# Modern Software Approaches For Fast Algorithms

**Srinath Kailasa**

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

Department of Mathematics
University College London
September, 2023

## Declaration

I, Srinath Kailasa, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the thesis.

**Abstract**

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 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 Spack, which collectively support a vast and diverse ecosystem of scientific research.

Recent decades have seen the emergence of so called 'fast algorithms', these are algorithms that are able to optimally apply and invert dense matrices that exhibit a special low-rank structure in their off-diagonal elements. Such matrices arise in numerous areas of science and engineering, for example in the linear system matrices of boundary integral formulations of problems from acoustics and electromagnetics to fluid dynamics, geomechanics and even seismology. In the best case matrices can be stored, applied and inverted in $O(N)$, in contrast to $O(N^2)$ for storage and application, and $O(N^3)$ for inversion when computed naively. The 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.

This thesis is concerned with the development of such a software platform for fast algorithms. Modern programming environments are diverse, with unique trade-offs, in Chapter 1 we document our experience in transitioning from Python to Rust as a modern, ergonomic, programming environment for our software. Chapter 2 introduces fast algorithms in the context of current work streams, specifically the implementation of a library for distributed memory octrees, and a distributed memory fast multipole method (FMM) built on top of this. Chapter 3 documents active areas of research and development for these projects, and proposed strategies for outstanding issues. Finally, Chapter 4 looks ahead to the requirements for the timely conclusion of this project as well as proposed extension areas.

# Contents

# Modern Programming Environments for Science

## 1.1 Developing Scientific Software

Scientific software development presents a unique set of challenges. Although development teams are frequently small, they are tasked with producing highly optimized code that must be deployed across a myriad of hardware and software platforms. Moreover, there is a pressing need for comprehensive documentation and rigorous testing to ensure reproducibility. Given that many of these softwares arise within doctoral programs or other short-term projects, there is a tendency to tailor software development to showcase a specific project's objectives. Whether that be to demonstrate a convergence result of a specific methodological improvement, or offer a new benchmark implementation of an algorithm. Consequently, once the principal results are achieved, these software projects often become orphaned, lack compatibility with a range of development platforms, or aren't adaptable to related challenges and subsequent research by other teams.

A recent survey of 5000 software tools published in computational science papers featured in ACM publications found that repositories for computational science papers had a median active development span of a mere 15 days. Alarmingly, one third of these repositories had a life cycle of less than one day [2]. Implying that upon the publication of the affiliated paper, software typically gets abandoned or, at best, receives private maintenance. This trend underscores the challenge of dedicating sustained resources in an academic setting to software upkeep, even when such maintenance is vital for reproducibility. It may also hint at a deficiency in professional software engineering expertise among computational researchers whose principal expertise lies elsewhere.

Therefore, confronting the challenge of developing maintainable research software relies on the choice of programming environment. Developers need to have a frictionless system for testing, documenting, using existing open-source solutions and building extensions to their code. Software design has to be general enough to extend to new algorithmic developments, but also malleable enough for external developers and users to adapt software to new usecases as well as their own needs. Building software for diverse software environments and target architectures should also be painless as possible to encourage large-scale adoption. Additionally, domain scientists who are typically not experienced in low-level software development require interfaces to familiar high level languages, which must be easy to maintain for core-developers.

In the early stages of this research project we experimented with Python, a high-level interpreted language, that has become a de-facto standard in data-science and

numerical computing for a wide variety of domain scientists. Recent years have seen the development of tools that allow for the compilation of fast machine-code from Python, allowing for multi-threading, and the targeting of both CPU and GPU architectures [4]. This approach takes advantage of the LLVM compiler toolchain for generating fast machine code from Python via the Numba library, and is similar to other approaches to creating fast runtimes for high-level languages such as Julia [1]. We built a prototypical fast algorithm, the fast multipole method (FMM), in Python to test the efficacy of this approach. However, we found that for complex algorithms writing performant Numba code can be challenging, especially when performance relies on low-level management of memory [3]. We summarise this experience in section 1.2. We identified Rust, a modern low-level compiled language, as an excellent programming environment for our software. Rust has a number of excellent features for scientific software development, most notably the introduction of a 'borrow checker', that enforces the validity of memory references at compile time preventing the existence of data races in compiled Rust code, as well as its runtime 'Cargo', which offers a centralised system for dependency management, compilation, documentation and testing of Rust code. We summarise Rust's benefits, as well as notable constraints, in section 1.3. We conclude this chapter by noting that language and compiler development for scientific computing is an active area of research, in section 1.4 we contrast Rust with other established programming environments to scientific software such as Julia as well as emerging platforms such as Mojo.

## 1.2 Pitfalls of Performant High Level Language Runtimes

## 1.3 Introducing Rust for Scientific Software

## 1.4 Future Developments

# Designing Software for Fast Algorithms

Q. What is the FMM, and how does it fit into an ecosystem of Fast Algorithms?

- Introduce the algorithmic intuition for the FMM very briefly. - Dump the algorithm in the appendix. - Zoo of FMMs/Hierarchical algorithms that have similar algorithm structure. - Utility relies on problem context. - Where some FMMs are preferable to others. - This makes a generic software hard ...- Some FMMs are amenable to generic implementation that target a wide-class of problems, black box. - KiFMM - How does it create multipole and local expansions - MFS. - Dump MFS further explanation in the appendix. - How does it compute the translation operators? - turn it into a least squares problem - bbFMM - How does it create multipole and local expansions? Cheb. - How does it compute translation? - Not sure, need to check.

Q. What are the main challenges associated with developing bbFMMs?

- Performant multipole to local translations. Which in bbFMMs are a matvec. - Main approaches to accelerate this are SVD and FFT. - How do these work, roughly? - Dump detailed explanation in the appendix. - Can count flops for different approaches - Implementation challenges - memory ordering for fft - fast ffts - blas3 for SVD - designing a generic interface for developers - fast trees, that can also work in dist memory - communication bottlenecks, and potential approaches

- dump all algorithms into appendix

Q. What about other fast algorithms, do they require any of the same stuff used in the FMM? What are the key areas to make generic as an implementer, has this been done before?

- Inverse FMM H matrix inverses, other fast direct solver approaches all rely on fast trees - trees are actually widely used in other sci comp tasks - current tree software, p4est - why can we not just use this? - want cross platform builds trivial - want to design a hierarchy of interfaces that can be used for our application, but also easily plug in to other work.

# Open Work Streams

Q. Active areas of development - field translations and comparison with state of the art - can take some benchmarks with dummy library - highlight the communication bound - can measure flop rate for translation library - can do roofline analysis of blas3 optimistion on modern processors for svd approach, though note that this is not necessarily the end of the road, as we have yet to examine the rank behaviour for different kernels. - communicator optimisations in trees - what has been implemented in p4est, and can this help us?

# Conclusion

- Comparison study of M2L - Completion of distributed FMM - Longer term/ final project - incorporation of fast direct solvers into software framework.

# Appendix

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

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

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

Deriving the local expansion centered around the origin, where the bounding box of the targets, $\Omega_t$, is well separated from the source box, $\Omega_s$,

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

from the multipole expansion relies on the following expressions,

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

and,

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

Substituting these expressions into (A.1), translated to be centred on $\Omega_t$

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

Identifying the local expansion coefficients as,

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

and,

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

# HykSort

The parallel splitter selection and HykSort algorithms are provided below. In terms of complexity analysis, we adapt the analysis provided in section 3.4 of [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,

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

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

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

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

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

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

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

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

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

**Algorithm 1 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 2 HykSort**

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

# Bibliography


[1] Jeff Bezanson et al. “Julia: A fresh approach to numerical computing”. In: *SIAM review* 59.1 (2017), pp. 65–98.

[2] Wilhelm Hasselbring et al. “Open source research software”. In: *Computer* 53.8 (2020), pp. 84–88.

[3] 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.

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

[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.