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
June, 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

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

The unification of software for the forward and inverse application of these operators in a single set of open-source libraries optimised for distributed computing environments is lacking, and is the central concern of this research project. We propose the creation of a unified solver infrastructure that can demonstrates good weak scaling from local workstations to upcoming exascale machines. Developing high-performance implementations of fast algorithms is challenging due to highly-technical nature of their underlying mathematical machinery, further complicated by the diversity of software and hardware environments in which research code is expected to run.

This subsidiary thesis presents current progress towards this goal. We begin with overview of the fast algorithms of interest, and a summary of the goals and expected impact of this research in Chapter 1. Chapter 2 details an early investigation of Python with Numba, an LLVM based 'just-in-time' (JIT) compiler, as a tool for building out our software infrastructure. Despite its many advantages, including cross-platform support and large numerics ecosystem, we find Python and high-level languages in general to be inadequate for our purposes. In chapter 3 we propose Rust, a modern and fast developing systems programming language as our proposed solution for ergonomic and high-performance codes for computational science. In chapter 4 we detail current open work streams in this research project, specifically current completed software outputs, computational investigations into optimal implementations of certain features of fast algorithms. We focus for now on the 'fast multipole method' (FMM), an algorithm for the fast application of these special low-rank matrices to vectors, with the extension to fast-inverses which re-use much of the FMM's structure, planned as a future extension. Chapter 5 summarises our goals for the near, and medium term, as well as the expected outputs of this research.

Contents

Introduction

- The goal of this research project is to contribute significantly towards the design and development of a highly-parallelised, scalable, software infrastructure for 'Fast Algorithms' - 'Fast Algorithms' - explain how they arise from desnse matrices with hierarchical structures that allows for numerical compression of off-diagonals for optimal application/inversion. - Specific outputs: - Parallel tree library - Parallel FMM library - Re-use of data structures to create fast direct solver library. - Context - Next gen BEM software - can scale from laptops to supercomputers - deployed with ease with interpreted language interfaces for non-computational researchers - Problems that can be studied - virus simulation - geophysics - ML - Impact - solvers in this domain are either - designed to numerically validate an algorithmic approach - heavily optimised to achieve performance - nothing extensible really available - as a testbed for algorithmic experimentation - extension to differing applications in which these algorithms apply. - or as a testbed for new implementation approaches - new hardware etc. - the impact of a particular method is defined by how easy it is for software to be extended OR to be run as a black box.

A Python FMM

- What is the FMM? - What are kernel-independent FMMs? - Benefits of this approach - Computational bottlenecks and open implementation issues - Tree construction - Design and implementation of translation operators - Simple software interfaces extensible to new techniques/hardware and accessible from familiar interpreted languages (python/matlab etc)

- Why did we try a Python FMM? - Fulfills many of the requirements we need for our infrastructure - large ecosystem of software for science - easy to deploy - easy to write and extend - JIT compiled approaches make it easy to extend to new hardware targets using LLVM infrastructure.

- Why did it not succeed? - Why JIT compiled approaches are not sufficient. - What alternatives are emerging (Mojo) and why these aren't necessarily a competitor to compiled languages. - What do we need from a language for our approach? And why is Rust a suitable alternative?

Rust as Tool For Computational Science

- Its benefits and pitfalls. - Traits for bottom up design. - Memory safety, contrast with C++/C - Python/C interfaces - Cargo as a package manager/deployment tool. - Comparison with major competitors - C/C++ and Fortran - Also JIT approaches, as well as MoJo's approach.

Current Work Streams

- Brief introduction to the FMM we use (KiFMM) - Why are the translation operators and tree construction the biggest bottlenecks and what past work has been done to optimise these pieces? - SVD based M2L - description of approach - computational optimisations (BLAS3, precomputation with randomised SVD, multithreading) - FFT based M2L - description of approach - computational optimisations (explicit SIMD, multithreading) - Contrast approaches in terms of complexity, as well as practical considerations - Identify a gap in the literature to directly compare the optimisation approaches on modern hardware. - propose this as upcoming paper.

- High performance distributed trees - Our algorithmic approach and why its the most appropriate for our usecase - double global sort etc justified. - Design approach to easily extend single to multiple nodes. - Key shared traits and how these help. - some preliminary scaling experiments. - How many octants can I get scaling on a single node? - How many octants can I get on Kathleen? - Unimplemented innovations to improve trees and hyksort (communicator optimisations) - Sub communicators to improve strong scaling in M2M/L2L - Review p4est's optimisations as they are state of the art - suggest how we can incorporate/improve on these - how we differ (two global sorts etc.)

- Design of Rust based FMM on a single node - Brief discussion on how this would translate to a multinode setting via trait implementations. - Explain current status of project code.

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, Ω_t, is well separated from the source box, Ω_s,

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

from the multipole expansion relies on the following expressions,

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

and,

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

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

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

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 [1]. The main costs of SampleSort is sorting the splitters and the MPI collectives for data reshuffling. This can lead to a load imbalance and network congestion, represented by a constant c below,

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

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

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

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

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

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

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

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

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

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

Algorithm 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] 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.