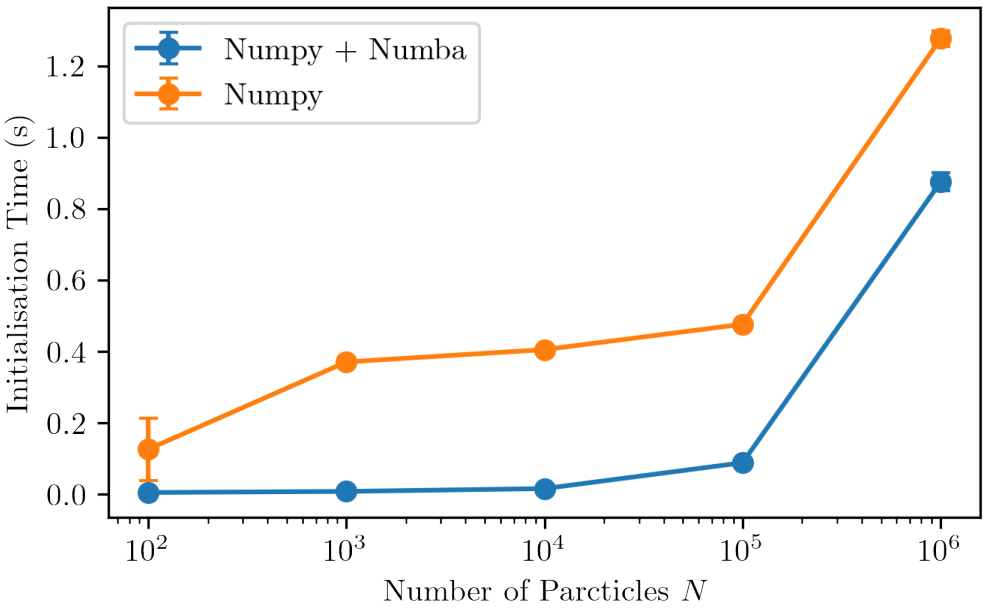

Figure 3.1: Comparison of octree initialisation time versus number of particles N when using Numpy and Numba, to just using Numpy.

approximately constant over file size. The reasons for this are due to the underlying techniques used by each method. Roughly speaking, HDF5 stores raw numeric data alongside a header file describing its dimensions, as a contiguous byte stream on disk which allows for optimised methods to quickly search and retrieve data [6]. Object serialisation is designed to work with arbitrary Python objects, raw numeric data is not stored, rather it is first compressed into an intermediate representation, that results in a byte stream which is then stored on disk. This data is stored alongside the metadata for the object, such as it's initialisation code, and even any dependent objects it may have. The intermediate representation allows any objects to be serialised in the same manner [22].

The effect of multiprocessing on time to calculate the operator matrix precomputations is illustrated in figure (3.7). Here we solve for the operator matrices of the model electrostatic problem in three dimensions (1.2) for a 1000 randomly distributed particles placed in a unit cube. These are taken to be both sources and targets, with unit charges placed on all sources. The problem is discretised using an octree with a maximum depth of 3 levels, and we vary the number of processor cores accessible to the script. Relevant surfaces are calculated to order $p = 2$, and therefore discretised with 8 quadrature points (2.3). The parameters for the relative sizes of different surfaces are the same as those provided in Chapter 2, Section 2.3. The speedup offered by increasing the number of processor cores is non-linear, this is indicative of a communication overhead, and the fact that multiprocessing has been implemented to a very basic degree with the required data being copied to each process, even if very large.

We benchmark the asymptotic run time of PyExaFMM in comparison to direct computation in figure (3.8), with same octree parameters as used for benchmarking the operator precomputation, but with an increasing number of target and source particles N for our model problem (1.2). The calculations are run three times for each problem size for statistics. Assuming a power relationship between run time t and problem size N,

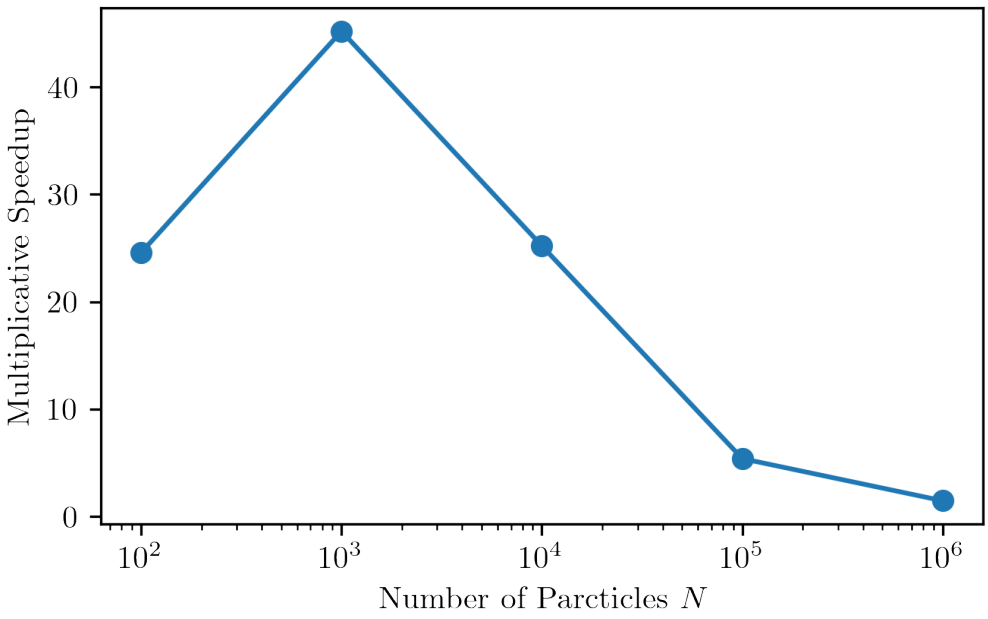

Figure 3.2: Multiplicative speedup offered by using Numba on top of just Numpy for initialising octrees. The analysis used to calculate the errors is provided in Appendix A.3, Section A.3.1.

$$t = N^x, \tag{3.1}$$

where x is unknown, taking logarithms allows one to apply least squares fitting to fit a linear relationship to,

$$\log(t) = x \log(N), \tag{3.2}$$

computing the value of x from the gradient. Performing this calculation for the results from PyExaFMM and direct calculation we report $x = 1.14$ (2 d.p) for PyExaFMM, and $x = 1.98$ (2 d.p) for the direct computation. This implies an approximate asymptotic complexity of $O(N)$, and $O(N^2)$ for PyExaFMM and direct methods respectively in line with what is expected, with the limitation that this has been tested on relatively small inputs of only $O(10^3)$ particles.

We report that a simulation with the above octree and $N = 3000$ takes 17.1 ± 0.2 s, over the three runs with error calculated from standard deviation and reported to 1 s.f. From correspondence with the authors of ExaFMM-T, we can report their corresponding benchmark figure, calculated on a 14 Core Intel i9-7940X CPU with $N = 10^6$ particles randomly distributed in a unit cube, it takes 0.52 s to perform a $p = 4$ degree computation, including tree construction but excluding the operator precomputations, in single precision [24]. Time limitations for this project makes it difficult to perform comparisons using equivalent hardware, however disparities in hardware are unlikely to explain the magnitude of superiority of ExaFMM-t's performance in comparison to PyExaFMM.

ExaFMM-T implements a range of optimisations which represent future avenues of software development for PyExaFMM. Specifically, they distribute the P2P and M2L operations on GPUs, as discussed in Chapter 2, Section 2.3, as well as using OpenMP to share precomputed surfaces across processes in the operator precomputations. PyExaFMM does not distribute any calculations on GPUs, and wastefully

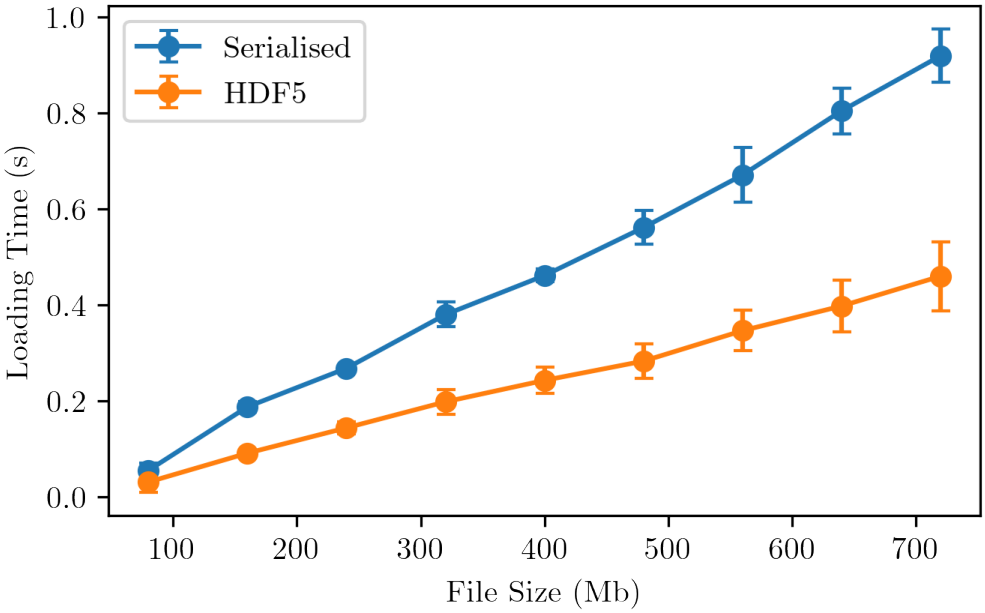

Figure 3.3: Loading times versus File Size comparing object serialisation to HDF5.

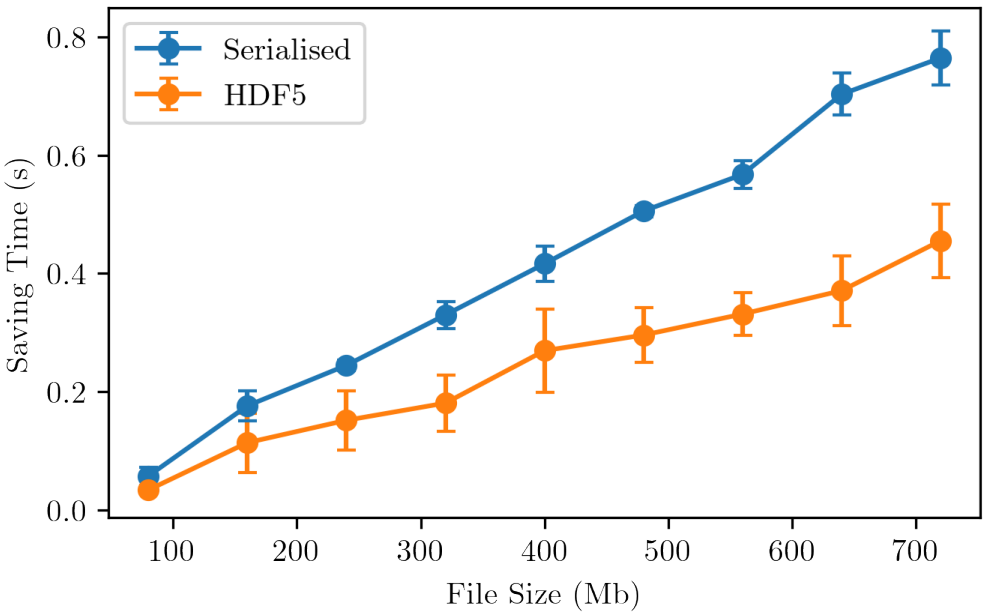

Figure 3.4: Saving times versus File Size comparing object serialisation to HDF5.

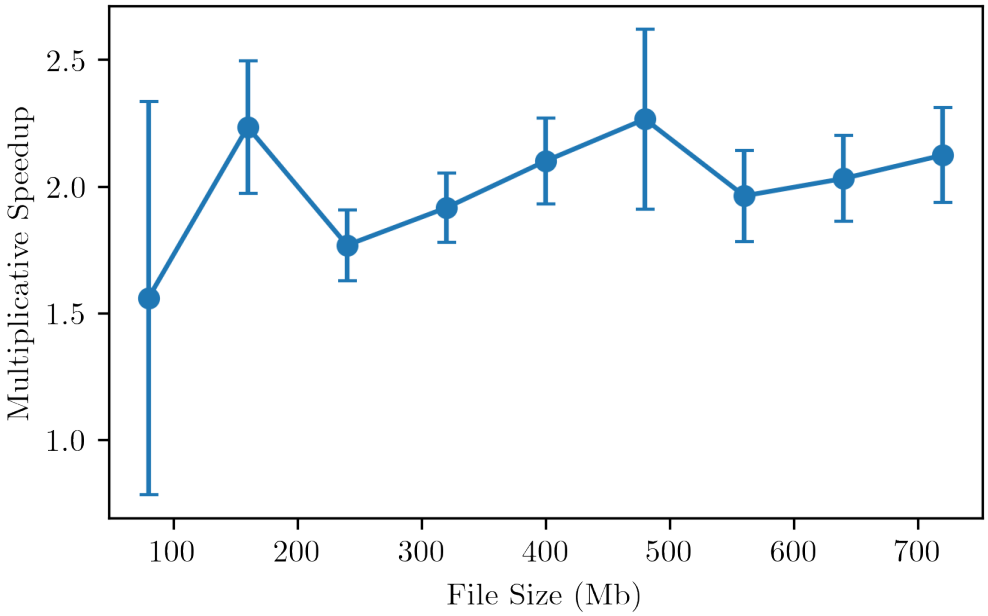

Figure 3.5: The multiplicative speedup offered by HDF5 over object serialisation as a function of file size for loading files. The analysis used to calculate the errors is provided in Appendix A.3, Section A.3.1.

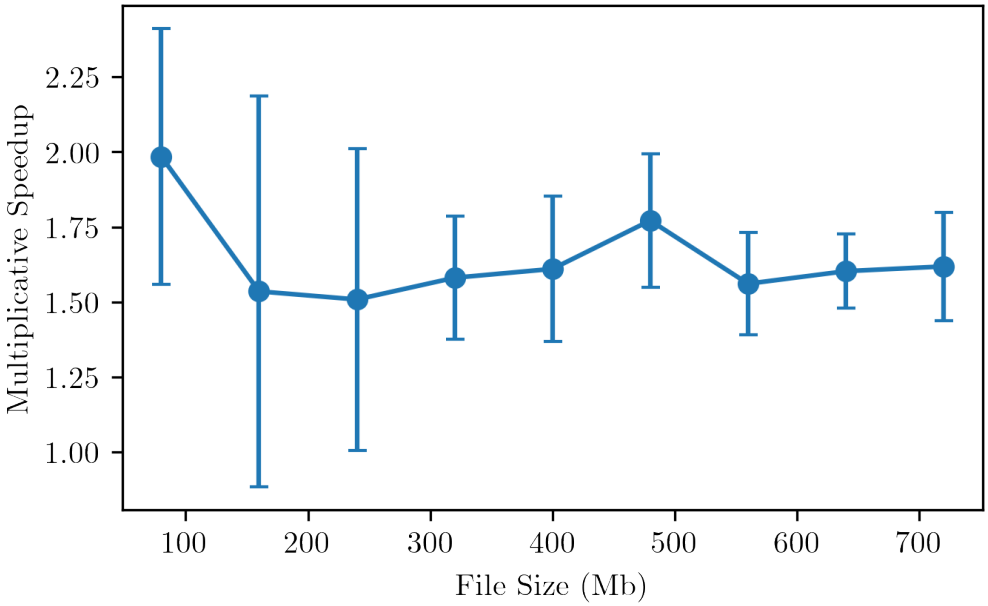

Figure 3.6: The multiplicative speedup offered by HDF5 over object serialisation as a function of file size for saving files. The analysis used to calculate the errors is provided in Appendix A.3, Section A.3.1.

Method	Numpy + Numba	Numpy
Get Level of a Key	280 ns ± 276 ns	1.01 µs ± 265 ns
Get Offset of Key	29.6 µs ± 291 µs	804 ns ± 324 ns
Remove Offset from a Key	30.7 µs ± 301 µs	1.83 µs ± 566 ns
Get Octant of a Key	23.6 µs ± 231 µs	2.16 µs ± 2.71 µs
Get Octant Index From Key	64.4 µs ± 632 µs	10.7 µs ± 3.82 µs
Get Key From Octant Index	36.4 µs ± 356 µs	11.1 µs ± 3.54 µs
Get Octant Index From Cartesian Coordinate	50.4 µs ± 484 µs	17.3 µs ± 21 µs
Get Key From Cartesian Coordinate	2.71 µs ± 1.58 µs	30.4 µs ± 20.3 µs
Get Box Center From Octant Index	49.2 µs ± 476 µs	21.7 µs ± 27.2 µs
Get Box Center From Key	60.5 µs ± 581 µs	22.4 µs ± 7.52 µs
Get Parent Key	29.7 µs ± 292 µs	2.47 µs ± 546 ns
Get Child Keys	37.6 µs ± 361 µs	8.62 µs ± 11.2 µs
Get All Neighbor Keys	98.4 µs ± 803 µs	445 µs ± 80.1 µs
Get All Possible Keys In Interaction List	98.4 µs ± 803 µs	1.67 ms ± 225 µs

Table 3.1: Benchmarking tree traversal functions implemented in PyExaFMM by comparing Numpy functions JIT compiled with Numba, to plain Numpy functions. Errors are taken from the standard deviation over trials. Results and errors are truncated to 3 significant figures.

copies surface data to each process for the operator precomputations. Furthermore, ExaFMM-t optimises the application of the kernel function, firstly by the inverse operation in (1.2) using a Newton iteration to approximate $x = 0.5x(w - ax^2)$ to approximate $x \approx a^{-1/2}$, the computation for which highly-optimised implementations exist [18, 21]. Secondly **AVX** vectorisation is used to apply the kernel function. Roughly speaking, AVX vectorisation allows for parallel execution of SIMD type instructions in a CPU. With similar optimisations, PVFMM is able to report a 3.8 times multiplicative speed up, in comparison to naive applications of the Laplace kernel [19].

3.2 Optimum Target Rank

Using a low-rank SVD approximation for the M2L operator matrix naturally leads to the question of what the optimum choice for target rank for k is (see Chapter 2, Section 2.4). Naturally, we want to minimise k to reduce the asymptotic complexity of the application of the compressed M2L operator. However, it's clear that if k is taken to be too low, we will be losing potentially important data from the original uncompressed M2L matrix. However, if k is too high we will be keeping redundant singular vectors which do not contribute significantly to the final result for potential.

For this experiment we use a 3 level octree containing 1000 randomly distributed particles placed in a unit cube, these are taken to be sources as well as targets, and unit charges have been placed on the sources. We compute $p = 3$ degree multipole and local expansions therefore discretising each surface with 26 quadrature points. Our metric of accuracy is the mean percentage error of the PyExaFMM result, with respect to the direct result. This mean is taken across errors for all target particles. For the above tree topology and expansion order p, we find a a mean percentage error of 1.96 % (2 d.p) when the M2L matrix is left with full-rank. This experiment

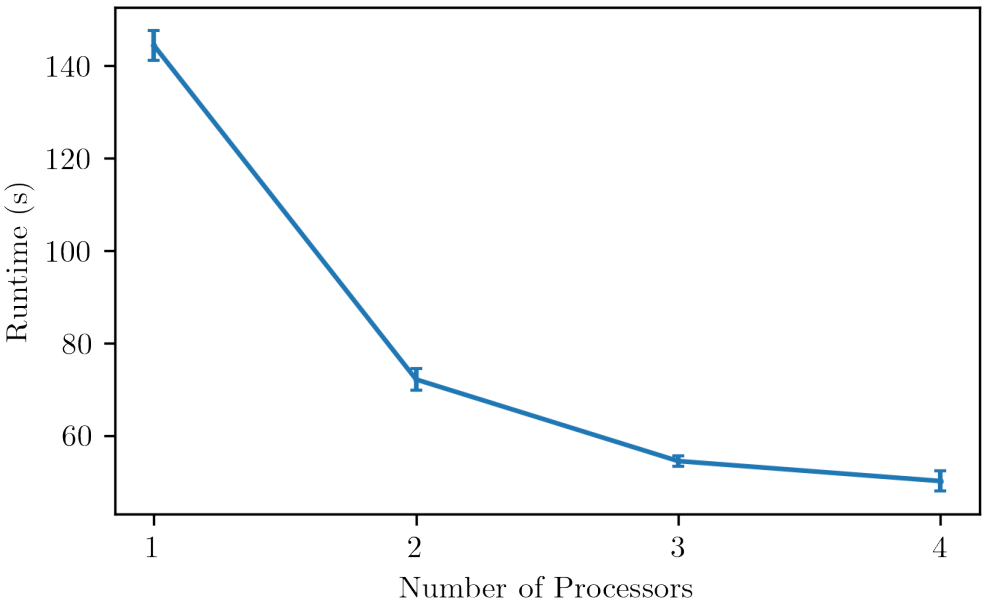

Figure 3.7: Runtime versus number of processors used for operator precomputations for the model problem (1.2). Error bars are plotted using standard deviation.

is repeated for different low-rank approximations, the results of which are illustrated in figure (3.9).

Figure (3.9) indicates that singular vectors corresponding to rank greater than 6 are actually adding noise to the results of PyExaFMM, with an optimum target rank k in the range 4-6 for the above tree topology. Currently PyExaFMM does not attempt to find an optimum rank for a given tree topology. however this represents a future body of work. Furthermore we see that even rank a one approximation leads to a marginal increase in error. Specifically a rank one approximation leads to a mean percentage error of 1.99 (2 d.p), compared with 1.96 (2 d.p) for the mean percentage error with the full-rank M2L matrix. Future work should investigate whether a bound for the error of a rank one approximation of the M2L operator matrix, and the mean percentage error exists. PyExaFMM could adopt this bound into the software implementation, to offer the user a trade-off between speed and accuracy due to the compression of the M2L operator matrix.

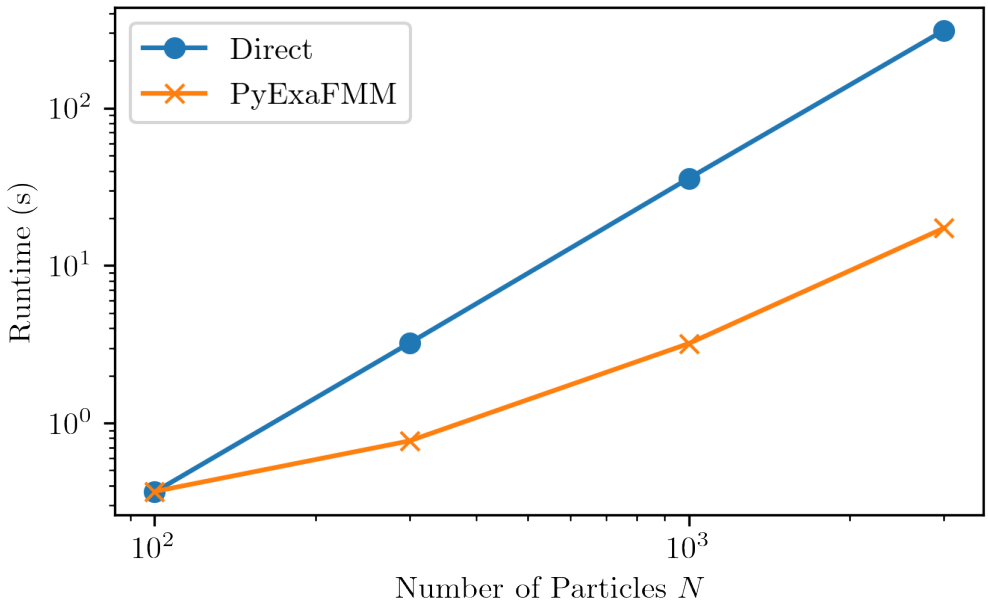

Figure 3.8: Runtime versus problem size to solve the model problem (1.2) using direct calculations and PyExaFMM.

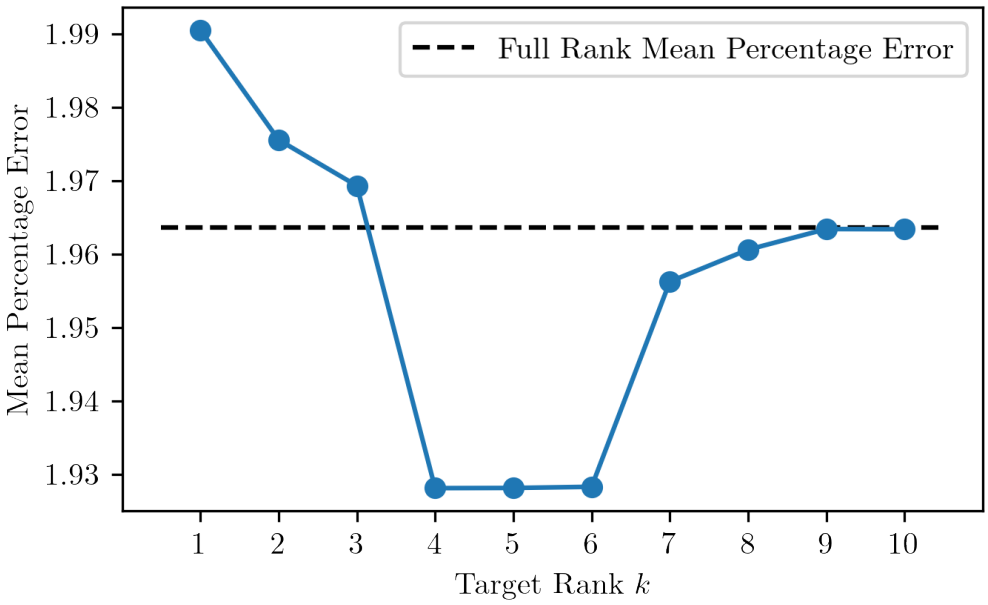

Figure 3.9: Mean percentage error of PyExaFMM with respect to direct computation, where the mean is taken across all particles in the simulation. The mean percentage error for the full-rank simulation is illustrated with a dashed horizontal line.

Conclusion

In summary, the main contribution of the work underlying this thesis is PyExaFMM; a three dimensional KIFMM simulation library with some parallel features. The software has been designed to be well testable and extensible, however it is currently someway behind state of the art implementations in terms of speed [19, 24]. Furthermore, from table (3.1), we see that some current optimisations, namely JIT compilation for some Numpy based functions, have been naively applied. To bring PyExaFMM in line with state of the art KIFMM software, extensions which fully take advantage of modern computing hardware will have to be implemented. Modern heterogenous computers have access to multiple multi-core CPU and GPU units, with multiple levels of memory-cache, and vectorisation available at the processor level [19]. The current optimisations within PyExaFMM do not take advantage of shared or distributed memory parallelism. As mentioned in Chapter 1, Section 1.1, the near-field P2P calculations can be transferred to GPUs for acceleration [16], alternatively as discussed above fast Newton iterations and AVX vectorisation at the CPU level is used to accelerate the P2P calculations in both major KIFMM implementations [19, 24]. These optimisations are dependent on the available hardware, however they share a common approach in distributing a single P2P instruction, across multiple particle interactions, following the SIMD paradigm. The calculation of the far-field M2L operator matrices can also be accelerated by the above techniques. Furthermore, randomised SVD compression [8, 20] can be implemented to also take advantage of shared memory parallelism as described in Chapter 2, Section 2.4, reducing further the cost of computing low-rank approximations of the M2L operator matrices. The mixed performance of JIT compilation for the construction of trees, leaves a lot of room for for further optimisation in PyExaFMM. Specifically, state of the art implementations construct adaptive trees in parallel, taking advantage of the distributed memory programming paradigm with MPI [19]. PVFMM, for example, chunks particle data across processors, constructing subtrees in parallel, and uses MPI to pass multipole and local expansion coefficients to other processes as required during the main FMM loop. In addition to the above optimisations, the current PyExaFMM codebase can be further sanitised. Specifically, integration tests that test the way in which modules interact, should be added on top of the current unit test suite. Additionally, it should be a priority to perform more detailed code profiling to clearly identify the most significant memory and CPU bottlenecks.

Despite its limitations, PyExaFMM achieves the complexity bound of the FMM algorithm. Furthermore, it represents a significant first step towards the goal of an open source Python KIFMM implementation which sacrifices as little computational performance as possible, in comparison to major compiled language implementations.

Appendix

A.1 FMM Algorithm Specification

This pseudo-code is adapted from [12].

Initialisation: Choose a level of refinement of $n \approx \log_8 N$ and precision ϵ, set $p = \lceil -\log_2(\epsilon) \rceil$[1].

Upward Pass

Step 1

Form multipole expansions of potential field due to particles in each box at leaf level.

do $ibox = 1, .., 8^l$
Form p^{th} degree multipole expansion for each leaf box.
end

Step 2

Translate Multipole expansion to coarser levels from the bottom up.

do $l = n - 1, .., 0$
do $ibox = 1, .., 8^n$
Perform M2M operations.
end
end

Downward Pass

Computations at the coarsest possible level. For a given box, done by including interactions with those boxes which are well separated, and whose interactions have not been accounted for at the parent level.

Step 3

Form local expansion about center of each box at each level $l \leq n - 1$, describes field due to all particles that are not contained in the current box, it's near neighbours or it's secondary near neighbors.

[1]Different authors have different suggestions for p, the authors of [17] use $p = \log_c \epsilon$ with $c = \frac{4-\sqrt{3}}{\sqrt{3}}$

do $l = 1, ..., n-1$
  do $ibox = 1, ..., 8^l$
    Perform M2L translations.
  end
  do $1, ..., 8^l$
    Perform L2L operations.
  end
end

Step 4

After this step, local expansions are available at the leaf level. One can use this to evaluate potential at leaves from all particles in the far field.

do $ibox = 1, ..., 8^n$
  Find local expansion at leaf level, by doing M2L from interaction list.
end

Step 5

Evaluate local expansions at particle positions in all leaves

do $ibox = 1, ..., 8^n$
  For every particle in *ibox*'th box, evaluate local expansion.
end

Step 6

Compute nearest neighbors directly,
do $ibox = 1, ..., 8^n$
  For every particle in *ibox*'th box, compute potential directly with nearest neighbors.
end

Step 7

do $ibox = 1, ..., 8^n$
  Add direct and far field terms together for every particle in the *ibox*
end

A.2 Analytic FMM Operators for 3D Laplace Kernel

The expressions presented here are first derived in [12]

For l charges of strengths $q_1, ..., q_l$ located inside sphere D of radius a center at $Q = (\rho, \alpha, \beta)$, and that for points $P = (r, \theta, \phi)$ outside D potential given by the following **Multipole Expansion**,

$$\Phi(P) = \sum_{n=0}^{\infty} \sum_{m=-n}^{n} \frac{O_n^m}{r'^{n+1}} \cdot Y_n^m(\theta', \phi')) \tag{A.1}$$

Where the points P and Q are defined such that $P - Q = (r', \theta', \phi')$. Then for any point P outside sphere D_1 of radius $a + \rho$, The multipole expansion can shifted with the following **M2M operation**,

$$\Phi(P) = \sum_{j=0}^{\infty} \sum_{k=-j}^{j} \frac{M_j^k}{r^{j+1}} \cdot Y_j^k(\theta, \phi) \tag{A.2}$$

Where,

$$M_j^k = \sum_{n=0}^{j} \sum_{m=-n}^{n} \frac{O_{j-n}^{k-m} \cdot J_m^{k-m} \cdot A_n^m \cdot A_{j-n}^{k-m} \cdot \rho^n \cdot Y_n^{-m}(\alpha, \beta)}{A_j^k} \tag{A.3}$$

and,

$$J_m^{m'} = \begin{cases} (-1)^{\min(|m'|,|m|)} & \text{if } m \cdot m' < 0 \\ 1 & \text{otherwise} \end{cases} \tag{A.4}$$

$$A_n^m = \frac{(-1)^n}{\sqrt{(n-m)! \cdot (n+1)!}} \tag{A.5}$$

For l charges of strengths $q_1, ..., q_l$ located inside sphere D_Q of radius a center at $Q = (\rho, \alpha, \beta)$, and that $\rho > (c+1)a$ with $c > 1$. The corresponding multipole expansion, converges inside sphere D_0 of radius a centered at origin. Inside D_0 the potential due to charges has the local expansion,

$$\Phi(P) = \sum_{j=0}^{\infty} \sum_{k=-j}^{j} L_j^k \cdot Y_j^k(\theta, \phi) \cdot r^j \tag{A.6}$$

where,

$$L_j^k = \sum_{n=0}^{\infty} \sum_{m=-n}^{n} \frac{O_n^m \cdot J_k^m \cdot A_n^m \cdot A_j^k \cdot Y_{j+n}^{m-k}(\alpha, \beta)}{A_{j+n}^{m-k} \cdot \rho^{j+n+1}} \tag{A.7}$$

where A_n^m same as above,
but,

$$J_m^{m'} = \begin{cases} (-1)^{n'}(-1)^{\min(|m'|,|m|)} & \text{if } m \cdot m' < 0 \\ (-1)^{n'} & \text{otherwise} \end{cases} \tag{A.8}$$

n' refers to j, m' refers to k

and that for points $P = (r, \theta, \phi)$ outside D potential given by multipole expansion. This is also known as the multipole to local, or **M2L Operation**.

Let $Q = (\rho, \alpha, \beta)$ be the origin of a local expansion,

$$\Phi(P) = \sum_{n=0}^{p} \sum_{m=-n}^{n} O_n^m \cdot Y_n^m(\theta', \phi') \cdot r'^n \tag{A.9}$$

Where $P = (r, \theta, \phi)$ and $P - Q = (r', \theta', \phi')$.

$$\Phi(P) = \sum_{j=0}^{p} \sum_{k=-j}^{j} L_j^k \cdot Y_j^k(\theta, \phi) \cdot r^j \tag{A.10}$$

where,

$$L_j^k = \sum_{n=j}^{p} \sum_{m=-n}^{n} \frac{O_n^m \cdot J_{n-j,m-k}^m \cdot A_{n-j}^{m-k} \cdot A_j^k \cdot Y_{n-j}^{m-k}(\alpha, \beta) \cdot \rho^{n-j}}{A_j^k} \tag{A.11}$$

where A_n^m same as above,

$$J_{n,m}^{m'} = \begin{cases} (-1)^n(-1)^m & \text{if } m \cdot m' < 0 \\ (-1)^n(-1)^{m'-m} & \text{if } m \cdot m' > 0 \text{ and } |m'| < |m| \\ (-1)^n & \text{otherwise} \end{cases} \tag{A.12}$$

This operation defines the local to local, or **L2L Operation**.

A.3 Error Calculations

A.3.1 Speedup Error Propagation

The speedup offered by Numba JIT compilation in comparison to just using Numpy is calculated using,

$$\text{Speedup} = \frac{t_{\text{numpy}}}{t_{\text{JIT}}} \tag{A.13}$$

where t_{JIT} and t_{numpy} are the mean times taken to complete a solve a given problem for each method respectively.

The error is then propagated using the standard formula [15],

$$\frac{\sigma_{\text{speedup}}}{\text{Speedup}} = \sqrt{\left(\frac{\sigma_{\text{JIT}}}{t_{\text{JIT}}}\right)^2 + \left(\frac{\sigma_{\text{numpy}}}{t_{\text{numpy}}}\right)^2} \tag{A.14}$$

where σ_{JIT} and σ_{numpy} are the standard deviations from the runs for each problem size.

Glossary

apache Open source software foundation that supports core open source projects across a diverse range of problem areas. . 28

AVX Advanced Vector Extensions, a 256-bit or 512-bit vector register for Intel Sandy Bridge **CPU** architectures. . 38, 41

check potential The potential calculated from source particles, or an equivalent density distribution at a check surface. . 12, 13

check surface Defines the surface at which the potential caused by source points, and the equivalent density formulation coincide. . 12, 13, 16

CPU Central Processing Unit. 23, 35, 38, 41

CUDA 'Compute Unified Device Architecture', is a parallel computing API designed for the development of programs for **GPU**s . 11, 17

data-level parallelism In a multiprocessor system, data level parallelism is achieved by distributing data amongst different compute nodes to be executed upon in parallel. See **CUDA** . 11

equivalent density An equivalent representation of the potential generated by a discrete/continuous distribution. This equivalent density is supported at discrete points on an **equivalent surface**. . 12, 13

equivalent surface An equivalent surface supports equivalent density points. . 12, 13, 16

far field An set of particles in the far field are considered to be far away enough from the particle of interest that they are suitably described by a convergent multipole expansion. . 5, 9, 12, 13

FFT Fast Fourier Transform. . 16, 17, 28

FLOPS Floating Point Operations per Second, as a measure of computer performance. . 27

FMM The Fast Multipole Method. . 2, 4, 5, 11, 15–19, 21, 30, 41

GPU 'Graphics Processing Unit', are specialised processors specifically designed for rapid parallel execution across large blocks of data. Generally consist of $O(10^3)$ cores, and are suited to handling **SIMD** type instructions, such as matrix vector products. . 11, 17, 23, 35, 41

heterogenous In reference to the availability of different kinds of compute units on a single machine, typically in reference to machines containing both CPUs and GPUs. . 27

high level interpreted language A High-Level language is one that offers an API and primitive data structures that strongly abstract from the details of the computer on which it is being run. An interpreted language is one in which codes are run directly via an *interpreter*, rather than first being compiled. In reality interpreted languages run in a 'virtual machine', which takes input code, transforms it to byte-code which is then translated into machine level instructions. This approach allows for greater portability of code, and developer productivity, at the expense of space and time overhead in running software. . 11

instruction-level-parallelism Instruction-level parallelism refers to the parallel execution of different instructions on a specific thread or process in a multi-processing system. . 28

interaction list Boxes which are the children of the near-neighbours of the a box's parent box, but are not adjacent to the box itself. . 9, 11, 21

interpreted In reference to an interpreted programming language. Here instead of direct compilation of source code into machine level instructions, source code is passed to an 'interpreter', which compiles a byte-code on the programmers behalf. This byte-code is sent to a compiler as it arrives, giving the programmer the illusion of interactive programming. . 19, 28

JIT Just-In-Time compilation 21, 29, 30, 33, 38, 41

JSON JavaScript Object Notation. 30, 31

key In the context of space filling curves, refers to the encoded position along the length of a curve. . 20, 23

KIFMM The 'Kernel Independent' Fast Multipole Method. . 2, 12, 15–19, 21, 23, 24, 28, 41

L2L Translate from local of parent box to local expansion of child box.. 9, 12–14, 16, 19, 23, 24, 31

L2P Evaluate local of leaf box at each target particle in a leaf box.. 9

LAPACK 'Linear Algebra Package', standard library for optimised numerous numerical solvers, including for eigenvalue problems, singular value decompositions, and linear least square problems. . 27

M2L Translate from multipole of box A to local expansion of box B.. 9, 12–14, 16–19, 23–25, 27, 28, 31, 35, 38, 39, 41

M2M Translate from multipole of child box to multipole expansion of parent box.. 9, 12–14, 16, 19, 23, 24, 31

MPI Message Passing Interface for implementing distributed memory parallelism. Each process is given it's own local memory, and can communicate and pass data and computational results to other processes being run in parallel. . 17, 21, 41

near field An set of particles in the near-field are considered to violate the the criteria for describing them with a convergence multipole expansion. . 8–10, 12

near neighbours Two boxes in computational tree are near neighbours if they are at the same level of refinement, and share a boundary point. . 9, 21, 42

Object Oriented Language Programming paradigm in which data and methods are abstracted into 'objects'. This allows for the separation of data and logic into structures that encapsulate their logical grouping. . 19, 28

OpenMP Open MultiProcessing is an API for implementing shared-memory parallelism. Each process is run on a separate thread, but a global memory space is shared. . 17, 35

operator matrix In the context of the cached **KIFMM** operators, refers to the matrix applied to a known source density in order to calculate an unknown source density. . 23

P2M Particle to multipole expasion operation.. 9, 13

P2P Evaluate particle to particle interactions directly. . 35, 41

PDE Partial Differential Equation. . 12

post-order In reference to the traversal of heirarchical trees, post-order traversal is moving from the finest (leaf) level to the coarsest (root) level. . 8

pre-order In reference to the traversal of heirarchical trees, pre-order traversal is moving from the coarsest (root) level to the finest (leaf) level. . 9

PyExaFMM The Python implementation of the **KIFMM** presented as a part of this thesis. . 2, 15, 17–25, 27–35, 38–41

SIMD Single Instruction, Multiple Data. . 24, 38, 41

source densities Source densities are associated with their respective **source particles**. In electromagnetic problems, these would correspond to charges. In gravitational problems, these would correspond to mass. . 12

source particles Particles are described as sources if they are consifered to contribute to the source field being evaluated at the target particles. They may or may not refer to the same set of particles as the target particles, for example in the classic N-Body problem, the target particles and the source particles are the same set. . 8, 9, 12, 13

SVD Singular Value Decomposition. . 15, 19, 24–28, 32, 38, 41

target particles Particles are described as targets if the potential due to a source field is being evaluated at these particle's positions, but they themselves are not being considered as a source for the field being evaluated. They may or may not refer to the same set of particles as the source particles, for example in the classic N-Body problem, the target particles and the source particles are the same set. . 9, 12, 20, 21

task-level parallelism Task-level parallelism is achieved when multiple threads or processes, running the same, or differing, code, in a multiprocessing system are executed with the same, or differing, data. . 11, 17, 27

well separated Two boxes in computational tree are near neighbours if they are at the same level of refinement, and are not near neighbours. . 42

Bibliography

[1] *Apache Mahout - Distributed Linear Algebra.* URL: https://mahout.apache.org/.

[2] *Apple SM0512G PCIe 500GB SSD Benchmarking.* URL: https://ssd.userbenchmark.com/SpeedTest/25611/APPLE-SSD-SM0512G.

[3] John A. Board et al. "Accelerated molecular dynamics simulation with the parallel fast multipole algorithm". In: *Chemical Physics Letters* 198.1 (1992), pp. 89 –94. ISSN: 0009-2614. DOI: https://doi.org/10.1016/0009-2614(92)90053-P. URL: http://www.sciencedirect.com/science/article/pii/000926149290053P.

[4] Paul M. Campbell et al. *Dynamic Octree Load Balancing Using Space-Filling Curves.* Tech. rep. CS-03-01. Williams College Department of Computer Science, 2003.

[5] Barry A. Cipra. "The Best of the 20th Century: Editors Name Top 10 Algorithms". In: *SIAM News* 33.4 (2000).

[6] A. Collette. *Python and HDF5.* O'Reilly Media, Incorporated, 2013. ISBN: 9781449367831. URL: https://books.google.co.uk/books?id=a_yXngEACAAJ.

[7] Michael A Epton and Benjamin Dembart. "Multipole translation theory for the three-dimensional Laplace and Helmholtz equations". In: *SIAM Journal on Scientific Computing* 16.4 (1995), pp. 865–897.

[8] N. Benjamin Erichson et al. "Randomized Matrix Decompositions Using R". In: *Journal of Statistical Software* 89 (June 2019). DOI: 10.18637/jss.v089.i11.

[9] Frank Ethridge and Leslie Greengard. "A New Fast-Multipole Accelerated Poisson Solver in Two Dimensions". In: *SIAM J. Sci. Comput.* 23.3 (Mar. 2001), 741–760. ISSN: 1064-8275. DOI: 10.1137/S1064827500369967. URL: https://doi.org/10.1137/S1064827500369967.

[10] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software.* 1st ed. Addison-Wesley Professional, 1994. ISBN: 0201633612.

[11] L Greengard and V Rokhlin. "A fast algorithm for particle simulations". In: *Journal of Computational Physics* 73.2 (1987), pp. 325 –348. ISSN: 0021-9991. DOI: https://doi.org/10.1016/0021-9991(87)90140-9. URL: http://www.sciencedirect.com/science/article/pii/0021999187901409.

[12] Leslie Greengard. "The Rapid Evaluation of Potential Fields in Particle Systems". PhD thesis. Yale University, 1987.

[13] Leslie Greengard and June Yub Lee. "A direct adaptive poisson solver of arbitrary order accuracy". English (US). In: *Journal of Computational Physics* 125.2 (May 1996), pp. 415–424. ISSN: 0021-9991. DOI: 10.1006/jcph.1996.0103.

[14] David J. Griffiths. *Introduction to Electrodynamics*. 4th ed. Cambridge University Press, 2017. DOI: 10.1017/9781108333511.

[15] I. Hughes and T. Hase. *Measurements and Their Uncertainties: A Practical Guide to Modern Error Analysis*. OUP Oxford, 2010. ISBN: 9780199566327. URL: https://books.google.co.uk/books?id=AbEVDAAAQBAJ.

[16] Wen-Mei W. Hwu. *GPU Computing Gems Emerald Edition*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN: 0123849888.

[17] 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.

[18] Chris Lomont. *Fast Inverse Square Root*. Purdue University, 2003.

[19] Dhairya Malhotra and George Biros. "PVFMM: A Parallel Kernel Independent FMM for Particle and Volume Potentials". In: *Communications in Computational Physics* 18.3 (2015), 808–830. DOI: 10.4208/cicp.020215.150515sw.

[20] Per-Gunnar Martinsson Nathan Halko and Joel Tropp. "Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions". In: *SIAM Review* 53 (Jan. 2011), pp. 217–288. DOI: 10.1137/090771806.

[21] *Origin of Quake3's Fast InvSqrt()*. URL: https://www.beyond3d.com/content/articles/8/.

[22] *Pickle - Python object serialization*. URL: https://docs.python.org/3/library/pickle.html.

[23] Luciano Ramalho. *Fluent Python*. 1st. O'Reilly Media, Inc., 2015. ISBN: 1491946008.

[24] Lorena A. Barba Rio Yokota. *ExaFMM User's Manual*. 2011. URL: http://www.bu.edu/exafmm/files/2011/06/ExaFMM-UserManual1.pdf.

[25] Vladimir Rokhlin. "Rapid Solution of Integral Equations of Theory in Two Dimensions". In: *Journal of Computational Physics* 86.2 (Feb. 1990), pp. 414–439. DOI: 10.1016/0021-9991(90)90107-C.

[26] 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.

[27] Lloyd N. Trefethen and David Bau. *Numerical Linear Algebra*. SIAM, 1997. ISBN: 0898713617.

[28] Leah A. Wasser. *Hierarchical Data Formats - What is HDF5?* URL: https://www.neonscience.org/about-hdf5.