Mostly Painless Scientific Computing With Rust

SRINATH KAILASA and TIMO BETCKE (ADVISOR), University College London, United Kingdom

Scientific software is required to be fast, painless to change, and easy to deploy. Historically, compiled languages such as C/C++ and Fortran have been preferred when writing software with the highest performance requirements. However these languages are complex, and the resulting software is challenging to maintain and deploy across platforms. We present our recent software projects written in Rust, a fast-growing, ergonomic, systems-level programming language with a toolchain designed for high-performance and simple cross platform builds. We illustrate the current state of the scientific computing ecosystem in Rust, through our experience developing high-performance MPI-distributed software for computational physics problems.

CCS Concepts: • **Applied computing** → Physical sciences and engineering; Life and medical sciences; • **Software and its engineering** → **Designing software**.

Additional Key Words and Phrases: rust, scientific software engineering, octrees, high performance computing

1 THE CONSTRAINTS OF ACADEMIC SOFTWARE DEVELOPMENT

In academia a small team, or commonly a solo developer, is responsible for the full software development life-cycle (SDLC). From planning and scoping a project's requirements, to designing, documenting, testing and finally deployment and maintenance. Inevitably, research software is often fragile as compromises are made on various aspects of the SDLC in order to meet a project's scientific goals as quickly as possible.

To raise developer productivity and therefore software quality, the academic software community has adopted interpreted languages such as Julia [3], Python [11] and Matlab as a de-facto standard. These languages are easy to compile and deploy cross-platform. Despite the emergence of tools to accelerate these languages [5–7, 10], compiled languages are preferred for software aiming to achieve the highest levels of performance. Fortran and C/C++ dominate, with a common approach being to develop interpreted language bindings on-top of the underlying implementation to encourage adoption. The complexity of development and deployment in these languages drains developer productivity as they are drawn away from their science.

2 RUST AS A SOLUTION

We propose Rust as an alternative to other compiled languages. Two of the biggest drains to productivity, dependency management and memory leaks, are effectively solved by Rust and its toolchain Cargo. Dependencies are specified in TOML files, which Cargo uses for compilation with a single command. Rust's 'borrow checker' forces a developer to manage ownership of each data structure, catching memory errors at compile time. This guarantees no memory leaks in compiled code, without the need for a garbage collector. Testing and documentation are critical for high-quality software projects. Cargo makes it seamless to develop tests and write documentation for Rust code. Cargo's documentation command transforms markdown docstrings into high-quality html webpages, and Cargo's test infrastructure supports both unit and integration testing.

This is in stark contrast to the current paradigm of dependency management, compilation, testing and documentation in Fortran or C/C++. Where the complexity of dependency management via package managers and CMake is colloquially referred to as 'dependency hell'. Third party dependencies are required for testing and documentation in C/C++ and Fortran, further contributing to dependency management issues, and imposing an artificial barrier to entry for writing quality software.

Despite being a young language, Rust already has well developed features for scientific computing including parallel multithreading support [13], mature MPI bindings [12] and simple Python binding development [1]. Rust's expressive syntax, inspired by both functional and imperative paradigms, is familiar to programmers coming from any language background. The trait system, for specifying shared behaviour, and the complete removal of objects from the language make large Rust projects significantly easier to read, and faster to write.

3 RUSTY TREE

Our present research is focused on developing high-performance solvers for boundary integral equations [2], for which octrees are a foundational data structure. Octrees spatially decompose a three-dimensional cube, and are defined by a 'root' node enclosing a physical domain of interest, which is recursively partitioned into eight children, until a desired resolution is reached. Creating octrees that can be distributed across parallel computing systems is challenging, demonstrated by the limited availability of open-source software [4, 8], despite their ubiquity in scientific computing applications from adaptive finite-element methods to many-body algorithms.

We present Rusty Tree [9], an implementation of MPI-distributed octrees with a complete Python interface. Rusty Tree is a proof of concept for Rust as a tool for high-performance computational science. In this poster, we document our experience in developing Rusty Tree, and use it as a tool to explore the current landscape of scientific computing with Rust. From multithreading, and developing a Python interface, to distributing applications with MPI, writing Rusty Tree involved using most of Rust's scientific computing ecosystem. We demonstrate weak scaling performance by constructing octrees with up to 640 million randomly distributed points. We aim to illustrate Rust's viability for high-performance scientific computing, however we also document the significant challenges, especially in the standardisation of libraries, that remain in developing the fledgling scientific Rust community, and our future Rust based research directions in light of this.

ACKNOWLEDGMENTS

Srinath Kailasa is supported by EPSRC Studentship 2417009 and a G-Research PhD Grant. Timo Betcke is supported by EPSRC Grants EP/W007460/1 and EP/W026260/1

REFERENCES

[1] 2022. *Maturin.* https://github.com/PyO3/maturin

[2] Timo Betcke and Srinath Kailasa. 2022. *Rusty Fast Solvers: Fast solvers for integral equations in Rust, with Python interfaces.* https://github.com/rusty-fast-solvers/

[3] Jeff Bezanson, Stefan Karpinski, Viral B Shah, and Alan Edelman. 2012. Julia: A fast dynamic language for technical computing. *arXiv preprint arXiv:1209.5145* (2012).

[4] Carsten Burstedde, Lucas C. Wilcox, and Omar Ghattas. 2011. p4est: Scalable Algorithms for Parallel Adaptive Mesh Refinement on Forests of Octrees. *SIAM Journal on Scientific Computing* 33, 3 (2011), 1103–1133. https://doi.org/10.1137/100791634

[5] Charles R. Harris et. al. 2020. Array programming with NumPy. *Nature* 585, 7825 (Sept. 2020), 357–362. https://doi.org/10.1038/s41586-020-2649-2

[6] James Bradbury et. al. 2018. *JAX: composable transformations of Python+NumPy programs.* http://github.com/google/jax

[7] Pauli Virtanen et. al. 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* 17 (2020), 261–272. https://doi.org/10.1038/s41592-019-0686-2

[8] Milinda S. Fernando and Hari Sundar. 2020. *Dendro-5.01: A data parallelism library for Rust.* https://github.com/paralab/Dendro-5.01

[9] Srinath Kailasa and Timo Betcke. 2022. *Rusty Tree: Distributed octrees in Rust with Python interfaces.* https://github.com/rusty-fast-solvers/rusty-tree

[10] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. 2015. Numba: A LLVM-Based Python JIT Compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC* (Austin, Texas) *(LLVM '15)*. Association for Computing Machinery, New York, NY, USA, Article 7, 6 pages. https://doi.org/10.1145/2833157.2833162

[11] Fernando Pérez, Brian E. Granger, and John D. Hunter. 2011. Python: An Ecosystem for Scientific Computing. *Computing in Science Engineering* 13, 2 (2011), 13–21. https://doi.org/10.1109/MCSE.2010.119

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

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