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/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 so called 'fast algorithms'. These algorithms have emerged in recent decades 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.

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 optimised 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 [16]. 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 [19]. This approach takes advantage of the LLVM compiler infrastructure for generating fast machine code from Python via the Numba library, and is similar to other approaches to creating fast compiled code from high-level languages such as Julia [4]. We built a prototypical single-node multithreaded implementation of a 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 [17]. We summarise this experience in section 1.2. We identified Rust, a modern low-level compiled language, as a promising 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 emerging programming environments for scientific software.

1.2 Pitfalls of Performant High Level Language Runtimes

The evolution of computational science has been marked by the introduction and adoption of high-level interpreted languages that are designed to be user friendly, and cross platform. Starting with Matlab in the 1970s, followed by Python in the 1990s, and more recently Julia in the 2010s. High-level languages have significantly impacted scientific computing, offering efficient implementations of algorithms and introducing optimised data structures via projects such as NumPy and SciPy, as well as ergonomic cross platform build systems. While these languages facilitate easier experimentation, achieving peak performance often necessitates manual memory management or explicit instruction set level programming to ensure vectorisation on a given hardware target. A natural question for us when deciding on a programming environment for this project was whether advancements in high-level languages were sufficient for the implementation of 'fast algorithms'. If they proved to be sufficient our software would be free of the so called 'two language' problem, in which a user friendly interfaces in a high-level language provides a front-end for a compiled language implementation of more data-intensive operations. This problem plagues academic software development, as resulting software relies on a brittle low-level interface between the high-level front-end, and low-level back-ends for performance critical code sections.

Recent strategies to enhance the performance of high-level languages have involved refining compiler technology. These projects are advertised as tools that allow developers to use high-level languages for quick algorithm experimentation, while leaving it to the compiler to produce efficient machine code. Benchmarks are usually offered with respect certain numerical operations that can be vectorized, such as iteration over aligned data structures, with compilers taking care to unroll

loops and apply inlining to inner function calls.

Noteworthy examples are the Numba project for Python and the Julia language. Both leverage the LLVM compiler infrastructure, which aims to standardize the back-end generation of machine code across various platforms. This approach, known as 'just in time' or JIT compilation, generates code at runtime from the types of a function's signature. Figure 1.1 provides a sketch of how Numba in particular generates machine code from high-level Python code. The LLVM compiler supports numerous hardware and software targets, including platforms like Intel and Arm, and provides multithreading through OpenMP. Additionally, these compilers offer native support for GPU code generation. An important difference between Numba and Julia is that Numba is simply a compiler built to optimise code written for the numerical types created using the NumPy library, and Julia is a fully fledged language. Numba contains implementations of algorithms a subset of the scientific Python ecosystem, specifically functionality from the NumPy project for array manipulation and linear algebra.

As many performance benchmarks for these programming environments are typically provided for algorithms that rely on simpler data structures, we decided to test these high-level environments for more complex algorithms before making a choice about our programming environment for this project. We implemented a single node multi-threaded FMM using Python's Numba compiler, identifying two notable pitfalls with this approach, the full results of which were recently published in Computing in Science and Engineering [17].

Firstly, JIT compilation of code imposes a significant runtime cost that is disruptive to development. Compiled functions are by default not cached between interpreter sessions in either Julia or Numba code. Our FMM code takes 15 ± 1s to compile when targeting a i7-9750H CPU with x86 architecture, where the benchmark is given with respect to seven runs. This is comparable to the runtime of the FMM software for a typical benchmark FMM problem [1]. For smaller problem sizes, the compilation time dominates runtime. Ahead of time (AOT) compilation is partially supported in Numba, and can be used to build binaries for distribution to different hardware targets, e.g. x86 or ARM. However support for AOT compiled code is currently second class, the machine code created in this manner are usable only from the Python interpreter and not from within calls made from other Numba compiled functions, and it's currently staged for deprecation and replacement. We note that Julia does support AOT compilation via the PackageCompiler.jl module. This allows for different levels of AOT compilation, from creating a 'sysimage' which amounts to a serialised file containing the compiled outputs of a Julia session, a relocatable 'app' which includes an executable compiled to a specific hardware target alongside Julia itself, to creating a C library which can be precompiled for a specific hardware target such as x86. However, this requires relatively advanced software engineering skills, and raises the barrier to entry for high-performance computing with Julia.

Thus switching to such a programming environment requires developers to create a workflow that keeps interpreter sessions active for as long as possible in order to reduce the impact of long compilation times, or write build scripts for AOT compilation for a specific hardware target similar to compiled languages. As one of the key advantages of high-level languages is their developer friendliness and simple

[1] 1e6 particles distributed randomly, using order $p = 6$ expansions for a Laplace problem takes approximately 30s on this hardware using our software. See Chapter 2 for more details on the FMM and the significance of these terms.

build systems this acts as an impediment.

Downstream users of software, who may also be using JIT compilers for their own code, are faced with significant compilation times, and potentially intricate build steps unless catered for by the original library's developers. This problem compounds in the case of distributed memory programs using MPI, as JIT compilation imposes runtime costs to the entire program. Unless a project has been distributed as a binary, by default MPI runs are not interactive, and therefore require a recompilation for each run. This isn't to say MPI programs written in Julia or Numba haven't been scaled to large HPC systems, however their usage does carry a cost in terms of developer workflow, and potentially program runtime, which can be significant if one is trying to reach the highest levels of performance.

Secondly, our experience in developing the FMM in Numba made clear how difficult it can be to anticipate the behaviour of Numba when considering how to optimise functions with different implementations of the same logic. Consider the code in Listing 1.1, here we show three logically equivalent ways of performing two matrix-matrix products, and storing a column from the result in a dictionary. The runtimes of all three implementations are shown in Table 1.1 for different problem sizes. We choose this example because this operation of matrix-matrix multiplication is well supported by Numba, and data instantiation, whether from within or external to a Numba compiled function, should in principle make little difference as the Numba runtime simply dereferences pointers to heap allocated memory when entering a Numba compiled code segment. This example is designed to illustrate how arbitrary changes to writing style can impact the behavior of Numba code. The behavior is likely due to the initialisation of a dictionary from within a calling Numba function, rather than an external dictionary, and having to return this to the user. However, the optimisations taken by Numba are presented opaquely to a user and it's unclear why there are performance variations at all.

In developing our FMM code we faced significant challenges in tweaking our code and data structures to maximise the performance achieved from within Numba. From manually inlining subroutines as in Listing 1.1, to testing where to allocate data. Our final code took a significant amount of time to develop, approximately six months, and was organised in a manner that optimised for performance rather than readability. Thus, despite being a *compiler* for numerical Python, Numba behaved in practice more like a *programming framework* which a developer had to adhere to strictly in order to achieve the highest-level of performance. The disadvantage of this is that the framework is both relatively restrictive, but also presented opaquely to a user.

Therefore, while being an amazing technology with great features, Numba was not decided to be a suitable choice for our programming environment. Its ability to target a diverse set of hardware targets from Python, leveraging the power of LLVM to write multithreaded, and autovectorised code, as well as writing CPU and GPU code from within Python, while allowing downstream projects to develop in a familiar language with a large open-source ecosystem and simple dependency management demonstrate the utility of this remarkable tool. However, the constraints of high level languages, specifically the inability to manage memory at a system level, as well as the development costs of JIT compilation, and Numba's framework-like behaviour demonstrate that while being useful, it is preferable to write our software platform using a systems-level language where high-performance can be assumed. Our criticisms of Numba also apply to Julia. However we note that Julia has certain advantages over Numba including its design and syntax focussed on mathematical

computing, native support for multithreading, a richer type system and a large open-source community with a scientific computing focus.

System level languages have progressed commensurately to high-level languages over recent decades. Modern languages such as Go and Rust, offer runtimes with informative compilers, inbuilt documentation and testing facilities, as well as LLVM backends to support code generation across platforms, removing much of the complexity associated with building and distributing software written in C/C++. Rust in particular, uninhibited by a garbage collector as for Go, makes it easy to interoperate Rust code with libraries built in C/C++ via its foreign function interface, making it simple to leverage existing open-source scientific software. We therefore conclude that the two language problem is minimised in comparison to the past, and modern system level programming languages potentially offer an effective solution for building academic software.

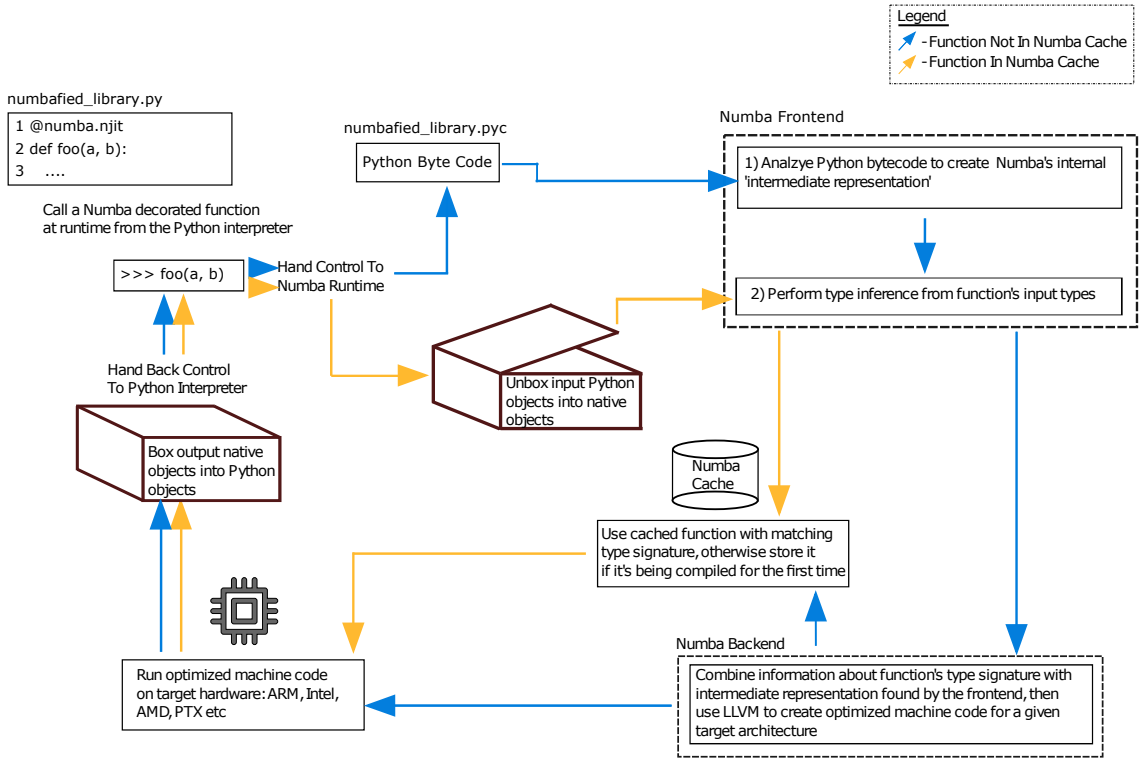


Figure 1.1: A visualisation of the Numba runtime system. A function decorated with 'njit' acts as an instruction to the runtime to check a database for any Numba-compiled functions with a matching function signature, if this doesn't exist Numba generates a new compiled function and caches it using LLVM. Future calls of this function use this cached version in place of the Python interpreter. Code relies on Numba's runtime to correctly 'box' and 'unbox' native Python objects into data structures compatible with Numba code, which operates on a subset of numerical data structures created using the NumPy library. Figure adapted from [17].

```
import numpy as np
import numba
import numba.core
import numba.typed

# Initialise in the Python interpreter
data = numba.typed.Dict.empty(
    key_type=numba.core.types.unicode_type,
    value_type=numba.core.types.float64[:]
)

data['initial'] = np.ones(N)

# Subroutine 1
@numba.njit
def step_1(data):
    """
    Initialise a matrix and perform a matrix matrix product,
    storing a single column in the data dictionary.
    """
    a = np.random.rand(N, N)
    data['a'] = (a @ a)[0,:]


# Subroutine 2
@numba.njit
def step_2(data):
    """
    Initialise a matrix and perform a matrix matrix product,
    storing a single column in the data dictionary.
    """
    b = np.random.rand(N, N)
    data['b'] = (b @ b)[0,:]


@numba.njit
def algorithm_1(data):
    """
    First implementation.
    """
    step_1(data)
    step_2(data)


@numba.njit
def algorithm_2(data):
    """
    Second implementation.
    """
    # This time the storage dictionary is created within the
    # Numba function, so the types are inferred by the Numba
    # runtime, this also avoids a boxing cost to create a
   Numba
```

```
    # type from a Python one.
    data = dict()
    data['initial'] = np.ones(N)
    step_1(data)
    step_2(data)
    return data

@numba.njit
def algorithm_3(data):
    """
    Third implementation.
    """

    # This time, the subroutines are manually inlined
    # by the implementer, as well as the initialisation
    # of the results dictionary locally, as in algorithm_2.

    data = dict()
    data['initial'] = np.ones(N)

    def step_1(data):
        a = np.random.rand(N, N)
        data['a'] = (a @ a)[0,:]


    # Subroutine 2
    @numba.njit
    def step_2(data):
        b = np.random.rand(N, N)
        data['b'] = (b @ b)[0,:]

    step_1(data)
    step_2(data)
    return data
```

Listing 1.1: Three ways of writing a trivial algorithm in Numba, that performs some computation and saves the results to a dictionary. Adapted from Listing 2 in [17]

1.3 Introducing Rust for Scientific Software

Rust is a modern system-level programming language, introduced by Mozilla in 2015 as a direct replacement for C/C++, and is bundled with features that favour safety for shared memory programming. Having identified it as a suitable candidate for our programming environment, we list a few of its key benefits in this section.

Fortran and C/C++ have have continued to dominate high-performance scientific computing applications, the main criticism of these languages for academic software is their relatively poor developer experience. C/C++ especially has significant flexibility in the compilers, documentation, build systems and package managers that developers can choose to work with, as well as support for multiple paradigms and a growing syntax. Rust stands in contrast to this with a single centrally supported runtime system, Cargo, with common standards for testing and documentation. Additionally, there is only a single Rust compiler, rustc. This inflexibility, in

Table 1.1: Performance of different algorithms from Listing 1.1, taken on an i7 CPU and averaged over seven runs for statistics.

Algorithm	Matrix dimension	Time (μs)
1	$\mathbb{R}^{1\times1}$	1.55 ± 0.01
1	$\mathbb{R}^{100\times100}$	304 ± 3
1	$\mathbb{R}^{1000\times1000}$	$29,100 \pm 234$
1	$\mathbb{R}^{1\times1}$	2.73 ± 0.01
1	$\mathbb{R}^{100\times100}$	312 ± 3
1	$\mathbb{R}^{1000\times1000}$	$25,700 \pm 92$
1	$\mathbb{R}^{1\times1}$	2.71 ± 0.01
1	$\mathbb{R}^{100\times100}$	312 ± 1
1	$\mathbb{R}^{1000\times1000}$	$25,700 \pm 140$

addition to a strongly preferred way of organising Rust code via its Traits system, makes Rust libraries significantly more uniform and readable than corresponding C++ code. Indeed installing a Rust library, or building a binary, is often as simple as running a single command from a terminal, or adding a single line to a TOML dependency file.

The lack of a uniform building and packaging standards in C/C++ means that some projects go as far as to implement a custom build system, such as the Boost library [2]. With the exception of Fortran, which has made recent strides to develop a standardised modern package manager and build system, inspired by Rust's Cargo [13], C and C++ do not have a single officially supported package manager or build system. The resulting landscape is a multitude of package managers [28, 34, 11] and build systems [33, 3, 26] a few of which we have cited here, all of which replicate each others functionality, none of which are universally accepted or implemented across projects nor officially supported by the C++ software foundation. Figure (1.2) provides an overview of a few of the myriad approaches taken in other languages. To manage this complexity, builds are often defined using a metabuild system, most commonly CMake. CMake is a scripting language, and as a meta build system it takes a specification of local and third party dependencies and hardware targets, and generates Makefiles. CMake gives developers a great deal of flexibility, it is multi-platform, and language agnostic, however using it directly is not straightforward to maintain projects as dependencies grow, and to keep on top of dependency versions. Indeed, there is a significant body of literature discussing best practices with CMake [27]. However, CMake is not responsible for downloading and installing third party packages or verifying their relative compatibility, implementing its best practices is again left to users. Cargo's relative simplicity mirrors the simple build systems of high-level languages such as Python or Julia, and removes one of the main causes of development pain when working with compiled languages, and makes it significantly easier for small teams to publish software that can be easily deployed by downstream users regardless of their system's architecture or operating system.

A unique feature of Rust is its approach to ensure safe shared memory programming, enforced by its compile time 'borrow checker'. Every reference in Rust has an associated 'lifetime' defined by its scope, and a singular 'owner'. Which enforce the programming pattern of 'resource allocation is initialisation' (RAII). The basic rule is that references are owned within a scope, and dropped when out of scope. The

borrow checker enforces this at compile-time, in a multithreaded context this makes it impossible to have a compiled Rust binary that has dangling pointers or double-free errors. For mutable data, the borrow checker ensures that there is only a single mutable reference at any given time in the program's runtime, ensuring that there can never be a race condition in compiled Rust code. Identifying pointer-related bugs is one of the main challenges in multi-threaded programming, with safe 'smart pointers' being optional in C++, implementing RAII is left to developers.

Rust is 'multi-paradigm', supporting both object oriented, and functional styles of programming. Method calls are often chained, in a functional-like style, however users can still implement methods on structs as in other object oriented languages. The unique feature introduced by Rust is its Traits system, for specifying shared behaviour, that supersedes object-oriented design. Traits are a similar to C++ 21's interfaces, in that they provide a way to enforce behaviour, rather than embedding it into a type as with traditional inheritance. However unlike C++, Rust Traits allow you to write blanket implementations, and implement interfaces for types you didn't define in your own code making them significantly more powerful. This means that behaviour can be built 'bottom up', rather than 'top down' as with object orientation, making it much easier for readers to identify the expected behaviour of a given Rust type by simply reading which Traits it implements. In a scientific context we are usually concerned with the organisation, reading and writing of data, commonly adhering to a design philosophy known to as data-oriented design. Traits allow us to inject additional behaviour on types without having to worry about potentially complex inheritance hierarchies.

Rust's runtime includes a test runner, a documentation generator, and a code formatter. As with other Rust features, these are maintained in lock step with the language specification, and with reference to other Rust developments. This imposes universal constraints on all Rust projects, allowing for objectively defined 'good' Rust code, rather than relying on various standards of best practices that vary between projects and organisations. Furthermore the Rust compiler is highly informative, providing hints to developers on best practices for their code, as well as potential sources of bugs or code rot, by notifying users of common anti-patterns or unused variables and functions.

Despite being a young language, Rust already supports a mature ecosystem of libraries for scientific computing with high-level multithreading support [30], numerical data containers [25], and tools for generating interfaces to Python via its C ABI [23]. Many tools are yet to be ported into native Rust, however high quality bindings exist for core tools such as MPI [29], BLAS and LAPACK [5], with simplified build steps often requiring only a few extra lines in the dependency TOML file. The problem with interfacing with tools written in other languages is again related to building software, however Cargo offers tools to build software written in other languages and integrate it with Rust code via the 'build.rs' package, which allows one to leverage existing build systems written for software written in foreign languages. This detracts from the benefits offered by Cargo as a unified package manager and build system, raising similar problems to those encountered when building software in other compiled languages. However, we observe that this remains a concern of the software's developer, who is responsible for providing build scripts for the operating systems and hardware platforms that they wish to support, and from a downstream user perspective their build process remains the same as with pure Rust packages, where the dependency is defined in their dependency TOML file. We also note that Rust is missing key tools for scientific computing, such as a code generation

for GPUs, however with mounting interest in Rust from the scientific computing community this is an active area of development.

1.4 Emerging Developments

Despite the above criticisms, high-level languages as tools for high-performance scientific computing remain an intense area of research and development. 'Mojo' is a new programming language, along with a compiler. It's built as a superset of Python, specifically with the two-language problem in mind. Additionally, it attempts to address the 'three language problem', whereby languages also target exotic hardware such as GPUs and TPUs [20].

Led by a team that includes the original developers of LLVM, Mojo aims to simplify the development of high-performance applications in a Python-like language, that acts as a superset of Python. Moreover, it seeks to make these applications deployable across most hardware and software targets, ensuring compatibility with Python's vast open-source libraries and straightforward build tools.

This is achieved by building on the MLIR compiler infrastructure. MLIR can be thought of as a generalisation of LLVM, catering to CPUs, GPUs, and novel ASICs for AI. The team chose to develop around Python to leverage its extensive existing user base in computational and data sciences.

Currently, Mojo remains a closed-source language and is actively being developed by its parent company, Modular. Thus, even though it appears promising, it's not yet in a state suitable for experimentation. Nevertheless, Mojo showcases the potential of a future programming environment that might definitively 'solve' the problems developers face when selecting a programming environment for academic software.

Installing and Building Software in C++/Fortran

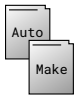
Builds for open source software are often **Readme Driven.** In this common approach developers provide a set of instructions for how to install a project's dependencies and the project itself. Common approaches include:

Dependency Management Methods

1) Simply add all dependencies to source tree of your project. Projects with a large number of dependencies can grow to have millions of lines of code, which can have a drastic effect on compilation times. Large C++ projects for example can take between several minutes to several hours to compile from source

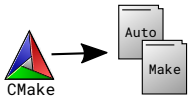
2) Use a system package manager, and source from repositories such as GitHub to install globally. These can then be found by build systems. This makes it difficult to build isolated build environments, and configure builds with different versions, or sets, of dependencies.

Build Methods



1) Developer provided Make and Autotools scripts. lowest barrier to entry but build system will use globally installed dependency libraries, unless alternative provided. Makes multi-platform builds, or those using different software versions challenging.

2) Developer provided CMake scripts, to generate build systems for different environments. Again, reliant on globally installed dependency libraries.



Neither of these methods can check for dependency conflicts, which is left up to the developer to resolve.

Modern Package Management

Modern package managers allow for maximum safety and flexibility. They support multiple operating systems, compilers, build systems, and hardware microarchitetures, and are usually defined by recipes written in a simple scripting language such as Python, and can be used to generate build system scripts such as Makefiles and CMake scripts. The also support dependency tree checking for conflicting requirements, leading to stable builds. Two popular leading tools for HPC in C++ and Fortran are **Spack** and **Conan**.

Spack

+ Supported on Linux and MacOS.
+ Package recipes specified with Python scripts.
+ Over 5000 commonly used packages available.
+ Supports isolated environments, supporting building packages with from specific constraints, and clean global environment.
+ Can target hardware microarchitectures.
+ Growing support for binary package installation, though not available on all hardware targets.
+ Compatible with all major build systems, such as CMake and SCons
- No Windows support.

Conan

+ Supported on Linux, MacOS and Windows
+ Package recipes specified with Python scripts.
+ Supports isolated environments, supporting building packages with from specific constraints, and clean global environment.
+ Support for binary packages, speeding up installation times.
+ Compatible with all major build systems, such as CMake and SCons.

However, we note that even in 2021 modern package managers are still only used in a tiny fraction of all projects. For example Conan, only accounts for 5% of all C++ projects surveyed by JetBrains[1] in comparison to 21% of projects still relying on the system package manager, 26% simply including a dependencies source code as a part of a project's source tree, 23% using the readme driven build of each specific dependency, and 21% installing non-optimised precompiled binaries from the internet. Indeed only a small minority, 22%, used a package manager of any kind, with no solution taking a majority of even this market share. This is in stark contrast to theuniformity of the situation in Rust, in which all projects use Cargo as a build system and package manager and rustc as a compiler.

1. https://www.jetbrains.com/lp/devecosystem-2021/cpp/

Figure 1.2: An overview of building software in other compiled languages.

Designing Software for Fast Algorithms

2.1 The Fast Multipole Method

In this section we introduce the Fast Multipole Method (FMM), a foundational fast algorithm, and the implementation of which is central to our software infrastructure. We begin by describing the FMM, first introduced by Greengard and Rokhlin in the 1980s, in the context of the solution of the N-body potential calculation problem of electrostatics, or gravitation, which gives rise to much of the terminology in this field [14]. We proceed to describe more modern algorithmic approaches, which retain many of the core algorithmic features of the original FMM while being more amenable to software implementation on modern computer hardware [21, 12]. We note that the FMM in its most basic form corresponds to a matrix-vector product, and we therefore conclude by briefly contrasting the FMM with similar methods for computing this quantity, as well as noting methods for computing the approximate inverse of FMM matrices, which correspond to a form of direct solver, termed 'fast direct solvers'.

Consider the following N-body problem which appears in the calculation of electrostatic, or gravitational potentials from a set of point charges, or masses,

$$\phi_j = \sum_{i=1}^{N} K(x_i, x_j) q_i \tag{2.1}$$

Here, q_i is a point charge/mass, corresponding to N particles at positions x_i and the 'kernel' is defined as,

$$K(x, y) = \begin{cases} \log \|x - y\| & \text{in } \mathbb{R}^2 \\ \frac{1}{4\pi\|x-y\|} & \text{in } \mathbb{R}^3 \end{cases} \tag{2.2}$$

Writing the component form (2.2) as its corresponding linear system,

$$\phi = K\mathbf{q} \tag{2.3}$$

We note that the matrix K is *dense*, with non-zero off diagonal elements, and that this implies a global data dependency between all point charges/masses. This global data dependency had previously inhibited numerical methods for N-body problems as a naive application of this matrix requires $O(N^2)$ flops, and an finding an inverse using a linear algebra technique such as LU decomposition or Gaussian

Elimination requires $O(N^3)$ flops. The key insight behind the FMM, and subsequent fast algorithms, was that the interactions between physically distant groups of points could be compressed with a bounded accuracy if the kernel function exhibits amenable properties, specifically if it rapidly decays as the distance between two point sets increases. We demonstrate this in figure 2.1.

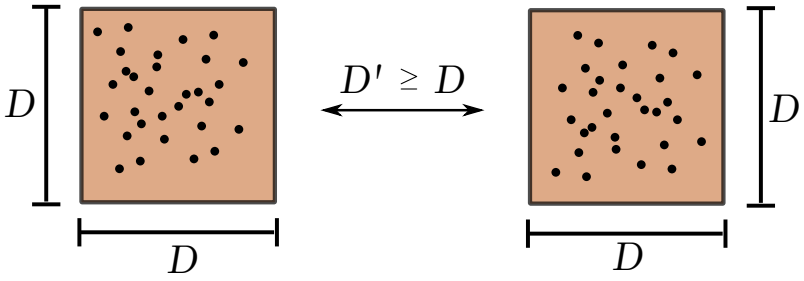


Figure 2.1: Given two boxes in $\mathbb{R}^d$ ($d = 2$ or 3), $\mathcal{B}_1$ and $\mathcal{B}_2$, which each enclose a corresponding set of points. The off-diagonal blocks in the matrix $K_{\mathcal{B}_1\mathcal{B}_2}$ and $K_{\mathcal{B}_2\mathcal{B}_1}$ are considered low rank and amenable to compression for the FMM when the distance separating them is at least equal to their diameter.

From figure 2.1 we see that the FMM relies on a discretisation that allows us to systematically compress far-interactions. To see how we obtain this, we start by placing our problem in a square/cube domain containing all the points, denoting this with Ω. Returning to (2.3), we must calculate the potential vector $\phi \in \mathbb{R}^N$, from a charge/mass vector $\mathbf{q} \in \mathbb{R}^N$ by applying $K \in \mathbb{R}^{N\times N}$. We begin with the situation in figure 2.1, where we have two sets of particles contained in boxes separated by some distance that makes their interaction amenable to compression, the FMM literature often describes such boxes as being 'well separated'. Labelling the boxes as Ω_s and Ω_t, containing 'source' particles $\{y_j\}_{j=1}^M$, with associated charges/masses q_j, and 'target' particles $\{x_i\}_{i=1}^N$ respectively, we seek to evaluate the potential introduced by the source particles at the target particles. As the interaction assumed to be amenable to compression we are able to write an approximation to (2.1) using a low-rank approximation for the kernel in terms of tensor products as,

$$K(x,y) \approx \sum_{p=0}^{P-1} B_p(x)C_p(y), \quad \text{when } x \in \Omega_t,\ y \in \Omega_s \tag{2.4}$$

Where P is called the 'expansion order', or 'interaction rank'. This is equivalent to approximating the kernel with an SVD using its leading singular values. We introduce index sets I_s and I_t which label the points inside Ω_s and Ω_t respectively, and find a generic approximation for the charges/masses,

$$\hat{q}_p = \sum_{j\in I_s} C_p(x_j)q_j,\ p = 0, 1, 2...P-1 \tag{2.5}$$

Using this we can evaluate an approximation to the potential due to these particles in the far-field as,

$$\phi_i \approx \sum_{p=1}^{P-1} B_p(x_i)\hat{q}_p \tag{2.6}$$

In doing so we accelerate (2.1) from $O(ML)$ to $O(P(M+L))$. As long as we choose $P \ll M$ and $P \ll L$, we recover an accelerated matrix vector product for calculating the potential interactions between the set of sources and targets from two well separated boxes. The rapid decay behaviour of the kernel ensures that we can recover the potential in Ω_t with high-accuracy even if P is small. We note that although we represent the sources/targets as corresponding to different particle sets in this derivation they may be equivalent.

We deliberately haven't stated how we calculate B_p or C_p. In Greengard and Rokhlin's original FMM these took the form of analytical multipole and local expansions of the kernel function [14]. To demonstrate this we derive an expansion in the $\mathbb{R}^2$ case, taking c_s and c_t as the centres of Ω_s and Ω_t respectively,

$$\begin{aligned} K(x,y) = \log(x-y) &= \log((x-c_s)-(y-c_s)) \\ &= \log(x-c_s) + \log(1-\frac{y-c_s}{x-c_s}) \\ &= \log(x-c_s) - \sum_{p=1}^{\infty} \frac{1}{p}\frac{(y-c_s)^p}{(x-c_s)^p} \end{aligned} \tag{2.7}$$

where the series converges for $|y-c_s| < |c-c_s|$. We note (2.7) is exactly of the form required with $C_p(y) = -\frac{1}{p}(y-c_s)^p$ and $B_p(x) = (x-c_s)^{-p}$. We define a 'multipole expansion' of the charges in Ω_s as a vector $\hat{\mathsf{q}}^{\mathsf{s}} = \{\hat{q}_p^s\}_{p=0}^{P-1}$,

$$\begin{cases} \hat{q}_0^s = \sum_{j \in I_s} q_j \\ \hat{q}_p^s = \sum_{j \in I_s} -\frac{1}{p}(x_j - c_s)^p q_j, \;\; p = 1,2,3...,P-1 \end{cases} \tag{2.8}$$

The multipole expansion is a representation of the charges in Ω_s and can be truncated to any required precision. We can use the multipole expansion in place of a direct calculation with the particles in Ω_s. As the potential in Ω_t can be written as,

$$\phi(x) = \sum_{j \in I_s} K(x,y) q_j = \log(x-c_s)\hat{q}_0^s + \sum_{p=1}^{\infty} \frac{1}{(x-c_s)^p}\hat{q}_p^s \tag{2.9}$$

Greengard and Rokhlin also define a local expansion centered on Ω_t, that represents the potential due to the sources in Ω_s.

$$\phi(x) = \sum_{l=1}^{\infty} (x-c_t)^l \hat{\phi}_l^t \tag{2.10}$$

with a simple computation to derive the local expansion coefficients $\{\hat{\phi}_p^t\}_{p=0}^{\infty}$ from $\{\hat{q}_p^s\}_{p=0}^{P-1}$ (see app. A).

For our purposes it's useful to write the multipole expansion in linear algebraic terms as a linear map between vectors,

$$\hat{\mathsf{q}}^s = \mathsf{T}_s^{P2M} \mathsf{q}(I_s) \tag{2.11}$$

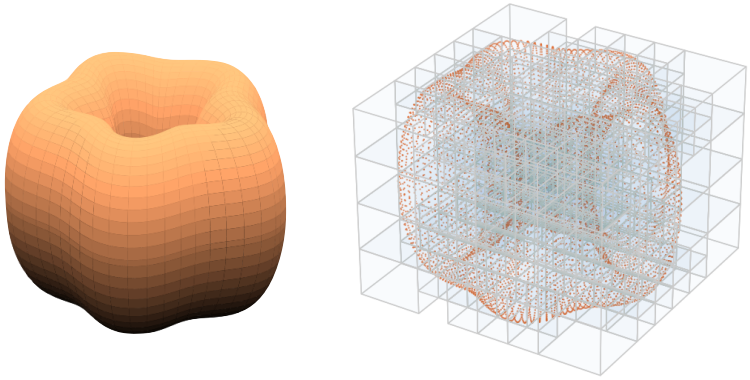


Figure 2.2: An adaptive octree for random point data placed on the surface of a 'wiggly torus' test geometry. The user defines the level of recursion via a threshold for the maximum number of particles in a given node.

where T_s^{P2M} is a $P \times N_s$ matrix, analogously for the local expansion coefficients we can write,

$$\hat{\phi}^t = \mathsf{T}_{t,s}^{M2L} \hat{\mathsf{q}}^s \tag{2.12}$$

where $\mathsf{T}_{t,s}^{M2L}$ is a $P \times P$ matrix, and the calculation of the final potentials as,

$$\phi^t = \mathsf{T}_t^{L2P} \hat{\phi}^t \tag{2.13}$$

where T_t^{L2P} is a $N_t \times P$ matrix. Here we denote each *translation* operator, T^{X2Y}, with a label read as 'X to Y' where L stands for local, M for multipole and P for particle. Written in this form, we observe that one could use a different method to approximate the translation operators than explicit kernel expansions to recover our approach's algorithmic complexity, and this is indeed the main difference between different implementations of the FMM.

We have described how to obtain linear complexity when considering two isolated boxes, however in order to recover this for interactions between *all particles* we rely on a hierarchical partitioning of Ω using a data structure from computer science called a *quadtree* in $\mathbb{R}^2$ or an *octree* in $\mathbb{R}^3$. The defining feature of these data structures is a recursive partition of a bounding box drawn over the region of interest (see fig. 2.2). This 'root node' is subdivided into four equal parts in $\mathbb{R}^2$ and eight equal parts in $\mathbb{R}^3$. These 'child nodes' turn are in turn recursively subdivided until a user defined threshold is reached based on the maximum number of points per leaf node. These trees can be 'adaptive' by allowing for non-uniform leaf node sizes, and 'balanced' to enforce a maximum size constraint between adjacent leaf nodes [32].

In addition to the T^{P2M}, T^{M2L} and T^{L2P} the FMM also require operators that can translate the expansion centre of a multipole or local expansion, T^{L2L}, T^{M2M}, an operator that can add the contribution of a set of points to a given local expansion T^{P2L}, and apply a multipole approximation to a set of points, T^{M2P}, finally we need define a $P2P$ operator, which is short hand for direct kernel evaluations. Algorithm (3) in Appendix C provides a sketch of the full FMM algorithm which combines these operators.

In summary the algorithm consists of two basic steps. During the first step, the upward pass, the tree is traversed in postorder [1]. At the leaves, multipole expansions

[1]The children of a box are visited before the box itself.

are built using T^{P2M}. At each non-leaf node, the multipole expansion is shifted from its children using T^{M2M} and summed. In the second step, the downward pass, the tree is traversed in preorder[2]. The local expansions are computed by first translating the multipole expansions of boxes which are the children of the parents of a box, but are not adjacent to it, using T^{M2L} and are summed, a second contribution is found from a box's parent using T^{L2L}. The sum of these two parts encodes all the contribution from sources particles in boxes with are not adjacent to itself. For non-uniform trees there also may be a contribution from boxes in near field of B, i.e. neighbour children/descendents, calculated using T^{P2L}. This is computed at the leaf level only. Having assembled the far-field contribution at B in its local expansion we evaluate it at the particles it contains using T^{L2P}, combining it with its near interactions from particles in adjacent boxes using $P2P$, as well as boxes which are non-adjacent but for which the multipole expansion still applies using T^{M2P} in a box's near-field, i.e. the children and descendants of adjacent boxes at the same level.

The complexity bound of the FMM algorithm is due to the hierarchical nature of the data structure used in its computation. In the upward pass, each box only considers either itself. This leads to a complexity of $O(N)$, as the number of leaf boxes is bounded by the number of particles. In the downward pass, the number of boxes each box considers when computing translations is bounded by the number of non-adjacent children of its parent box for the T^{M2L} step. As this is a constant (27 in $\mathbb{R}^2$ and 189 in $\mathbb{R}^3$), this step is again $O(N)$ in complexity, resulting in a total complexity of $O(N)$ for this step. In the T^{L2L} and T^{L2P} operators each box considers only itself. For the leaf level computations, the $P2P$,T^{P2L} and T^{M2P} steps are bounded to be contained in boxes in B's near field [3], and T^{P2L} only considers each box B itself. This leads to an overall algorithmic complexity of $O(N)$. We note that this complexity bound is entirely dependent on the properties of the kernel. Highly oscillatory kernels are not as rank-deficient, however corresponding algorithms inspired by the FMM have been developed for them, however here the complexity increases to $O(N\log(N))$ [24]. The conversion of an $O(N^2)$ problem to one that can be solved in $O(N)$ was a groundbreaking discovery, and lead to the admissibility of a large number of scientific problems to computer simulation. As a result the FMM has been described as 'one of the top ten algorithmic discoveries of the twentieth century', taking its place alongside revolutionary algorithms such as Quicksort and the Fast Fourier Transform [10].

2.2 Algebraic FMM Variants

In its original analytical form the applicability of the FMM is limited by the requirement for an explicit multipole and local expansions, as well as a restriction to matrix vector products. Subsequent decades saw the development of 'algebraic' analogues to the original algorithm. These methods are similarly based on a hierarchical partitioning, whether that be of the point data using a recursive tree as with the original FMM [21, 12], or by operating on the matrix implied by the FMM's algorithmic structure directly [15, 7, 9]. The latter methods are collectively known as $\mathcal{H}$-matrix methods. Representing the FMM operation in this way has allowed ex-

[2]The children of a box are visited after the box itself.

[3]In the worst case, for highly non-uniform point distributions, these steps can be subsumed into the $P2P$ step, allowing us to maintain the bound.

tension of the FMM to other matrix computations, such as matrix-matrix products, as well as approximations of its inverse [1]. Notably, many of these methods don't necessarily rely on explicit multipole/local expansions for approximating fields as in the original FMM in the previous section. Examples such as the 'kernel independent FMM' (kiFMM) of Ying and co-authors instead relies on the method of fundamental solutions to approximate the fields and requires only evaluating kernel values, and is applicable to a wide range of kernels from second-order linear non-oscillatory elliptic PDEs with constant coefficients such as the Laplace equation. This approach relies on some analytic considerations, however methods also exist which are interpolatory - such as the 'black box FMM' (bbFMM) of Fong and Darve [12], which similarly only relies on kernel evaluations and a Chebyshev scheme to approximate fields.

In our software we choose to implement the kiFMM of Ying and coauthors, which we explain in detail below adapting the discussion from Section 3 of [21]. This method shares advantages with other algebraic FMM methods, of generality to a large class of problems, as well as opportunities to optimise computer implementations through caching strategies which we explore in Section 2.2. This approach relies on a spatial discretisation of the problem domain via an quad/octree as in the original FMM, as well as the method of fundamental solutions (MFS) for approximating the fields from point charges/masses. This method has been demonstrated to scale well on shared [35] as well as distributed memory [22] systems, achieve similar accuracy to the analytical variant, with relatively low pre-computation required. The underlying data structure of the quad/octree has been a significant area of research and development, with established high-performance methods for their construction in shared/distributed memory environments [32, 31, 8].

Consider the kernels for second-order constant coefficient non-oscillatory PDEs, such as that of the Laplace equation in (2.2). Such kernels satisfy the underlying PDE everywhere except the singularity location, and are smooth away from this singularity. These problems admit a unique solution for interior/exterior Dirichlet boundary value problems (see Appendix D for a demonstration of this for the Laplace equation). The authors of [21] rely on the smoothness and uniqueness of the Dirichlet boundary value problems as basic properties to develop their FMM formulation. The problem setting as before is the calculation of (2.1) for the Laplace kernel, for a set of N point sources y_i, $i = 1...N$, which we will associated with N source densities q_i, an target locations y_j, $j = 1...M$. As before, the source and target locations may coincide. We use an index set I_s^B and I_t^B to identify the sources and targets we are considering in a particular interaction. We assume that our problem is in $\mathbb{R}^3$, however the exposition is essentially the same as in $\mathbb{R}^2$.

Assuming that we have constructed our hierarchical tree partitioning, which may be adaptive, the first step is to approximate the fields due to particles contained in each leaf box. We specify more concretely that for a given box, B, its 'near field range', $\mathcal{N}^B$, is the set of 27 boxes at the same level of a tree which are adjacent to it, i.e. they share a corner, face or edge as well as B itself. Its 'far field range', $\mathcal{F}^B$, is simply the boxes which are the complement of this.

We approximate the potential in $\mathcal{F}^B$ from the source densities $\{q_i : i \in B\}$ in B using the potential from an *equivalent density distribution*, $q^{B,u}$, supported at prescribed locations $y^{B,u}$ (see the left box in Figure 2.3). Where $q^{B,u}$ is called the *upward equivalent density* and $y^{B,u}$ is called the *upward equivalent surface*. This amounts to representing the potential with a 'single layer' potential [18],

$$\phi(x) := \int_{y \in y^{B,u}} q^{B,u}(y) K(x,y) ds(y), \;\; x \in \mathbb{R}^3 \setminus y^{B,u} \tag{2.14}$$

To guarantee the smoothness of the potential produced by $q^{B,u}$ in the far-field, its support $y^{B,u}$ must not overlap with $\mathcal{F}^B$ due to the singularities in this integral from the kernel function when evaluated on the equivalent surface. Secondly, we note that the equivalent surface must enclose B from the definition of the single layer potential [18]. We see that the equivalent surface must be placed in between the box and the boundary of $\mathcal{F}^B$.

The potential induced by our equivalent densities and upward equivalent surface satisfies the Laplace equation. Therefore, due to the uniqueness of the exterior Dirichlet boundary value problem for this kernel (as well of kernels of a similar type), we reason that the potential calculated using (2.1) directly from the source particles must be equivalent to that calculated using (2.14) in $\mathcal{F}^B$, or anywhere between $y^{B,u}$ and $\mathcal{F}^B$. Thus we place an intermediate surface called the *upward check surface* between $\mathcal{F}^B$ and $y^{B,u}$, denoting it with $x^{B,u}$. The potential computed at this surface is called the *upward check potential*, which we denote by $\phi^{B,u}$.

We write this as,

$$\int_{y^{B,u}} K(x,y) q^{B,u} dy = \sum_{i \in I_s^B} K(x,y_i) q_i = \phi^{B,u} \;\; \text{, for any } x \in x^{B,u} \tag{2.15}$$

Solving for the equivalent densities is an equivalent method of approximating the far-field potential induced by the points in B. We identify this with a *multipole expansion*. Now considering source densities which are not contained in a box B (see right box of Figure 2.3) but in its $\mathcal{F}^B$, we can construct an equivalence for local expansions using a similar approach. To ensure the existence of the *downward equivalent densities* $q^{B,d}$, the *downward equivalent surface*, $y^{B,d}$, must be located between B and $\mathcal{F}^B$ and the potentials generated by the source points are matched to those generated by the equivalent points on a *downward check surface*, $x^{B,d}$ that encloses the box and is itself enclosed by $y^{B,d}$ in order to calculate a *downward check potential* $\phi^{B,d}$.

$$\int_{y^{B,d}} K(x,y) q^{B,d} dy = \sum_{i \in I_s^{\mathcal{F}^B}} K(x,y_i) q_i = \phi^{B,d} \;\; \text{for any } x \in x^{B,d} \tag{2.16}$$

In $\mathbb{R}^3$ the authors chose to represent the equivalent and check surfaces as cubes, with the equivalent/check points arranged regularly over the surfaces. We choose the same as it leads to implementation benefits when designing the field translation operators (see Section 1.3 for how we take advantage of this).

Equations (2.15) and (2.16) are examples of *Fredholm integral equations of the first kind*, the inversion of which is ill-posed. To solve these equations, we must first discretise, for which we made use of the *Method of Fundamental Solutions* (MFS), a technique whereby the potential is approximated by a linear combination of kernel function evaluations at a set of discrete points with associated densities,

$$\phi(x) \approx \phi^N(x) = \sum_{y_i \in y^{B,u}} K(x,y_i) q_i \tag{2.17}$$

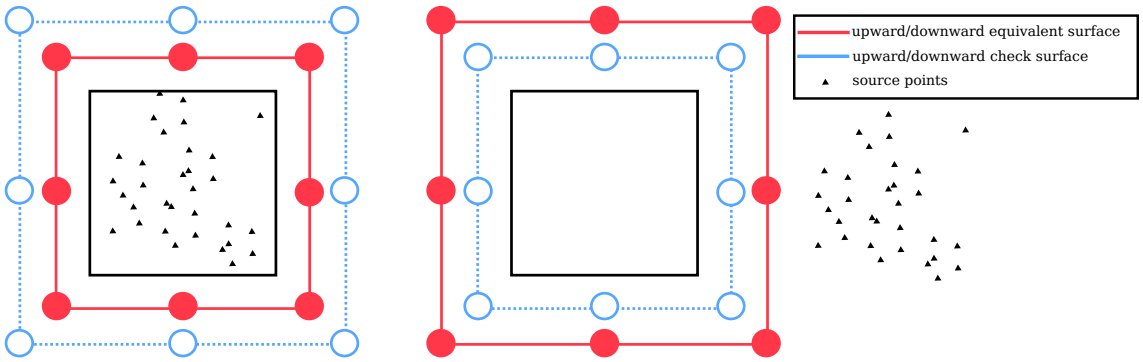


Figure 2.3: We illustrate the equivalent/check surfaces and associated boxes in $\mathbb{R}^3$, where we show a corresponding cross section. In the first figure, we illustrate the situation in the 'P2M' operation, where we are trying to construct an approximation to the potential generated by points in a box by matching it to that generated by a set of equivalent density points placed on a fictitious surface enclosing it. In the second figure we illustrate the 'P2L' operation, where we are now trying to construct an approximation to the potential generated by points in a box's far-field within the box.

Where N denotes the number of equivalent points of density are placed on the equivalent surface. To see how this approximates the single-layer operator we used above to calculate the potential we simply replace the continuous density functions in (2.14) with $\sum_{j=1}^{N} q_i \delta(y - y_j)$, recovering (2.17). Writing (2.15) or (2.16) in matrix form reflecting the discretisation via MFS,

$$Kq = \phi \tag{2.18}$$

We can solve the problem with Tikhonov regularisation,

$$q = (\alpha I + K^* K)^{-1} K^* \phi \tag{2.19}$$

which converts it into a *Fredholm integral equation of the second kind* which are well-posed. The regularisation parameter α is found empirically. Computing (2.15) is equivalent to the T^{P2M} operator described in the previous section for the analytical FMM, (2.16) is equivalent to an T^{P2L} operator which was not explicitly described. As before, these operators take a set of charges, and create expansions that allow us to evaluate their potential in the far-field, however the method of creating the expansions is clearly significantly different. The most notable change being that our above formulation *does not require explicit analytical expansions of the kernel function*, only making use of kernel evaluations.

To complete the kiFMM we need analogues to the field translation operators, T^{M2L}, T^{M2M}, T^{L2L}. We illustrate the required surfaces for each of these operators in Figure 2.4. For T^{M2M} we must translate the upward equivalent density of A to the centre of its parent box B, solving the following equation for $q^{B,u}$,

$$\int_{y^{B,u}} K(x,y)\phi^{B,u}(y)dy = \int_{y^{A,u}} K(x,y)\phi^{A,u}(y)dy, \text{ for all } x \in x^{B,u} \tag{2.20}$$

As before, $y^{A,u}$ must enclose the child box A. During the downward pass, we must evaluate the downward equivalent density, corresponding to a local expansion, from

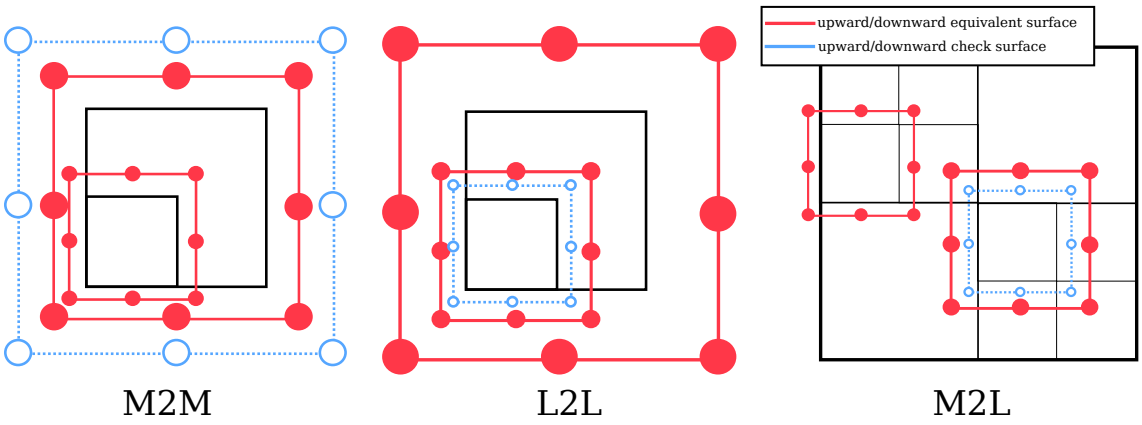


Figure 2.4: As in Figure 2.3, we illustrate the translations as cross sections.

the multipole expansions of non-adjacent boxes in B's parent neighbour's children. We similarly write down for T^{M2L},

$$\int_{y^{B,d}} K(x,y)\phi^{B,d}(y)dy = \int_{y^{A,u}} K(x,y)\phi^{A,u}(y)dy, \text{ for all } x \in x^{B,d} \tag{2.21}$$

Finally, the local expansions of each box B must be translated to the centre of each of its children A,

$$\int_{y^{B,d}} K(x,y)\phi^{B,d}(y)dy = \int_{y^{A,d}} K(x,y)\phi^{A,d}(y)dy, \text{ for all } x \in x^{B,d} \tag{2.22}$$

For each of these translation operators we use the discretisation based on MFS discussed above, and solve using Tikhonov regularisation. For the T^{L2P}, $P2P$ and T^{M2P} operators we simply use direct kernel evaluations at the target points with respect to the equivalent densities represented by the multipole or local expansion coefficients or the true densities at the source points.

2.3 Implementation Challenges for the Kernel Independent Fast Multipole Method

In this section we introduce analytical features of the kernels compatible with the kiFMM that have consequences for computer implementations, such as the ability to pre-compute and cache matrices that correspond to translation operators. We also discuss the octree datastructure in more detail, highlighting its key bottlenecks, concluding with a discussion on implied computational and storage complexity.

The defining feature of the kernels compatible with the FMM is that they they display a rapid decay behaviour as the distance between interactions increases, known formally as *asymptotic smoothness*. A kernel is described as asymptotically smooth if there are constants $C_{\text{as1}}, C_{\text{as2}} \in \mathbb{R}_{>0}$ that satisfy,

$$\|\partial_x^\alpha \partial_y^\beta K(x,y) \leq C_{\text{as1}}(C_{\text{as2}}\|\|x-y\|_2)^{-|a|-|b|}\alpha + \beta|K(x,y)| \tag{2.23}$$

for all multi-indices $\alpha, \beta \in \mathbb{N}_0^3$ and all $x, y \in \mathbb{R}^3$. It's this smoothness that allows us to limit the definition of $\mathcal{N}^B$ to its neighbouring boxes in the FMM. This leads to $|\mathcal{N}^B| = 3^3 = 27$ in $\mathbb{R}^3$.

Table 2.1: Number of near and far field boxes for a given box B, depending on the type of kernel we're considering.

Kernel Type	boxes in $\mathcal{N}^B$	boxes in $\mathcal{F}^B$
trans. invar.	≤ 27 per box	≤ 316 per level
trans. invar. + homog.	≤ 27 per box	≤ 316 in total

Far-field interactions are handled by the T^{M2L} step, we saw above how this is limited to interactions with boxes which are children of B's parent, but non-adjacent to B. Therefore we see that in a multi-level scheme the $\mathcal{N}^B$ contains all of the $6^3 = 216$ near and far field interactions of B's children. As far-field interactions necessarily exclude the near field, this leads to a maximum of $6^3 - 3^3 = 189$ boxes in each child box's far field that must be considered, each corresponding to a single T^{M2L}. This is one of the *key bottlenecks* of efficient FMM implementations. There are numerous ways of *sparsifying* this step, either by using some additional numerical technique such as an SVD to compress the matrices corresponding to T^{M2L} as they are known to be low-rank, or by using an exact method such as an Fast Fourier Transform (FFT) - which relies on our decision to place the equivalent densities on a regular grid as then the T^{M2L} can be re-interpreted as a convolution. We discuss the trade-offs of these approaches in detail in Section 2.3, as optimal implementations of this sparsification are of central importance to our software's performance.

We note that precomputations can be further reduced if kernels also exhibit translational invariance,

$$K(x, y) = K(x + v, y + v) \tag{2.24}$$

where $v \in \mathbb{R}^3$, we can compute T^{M2L} for a restricted subset of the total possible interactions for the eight child boxes. Indeed, the union of all possible far-field interactions for the eight child boxes gives $7^3 - 3^3 = 316$ possible interactions. This comes from the fact some subset of translation vectors are non-overlapping for each child, resulting in a total of 7^3 total interactions for all children, subtracting the near-field interactions which are the same for all children gives the result. Furthermore this means that we must pre-compute just 316 unique T^{M2L} if our kernel has these favourable properties. This is of course dependent on choosing the same equivalent and check surfaces, and density locations, for each box. If so, these can be pre-computed and cached for a given expansion order, corresponding to all possible T^{M2L} at each level.

If an asymptotically smooth, translationally invariant, kernel is also homogeneous to degree n,

$$K(\alpha r) = \alpha^n K(r) \tag{2.25}$$

where $\alpha \in \mathbb{R}$, implies that when we scale the distance between a source and target box by α the potential is scaled by a factor of α^n, where n depends on the kernel function in question, we can compute T^{M2L} for a *single* level of the tree, and scale the result at subsequent levels. This is summarised in Table 2.1.

As we've chosen the location of point densities to be fixed relative to each box, the evaluation of (2.20) and (2.22) can equivalently be pre-computed. In this case,

there are just 8 unique matrices corresponding to T^{M2M} and T^{L2L}, corresponding to the relative positions between a parent box and its children.

We observe the T^{P2M} is calculated for all the nodes of a quad/octree in a discretisation, as observed in the previous section the level of discretisation is either specified by the user, or set by a user defined constraint N_{crit} which specifies a maximum number of points contained within each leaf box. This leads to $O(N \cdot N_{\text{crit}})$ to find the check potentials for each leaf, therefore we note that N_{crit} must be kept small to ensure linear complexity is maintained for this operation. To compute the equivalent density, we rely on an application of the inverted integral operator in (2.15). The size of this matrix is determined by the expansion order P, which determines the number of quadrature points taken on the equivalent/check surfaces. As the points are taken to be placed on a regular grid over this surface, the number of quadrature points N_{quad} relates to P as,

$$N_{\text{quad}} = 6(P-1)^2 + 2 \tag{2.26}$$

Therefore, this matrix vector product is of $O((P-1)^4)$, leading to $O(N((P-1)^4 + N_{\text{crit}}))$ complexity for the entire operator. The complexity of T^{P2L} is the same, as it amounts to the same calculation, albeit to form a local expansion. The precomputation of the translation operator requires an SVD for the integral operator in (2.15), the complexity of which is dependent on the algorithm chosen, with implementation specific optimisations. However, for a $\mathbb{R}^{m\times n}$ matrix where m and n are of similar size, the 'DGESVD' implementation of LAPACK, which we use in our software framework, has a complexity of $O(n^3)$ therefore this precomputation is of $O((P-1)^6)$. The storage required is of $O((P-1)^4)$ for the inverted matrix.

Applying similar analysis for computing (2.20) and (2.22), we arrive at computational complexities of $O(N \cdot (P-1)^4)$ for calculating the check potentials, and then evaluating the equivalent densities via two matrix-vector products. For the M2L step, if we do not choose to implement any sparsification, we also obtain $O(N \cdot (P-1)^4)$ for the asymptotic complexity of applying T^{M2L}, however here there are large lurking constants. Specifically, each box B has to apply an T^{M2L} up to 189 times in $\mathbb{R}^3$ or 27 times in $\mathbb{R}^2$. The storage complexity is determined by the kernel, as shown by table 2.1, the properties of the kernel can reduce the number of precomputations that can be cached.

Similar large constants lurk in the leaf level computations $P2P$, T^{M2P}, T^{P2L}. These all consider boxes contained in B's near field. For non-uniform point distribution there could be very large number of boxes contained in the near field. Therefore to reduce this we enforce a 'balancing' condition, as mentioned above. The most common condition is a '2:1 balance', whereby neighbouring boxes are restricted to be no more than twice as large as each other [22]. This restricts the size of the number of adjacent boxes to B to be at most 52 in $\mathbb{R}^3$. For these boxes we will calculate the potential for points in B using kernel evaluations directly via $P2P$. For the remainder of its near field, we'll use the T^{M2P} operator. The first T^{M2P} takes boxes in its near field for which the multipole expansion still applies at B, this means that leaf boxes that coincide with B's neighbours children that are non-adjacent to B. At most there are 156 such boxes in $\mathbb{R}^3$. There may also be an additional contribution to the local expansion at B not captured in T^{L2L} and T^{M2L}, from boxes in the near-field of B's parent which are the leaf level, and considered in the far-field of B itself, and are the level of B's parent. These contributions are found via T^{P2L}. With our balancing condition there are at most 19 such boxes in $\mathbb{R}^3$.

We note that for uniformly refined trees T^{M2P} and T^{P2L} aren't calculated as their corresponding interactions are subsumed into $P2P$. The $P2P$ operator is therefore of $O(52N \cdot N_{\text{crit}})$, with no storage cost as this is applied directly to points. Similarly, the complexity of T^{M2P} is $O(156N \cdot (P-1)^2 N_{\text{crit}})$, T^{L2P} is of $O(N(P-1)^2 \cdot N_{\text{crit}})$. The complexity of T^{P2L} is $O(19N \cdot ((P-1)^4 + N_{\text{crit}}))$, where we include the cost of calculating the check potential as in T^{P2M}. The complexities of each step are summarised in table 2.2.

In practice, the largest bottlenecks are the $P2P$ and T^{M2L} steps. When implemented naively, T^{M2L} contains a large number of BLAS level 2 operations, and is calculated for every non-leaf box. An obvious optimisation is to re-order data such that a single BLAS level 3 operation is computed for each box. This increases the ratio of computations to memory accesses by converting a series of matrix-vectors product into a single matrix-matrix product for each box. Additionally, the SVD taken for the integral operators above can be cut-off due to the low-rank nature of the above operators, leading to a reduced storage and application complexity. We explore the rank behaviour in more detail in Chapter 3 for the Laplace kernel. If instead one chooses to take an FFT, which offers lower complexity for each T^{M2L} while being an exact translation, one has to compute the FFT corresponding to each unique translation which as we've seen is kernel dependent, and perform an inverse FFT to evaluate the check potentials at the equivalent density points. This leads to a lower overall complexity for both compute and storage - however, as we observe in Section 3, memory accesses are critical to performant implementations. The performance of the $P2P$, T^{M2P} and T^{L2P} operators are determined most significantly by the ability to rapidly evaluate the kernel function over sets of points. This is a ripe area for compute optimisations, such as GPU off-loading, or developing vectorised CPU code, and is therefore highly-dependent on the available computational resources and their respective architectures. Practical implementations should be flexible enough to for developers to 'plug-in' different implementations in order to experiment with new hardware.

The second major bottleneck with the kiFMM and FMMs more generally, especially in a distributed setting, is the quad/octree datastructure. The tree is crucial to performance as being able to rapidly query, and communicate via the tree, especially in a distributed setting will determine the overall runtime, as communication latency is a leading order variable in the distributed FMM's complexity [36]. Though there has been significant scholarship in developing high-performance tree libraries for parallel and distributed settings [8, 32, 31], we observe that relatively little work has been done to examine and develop tree libraries with the FMM explicitly in mind, we explore potential optimisations in Chapter 3. Specifically, the downward pass involves operators that summarise the global interaction for a given box, and will necessarily involve communication across node boundaries in a distributed setting. However, communication is not 'global' in the sense that B's interactions will fall into a halo represented by its parent's neighbours, therefore optimising the specification of communicators when using MPI can potentially lead to significant differences in overall algorithm runtimes. Additionally, the broad applicability of parallel trees necessitates a design of software that is relatively de-coupled from the FMM code to encourage adoption in other communities. These two concerns determine our own tree library presented in the following chapter.

Table 2.2: Asmyptotic complexities of each kiFMM operator in $\mathbb{R}^3$, with constants left to demonstrate relative contrast. For T^{M2L} we provide a few different complexity estimates, starting with 'naive' implementation which applies the inverted integral equation (2.21) up to 189 for each box as a matrix vector product, with no sparsification. Then the complexities with respect to different kernel properties. For an T^{M2L} sparsified via the SVD we indicate a cut-off rank with κ, for an FFT based approach as we are often working with real data for the densities we can retain only half the calculated frequencies to save on storage costs. For the T^{P2L}, T^{M2P} and $P2P$ operators we provide estimates for 2:1 balanced trees, as we restrict ourselves to this a practical implementation.

Operator	Computational Complexity	Storage Complexity
T^{P2M}	$O(N(36(P-1)^4 + N_{\text{crit}}))$	$O(36(P-1)^4)$
T^{M2M}	$O(36N(P-1)^4)$	$O(8 \cdot 36(P-1)^4)$
T^{L2L}	$O(36N(P-1)^4)$	$O(8 \cdot 36(P-1)^4)$
T^{M2L}_{naive}	$O(189N \cdot 36(P-1)^4)$	$O(189N \cdot 36(P-1)^4)$
$T^{M2L}_{\text{trans. inv.}}$	$O(189N \cdot 36(P-1)^4)$	$O(316 \log(N) \cdot 36(P-1)^4)$
$T^{M2L}_{\text{homog+trans. inv.}}$	$O(189N \cdot 36(P-1)^4)$	$O(316 \cdot 36(P-1)^4)$
$T^{M2L}_{\text{homog+trans. inv. + SVD}}$	$O(189N \cdot 6(P-1)^2 \cdot \kappa)$	$O(316 \cdot 6(P-1)^2 \cdot \kappa)$
$T^{M2L}_{\text{homog+trans. inv. + FFT}}$	$O(189N \cdot 4(P-1)\log(P-1))$	$O(316 \cdot 36(P-1)^4)$
$T^{M2L}_{\text{homog+trans. inv. + Real FFT}}$	$O(189N \cdot 4(P-1)\log(P-1))$	$O(316 \cdot 18(P-1)^4)$
T^{L2P}	$O(N(36(P-1)^4 + N_{\text{crit}}))$	$O(1)$
$T^{P2L}_{\text{balanced}}$	$O(19N \cdot (36(P-1)^4 + N_{\text{crit}}))$	$O(1)$
$T^{M2P}_{\text{balanced}}$	$O(156N \cdot (36(P-1)^4))$	$O(1)$
$P2P_{\text{balanced}}$	$O(52N \cdot N_{\text{crit}})$	$O(1)$

2.4 Building for Re-Use in Fast Algorithm Software

High performance software for the core data structure of quad/octrees is essential to all fast algorithms. Indeed, these hierarchical discretisations are found across scientific computing applications [REFERENCES FOR APPLICATIONS OF TREES]. Similarly, methods for sparsifying the T^{M2L} operator, whether that be via an SVD or FFT, can also be re-used in the implementation of fast direct solvers, or alternative algebraic FMMs such as the bbFMM of Fong and Darve [12]. The same is true of software for evaluating kernel functions in $P2P$ operator. We identify the decoupling of these separate components to be rare in other softwares for fast algorithms [22, 35, 6], which are often presented as a monolith to users, making it difficult for developers to extend or adapt this software to their use-case.

SOLID

Open/close - software entities should be open for extension, but close for modification.. - implemented by default with traits

Interface segragation - no code should depnd on methods it doesn't use - i.e. interfaces should be split up so that each one only specifies methods that a caller actually requires

Dependency inversion - high level modules should be independent of low-leve implementation details

Single responsibility - a module/software unit is responsible only for one thing

- liskov substitution - an object/class should be replacable by a sub class without breaking the code.

Rust trait coherence and orphan rules

- coherence - each trait has only one implementation for a given type. - Rust refuses to compile code with multiple implementations - but incoherence may occur far away from crates with conflicting implementations - e.g. hashtables implement hash on your types

- orphan rules - overlap rules - prevent you from writing two impl blocks that overlap, i.e. apply to some of the same types

- orphan rules - forbid you from writing an impl where both the trait and the type are defined in a different crate - prevent dependency hell, where multiple crates write conflicting impls - allow crates to change their impls without it being considered a breaking change, as downstream users may have overlapping impls.

- either the trait or the type must be from the same crate.

Our software is designed to be de-coupled as independent sub-components. We achieve this using Cargo, which allows the development of 'workspaces'. This is a modern language feature, without a direct parallel provided in competitors such as Fortran or C/C++. With workspaces, all sub-packages (known as 'crates' in Rust) share a common build directory - speeding up build compile times, and ensuring that dependencies are uniformly versioned across all crates, specified by a root level TOML dependency file. Indeed, crates within a workspace can be specified to be binaries, libraries, or both, allowing for sub-components to easily be deployed independently or used as a dependency by a downstream developer.

- Explain in detail how traits work to specify shared behaviour in comparison to oop

- what are the trad-offs with traits (orphan rule) - what are the benefits (implement for non-local types) - e.g. developers can change our kernel/translation

implementations - implement open/close principle - composition - diamond problem, e.g. multiple inheritors which implement a base class, is not a thing. - all the polymorphism is compile time (runtime via trait objects?) - I don't think this is a big criticism really, relatively similar. - FMM/Fast algorithms naturally fall into sub-components - e.g. trees, translation operators, share large parts of algorithm structure. - packaging data with behaviour as with oop is difficult, as data structures are often very different. - much easier to implement a trait over a type, vs creating a completely new class for each type - clearer to glue together using trait contracts.

- How the design of, say translation operators, is relatively generic and plug and play. - same for P2P, via traits as well, can plug in a GPU implementation say.

Open Work Streams

3.1 Distributed Octrees

- Implementation details of octrees, and overview of current research. - how are we explicitly doing Morton encodings? - how are parallel trees constructed - global sorts for balancing - hyksort, and its importance

- Our octree algorithm and software - bottlenecks for fast algorithm implementation, and potential communicator optimisations - a single node benchmark at the very least, can defer multi-node one for now. - how are single/multinode tree implementations made easier with Rust traits?

3.2 Fast Field Translations

- take stuff from M2L writeup on transfer vectors here

3.2.1 SVD Field Translations

- Can largely copy discussion from M2L write-up for specifics on how this works.

- BLAS3 optimisation for this algorithm, again taken from write-up.

- offer a benchmark for a test problem, which can be contrasted with FFT.

- Rank experiment for Laplace, different orders, where is the cut-off, and what is the accuracy?

- Can this be improved by analytical considerations as well? What are they? The explicit rank behaviour of kernels is not known, even numerically, if this was investigated in more detail we could inform the design of an SVD based compression.

3.2.2 FFT Field Translations

- Use M2L write-up

- offer a benchmark from exafmm-t, and a comment on how it works in terms of memory ordering

- Go through a new algorithm, and how this could help. I already have the discussion on transfer vectors from before and don't need to re-write it. - offer a benchmark for same test problem using our dummy software, as this is the best I can do for now - document the issue with memory ordering, and how this is rectified with the new algorithm.

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, a distributed load balanced octree library designed for high-performance, as well as significant inroads to a distributed FMM based on this.

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. 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, as well as experiment with the communicator optimisations outlined above for parallel distributed trees.

If time permits, we hope to use our software infrastructure to implement an example of a fast direct solver, and benchmark this with a computation of a vector kernel, such as the Time Harmonic Maxwell Equations.

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

Working in the setting in which we derived the multipole expansion in equation (2.9),

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

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 [31]. 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.

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

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 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

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 3 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

Well Posedness of Dirichlet BVP For Laplace Equation

Consider the Laplace equation,

$$\Delta\phi = 0 \tag{D.1}$$

Solutions of which are described as 'harmonic'. The fundamental solutions, or kernel, (2.2) are harmonic in $\mathbb{R}^n \setminus \{y\}$. To formulate a boundary value problem for the Laplace equation, consider a bounded domain $\Omega_- \in \mathbb{R}^n$ which supports C^2 functions, with a boundary $\partial\Omega_-$, its open complement $\Omega_+ := \mathbb{R}^n \setminus \bar{\Omega_-}$, as well as a unit normal n pointing outwards into its exterior Ω_+. We can define two prototypical boundary value problems based on a Dirichlet boundary condition,

Definition D.0.1 (Exterior Dirichlet Problem) *Find a function $\phi \in C^2(\Omega_+) \cap C(\bar{\Omega}_=)$ which is harmonic over Ω_+ and satisfies,*

$$\phi = f, \ \ on\ \partial\Omega_-$$

where f is a given continuous function. For $|x| \to \infty$ it's required that,

$$\phi(x) = O(1)$$

if $m = 2$, and

$$\phi(x) = o(1)$$

if $m = 3$, uniformly in all directions $x/|x|$

Definition D.0.2 (Interior Dirichlet Problem) *Find a function $u \in C^2(\Omega_-) \cap C(\bar{\Omega}_-)$ which is harmonic over Ω_- and satisfies,*

$$\phi = f, \ \ on\ \partial\Omega_-$$

where f is a given continuous function.

Problems of this form appear with great frequency in Physics and Engineering. They appear in electrostatics, heat flow, and fluid flow and many many more fields. We want to establish well-posedness, i.e. uniqueness, in the solution of each of these problems.

Theorem D.0.1 (Dirichlet Problems) *The interior and exterior Dirichlet problems are unique.*

Proof:

Let ϕ_1 and ϕ_2 be two harmonic functions in some region Ω, satisfying the Dirichlet boundary condition on $\partial\Omega$. Then $\phi := \phi_1 - \phi_2$ is also harmonic with $\phi = 0$ Dirichlet boundary conditions. Using the minimum-maximum principle we see that $\phi \equiv 0$ in Ω. As ϕ is of $O(1)$ (or $o(1)$, depending on dimension) in the exterior uniformly in all directions, we see that $\phi \equiv 0$ in the exterior too. Thus proving our claim.

Bibliography

[1] Sivaram Ambikasaran and Eric Darve. "The inverse fast multipole method". In: *arXiv preprint arXiv:1407.1572* (2014).

[2] *B2: makes it easy to build C++ projects, everywhere*. 2022. URL: `https://github.com/boostorg/build`.

[3] *Bazel - a fast, scalable, multi-language and extensible build system*. Version 5.3.2. 2022. URL: `https://github.com/bazelbuild/bazel`.

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

[5] *BLAS and LAPACK for Rust*. 2022. URL: `https://github.com/blas-lapack-rs`.

[6] Steffen Boerm et al. *H2Lib: A software for hierarchical and H2 matrices*. Version 3.0.0. URL: `https://github.com/H2Lib/H2Lib`.

[7] Steffen Börm, Lars Grasedyck, and Wolfgang Hackbusch. "Introduction to hierarchical matrices with applications". In: *Engineering analysis with boundary elements* 27.5 (2003), pp. 405–422.

[8] Carsten Burstedde, Lucas C. Wilcox, and Omar Ghattas. "`p4est`: Scalable Algorithms for Parallel Adaptive Mesh Refinement on Forests of Octrees". In: *SIAM Journal on Scientific Computing* 33.3 (2011), pp. 1103–1133. DOI: `10.1137/100791634`.

[9] Shiv Chandrasekaran et al. "A fast solver for HSS representations via sparse matrices". In: *SIAM Journal on Matrix Analysis and Applications* 29.1 (2007), pp. 67–81.

[10] Barry A Cipra. "The best of the 20th century: Editors name top 10 algorithms". In: *SIAM news* 33.4 (2000), pp. 1–2.

[11] *Conan - The open-source C/C++ package manager*. Version 1.53.0. 2022. URL: `https://github.com/conan-io/conan`.

[12] William Fong and Eric Darve. "The black-box fast multipole method". In: *Journal of Computational Physics* 228.23 (2009), pp. 8712–8725.

[13] *Fortran Package Manager (fpm)*. Version 0.7.0. 2022. URL: `https://github.com/fortran-lang/fpm`.

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

[15] Wolfgang Hackbusch. "A sparse matrix arithmetic based on H-matrices. part i: Introduction to H-matrices". In: *Computing* 62.2 (1999), pp. 89–108.

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

[17] 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.

[18] Rainer Kress. *Linear integral equations*. Vol. 82. 2014, pp. i–412. ISBN: 9781461495925. DOI: 10.1007/978-1-4614-9593-2.

[19] 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.

[20] Chris Lattner. *Mojo - a new programming language for all AI developers*. https://www.modular.com/mojo. 2023.

[21] 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.

[22] Dhairya Malhotra and George Biros. "PVFMM: A parallel kernel independent FMM for particle and volume potentials". In: *Communications in Computational Physics* 18.3 (2015), pp. 808–830.

[23] *Maturin*. Version 0.13.0. 2022. URL: https://github.com/PyO3/maturin.

[24] Matthias Messner, Martin Schanz, and Eric Darve. "Fast directional multilevel summation for oscillatory kernels based on Chebyshev interpolation". In: *Journal of Computational Physics* 231.4 (2012), pp. 1175–1196.

[25] *ndarray: an N-dimensional array with array views, multidimensional slicing, and efficient operations*. Version 0.15.6. 2022. URL: https://github.com/rust-ndarray/ndarray.

[26] *SCons - a software construction tool*. Version 4.4.0. 2022. URL: https://github.com/SCons/scons.

[27] Craig Scott. *Professional CMake: A Practical Guide*. 2018.

[28] *Spack - A flexible package manager that supports multiple versions, configurations, platforms, and compilers.* Version 0.18.1. 2022. URL: https://github.com/spack/spack.

[29] Benedikt Steinbusch and Andrew Gaspar et al. *RSMPI: MPI bindings for Rust*. Version 0.5.4. 2018. URL: https://github.com/rsmpi/rsmpi.

[30] Josh Stone and Niko Matsakis et. al. *Rayon: A data parallelism library for Rust*. Version 1.5.3. 2022. URL: https://github.com/rayon-rs/rayon.

[31] 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.

[32] 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.

[33] *The Meson Build System*. Version 0.63.3. 2022. URL: https://github.com/mesonbuild/meson.

[34] *VCPKG - C++ Library Manager for Windows, Linux, and MacOS*. Version 2022.10.19. 2022. URL: https://github.com/microsoft/vcpkg.

[35] Tingyu Wang, Rio Yokota, and Lorena A Barba. "ExaFMM: a high-performance fast multipole method library with C++ and Python interfaces". In: *Journal of Open Source Software* 6.61 (2021), p. 3145.

[36] Rio Yokota, George Turkiyyah, and David Keyes. "Communication complexity of the fast multipole method and its algebraic variants". In: *Supercomputing Frontiers and Innovations* 1.1 (2014), pp. 62–83. ISSN: 23138734. DOI: 10.14529/jsfi140104. arXiv: 1406.1974.