GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

HPS

Seminar Report

Rust for HPC Programming

Henrik Jonathan Seeliger
Matrikelnummer: 20534843

Supervisor: Artur Wachtel

Georg-August-Universität Göttingen
Institute of Computer Science

September 27, 2024

Abstract

This report examines the suitability of the Rust programming language for the field of HPC programming. In order to achieve this, the Marching Cubes algorithm is implemented using both the C programming language and Rust, creating a foundation for a comparative analysis of Rust. Additionally, the performance and scalability implications of Rust are investigated by employing benchmarks. The results demonstrate that Rust is a valuable alternative to C in terms of maintainability, language features and performance. However, limitations in the parallel computing capabilities of Rust have been identified that indicate performance challenges when utilizing Rust in parallel computing scenarios.

Repository

The source code of the implementations, as well as the source of this document and the accompanying presentations, can be accessed using the following URL:

https://gitlab.gwdg.de/h.seeliger/rust-for-hpc-programming

Declaration of Authenticity

Declaration on the Use of ChatGPT and Comparable Tools in the Context of Examinations

In this work I have used ChatGPT or another AI as follows:

- ☑ Not at all
- ☐ During brainstorming
- ☐ When creating the outline
- ☐ To write individual passages, altogether to the extent of 0% of the entire text
- ☐ For the development of software source texts
- ☐ For optimizing or restructuring software source texts
- ☐ For proofreading or optimizing
- ☐ Further, namely: -

I hereby declare that I have stated all uses completely.
Missing or incorrect information will be considered as an attempt to cheat.

Contents

List of Figures

List of Tables

List of Listings

Acronyms

CT – computed tomography.

GWDG – Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen.

HPC – high-performance computing.

OS – operating system.

1 Introduction

1.1 Motivation

Writing safe software presents a major challenge in software development. The term *safe* refers to several software quality characteristics like *faultlessness* and *resistance*, defining how robust the software is against bugs and possible vulnerabilities [1]. Costing the U.S. economy approximately $1.81 trillion in 2022, operational failures and their prevention have global significance [2].

This importance of software safety also applies to the context of HPC. Given the nature of HPC, long-running software processes are used in many cases. If a software failure occurs at any time of the process, the process usually has to be restarted, meaning a loss of time and resources. Besides crashes, failures may also be logical, possibly resulting in malformed results.

While the necessity for safe software in HPC seems clear, the realization is often not. Languages like C or C++, characterized by offering the programmer full control and performance at the expense of safety e.g. in terms of memory [3], dominate the current field of HPC programming languages. Rust, on the other hand, is a programming language that sets performance and reliability in equal parts as its main goal [4], but is currently less-known than e.g. C and C++ [5]. An examination of using this programming language in the context of HPC may yield insights regarding the language's strengths and its suitability for the HPC field, tackling some of the current challenges of HPC programming.

1.2 Objectives

The main goal of this work is analyzing and evaluating the Rust programming language with regard to its use and suitability for the HPC environment. To achieve this, the same HPC-centered practical problem is solved using both the Rust programming language and C, which is one of the most widely used languages in HPC [6]. A comparison of both implementations as well as the examination of the Rust implementation is used to identify and elaborate the capabilities, advantages and value of Rust for the usage in HPC. This comparison covers both qualitative and quantitative aspects, including benchmarking of both implementations.

1.3 Structure

After explaining some general foundations for this work, the approach to achieving the objectives described earlier is presented. This is followed by a short explanation of both implementations, highlighting important key aspects. Subsequently, qualitative findings and benchmark results are presented, followed by an interpretation and discussion of these to achieve this work's main research goal. Finally, the report is concluded by summarizing the findings and presenting an outlook towards potential future work.

2 Background

This section briefly explains various concepts on which the report is based. After introducing the Rust programming language, the Marching Cubes algorithm is described shortly which serves as the foundation for the subsequent practical examinations.

2.1 Rust

After starting as a personal project of a Mozilla employee, the Rust programming language had its first stable release in 2015, making it a relatively young programming language [7]. As of 2024, it is widely adopted by leading software companies like Amazon [8] or Microsoft [9] and one of the most popular programming languages in the world [5].

Rust's main goal is to enable reliable and efficient software while enhancing developer productivity. It combines imperative and functional programming paradigms, while maintaining a systems-oriented approach that allows for high-level programming [4]. Performance and reliability in terms of e.g. memory safety are ensured by an unique memory management model, providing a manual but implicit approach without runtime overhead. These language features are accompanied by various tools designed to improve productivity, including a verbose compiler and a package management system with approximately 154,000 available packages as of August 2024 [10].

Considering the general requirements for HPC software, the described characteristics suggest Rust to be a possibly valuable addition to the field of HPC programming.

2.2 Marching Cubes Algorithm

The aim of the Marching Cubes algorithm proposed by W. E. Lorensen and H. E. Cline is to generate a surface mesh from a three-dimensional scalar field. Practical use cases include generating three-dimensional models from CT images, which are typically available in the form of layered two-dimensional images [11]. Besides a three-dimensional scalar field, a user-defined iso value must be given, defining the level of the generated surface. For example, this could be a specific CT number that indicates the presence of bones for generating a mesh of the bone structure.

The Marching Cubes algorithm works by dividing the three-dimensional scalar field into a three-dimensional grid. Using two adjacent grid slices, four vertices from each slice are defined as a cube. Figure 1 illustrates this construction of one cube. Given the user-defined iso value and the scalar values at the vertices' positions, each vertex is identified as being above or below the surface to be generated. This information is then used to identify one of 14 possible cube configurations (256 in total, reduced using symmetries) which determine how the

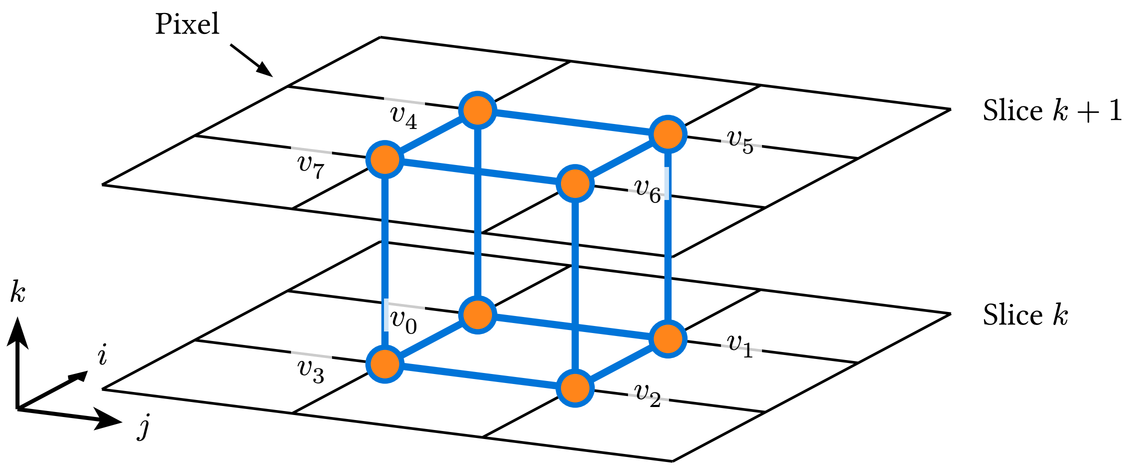

Figure 1: Cube construction in the Marching Cubes algorithm. Adapted from [11]

surface intersects with the inspected cube. In accordance with established practices for mesh representations [12], a specific number of triangles represents this intersecting surface. This process is repeated by "marching" over the grid while collecting all identified triangles, which is the origin of the algorithm's name. Ultimately, these triangles represent the generated surface.

Using a divide-and-conquer approach, the Marching Cubes algorithm can be parallelized effectively. As the algorithm operates on individual "cubes" with individual results, the grid can be partitioned and distributed among multiple processes. The resulting triangles can then be collected to create the overall resulting surface. This characteristic of the algorithm makes it an appropriate foundation for the practical examinations in this work, as parallel and distributed computing are important aspects of programming in the field of HPC. Besides this, the algorithm's input can be scaled and varied with minimal effort, enabling realistic workloads such as CT image processing as well as artifical workloads with different input sizes for scalability analysis.

3 Methodology

This section describes the approach and methods employed in this work for evaluating the Rust programming language regarding its suitability for HPC programming.

3.1 Study Design

To identify characteristics, advantages and capabilities of Rust in the context of HPC, the Marching Cubes algorithm as described in Section 2.2 is implemented using this language. Furthermore, the same problem is implemented using the C programming language. This enables insights regarding Rust itself, but also differences, advantages and disadvantages compared to C, which is currently one of the most widely used languages in HPC programming [6]. Besides its popularity, C has also been chosen for the comparison because Rust and C have a variety of design goals in common like systems-orientation or efficiency but use different approaches for achieving them [4]. Given the importance of parallel computing in HPC programming, both a sequential and a parallel implementation for each language are developed, respectively. This allows a more comprehensive examination regarding the parallelizability and scalability of Rust.

After implementing the Marching Cubes algorithm using Rust as well as C, the code and characteristics of the resulting software can be compared. First, qualitative aspects are compared, like available language constructs and semantics regarding memory safety. During this, distinctive features of Rust may be observed and identified. This includes further research regarding Rust's ecosystem with focus on HPC-related technology. Then, both implementations can be compared quantitatively regarding their performance and scalability by employing benchmarks.

It is important to clarify that the objective of this comparison is not to identify an universally superior programming language. Caution is required when comparing programming languages as many factors influence aspects of software like performance or reliability. Instead, this work focuses on the suitability of Rust for HPC programming with a comparison as a method of gaining insights, while many other factors may be considered in a comparison of programming languages.

3.2 Benchmarking

To quantify differences between the implementations and examine Rust's performance, a benchmarking process is employed. This process involves measuring and comparing runtimes of both the sequential and parallel implementations for each language. The results can then be used to identify performance characteristics of Rust. Benchmarking the parallel implementations may also give findings regarding the scalability of software written in Rust. To benchmark the parallel implementations and their scalability, the runtime of the programs given a specific number of processors is measured. Further information regarding used hardware, exact benchmark parameters, and implementation details can be found in Section 4.3.

4 Implementation

As previously described, implementing the Marching Cubes algorithm represents the foundation for this work's findings. This section describes key aspects of the different implementations.

4.1 Common Approach

Both the C and the Rust implementations use the same general approach to realize the Marching Cubes algorithm.

First, the input data is read from a common file format. Being a three-dimensional scalar field, a file format designed for matrix data is appropriate. The NumPy file format [13] has been chosen to represent input data to be read by the implementations, as it is flexible as well as widely adapted in various software ecosystems. This also enables efficient preparation of real and artificial test input data using the Julia programming language and its ecosystem. To test the implementations, real CT images from [14] are used, as visualized in Section 5. A parser and reader of NumPy files serve as the first part of the implementations and form the foundation of the algorithm's implementation.

After executing the Marching Cubes algorithm on the input data, a set of triangles representing the surface mesh is yielded. To visualize and verify the results, the mesh is written to disk using the STL file format [12]. This file format employs a straightforward approach to representing meshes using triangles, making it well suited for this application.

The implementation of the algorithm itself only differs slightly across the languages and their respective sequential and parallel versions. The parallel versions are derived from their respective sequential versions with minimal divergence for clarity and brevity.

4.1.1 Sequential

After reading parameters from the command line, e.g. the iso value of the surface to generate, and the input data into memory, the implementations stick closely to the procedure described in the algorithm's original publication [11]. For each position in the grid, eight neighboring vertices and their values are captured as a cube structure. This cube's configuration is then used to look up in a table which edges are intersected by the surface. Using the resulting edges and the vertice's positions and values, triangles representing the surface mesh inside the cube can be looked up in another table and be interpolated. Both the table of possibly intersected edges and possible triangle configurations can be pre-calculated and are taken from [15]. Using a form of collection, the triangles resulting from each cube are collected while iterating. After iterating over the entire grid, the triangle collection represents the result and can be written to disk using the aforementioned STL format.

4.1.2 Parallel

The Marching Cubes algorithm utilizes a divide-and-conquer approach, which allows for an uncomplicated parallelization. The parallelization strategy used with both programming languages is identical. As every "cube" is processed individually and does only depend on its own position with its respective field values, the grid can be divided in equally large partitions that can be distributed among processes. Each process then computes the mesh triangles for its partition. Afterwards, the triangles from all partitions can be collected into one result set.

This approach makes the main loop, which iterates over the grid, the central point of implementing parallelization. Both parallel versions employ parallelization by converting the sequential main loop into a parallel loop, but use different techniques and libraries. These techniques and libraries are one of the main key differences, described in the following section.

4.2 Key Differences

By employing different programming languages with different semantics, design decisions and features, some key difference exist between the algorithm implementations using Rust and C. Both versions are designed to be comparable, but some differences are necessary.

The most significant difference between both implementations is that the Rust implementation is designed for the examination of Rust: This implementation uses a variety of Rust's features to support a comprehensive exploration of the language. It prioritizes memory safety, a crucial aspect of the language as described later on, and utilizes various high-level constructs, while the C language does not employ special safeties regarding e.g. memory safety. This decision may have implications regarding performance which could be reflected in the comparison results.

An implementation detail connected to this aspect is the memory layout of the input data. In the C implementation, the input data, which is a three-dimensional array representing the scalar field, is loaded into memory without modification or typing. Read operations on the data are then performed using pointer arithmetic. In the Rust implementation, however, this is not possible without overriding specific compiler rules, which ensure the memory safety of Rust programs. Instead, the *ndarray* library [16] is used for safe and compliant data access, which the Rust compiler permits.

As described in Section 4.1.2, both implementations parallelize the algorithm by parallelizing the main loop iterating over the input data. However, both versions utilize different libraries and language constructs to achieve this. The C implementation uses the *OpenMP* API [17] and its compiler directives, while the Rust implementation utilizes Rust's language constructs and a specific library, called *Rayon* [18]. This library is designed for data parallelism in conjunction with the language features of Rust, enabling the parallelization of existing loops by employing the iterator pattern. As a considerable proportion of loops in Rust are based on the iterator pattern [4], this approach is well integrated with Rust and significantly simplifies the parallelization of loops.

4.3 Benchmarks

In order to quantify and compare the performance characteristics of the implementations, a benchmarking process has been developed for each language. This process measures the wall-clock time of the implementations, given fixed input data. Additionally, to analyze the scaling behavior, the parallel versions are measured with a varying number of processors. After measuring the wall-clock time given the number of processors, the speedup s can be calculated as

$$s = \frac{t(1)}{t(N)}$$

where $t(1)$ denotes the runtime of the respective parallel implementation using one processor and $t(N)$ denotes the runtime using N processors [19]. As the input data size remains constant throughout this process, only the strong scaling behavior is examined in this work.

Both languages use the same overall process for benchmarking, with only minor differences in the implementation. For C, the benchmarking logic is self-developed due to the unavailability of suitable libraries. Rust, however, utilizes the *Criterion.rs* library for implementing benchmarks with minimal effort [20]. After loading the input data, the algorithm is executed 20 times, with the elapsed time for each run recorded and stored in a machine-readable format. In the case of the parallel implementations, this process is repeated for a specified range of processor counts. The times are then averaged over each processor configuration.

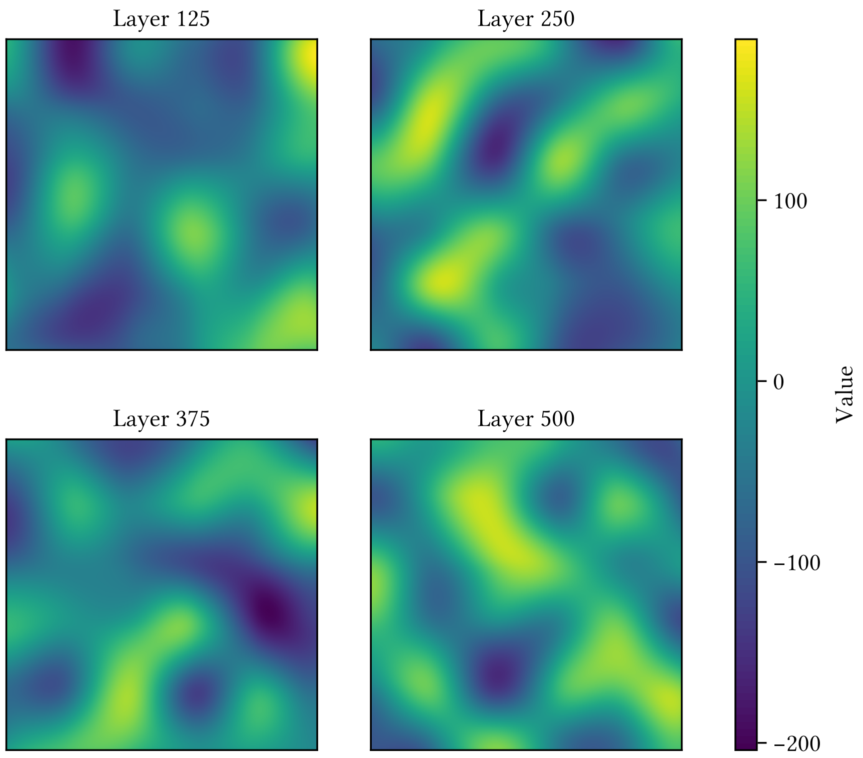

Figure 2: Different layers of the artificial input data for the benchmarks.

To enable controllable benchmarks, artificially generated input data is used. In this work, sampling from a three-dimensional coherent noise field is employed as generation method. Value cubic noise is selected as noise the type, as implemented in the *CoherentNoise* library [21]. Figure 2 provides a visual representation of the generated and utilized input data. For the purpose of the benchmarks, dimensions of 500 by 500 by 500 have been selected, which are comparable to the dimensions of real CT data, as shown in Section 5.1.4.

The benchmarks were conducted on an HPC system provided by the Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen (GWDG). As no distributed computing is implemented or involved, one node was allocated. Table B.1 shows this node's specificiations. The results of the benchmarks are described in the following section, while implementation details can be reviewed in the repository referenced on page I.

5 Results and Discussion

This section presents the observations and findings of this work, which were obtained through the previously described methodology. These findings are then employed to analyze the suitability of Rust in the context of HPC programming in order to answer this work's research question.

5.1 Qualitative Results

To assess the Rust programming language for use in HPC programming, qualitative characteristics of Rust have to be described and analyzed. This analysis is based on the implementations, with illustrative listings taken directly from the source code. Then, possible advantages and disadvantages emerging from these aspects can be stated and explained, thus contributing to an overall evaluation.

5.1.1 Syntax

Rust features a syntax similar to the syntax of C. Like C and numerous other widely used programming languages, it focuses on imperative programming, but it also incorporates elements of functional programming, which are described further in Section 5.1.2. Listing 1 shows one main function from the Rust implementation[1] to illustrate the syntax of Rust.

```rust
// [...]

fn main() -> Result<(), Box<dyn Error>> {
    let conf = Configuration::parse();

    let data: NpyArray<NpyLeI2, Ix3> = NpyArray::read(conf.input)?;
    let result = marching_cubes(&data, conf.iso_value);

    result.write_to_file(conf.output)?;

    println!("Success!");

    Ok(())
}
```

Listing 1: Main function of the sequential Rust implementation.

The syntax of Rust, as demonstrated in Listing 1, shares similarities with that of C. This aspect means that Rust may be more accessible for developers with prior experience in C and other languages with C-like syntax, which currently represent the majority of the most widely known programming languages [5].

[1]Multiple main functions exist because each implementation has multiple versions (sequential and parallel).

Furthermore, the Rust syntax features some additional functionality when compared to C, like string interpolation or optional typing. Lines 4 and 6 of Listing 1 show examples for optional typing: In line 4, the type of the variable can be omitted because the compiler is able to infer its type. In line 6, however, the compiler is not, meaning the type of `result` has to be stated explicitly as `NpyArray<NpyLeI2, Ix3>`.

In the context of general programming, the accessibility and features of Rust's syntax may prove advantegous. Its accessibility allows more developers to read, learn and maintain software written in Rust, which can lead to increased productivity. Its advanced features may also support this increased productivity. When combined, they can also positively impact the maintainability of software written in Rust. However, as the advanced features of Rust's syntax increase its complexity, it could possibly hurt the accessibility for developers with less experience in Rust or programming in general. These advantages and disadvantages can be directly transferred to the utilization of Rust in the context of HPC.

5.1.2 Semantics

Rust offers a variety of semantic features distinguishing it from other languages, particularly from C. One such feature is its type system, which differs considerably from that of C.

Both languages employ a static type system, meaning the types of variables have to be declared at compile time. However, in contrast to Rust, the type system of C is not as strong and flexible. One example for this is the possibility and common practice of using `void` pointers, which demonstrates the weak typing of C [22]. Besides that, C provides only a limited number of possibilities for defining and using custom types, like structures and enumerations, without support for features like type hierarchies. Rust, on the other side, promotes a strong type system with advanced capabilities. One illustrative example is the support for generics, as demonstrated in Listing 2. Combined with other features such as *traits*, which can be compared to interfaces in other programming languages, Rust's type system is not only robust, but also extensible and maintainable.

```rust
pub struct NpyArray<T, D>
where
    D: Dimension,
    T: NpyType,
{
    pub version_major: u8,
    pub version_minor: u8,
    pub header: Header<T>,
    pub data: Array<T::Inner, D>,
}

fn main() {
    let data: NpyArray<NpyLeI2, Ix3>
        = NpyArray::read(conf.input);
}
```

Listing 2: Example of a custom struct using generics in Rust.

The aspect of robustness is not exclusive to the type system of Rust. One characteristic of Rust that differentiates it from the majority of programming languages is its memory management design. In C, memory is managed explicitly which requires the developer to manually allocate and release memory as necessary. While providing the developer full control, it compromises the language's memory safety and therefore robustness [23]. An alternative to explicit memory management is garbage collection, as employed by, for example, the Java programming langage [22]. However, this approach can frequently result in a decline of performance because of introduced runtime overhead [24]. This is why Rust employs a memory management strategy that differs from those used in other programming languages, referred to as the *ownership system* [4]. This system defines a set of rules that are enforced by the compiler to ensure the memory safety of Rust programs. These rules are checked at compile time, minimizing runtime overhead and performance penalties for memory safety in theory [4]. Using this approach, Rust can enhance the robustness of software by preventing runtime errors at compile time, while simultaneously ensuring optimal efficiency and performance.

The ownership system and generics serve as examples of Rust's *zero-cost abstractions* [4], offering high-level functionality without compromising performance. An additional example of these abstractions can be observed in the iteration methods of Rust. In Rust, iteration is primarily realized through the use of the iterator pattern. This pattern employs abstraction to provide high-level constructs, thereby achieving high maintainability and readability with minimal performance overhead [4]. In addition to offering loops similiar to loops provided by C, Rust promotes the use of iterator-based constructs and methods, as observable, for example, in reactive programming [25]. Listing 3 shows the identical iteration logic implemented in Rust, employing either a for loop or iterator operations. Besides maintainability, this also enables the uncomplicated modification and enhancement of iterations. The library *Rayon* utilizes this approach to incorporate data parallelism into iterator-based loops [18].

```rust
let mut intersection_index = 0;
for (i, value) in cube.values.iter().enumerate() {
  if (*value as f32) < iso_value {
    intersection_index |= 1 << i;
  }
}
```

```rust
let intersection_index = cube
  .values
  .iter()
  .enumerate()
  .filter(|(_, v)| v < iso_value)
  .map(|(i, _)| i)
  .reduce(|acc, e| acc | (1 << e));
```

Listing 3: Iterations implemented in Rust using for loop and iterator operations.

In the field of HPC, maintainability, robustness and performance are crucial. Maintainable software enables developers to modify and transfer it with ease, while the robustness of sofware ensures that it yields correct results in a shorter time, as unexpected exceptions can be minimized. Furthermore, performance represents a fundamental aspect of HPC, influencing a variety of dimensions, including wall-clock time and the potential workload of simulations. As shown, the semantic features of Rust support and promote these aspects in software, thereby representing valuable contributions to the field of HPC programming with significant potential. However, these features introduce a certain degree of complexity to the language and require developers to learn and adopt them. The ownership system, for example, presents many developers a challenge to learn initially [4]. The additional complexity of Rust's semantics represent a relevant factor to consider before a possible utilization in HPC programming.

5.1.3 Ecosystem

Dependency management presents a challenge to many programming languages, especially to traditional languages with vast and diverse ecosystems, such as C [26]. This includes obtaining, configuring and building required libraries. For C, a variety of approaches and tools exist that address this issue. These, however, vary significantly across projects utilizing C, creating challenges regarding compatibility and complexity of dependency management.

To mitigate these issues, Rust provides an unified approach to tooling and dependency management. Besides an official online registry for Rust libraries, *crates.io*, Rust offers a custom build tool with support for dependency management, called *cargo*. As of August 2024, *crates.io* contains approximately 154,000 packages [10]. This approach allows for unified and uncomplicated dependency and build management. By decreasing the necessary effort of obtaining, configuring and building required libraries, the ecosystem of Rust increases the developer productivity significantly when compared to the ecosystem of C.

In the context of HPC, however, the selection of relevant libraries must also be considered. Software for HPC commonly benefits from the paradigms of parallel and distributed computing. For C, various libraries and APIs supporting an implementation of these paradigms exist, such as *OpenMP* for parallel computing and *MPI* for distributed computing [17], [27]. In this work, the OpenMP API was employed to achieve data parallelism in the C implementation, a technique commonly utilized in HPC programming. In the Rust ecosystem, the *Rayon* library is widely employed for implementing data parallelism [18]. This library offers various high-level constructs for parallel computing with a strong focus on declarative and reactive programming patterns. Listing C.1 and Listing C.2 show the parallelization of the parallel implementations' main loops using OpenMP and Rayon, respectively. By extending the iterator pattern which represents the common technique for iteration in Rust, Rayon provides an uncomplicated way of parallelizing loops, therefore minimizing the effort for implementing data parallelism. This enhances the developer productivity and the maintainability of the software, particularly in comparison to the approach of OpenMP and C, which relies on compiler-level `pragma` directives. This results in written code being verbose and intricate when compared to Rayon, as illustrated by Listings C.1 and C.2.

The distributed programming capabilities of Rust are not covered in this work. Besides scopes limitations, the selection of libraries for distributed computing in Rust is relatively limited. Although bindings to external libraries such as MPI exist [28], a native approach implemented in Rust and using its strengths could be a valuable addition to the Rust HPC ecosystem. Some approaches implemented in Rust exist, but are unmaintained as of August 2024, such as the *Constellation* framework [29].

5.1.4 Functional Results

As described in Section 3, the objective of the implementations is to generate a three-dimensional mesh from given input data. Given the same input data, all implementations yield the same result, indicating their functional correctness. Furthermore, the results can be visualized to support this assumption. Figure A.1 visualizes exemplary CT data provided by [14] and its corresponding result. Figure A.2 shows the resulting meshes as produced by the benchmarks.

5.2 Quantitative Results

This section compares and analyzes the results of the benchmarks. These benchmarks aim to provide a basis for evaluating the performance of Rust, particularly in comparison to C. As described in Section 4.3, artificially generated data is utilized as input data for the benchmarks.

5.2.1 Sequential Implementation

Implementation	Mean	Standard Deviation	Range
C	2.009 s	0.009	1.992 s – 2.021 s
Rust	1.748 s	0.001	1.746 s – 1.751 s

Table 1: Measured runtimes of the sequential implementations.
Presented values are approximations.

Table 1 presents the measured wall-clock times of the sequential algorithm implementations. The difference in runtime demonstrates that the Rust implementation outperforms the C implementation in terms of speed. The minimal standard deviation indicates a high degree of consistency among the measurements, which supports this conclusion.

C is widely regarded as a programming language that enables highly performant software [30]. Given that the Rust implementation exhibits a faster runtime than the C implementation, it can be concluded that Rust also enables the development of software with a notable degree of performance. Furthermore, the difference suggests that Rust may yield even more performant software than C. In the context of HPC programming, where performance is a crucial consideration, this renders Rust a viable alternative to C where runtime is a significant factor.

5.2.2 Parallel Implementation

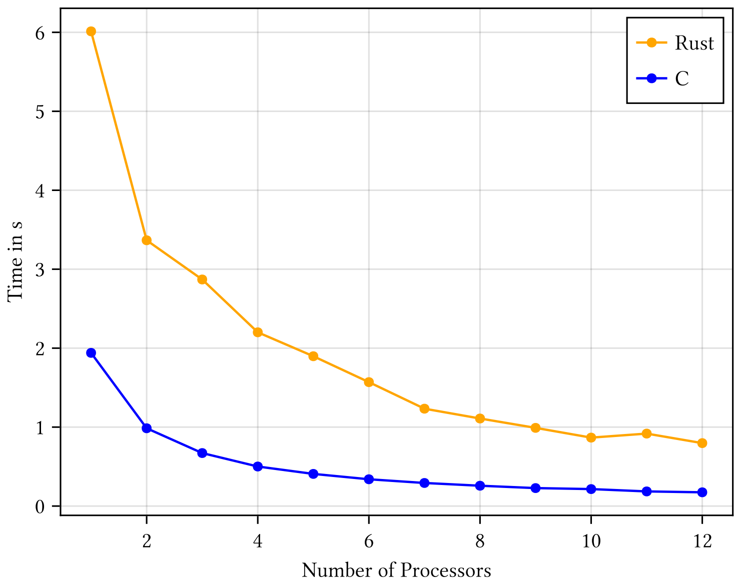

Figure 3: Runtimes of the parallel implementations depending on processor count.

Figure 3 depicts the wall-clock time of the parallel implementations in relation to the number of available processors. Both implementations exhibit a high runtime when a single processor is utilized, which decreases as the number of processors increases. However, the runtime of the Rust implementation is approximately four times higher than that of C across all processor configurations.

As detailed in Section 4.3, the speedup of both implementations can be calculated using the runtimes of the parallel implementations. Figure 4 illustrates the speedup for each implementation with $t(1)$ being the runtime of the parallel implementations using one processor.

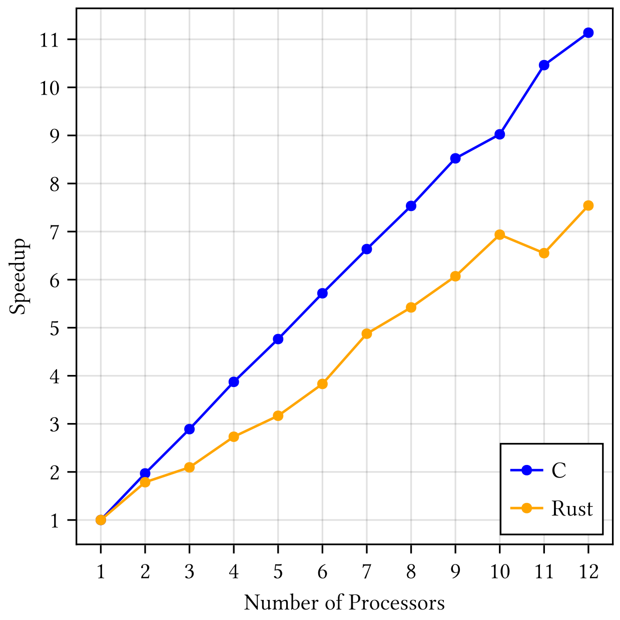

Figure 4: Speedup of the parallel implementations.

For C, the implementation approximates a linear speedup. The Rust implementation also demonstrates a speedup that increases with the number of processors. However, the increase is overall lower and more unstable, with a decrease observed from ten to eleven processors.

The findings regarding the parallel implementations are inconsistent with those regarding the sequential implementations. When the sequential versions are considered, the Rust implementation exhibits higher performance, whereas C demonstrates higher performance in the parallel versions. Additionally, the sequential Rust implementation is only outperformed by its parallel implementation from a processor count of six onwards. The runtime difference between the sequential versions indicates that a parallel Rust implementation may outperform both the sequential implementation and the parallel C implementation. However, the observations disprove this hypothesis.

This leads to the assumption that the parallelization of the sequential Rust version results in a decreased performance, whereas the parallel C version demonstrates the anticipated results for parallelization. Based on these results, Rust may prove to be less suitable for parallel computing scenarios in HPC programming when compared to C.

5.2.3 Performance Analysis

The results of the benchmarks indicate that Rust enables performant software in comparison to C when utilizing a sequential programming approach. However, when considering the presented observations, parallelizing Rust code may lead to drawbacks in performance, which also implies a decreased scalability. As performance and scalability represent important aspects in the context of HPC programming, Rust may therefore be seen as a suitable alternative in sequential programming scenarios, but not in parallel scenarios.

This conclusion must be regarded with a certain degree of caution as the performance and scaling behavior of software are complex and may be influenced by a number of factors. Consequently, an intense analysis of the implemetations is necessary to understand the performance implications of the programming languages. The inconsistencies and shortcomings observed in the parallel implementations emphasize the importance of such an analysis. Therefore, both implementations and their versions have to be examined to isolate potential causes. Despite deploying a range of techniques to accomplish this, no concrete evidence for bottlenecks could be obtained. One employed methodology is the analysis of *flame graphs*, with illustrative examples shown in Figure A.3 and Figure A.4. As can be observed, flame graphs can be complex, which presents a challenge when attempting to analyze them.

Nevertheless, a number of hypotheses can be proposed to explain the observed inconsistencies. One possible explanation for the runtime difference of the parallel versions is that Rayon and its parallelization logic differ from that of OpenMP fundamentally [17], [18]. Additionally, the application and introduction of Rayon in the Rust implementation may be inefficient, resulting in the observed runtime overhead. The memory layout of the input data may also be a significant factor, as they exhibit fundamental differences as described in Section 4.

Without clarification of these hypotheses and identification of concrete reasons for the benchmark inconsistencies, it is not possible to make an accurate evaluation of Rust's suitability for use in parallel programming scenarios, and consequently, in HPC. This represents one of this work's limitations.

5.3 Limitations

While this work provides insights into the suitability of Rust for HPC programing, it does not address all relevant aspects of the topic.

As previously stated, the analysis of the implementations providing the foundation for this work lacks sufficient depth. This causes the findings and implications of the benchmark results to be less expressive and universally applicable. Additionally, no evidence for the observed performance losses of the parallel Rust implementation has been identified, which could provide valuable insights for parallel and HPC programming using Rust.

A further limitation of this work is the number of factors employed in the comparative analysis of Rust and C. By considering additional factors beyond those described previously, a more comprehensive examination of Rust can be conducted, potentially providing more insights regarding Rust in the context of HPC.

The implementations developed in this work only employ parallel computing. As ditributed computing represents another important paradigm in the field of HPC, an evaluation of Rust's distributed computing capabilities could significantly enhance an evaluation of Rust for HPC programming.

6 Conclusion

6.1 Summary

In this work, the Rust programming language is examined regarding a potential application in the field of HPC programming. After implementing the same algorithm both using C and Rust, a foundation for the identification of Rust's capabilities, advantages and disadvantages is established through a comparative analysis with one of the most widely utilized programming languages in HPC.

The results demonstrate that Rust offers high maintainability and developer productivity, particularly in comparison to C. Furthermore, it enables the development of software with optimal performance characteristics. However, when parallel programming is employed, performance drawbacks can be observed, raising concerns about the suitability of Rust in multi-core environments and therefore in HPC. As the benchmarks and analysis methods employed in this work have limitations in depth, this aspect may be examined in more detail in the future to prove this assumption incorrect.

Upon summarizing these aspects of Rust, it can be concluded that while Rust represents a suitable language for HPC programming, this suitability depends on certain factors, with the experience of the developers and the necessity of parallel computing representing the determining aspects in this regard.

Besides this evaluation of Rust's suitability in the context of HPC, this work presents a code base for addressing a HPC-centered problem using Rust. The code base may serve as a foundation for solving other problems in HPC using Rust, and may also establish a foundation for future work in this research area.

6.2 Future Work

To enable a more comprehensive evaluation of Rust in the context of HPC, various aspects not covered in this work may be examined. These could include a more detailed performance analysis as well as additional benchmarking methods, such as examining the weak scaling behavior exhibited by software utilizing Rust. Furthermore, the suitability of Rust for distributed computing may be investigated, potentially leading to new applications for Rust and enhancing its overall suitability in the field of HPC.

References

[1] ISO/IEC 25010, "ISO/IEC 25010:2011, Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models." 2011.

[2] H. Krasner, "The Cost of Poor Software Quality in the US: A 2022 Report," Dec. 2022. Accessed: Aug. 08, 2024. [Online]. Available: https://www.it-cisq.org/wp-content/uploads/sites/6/2022/11/CPSQ-Report-Nov-22-2.pdf

[3] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic, "Hardbound: architectural support for spatial safety of the C programming language," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 2, pp. 103–114, Mar. 2008, doi: 10.1145/1353535.1346295.

[4] S. Klabnik and C. Nichols, *The Rust Programming Language: 2nd Edition*. San Francisco, CA: No Starch Press, 2023.

[5] Stack Exchange Inc., "2024 Stack Overflow Developer Survey." Accessed: Aug. 08, 2024. [Online]. Available: https://survey.stackoverflow.co/2024/

[6] V. Amaral *et al.*, "Programming languages for data-Intensive HPC applications: A systematic mapping study," *Parallel Computing*, vol. 91, p. 102584–102585, 2020, doi: https://doi.org/10.1016/j.parco.2019.102584.

[7] C. Thompson, "How Rust went from a side project to the world's most-loved programming language," MIT Technology Review. Accessed: Aug. 12, 2024. [Online]. Available: https://www.technologyreview.com/2023/02/14/1067869/rust-worlds-fastest-growing-programming-language/

[8] J. Barr, "Firecracker – Lightweight Virtualization for Serverless Computing." Accessed: May 22, 2024. [Online]. Available: https://aws.amazon.com/blogs/aws/firecracker-lightweight-virtualization-for-serverless-computing/

[9] T. Claburn, "Microsoft is busy rewriting core Windows code in memory-safe Rust." Accessed: May 24, 2024. [Online]. Available: https://www.theregister.com/2023/04/27/microsoft_windows_rust/

[10] The Rust Foundation, "crates.io: Rust Package Registry." Accessed: Aug. 12, 2024. [Online]. Available: https://crates.io/

[11] W. E. Lorensen and H. E. Cline, "Marching cubes: A high resolution 3D surface construction algorithm," *SIGGRAPH Comput. Graph.*, vol. 21, no. 4, pp. 163–169, Aug. 1987, doi: 10.1145/37402.37422.

[12] M. Szilvśi-Nagy and G. Matyasi, “Analysis of STL files,” *Mathematical and computer modelling*, vol. 38, no. 7–9, pp. 945–960, 2003.

[13] R. Kern, “NEP 1 — A simple file format for NumPy arrays.” Dec. 20, 2007. Accessed: Aug. 14, 2024. [Online]. Available: https://numpy.org/neps/nep-0001-npy-format.html

[14] S. Wan *et al.*, “Conscious-SEEG-Dataset.” OpenNeuro, Aug. 13, 2021. doi: 10.18112/openneuro.ds003754.v1.0.1.

[15] P. Bourke, “Polygonising a scalar field.” May 1994. Accessed: Aug. 14, 2024. [Online]. Available: https://paulbourke.net/geometry/polygonise/

[16] U. Sverdrup, J. Turner, and ndarray developers, “ndarray.” 2024. Accessed: Aug. 15, 2024. [Online]. Available: https://github.com/rust-ndarray/ndarray

[17] OpenMP Architecture Review Board, “OpenMP Application Programming Interface Specification Version 5.2.” Nov. 2021. Accessed: Aug. 15, 2024. [Online]. Available: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf

[18] The Rust Project Developers, “Rayon: A data parallelism library for Rust.” Mar. 24, 2024. Accessed: Jun. 26, 2024. [Online]. Available: https://github.com/rayon-rs/rayon

[19] M. Pondkule, R. Schade, and S. Potthoff, “Scaling,” HPC Wiki. Accessed: Aug. 17, 2024. [Online]. Available: https://hpc-wiki.info/hpc/Scaling

[20] J. Aparicio and Criterion.rs contributors, “Criterion.rs.” Aug. 2024. Accessed: Aug. 17, 2024. [Online]. Available: https://github.com/bheisler/criterion.rs

[21] M. Fiano and CoherentNoise.jl contributers, “CoherentNoise.jl.” Jun. 2023. Accessed: Aug. 17, 2024. [Online]. Available: https://github.com/lazarusA/CoherentNoise.jl

[22] B. Ray, D. Posnett, V. Filkov, and P. Devanbu, “A large scale study of programming languages and code quality in github,” in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, in FSE 2014. Hong Kong, China: Association for Computing Machinery, 2014, pp. 155–165. doi: 10.1145/2635868.2635922.

[23] E. D. Berger and B. G. Zorn, “DieHard: probabilistic memory safety for unsafe languages,” *SIGPLAN Not.*, vol. 41, no. 6, pp. 158–168, Jun. 2006, doi: 10.1145/1133255.1134000.

[24] M. Hertz and E. D. Berger, “Quantifying the performance of garbage collection vs. explicit memory management,” in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, in OOPSLA '05. San Diego, CA, USA: Association for Computing Machinery, 2005, pp. 313–326. doi: 10.1145/1094811.1094836.

[25] E. Bainomugisha, A. L. Carreton, T. v. Cutsem, S. Mostinckx, and W. d. Meuter, “A survey on reactive programming,” *ACM Comput. Surv.*, vol. 45, no. 4, Aug. 2013, doi: 10.1145/2501654.2501666.

[26] T. Gamblin *et al.*, “The Spack package manager: bringing order to HPC software chaos,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, in SC '15. Austin, Texas: Association for Computing Machinery, 2015. doi: 10.1145/2807591.2807623.

[27] Message Passing Interface Forum, “MPI: A Message-Passing Interface Standard Version 4.1.” Nov. 2023. [Online]. Available: https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf

[28] The rsmpi Developers, “MPI bindings for Rust.” May 03, 2024. Accessed: Jun. 26, 2024. [Online]. Available: https://github.com/rsmpi/rsmpi

[29] A. Mocatta, “constellation: Distributed programming for Rust..” Jul. 15, 2020. Accessed: Jun. 26, 2024. [Online]. Available: https://github.com/constellation-rs/constellation

[30] L. Prechelt, “An Empirical Comparison of Seven Programming Languages,” *Computer*, vol. 33, no. 10, pp. 23–29, 2000, doi: 10.1109/2.876288.

[31] Ferrous Systems GmbH, “[cargo-]flamegraph.” Oct. 20, 2022. Accessed: Jun. 26, 2024. [Online]. Available: https://github.com/flamegraph-rs/flamegraph

Appendix

A Figures

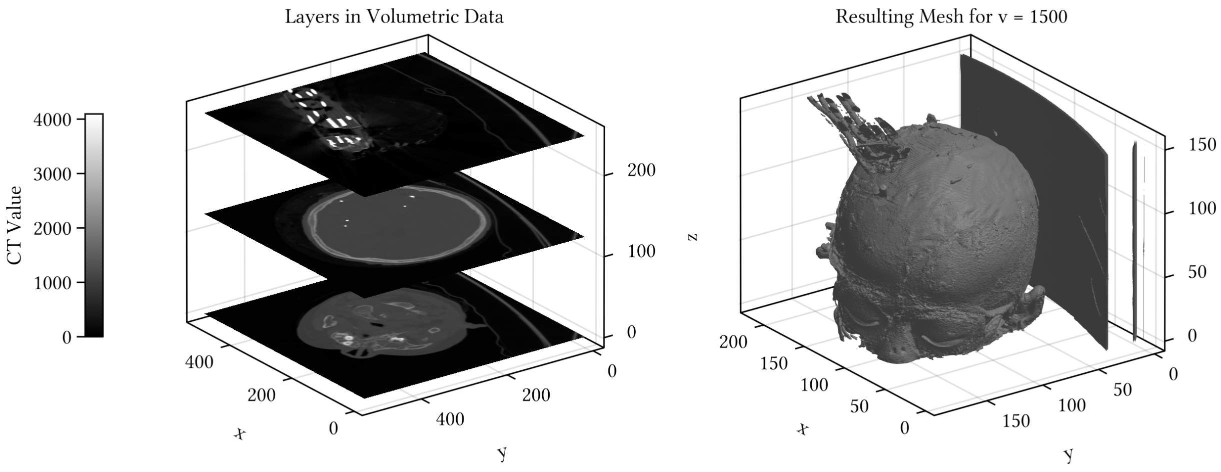

Figure A.1: Different CT slices and the resulting mesh.
Data from [14]

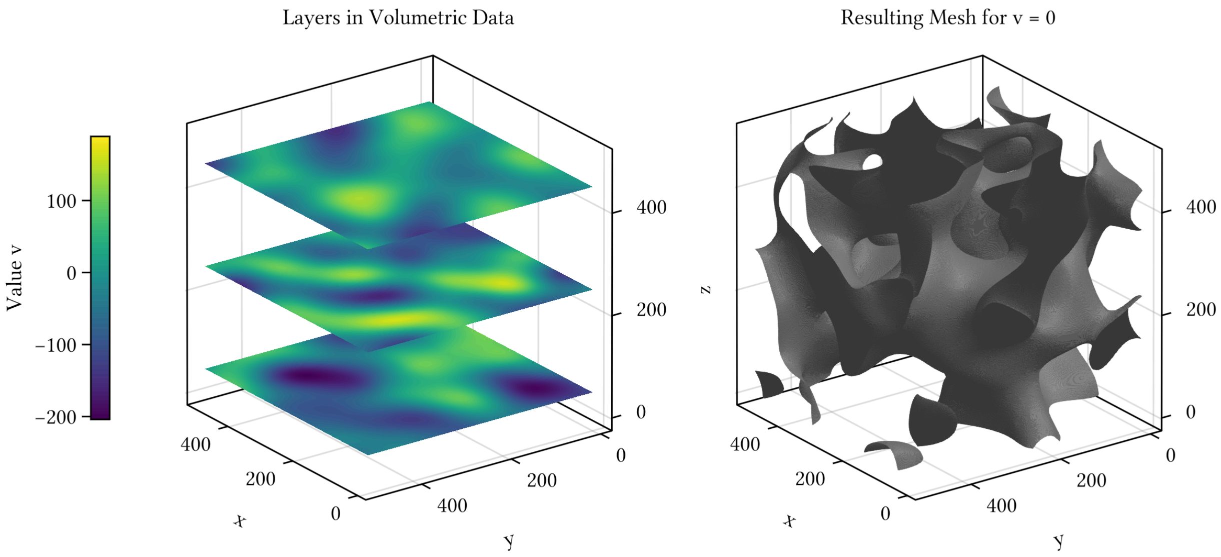

Figure A.2: Different slices of the benchmark data and the resulting mesh.

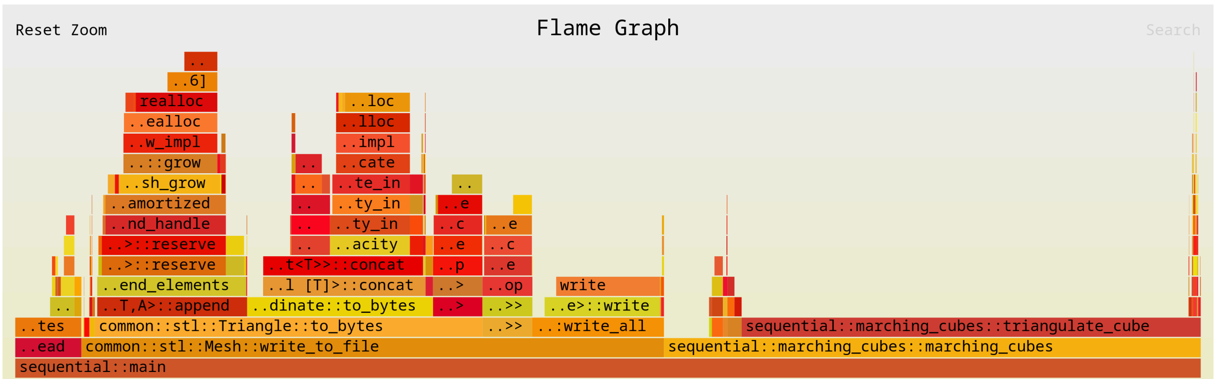

Figure A.3: Flamegraph of the sequential Rust version.
Generated using [31]

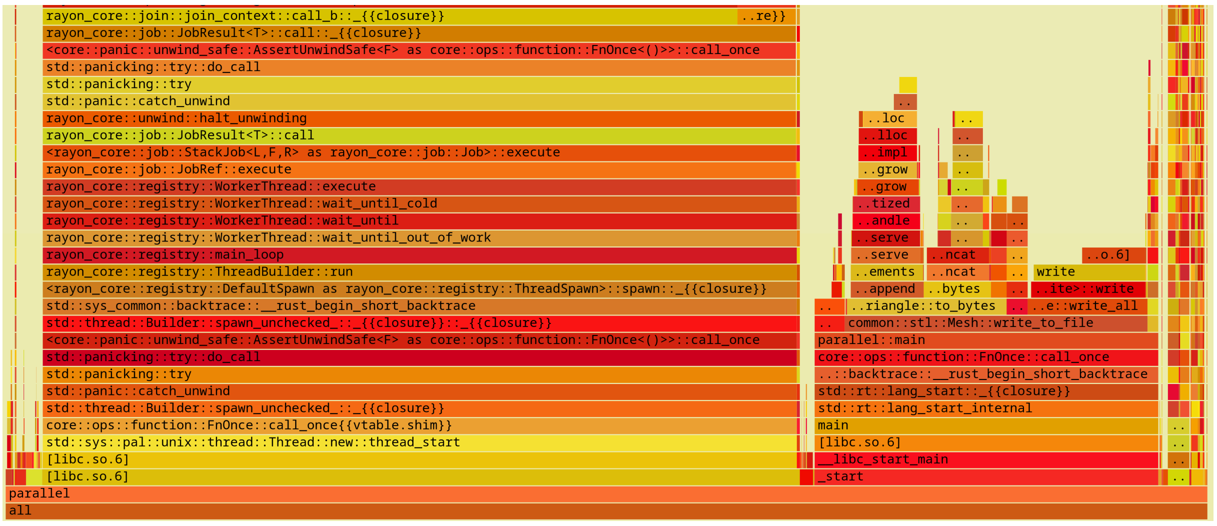

Figure A.4: Excerpt from the flamegraph of the parallel Rust version.
Generated using [31]

B Tables

Specification	Value
OS	Rocky Linux 8
CPUs	2
CPU cores (total)	64
CPU type	Zen3 EPYC 7513
RAM	497,810 MB

Table B.1: Specifications of the node utilized for the benchmarks.

C Code Samples

```c
// [...]
#pragma omp parallel
{

  struct triangle_list *process_results
    = triangle_list_create();

  #pragma omp for nowait
  for (size_t k = 0; k < data->header->shape[2] - 1; k++) {
    // [...]
    struct triangle_list *new_triangles
      = triangulate_cube(&cube, iso_value);

    triangle_list_append_list(
      process_results,
      new_triangles
    );
  }

  #pragma omp critical
  triangle_list_append_list(
    mesh->triangles,
    process_results
  );
}
```

Listing C.1: Parallel iteration in C using OpenMP.

```rust
let triangles: Vec<Triangle>
  = Zip::from(indices(indices_shape))
      .into_par_iter()
      .flat_map(|((i, j, k),)| {
        // [...]
        triangulate_cube(&cube, iso_value)
      })
      .collect();
```

Listing C.2: Parallel iteration in Rust using Rayon.