Modern Research Software For Fast Multipole Methods

Srinath Kailasa

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

Department of Mathematics

University College London

September, 2024

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

The completion of this thesis simply wouldn't have been possible without the strong prevailing wind of emotional support from my family the regularity of fun with my friends, and the sympathetic and dedicated teachers and colleagues I met at UCL and across the world. Thank you *all* for giving me this opportunity, I'm excited for what the future brings.

ఇది మా అమ్మ కోసం రాశాను

UCL Research Paper Declaration Form

Published Manuscripts

1. PyExaFMM: an exercise in designing high-performance software with Python and Numba.
    a) DOI: 10.1109/MCSE.2023.3258288
    b) Journal: Computing in Science & Engineering
    c) Publisher: IEEE
    d) Date of Publication: Sept-Oct 2022
    e) Authors: Srinath Kailasa, Tingyu Wang, Lorena A. Barba, Timo Betcke
    f) Peer Reviewed: Yes
    g) Copyright Retained: No
    h) ArXiv: 10.48550/arXiv.2303.08394
    i) Associated Thesis Chapters: 2
    j) Statement of Contribution
        i. Srinath Kailasa was the lead author and responsible for the direction of this research and the preparation of the manuscript.
        ii. Tingyu Wang offered expert guidance as on fast multipole method implementations, and critical feedback of the manuscript.
        iii. Lorena A. Barba served as an advisor, offering valuable insights into the structure of the manuscript, suggesting improvements to enhance clarity and impact of the work.
        iv. Timo Betcke provided significant advisory support, contributing to the design and framing of the research, interpretation of the results and provided critical feedback of the manuscript.

Unpublished Manuscripts

1. M2L Translation Operators for Kernel Independent Fast Multipole Methods on Modern Architectures.

a) Intended Journal: SIAM Journal on Scientific Computing

b) Authors: Srinath Kailasa, Timo Betcke, Sarah El-Kazdadi

c) ArXiv: 10.48550/arXiv.2408.07436

d) Stage of Publication: Submitted

e) Associated Thesis Chapters: 3, 4

f) Statement of Contribution

   i. Srinath Kailasa was the lead author and responsible for the direction of this research and the preparation of the manuscript.

   ii. Timo Betcke provided significant advisory support aid with interpretation of the results and provided critical feedback of the manuscript.

   iii. Sarah El-Kazdadi provided expert guidance on the usage of explicit vector programming, which was critical for achieving the final results presented.

2. kiFMM-rs: A Kernel-Independent Fast Multipole Framework in Rust

   a) DOI: TODO

   b) Intended Journal: Journal of Open Source Software

   c) Authors: Srinath Kailasa

   d) Stage of Publication: Submitted

   e) Associated Thesis Chapters: 3

e-Signatures confirming that the information above is accurate

Candidate:

Date:

Supervisor/Senior Author:

Date:

Abstract

Modern scientific computing is constrained by the dual requirement for practitioners to develop algorithms that exhibit optimal complexities and convergence, at the same time as remaining fast in practice, and malleable to new algorithmic and hardware advances. This thesis is concerned with the development of a software platform for the kernel independent Fast Multipole Method (kiFMM), a variant of the widely applied Fast Multipole Method (FMM) algorithm which finds far reaching application across computational science. Indeed, for certain dense rank-structured matrices, such as those that arise from the boundary integral formulation of elliptic Partial Differential Equations (PDEs), the FMM and its variants accelerate the computation of a matrix vector product from $O(N^2)$ to just $O(N)$ in the best case.

In this thesis we demonstrate the efficacy of our software's flexible design by contrasting implementations of a key bottleneck known as the Multipole to Local (M2L) field translation, and present a new highly optimised approach based on direct matrix compression techniques and BLAS operations, demonstrating its competitiveness with the current state of the art approach based on Fast Fourier Transforms (FFT) for kiFMMs. We show that we are able to achieve similar runtimes for three dimensional problems described by the Laplace kernel to the state of the art, and often faster depending on the available hardware, with a simpler approach. The introduced approach is well suited to the direction of development of hardware architectures, and demonstrates the importance of re-considering the design of algorithm implementations to reflect underlying hardware features.

The software itself is written in Rust, a modern systems programming language, with features that enable a 'data oriented' approach to design, where operations are centred on minimising memory movements. We describe our approach to the design of the software, the flexibility of which allows us to extend our software to problems described by the Helmholtz kernel, at low frequencies, as well as in a distributed memory setting.

Impact Statement

This thesis establishes norms and practices for developing practical implementations of the kernel independent Fast Multipole Method (kiFMM), which will be of significant utility to the developers specialising in this and related algorithms. During this research we re-visited established codes for the kiFMM, identified software construction techniques that can lead to more flexible implementations that allow users to experiment, exchange, and build upon critical algorithmic subcomponents, computational backends, and problem settings - which are often missing from competing implementations which focus achieving specific benchmarks. For example, the flexibility of the software presented in this thesis allows for the critical evaluation of key algorithmic subcomponents, such as the 'multipole to local' (M2L) operator which we presented in [3].

As the primary outputs are open-source software libraries [4, 2] which are embedded within existing open-source efforts, most significantly the Bempp project, with an existing user-base the software outputs of this thesis are likely to have a wide ranging impact in academia and industry influenced by the demand for these softwares. Furthermore, the adoption and promotion of Rust for this project, and within our group, establishes further the utility of this relatively new language for achieving high-performance in scientific codes, which in recent years has been the subject of growing interest in the wider high-performance scientific computing community.

Acronyms

FFT Fast Fourier Transforms. v

FMM Fast Multipole Method. v, 1, 3, 14

kiFMM kernel independent Fast Multipole Method. v, 14

M2L Multipole to Local. v, 14

PDEs Partial Differential Equations. v, 1

Contents

Introduction

The FMM was introduced to accelerate the application cost of dense matrices which exhibit a special low-rank structure in their off-diagonal blocks, the canonical example of which is the computation of electrostatic or gravitational potentials, described by the Poisson equation. The FMM reduces the application cost of the resulting matrix-vector product to calculate potentials from mass/charge densities from $O(N^2)$ to just $O(N)$, a significant speedup. As such matrices also arise in the discretisation of boundary integral equations formulation of elliptic Partial Differential Equations (PDEs), the FMM makes practical a vast variety of scientific and engineering simulations.

Despite the improved complexity, the operations that the FMM consists of contain practically significant constants in their complexity estimates, which can be particularly challenging to handle especially in three dimensions.

In the time since the FMM's first introduction in the late 1980s, software and hardware environments have

In Chapter 1 we review the Fast Multipole Method and its kernel independent variant, going over the key challenges in achieving high performance parallel implementations. Chapter 3 reviews the challenges of software engineering in research, documenting our experience with Python as an alternative for achieving low-level performance as well as our chosen platform Rust, a relatively new language emerging as a contender for performant and productive research software. Chapter 4 describes in detail the engineering approach of our software, and Chapter 5 demonstrates the utility of these techniques with a benchmark study of competing approaches to a critical algorithmic subcomponent, the multipole to local field translation, and its redesign enabled by our software's construction. Chapter 6 contains benchmarks of our codes, and we conclude with a reflection on this work in Chapter 7.

The Fast Multipole Method

The Fast Multipole Method (FMM) is an algorithm that accelerates the computation of potential evaluation problems of the form

$$\phi(x_i) = \sum_{j=1}^{M} K(x_i, y_j) q(y_j), \quad i = 1, ..., N \tag{2.1}$$

where the potential $\phi(x_i)$ at a set of target points $\{x_i\}_{i=1}^N$ due to a set of source points $\{y_j\}_{j=1}^M$ associated with the densities $\{q(y_j)\}_{j=1}^M$ where $K(,)$ is an interaction kernel. The FMM accelerates this from a naive $O(NM)$ to, in the best case depending on the interaction kernel, $O(N+M)$. Introduced by Greengard and Rokhlin [1], the FMM has had a wide ranging impact in the field of computational science due to the prevalence of calculations of the form (2.1) in science and engineering applications.

- Introduction to idea and justification of fast multipole methods - origin of idea, and difference with respect to similar ideas, and utility in the era of exascale computing. - their research context and utility, and reason for why implementing them is still a research question.

- Review of FMM literature for software

- Go through the details of current and past projects, and detail exactly where in this context this thesis fits in.

- open questions on software side addressed by this thesis. How to make a framework that is usable, and open to extension.

- open questions on the algorithm side addressed by this thesis, with a software framework in hand can compare subcomponents of the algorithm.

2.1 Kernel Independent Fast Multipole Method

- Review of the KiFMM and variants. Black Box FMM, Analytical FMM, Data Driven Techniques.

- Motivation for use from a software engineering and computational performance perspective.

- Data flow during the KiFMM.

- Performance characteristics and features of the kiFMM.

- Reflection on the kiFMM and modern software and hardware

2.2 Related Ideas

- H Matrix and H2 matrices, and wider setting of the FMM and related problems.

- Abduljabbar thesis contains a nice summary I can read.

2.3 Laplace and Helmholtz

- What are the computational problems in Lapalce FMMs?

- Precomputations

- M2L operator pre-computations.

The oscillatory case is considerably more complicated both in formulation and implementation.

Basic idea of rank decay in oscillatory case.

- Different approaches taken so far.

- Where is kernel independent FMM weak in oscillatory case? And why?

- What can be done here instead, as a stop gap?

2.4 Computational Structure of Fast Multipole Methods

- Parallelism levels in computing (ILP (Pipelining, Superscalar, Speculative execution), Data level (SIMD, GPU), Thread level TLP (multithreading, simultaneous multithreading and hyper threading), Process level (symettric and asymettrixc multitprocessing), task level, Distributed Parallelism e.g. MPI and MapReduce)

- Only some of these are relevant for scientific computing

- Examine FMM data flow and relate to levels of Parallelism and which will be taken advantage of by us, and which are yet to be examined.

- What is the trend in hardware and why is the FMM a good kernel for scaling in future computer systems?

- What are the principal difficulties we will encounter? Data organisation, and communication costs in a distributed setting.

- What about good FMM software? Specialised kernels and substructures are required to be generically interfaced.

- What parts of this are addressed by this thesis and where?

Modern Programming Environments for Science

The discussion in this chapter, including figures and diagrams, is adapted from the material first presented in [4].

3.1 Requirements for Research Software

- Requirements and constraints on research software development.

- As an example FMM softwares used in recent benchmark studies (ExaFMM variants, PVFMM) have been constructed during the course of doctoral or post-doctoral projects. This entails a significant 'key man' risk, in which when the project owner completes their course of research the project enters a decay state and is no longer actively maintained and developed. New developers, unfamiliar with the code bases which can grow to thousands of lines of code, and often written without reference to standard software engineering paradigms for designing and managing large code bases (continuous integration, software diagrams, and simple decoupled interfaces) will find it challenging to build upon existing advances, and resort to developing new code-bases from scratch, rediscovering implementation details that are often critical in achieving practical performance.

- In seeking to avoid this cycle we envisioned a project built in Python, which maximises the maintainability of a project due to its simple syntax and language construction. New developers who, as a standard, are often educated in Python in the natural sciences and engineering, will hopefully be familiar with the language in order to gain productivity as fast as possible. However, as we demonstrated in our paper ... This itself imposes significant constraints on performance, which is balanced by the 'usability' of the language, making it just as challenging as developing a complex code in a compile language.

- Modern compiled languages offer tools that enable developer productivity. Examples include Go, Rust, ...

- The complexity of methods leads to complex code surface areas which are difficult to maintain especially in an academic setting with few resources for professional software engineering practice.

- The diversity of hardware and software backends leads to increasing difficulty for projects to experiment with and incorporate computational advances.

- Hardware and software complexity, and gap between a one-off coding project and extensible maintainable software tooling.

- Review developments in computer hardware and software that make this easier to be more productive, but also more challenging to wrap together over time.

- Emerging and future trends, exemplified by the step change in compiled langauges in the new generation and the interest in Rust and similar langauges. The mojo project and what this says about the future.

3.2 Low Level or High Level? Balancing Programming Environments with Performance Requirements

- Summary of Python paper results, in summary complex algorithms necessitate complex code in order to achieve performance - specifically the requirement for programmers to be in charge of memory and for hot sections manually vectorise etc. Writing everything in a high-level language obfuscates the application code from the sections critical to performance

- Review of why this was thought to be a good idea, and why it might be worth trying again in the future.

- What problems does this paper address, wrt to the literature?

- Brief review of motivation and reasoning behind Rust, and which features we take advantage of

- Review of data oriented design, how this can be enabled with traits.

Designing Software for The Fast Multipole Method

4.1 Data Oriented Design with Rust Traits

- Motivation, and review, DOD book.

- How do traits enable data oriented design.

- Overview of the design of the software.

- Diagram for principal traits and how they link together in the final software.

- Why is this good for the future? Well, it leaves open extension to other approaches for any individual subcomponent.

- An example of this is the genericity over data type, kernel implementation, and field translation method, with a space for the kind of tree data structure.

4.2 FMM Software As A Framework

- Want to encourage as much code re-use as possible.

- The re-implementation of critical subcomponents should be avoided. A step towards this is the development low-level C interfaces which enable the construction of higher level interfaces in compatible languages.

- We've made a start to this with a low-level interface to the principal API of the FMM software.

- We also want to be able to deploy on as wide a range of target hardware as possible, and leave open extension to future systems, enabled by design, referencing diagram.

- High level diagram of how software components fit together

- Explain the diagrams (will need zoomed in diagrams for things like M2L data)

- Decouple implementation from

- Code generation for multiple targets enabled by Rust's llvm based compiler.

- C ABI as a compatiblity layer to other projects, success with this in developing Python wrappers and integration with NGBEM

- Flexible backends enabled by RLST package for BLAS and Lapack.

4.3 High Performance Trees

- Exactly how are Morton encodings done, and what are the drawbacks and alternatives.

- ORB (how does it work)

- Tree Construction approach and algorithms - Morton encoding via lookup tables - neighbour finding - interaction list construction (fast)

- What did we end up doing, and what is the justification for these being good enough.

Important implementation details - construction of interaction lists, neighbour finding. - construction of Morton encodings. - trade-offs of approach in shared and distributed memory - e.g. adaptive vs weakly adaptive trees. - problems with load balancing approach etc - justification - simplicity/works vs complex/private

4.4 High Performance FMM Operators

- FMM data flow, mapping its tree structure to a data flow diagram.

- Go through each operator of kiFMM and how it's been optimised, at a high level for M2L operators.

4.4.1 Point to Multipole (P2M)

- Formulation in a blocked manner

4.4.2 Multipole to Multipole (M2M) and Local to Local (L2L)

- Formulation as BLAS3

4.4.3 Point to Point (P2P)

- Computational challenge (SIMD/SIMD), and why this is the easiest to optimise.

4.4.4 Multipole to Local (M2L)

- Description of computational challenge due to memory layout proscribed by Morton encoding.

  - Formulation of the problem, and required precomputations.

  - Data structure required to be flexible over this.

  How future FMMs may look, shallow trees with large P2P.

Comparing Field Translation Approaches for Kernel Independent Fast Multipole Method

The discussion in this chapter, including figures and diagrams, is adapted from the material first presented in [3]

- Review of approaches, success and failures, and what works in the context of modern software and hardware systems.

- Can include section reviewing PVFMM approach

- Approaches for BLAS based field translation in some more detail than in the paper.

- Our approach, and why it works with reference to the software and hardware currently available.

- Articulate the significance of this result

- Why our approach is sustainable given long term trends in hardware and software.

- Why might it be useful for Helmholtz FMM ...

- What are important trends, and what have we actually done.

Experiments

6.1 Single Node

6.2 Multi Node

Conclusion

In this thesis we have presented progress on the development and design of a software framework for kernel independent Fast Multipole methods. We've documented outputs towards the broader goal of a sustainable framework which can be extended, with re-usable subcomponents. This research performed necessitated a significant investigation into the optimal programming environment for high-performance scientific computing that enabled high productivity within the constraints of academic software development. The resultant software enabled a new investigation of the critical M2L field translation operation, a key bottleneck in the kiFMM algorithm, and the development of a highly competitive approach well suited to emerging trends in computer hardware.

Due to the high-performance the FMM operator kernels for both the Laplace and low-frequency Helmholtz kernels demonstrated in this work as well as the creation of trees, we are in a good position to extend our software to a distributed setting. The decoupling of operator kernels from their implementation via the design of our software also enables future extensions to a heterogenous platforms in which batch BLAS for the M2L, and SIMT for the P2P operations

The Adaptive Fast Multipole Method Algorithm

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

Algorithm 1 Adaptive Fast Multipole Method.

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

Step 1: Tree construction
for each node $B$ in preorder traversal of tree, i.e. the nodes are traversed bottom-up, level-by-level, beginning with the finest nodes. do
    subdivide $B$ if it contains more than $s$ points.
end for
for each node $B$ in preorder traversal of tree do
    construct interaction lists, $U$, $V$, $X$, $W$
end for

Step 2: Upward Pass
for each leaf node $B$ in postorder traversal of the tree, i.e. the nodes are traversed top-down, level-by-level, beginning with the coarsest nodes. do
    P2M: compute multipole expansion for the particles they contain.
end for
for each non leaf node $B$ in postorder traversal of the tree do
    M2M: form a multipole expansion by translating the expansion centre of its children to its centre and summing their multipole expansion coefficients.
end for

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

Hyksort

The parallel splitter selection and HykSort algorithms are provided below. In terms of complexity analysis, we adapt the analysis provided in section 3.4 of [5]. The main costs of SampleSort is sorting the splitters and the MPI collectives for data reshuffling. This can lead to a load imbalance and network congestion, represented by a constant c below,

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

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

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

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

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

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

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

$$T_{Hk} = t_c \frac{N}{p} \log \frac{N}{p} + \left(t_c \frac{N}{p} + T_{ps} \right) \frac{\log p}{\log k} + T_{a2a} \tag{B.1}$$

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

Algorithm 2 Parallel Select

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

Algorithm 3 HykSort

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

Distributed Octrees

C.1 Useful Properties of Morton Encodings

These properties are taken from Appendix A in [6]

1. Sorting the leaves by their Morton keys is equivalent to pre-order traversal of an octree. If one connects the centers of the boxes in this order we observe a ‘Z’-pattern in Cartesian space. Nearby octants in Morton order are clustered together in Cartesian space.

2. Given three octants $a < b < c$ and $c \notin \text{Descendants}(b)$

$$a < d < c \forall d \in \{\text{Descendants(b)}\}$$

3. The Morton key of any box is less than those of its descendants.

4. Two distinct octants overlap only if and only if one is an ancestor of another.

5. The Morton key of any node and its first child are consecutive.

6. The first descendant at level l, $\text{FirstDescendant}(N, l)$ of any box N is the descendant at that level with the least Morton key.

7. The range $(N, \text{DeepestFirstDescendent}(N)]$ contains only the first descendants of N at different levels, and hence there can be no more than one leaf in this range in the entire linear octree.

8. The last descendant at level l of N, $\text{LastDescendant}(N, l)$ of any node N is the descendant at that level with the greatest Morton key.

9. Every octant in the range $(N, \text{DeepestLastDescendant}(N)]$ is a descendant of N.

C.2 Algorithms Required for Constructing Distributed Linear Octrees

These listings are adapted from [6].

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

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

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

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

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

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

Algorithm 7 Construct Distributed Octree (Parallel)

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

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

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

```
Input: A distributed list of octants $F$.
Output: A list of blocks $G$, $F$ redistributed but the relative order of the octants is preserved.
Work: $O(n)$, where $n$=len($F$).
Storage: $O(n)$, where $n$=len($F$).
$T \leftarrow$ CompleteRegion($F[1]$, $F[\text{len}(F)]$), using algorithm (5)
$C \leftarrow \{x \in T | \forall y \in T, \text{Level}(x) \leq \text{Level}(y)\}$
$G \leftarrow$ CompleteOctree($C$), using algorithm (9)
for $g \in G$ do weight($g$) $\leftarrow$ len($F_{global} \cap \{g, \{\text{Descendents (g)}\}\}$)
end for
$F \leftarrow F_{global} \cap \{g, \{\text{Descendants}(g)\} \forall g \in G\}$
```

Algorithm 9 Construct a Complete Linear Octree From a Set of Seed Octants Spread Across Processors (Parallel) - `CompleteOctree`

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

Bibliography

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

[2] Srinath Kailasa. "kifmm-rs: A Kernel-Independent Fast Multipole Framework in Rust". Submitted to Journal of Open Source Software. 2024.

[3] Srinath Kailasa, Timo Betcke, and Sarah El Kazdadi. *M2L Translation Operators for Kernel Independent Fast Multipole Methods on Modern Architectures.* 2024. arXiv: `2408.07436 [cs.CE]`. URL: `https://arxiv.org/abs/2408.07436`.

[4] Srinath Kailasa et al. "PyExaFMM: an exercise in designing high-performance software with Python and Numba". In: *Computing in Science & Engineering* 24.5 (2022), pp. 77–84.

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

[6] 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.