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

August, 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

I leave the past five years transformed personally and professionally, the completion of this thesis simply wouldn't have been possible without the strong prevailing wind of emotional support from my family and the regularity of fun with my friends, sympathetic and dedicated teachers and colleagues both at UCL and across the world. It's been a pleasure to grow as a person and as a scientist with your support over these years which have been incredibly formative. Thank you all for giving me this opportunity.

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: 1
    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Unpublished Manuscripts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: TODO

d) Stage of Publication: Submitted

e) Associated Thesis Chapters: 2

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) Intended Journal: Journal of Open Source Software

   b) Authors: Srinath Kailasa

   c) ArXiv: TODO

   d) Stage of Publication: Submitted

   e) Associated Thesis Chapters: 1, 2

e-Signatures confirming that the information above is accurate

Candidate:

Date:

Supervisor/Senior Author:

Date:

Abstract

Software has come to be a central asset produced during computational science research. Projects that build off the research software outputs of external groups rely on the software implementing proper engineering practice, with code that is well documented, well tested, and easily extensible. As a result research software produced in the course of scientific discovery has become an object of study itself, and successful scientific software projects that operate with performance across different software and hardware platforms, shared and/or distributed memory systems, can have a dramatic impact on the research ecosystem as a whole. Examples include projects such as the SciPy and NumPy projects in Python, OpenMPI for distributed memory computing, or the package manager and build system Spack, which collectively support a vast and diverse ecosystem of scientific research.

This thesis is concerned with the development of a software platform for Fast Multipole Methods. These algorithms have emerged in recent decades to optimally apply dense matrices that exhibit a special low-rank structure in their off-diagonal elements. Such matrices arise in numerous areas of science and engineering, for example in the linear system matrices of boundary integral formulations of problems from acoustics and electromagnetics to fluid dynamics, geomechanics and even seismology. In the best case matrices can be stored and applied in $O(N)$, in contrast to $O(N^2)$ for storage and application when computed naively. The diversity of the implementation approaches for these algorithms, as well as their mathematical intricacy, makes the development of an ergonomic, unified, software framework, that maximally re-uses data structures, is designed for high performance, distributed memory environments, and works seamlessly across platforms highly challenging. To date, many softwares for fast algorithms are often written to benchmark the power of a particular implementation approach, rather than with real user in mind.

Impact Statement

The bulk of the work contributed in this thesis is presented via substantial open-source software contributions to the Bempp and ExaFMM projects, most significantly the libraries kiFMM-rs [1] and pyexafmm [2]. The former software is a core dependency of the upcoming boundary element software Bempp-rs, a project with an existing and broad user base across industry and academia, however its standalone nature makes

Contents

Introduction and Background

Modern Programming Environments for Science

Designing Software for The Fast Multipole Method

3.1 Fast Multipole Methods

3.2 Kernel Independent Fast Multipole Method

3.3 Computational Issues

Bottlenecks in shared and distributed memory

Tree Construction

M2L kernels

P2P kernels (cite out work)

3.4 Software Issues

JOSS paper

Data oriented design for kiFMM-RS

Traits for flexibility

Code generation for multiple targets

Flexible backends

3.5 Performance Model

3.6 Distributed FMM

3.7 Helmholtz FMM

Experiments

4.1 Single Node

4.2 Multi Node

Conclusion

In this subsidiary thesis we've presented progress on the development of a new software infrastructure for fast algorithms. We've documented recent outputs towards this goal including foundational software as well as algorithmic techniques. The main outputs being an investigation into programming languages and environments most suitable for scientific computing, investigations to ensure an ergonomic design for our software, a distributed load balanced octree library designed for high-performance, as well as significant inroads to a distributed FMM based on this by studying sparsification schemes for the multipole-to-local translation operator T^{M2L}.

The immediate next steps of this project will be to publish our recent software results on octrees and the parallel FMM in an appropriate scientific journal, and release a first version of our software. The final stages of this project will focus on completing the outlined improvements to our translation operator library to achieve, and hopefully supersede the current state of the art, creating a new benchmark distributed FMM library that is open to extension to other fast algorithms.

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 [3]. 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
        $p_{recv} \leftarrow m((color - i) \bmod k) + (p_r \bmod m)$
        $R_i \leftarrow$ MPI_Irecv($p_{recv}, comm$)
    end parfor
    for $i \in 0, ..., k-1$ do
        $p_{recv} \leftarrow m((color - i) \bmod k) + p_r \bmod m$
        $p_{send} \leftarrow m((color + i) \bmod k) + p_r \bmod m$
        $j \leftarrow 2$
        while $i > 0$ and $i \bmod j = 0$ 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 [4]

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, 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 [4].

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($L$).
$F \leftarrow [\text{Octant}(p, \text{MaxDepth}), \forall p \in L]$
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] Srinath Kailasa. “kifmm-rs: A Kernel-Independent Fast Multipole Framework in Rust”. Submitted to Journal of Open Source Software. 2024.

[2] 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.

[3] 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.

[4] 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.