Modern Software Approaches For Fast Algorithms

Srinath Kailasa

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

Department of Mathematics
University College London
September, 2023

Declaration

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

Abstract

Software has come to be a central asset produced during computational science research. Projects that build off the research software outputs of external groups rely on the software implementing proper engineering practice, with code that is well documented, well tested, and easily extensible. As a result research software produced in the course of scientific discovery has become an object of study itself, and successful scientific software projects that operate with performance across different software and hardware platforms, shared and distributed memory systems, can have a dramatic impact on the research ecosystem as a whole. Examples include projects such as the SciPy and NumPy projects in Python, OpenMPI for distributed memory computing, or the package manager 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 optimized code that must be deployed across a myriad of hardware and software platforms. Moreover, there is a pressing need for comprehensive documentation and rigorous testing to ensure reproducibility. Given that many of these softwares arise within doctoral programs or other short-term projects, there is a tendency to tailor software development to showcase a specific project's objectives. Whether that be to demonstrate a convergence result of a specific methodological improvement, or offer a new benchmark implementation of an algorithm. Consequently, once the principal results are achieved these software projects often become orphaned, lack compatibility with a range of development platforms, or aren't adaptable to related challenges and subsequent research by other teams.

A recent survey of 5000 software tools published in computational science papers featured in ACM publications found that repositories for computational science papers had a median active development span of a mere 15 days. Alarmingly, one third of these repositories had a life cycle of less than one day [9]. 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 [11]. 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 [3]. 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 [10]. 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 optimized 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.

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 [10].

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 build systems this acts as an impediment.

[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.

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 helpful 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.

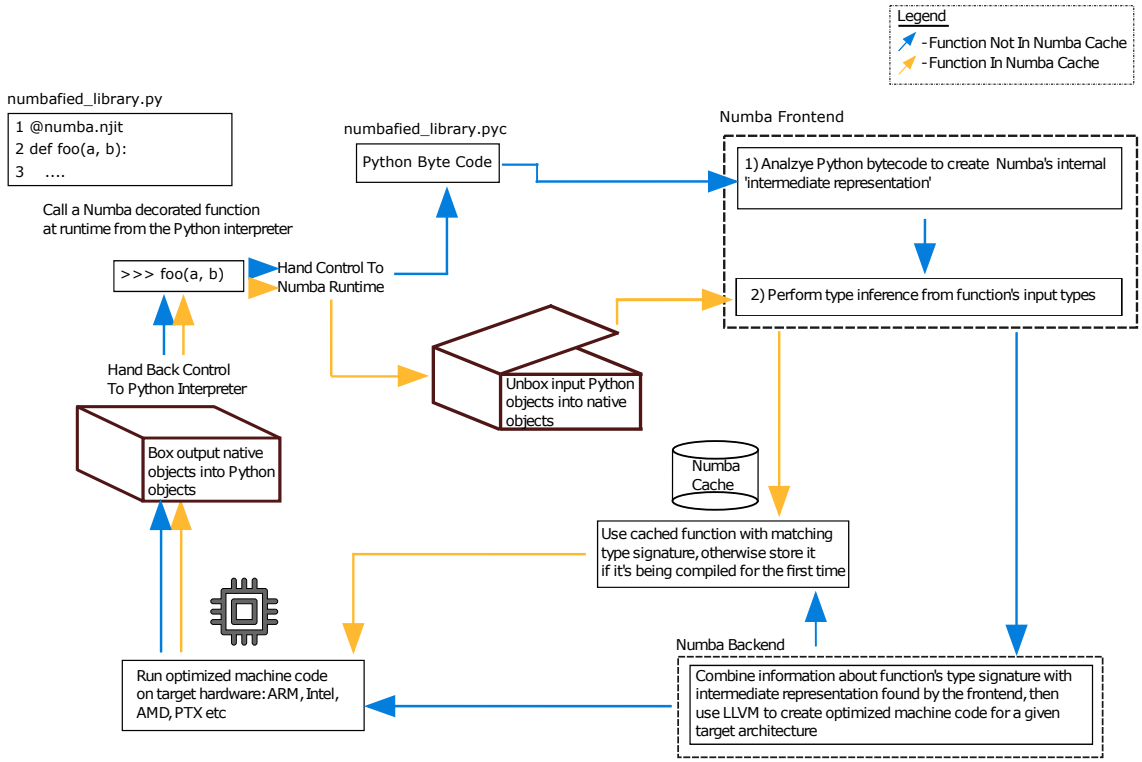


Figure 1.1: A visualisation of the Numba runtime system. A function decorated with 'njit' acts as an instruction to the runtime to check a database for any Numba-compiled functions with a matching function signature, if this doesn't exist Numba generates a new compiled function and caches it using LLVM. 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 [10].

```
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 [10]

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 [1]. 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 [7], C and C++ do not have a single officially supported package manager or build system. The resulting landscape is a multitude of package managers [18, 24, 5] and build systems [23, 2, 16] 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 [17]. 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 Trait 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 [20], numerical data containers [15], and tools for generating interfaces to Python via its C ABI [14]. Many tools are yet to be ported into native Rust, however high quality bindings exist for core tools such as MPI [19], BLAS and LAPACK [4], 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 Future 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 [12].

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.

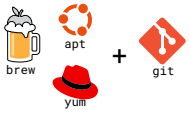
Currently, Mojo remains a closed-source language and is actively being developed by its parent company, Modular. Thus, even though it appears promising, it's not yet in a state suitable for experimentation. Nevertheless, Mojo showcases the potential of a future programming environment that might definitively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

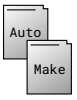
Dependency Management Methods

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



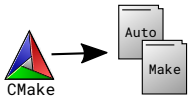
2) Use a system package manager, and source from repositories such as GitHub to install globally. These can then be found by build systems. This makes it difficult to build isolated build environments, and configure builds with different versions, or sets, of dependencies.

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 in the context of its original introduction for the solution of the N-body potential calculation problem of electrostatics, or gravitation, which gives rise to much of the terminology in this field [8]. 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 [13, 6]. 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,

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

Such problems appear frequently across science and engineering, most notably in the discretisation of boundary integral equation (BIE) formulations for elliptic partial differential equations. See Appendix A, for a discussion on how such a problem formulation arises from the discretisation of a BIE for elliptic PDEs [NOTE, add to the appendix on why elliptic pdes such as laplace, helmholtz and maxwell are nicely behaved as the fundamental solutions are well-known and smooth for these problems facilitating the development of bie formulations].

Writing the component form (2.1) 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 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 [APPENDIX FOR LU DECOMP/GE] 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.

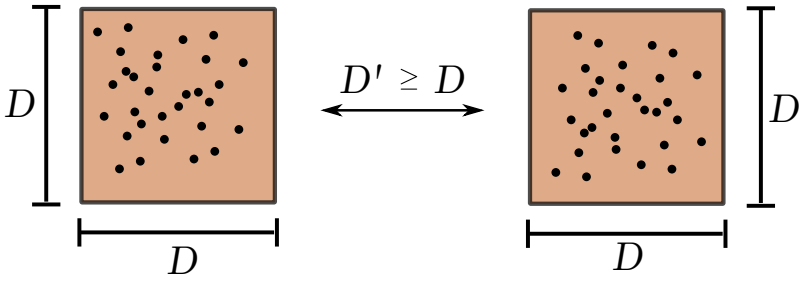


Figure 2.1: Given two boxes in $\mathbb{R}^d$ ($d = 2$ or 3), $\mathcal{B}_1$ and $\mathcal{B}_2$, which each enclose a corresponding set of points. The off-diagonal blocks in the matrix $K_{\mathcal{B}_1\mathcal{B}_2}$ and $K_{\mathcal{B}_2\mathcal{B}_1}$ are considered low rank and amenable to compression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

From figure 2.1 we see that the FMM must rely 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 (2.1), we must calculate the potential vector $\phi \in \mathbb{C}^N$, from a charge vector $\mathbf{q} \in \mathbb{C}^N$ by applying $K \in \mathbb{C}^{N\times N}$. These quantities are in general complex. 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 [APPENDIX NOTE ON SVD APPROXIMATION]. 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 deliberately haven't stated how we calculate B_p or C_p. In Greengard and Rokhlin's FMM these took the form of analytical multipole and local expansions of the kernel function [8]. 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 $\hat{\mathbf{q}}^{\mathbf{s}} = \{\hat{q}_p^s\}_{p=0}^{P-1}$,

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

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,

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

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 nodes, 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 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 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 [22].

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 apply a local expansion to a set of points T^{L2P}, 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 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

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

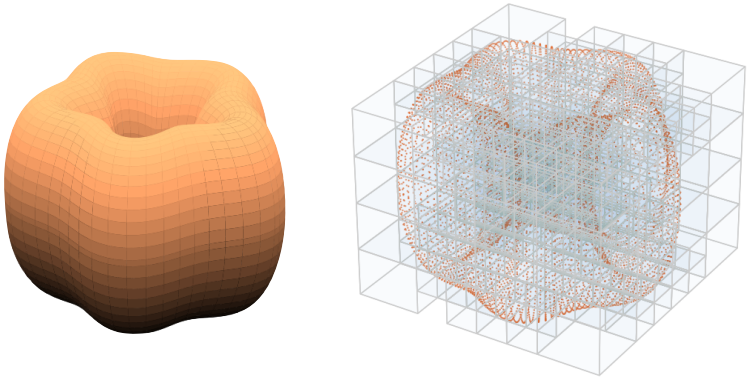


Figure 2.2: An adaptive octree for random point data placed on the surface of a 'wiggly torus' test geometry. The user defines the level of recursion via a threshold for the maximum number of particles in a given node.

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. Finally, for each box, the far-field interaction which is encoded in the local expansion at each box, is evaluated at the particles it contains using T^{L2P} is combined with the 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}.

2.2 Implementation Challenges for Black Box Fast Multipole Methods

2.3 Building for Re-Use in Fast Algorithm Software

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

Open Work Streams

3.1 Distributed Octrees

- Implementation details of octrees, and overview of current research.

- Our octree algorithm - bottlenecks in algorithm, and potential communicator optimisations - a single node benchmark at the very least, can defer multi-node one for now.

3.2 Fast Field Translations

3.2.1 SVD Field Translations

3.2.2 FFT Field Translations

Conclusion

Q. What do you hope to achieve in next 15 months? - distributed FMM library, stable interface, plugged into BEM code - ready for some large scale benchmarks, and test experiments

Q. What is an extension? - developing a fast direct solver using the software infrastructure - presenting a unified forward/backward solver package - first one on the market.

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 [21]. 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

Bibliography

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

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

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

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

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

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

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

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

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

[10] 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.

[11] 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.

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

[13] 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.

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

[15] *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.

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

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

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

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

[20] 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`.

[21] 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.

[22] 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.

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

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