HPS

h.seeliger@stud.uni-goettingen.de

Henrik Jonathan Seeliger

Rust Programming for HPC

Project Results

Table of Contents

HPC and Programming Languages

Programming Language Characteristics

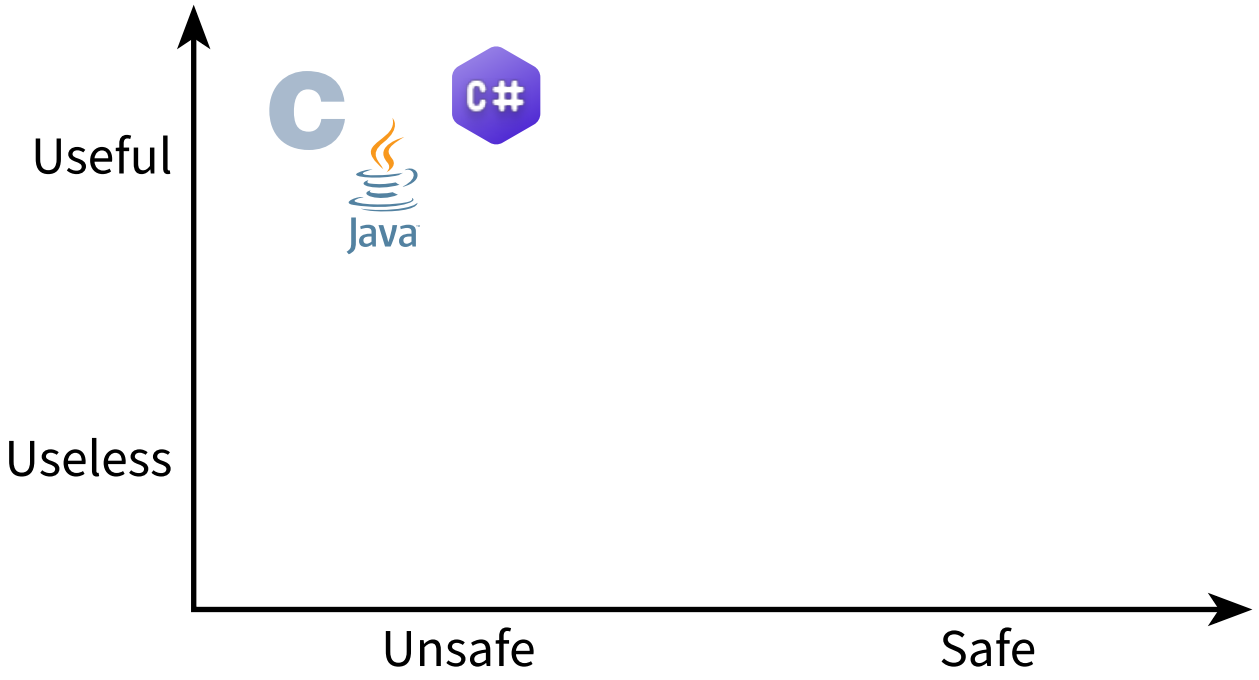
- Large number of aspects that characterize programming languages
- Examples: **Usefulness** and **Safety** [1]
- Usefulness: Performant and efficient resulting machine code, real-world applicability
- Safety: Robustness to bugs and failures

Simon Peyton Jones
Source: [2]

Big Picture of Programming Languages

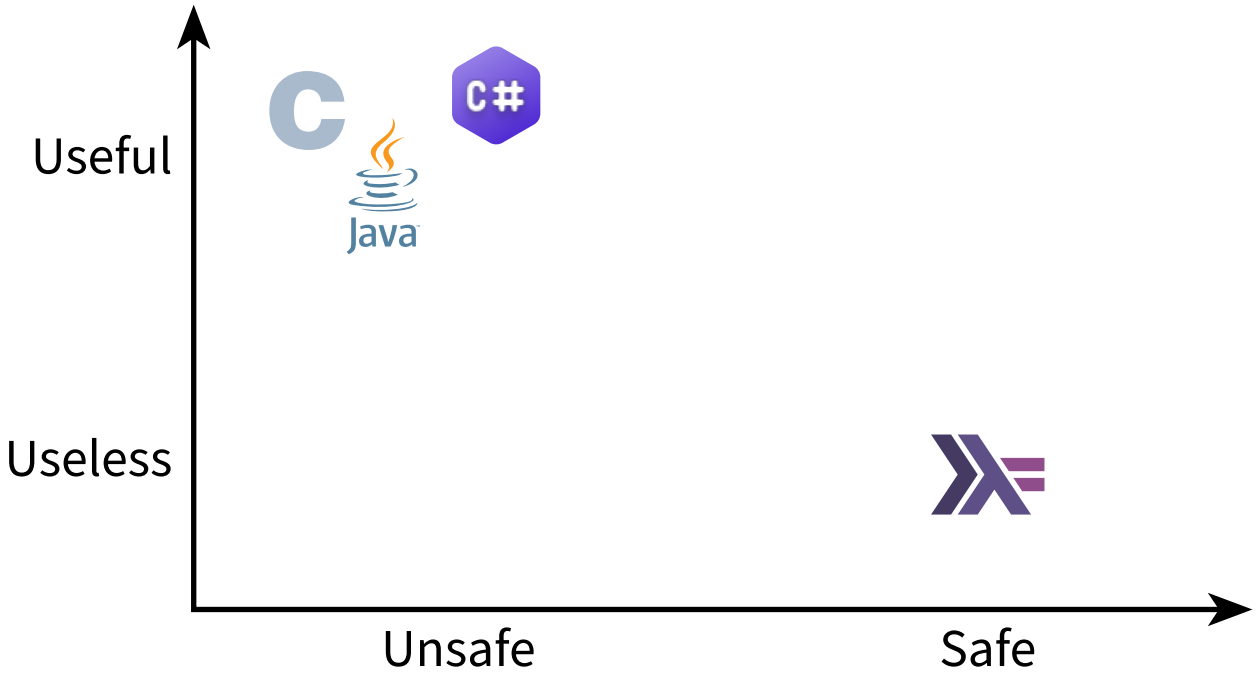
Adapted from [1], icons from [3]–[6]

Big Picture of Programming Languages



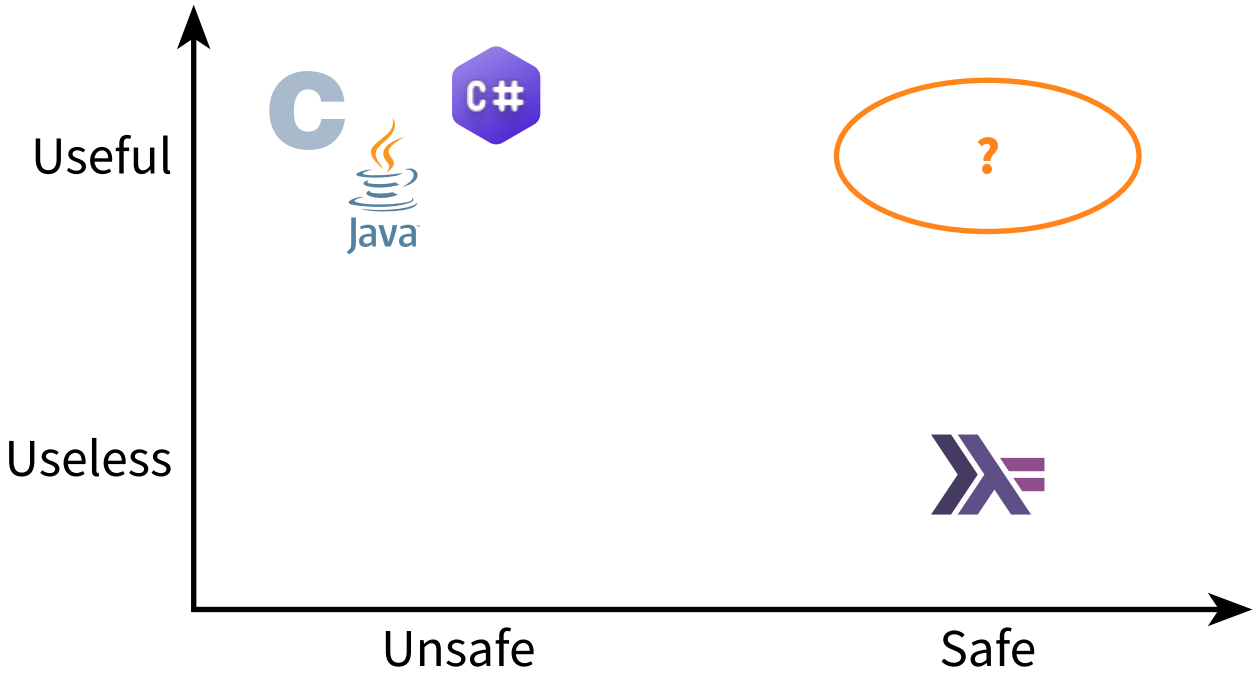
Adapted from [1], icons from [3]–[6]

Big Picture of Programming Languages



Adapted from [1], icons from [3]–[6]

Big Picture of Programming Languages



Adapted from [1], icons from [3]–[6]

Relevance for HPC

- Goal of high-performance computing:
  - Maximize computational power → solve computational problems as fast as possible [7]

- Enabled by software with optimal
  - Performance
  - Efficiency
  - Robustness

- Enabled by programming languages with optimal
  - Usefulness
  - Safety

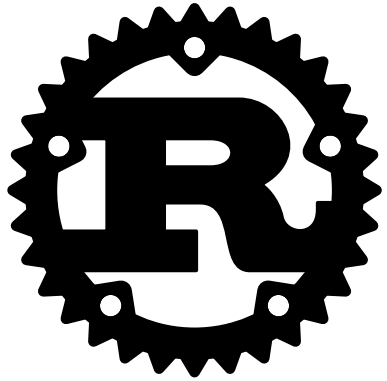
The Project

Current HPC Programming Languages

- Currently widely used languages in HPC are not perfect
  - Cutbacks in performance, safety, …
- Example: C
  - Performance at the cost of safety
  - E.g. memory safety and robustness
- An alternative for HPC that values the described requirements could be valuable

Rust

- Started in 2006, first stable release in 2015
- Now adopted by leading software companies, e.g. Google, Amazon and Microsoft [8]–[10]
- Since 2022 third supported language in Linux kernel development [11]
- Main goals: **Performance**, **Reliability** and **Productivity** [12]

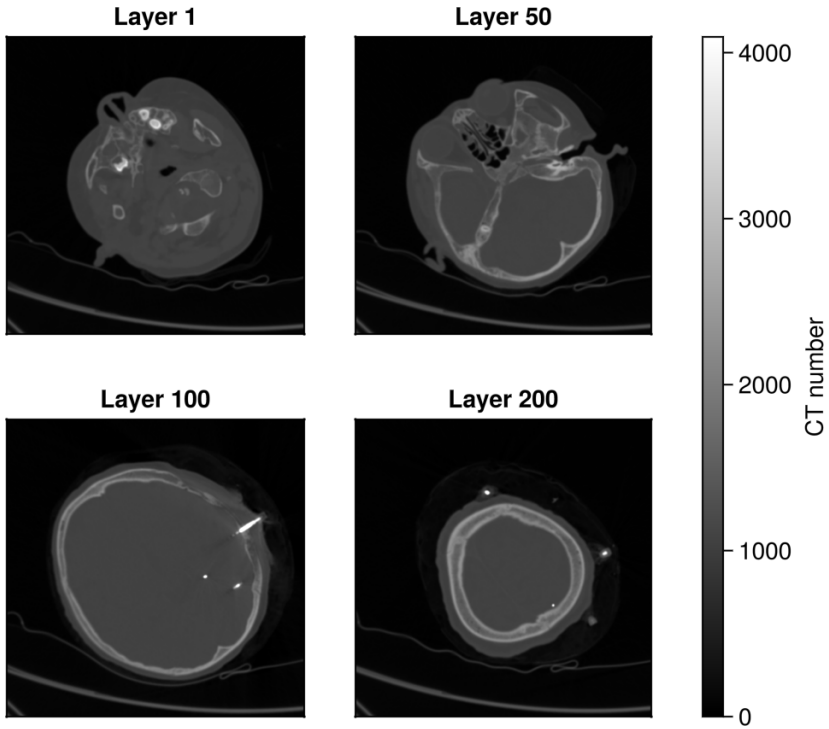


⇒ A viable alternative?

Practical Study

- Marching Cubes Algorithm: Three-dimensional field and iso value to surface mesh
- Implement in C and Rust

⇒ Compare both versions, identify Rust's capabilities and evaluate Rust for HPC programming



Data from [13]

Rust for HPC

Rust: General Characteristics

- C-like syntax
- Imperative with declarative features
- Focus on system level while providing high-level features
- Compiled to native machine code, linked statically by default

Syntax

Euclidean distance implemented in **C**

```c
#include <stdio.h>
#include <math.h>

int main() {
    int vec_a[] = {1, 2, 3};
    int vec_b[] = {4, 5, 6};

    int squared_sum = 0;
    for (int i = 0; i < 3; i++) {
        squared_sum += pow((vec_a[i] - vec_b[i]), 2);
    }

    double eucl_distance = sqrt(squared_sum);
    printf("Result: %f", eucl_distance);

    return 0;
}
```

Syntax

Euclidean distance implemented in **C**

```c
#include <stdio.h>
#include <math.h>

int main() {
    int vec_a[] = {1, 2, 3};
    int vec_b[] = {4, 5, 6};

    int squared_sum = 0;
    for (int i = 0; i < 3; i++) {
        squared_sum += pow((vec_a[i] - vec_b[i]), 2);
    }

    double eucl_distance = sqrt(squared_sum);
    printf("Result: %f", eucl_distance);

    return 0;
}
```

Euclidean distance implemented in **Rust**

```rust


fn main() {
  let vec_a: [i32; 3] = [1, 2, 3];
  let vec_b = [4, 5, 6];

  let mut squared_sum = 0;
  for (x1, x2) in vec_a.into_iter().zip(vec_b) {
      squared_sum += (x1 - x2).pow(2)
  }

  let eucl_distance = (squared_sum as f64).sqrt();
  print!("Result: {eucl_distance}");


}
```

Syntax

Euclidean distance implemented in **C**

```c
#include <stdio.h>
#include <math.h>

int main() {
    int vec_a[] = {1, 2, 3};
    int vec_b[] = {4, 5, 6};

    int squared_sum = 0;
    for (int i = 0; i < 3; i++) {
        squared_sum += pow((vec_a[i] - vec_b[i]), 2);
    }

    double eucl_distance = sqrt(squared_sum);
    printf("Result: %f", eucl_distance);

    return 0;
}
```

Euclidean distance implemented in **Rust**

```rust


fn main() {
  let vec_a: [i32; 3] = [1, 2, 3];
  let vec_b = [4, 5, 6];

  let mut squared_sum = 0;
  for (x1, x2) in vec_a.into_iter().zip(vec_b) {
      squared_sum += (x1 - x2).pow(2)
  }

  let eucl_distance = (squared_sum as f64).sqrt();
  print!("Result: {eucl_distance}");


}
```

Syntax

Euclidean distance implemented in **C**

```c
#include <stdio.h>
#include <math.h>

int main() {
    int vec_a[] = {1, 2, 3};
    int vec_b[] = {4, 5, 6};

    int squared_sum = 0;
    for (int i = 0; i < 3; i++) {
        squared_sum += pow((vec_a[i] - vec_b[i]), 2);
    }

    double eucl_distance = sqrt(squared_sum);
    printf("Result: %f", eucl_distance);

    return 0;
}
```

Euclidean distance implemented in **Rust**

```rust


fn main() {
  let vec_a: [i32; 3] = [1, 2, 3];
  let vec_b = [4, 5, 6];

  let mut squared_sum = 0;
  for (x1, x2) in vec_a.into_iter().zip(vec_b) {
      squared_sum += (x1 - x2).pow(2)
  }

  let eucl_distance = (squared_sum as f64).sqrt();
  print!("Result: {eucl_distance}");


}
```

Syntax

Euclidean distance implemented in **C**

```c
#include <stdio.h>
#include <math.h>

int main() {
    int vec_a[] = {1, 2, 3};
    int vec_b[] = {4, 5, 6};

    int squared_sum = 0;
    for (int i = 0; i < 3; i++) {
        squared_sum += pow((vec_a[i] - vec_b[i]), 2);
    }

    double eucl_distance = sqrt(squared_sum);
    printf("Result: %f", eucl_distance);

    return 0;
}
```

Euclidean distance implemented in **Rust**

```rust


fn main() {
  let vec_a: [i32; 3] = [1, 2, 3];
  let vec_b = [4, 5, 6];

  let mut squared_sum = 0;
  for (x1, x2) in vec_a.into_iter().zip(vec_b) {
      squared_sum += (x1 - x2).pow(2)
  }

  let eucl_distance = (squared_sum as f64).sqrt();
  print!("Result: {eucl_distance}");


}
```

Syntax: HPC Relevance

Advantages	Disadvantages
• Easy to learn for developers with C-like background • Advanced syntax features, e.g. string interpolation, improve readability and maintainability of source code	• Can get complicated and confusing

- Relevance for HPC:
  → Increased productivity, easy to learn, increased maintainability

Semantics: Declarative Features

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

Iteration in Rust using for-loop

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

Iteration in Rust using reactive constructs

Semantics: Advanced Typing

```rust
#[derive(Debug)]
pub enum NpyError {
    OpenFileError(io::Error),
    MissingMagicString,
    UnexpectedNumberOfBytesRead(usize, usize),
}

impl From<io::Error> for NpyError {
    fn from(value: io::Error) -> Self {
        NpyError::OpenFileError(value)
    }
}
```

Enums and methods in Rust

```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);
}
```

Structs and generics in Rust

Semantics: Memory Safety

- **Ownership**: Special set of rules governing Rust code

- Enforced by compiler
  - Type-safety at compile time
  - Can be overriden manually if required

- Enhances memory safety drastically and prevents memory bugs, e.g.
  - Memory leaks
  - Dangling pointers
  - Double frees

Semantics: HPC Relevance

Advantages	Disadvantages
• Safety with very little to no performance penalty • Comprehensive standard library for e.g. declarative programming • Excellent type system and extensibility	• Ownership system is unique; steep learning curve • Enforced safety may be exhaustive and unnecessary at points • Cost of higher-level features not always transparent

- Relevance for HPC:
  → Increased productivity and maintainability, safety for little to no overhead, extensibility
  → Cost of certain features and abstractions have to be considered

HPC Ecosystem

- Classic HPC paradigms and technologies:
  - Parallel computations with shared memory, e.g. OpenMP
  - Distributed computing with distributed memory, e.g. MPI

    → Rust's capabilities?

- Rust's semantics enable "fearless concurrency"
  - Comprehensive parallel computing support in standard library
  - Native thread support
  - Mutexes, reference counter, channels, etc.
  - Without sacrificing safety
  - Support for asynchronous programming

Rayon

- But: Like in e.g. C, manual threading for data parallelism can be exhaustive
  → Simple using OpenMP

- Rust's solution: *Rayon* [14]
  - Library for data parallelism
  - Uses Rust's declarative features and constructs for high-level data parallelism

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

Usage example of Rayon

Distributed Computing with Rust?

- At the time of writing, limited options exist for distributed computing using Rust

- Few native Rust solutions
  - Example: *constellation* [15]

- Like for many other languages, e.g. MPI bindings exist [16]
  - Bindings to given C implementation of MPI
  - Abstracts away “unsafe” C code
  - Provides Rust abstractions and higher-level constructs
  - Evaluation is out of scope for this project

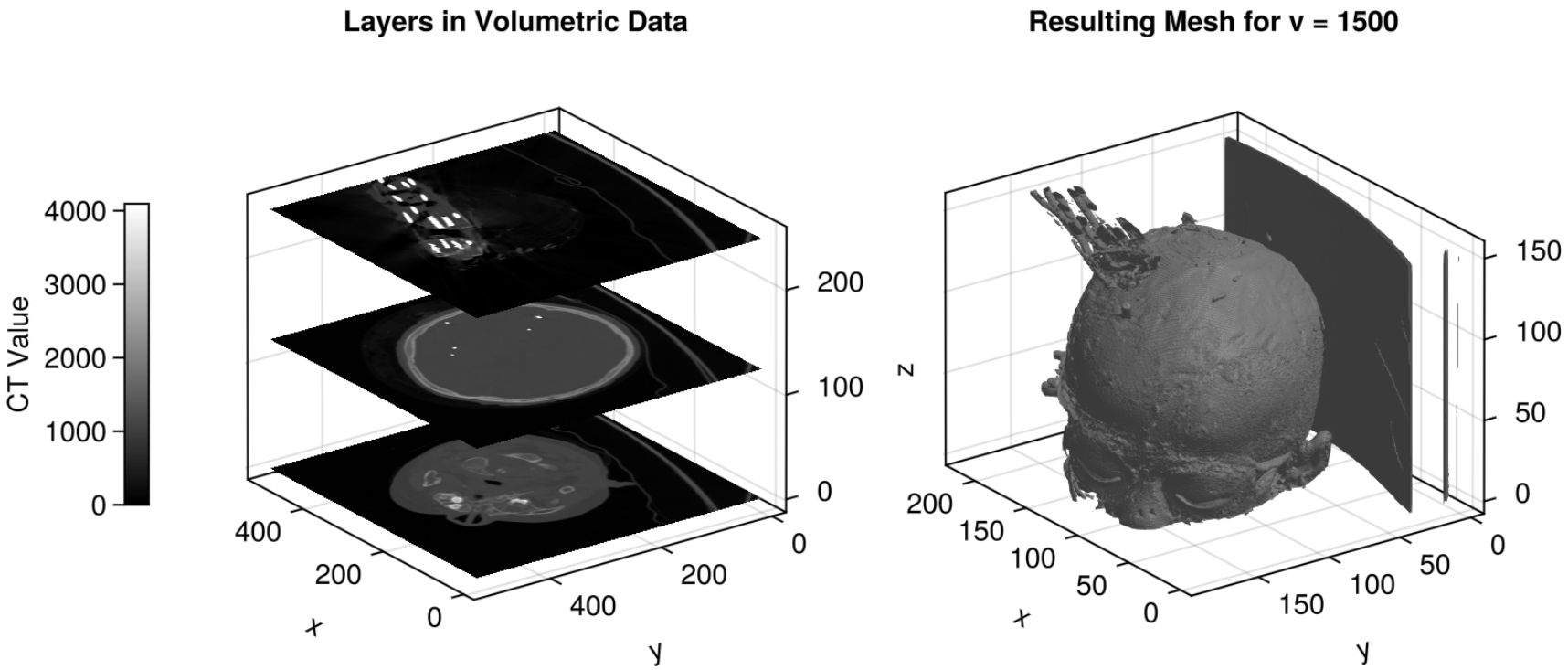
Rust Ecosystem: HPC Relevance

Advantages	Disadvantages
• Good support for concurrent and parallel programming • Safety implied by Rust's semantics • Vast standard library, well-supported external libraries	• Limited support for distributed computing

- Relevance for HPC:
  → Safe concurrency and parallelism with little effort
  → May be unsuited for distributed workloads

Implementation Results

Functional Results



Data from [13]

Implementation Details

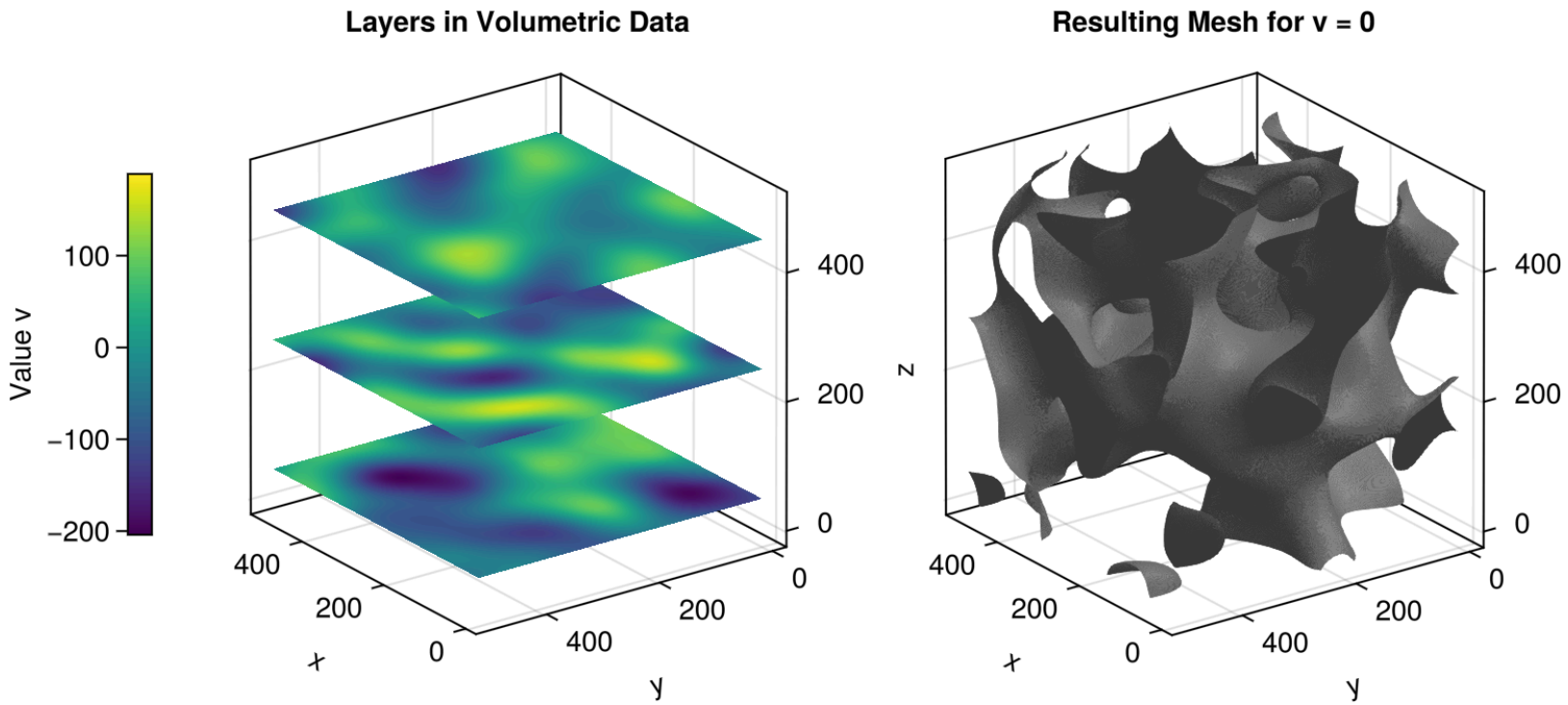
- Goal of implementing Marching Cubes in C as well as Rust:
  - Gain insights about Rust's capabilities
  - Get overview of Rust's performance compared to established HPC language

- Both implementations function in the same way
  - Two versions each: sequential and parallel (shared memory)
  - Rust version offers several enhancement using Rust-specific features, e.g.
    - Extensible typing
    - Error handling
    - Different data representation in memory

Benchmarking Process

- One benchmark for wall-clock time of the sequential version
- One benchmark for analyzing strong-scaling behaviour of the parallel version

Benchmark Data



Data generated using [17]

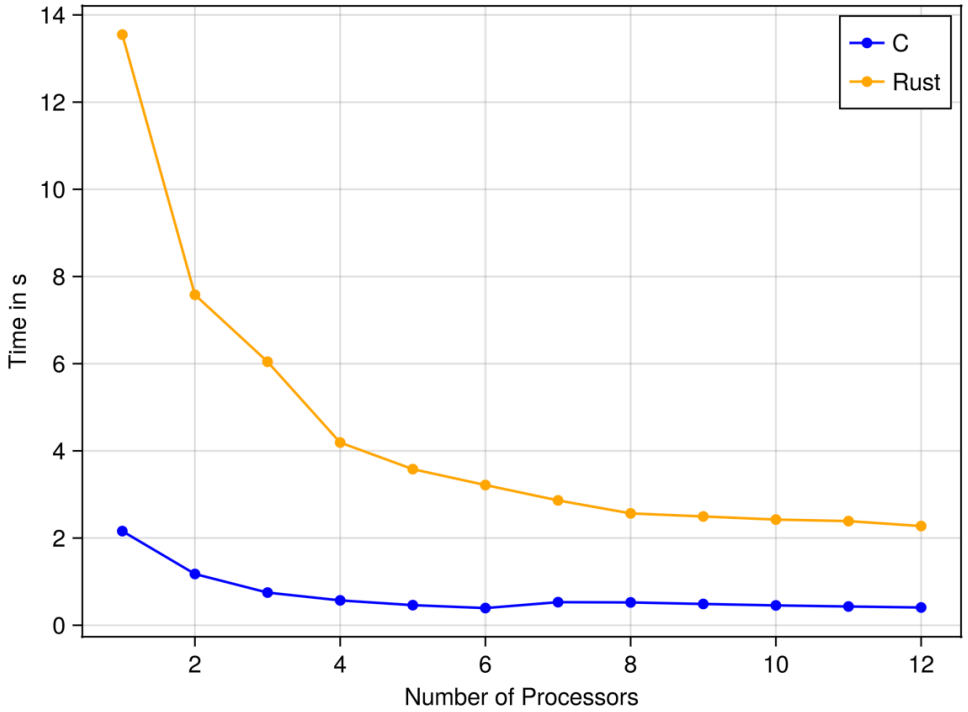
Benchmark Results: Sequential

Implementation	μ	σ	Range
C	2.11 s	0.02	2.09 s – 2.19 s
Rust	6.55 s	0.03	6.53 s – 6.63 s

Time needed for executing the algorithm

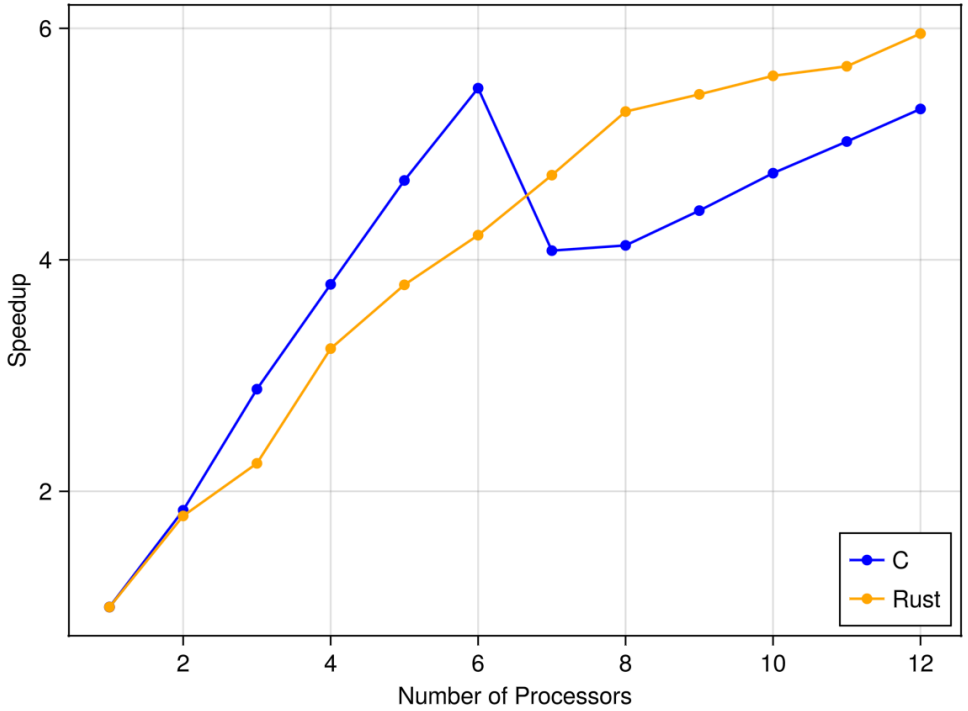
Executed on a Gigabyte AERO 15XV8 with an Intel Core i7-8750H CPU
(4.10 GHz, 12 threads) and DDR4 RAM (16GB, 2667 MT/s), 20 iterations executed

Benchmark Results: Parallel



Runtime of the parallel version, depending on the number of processors used

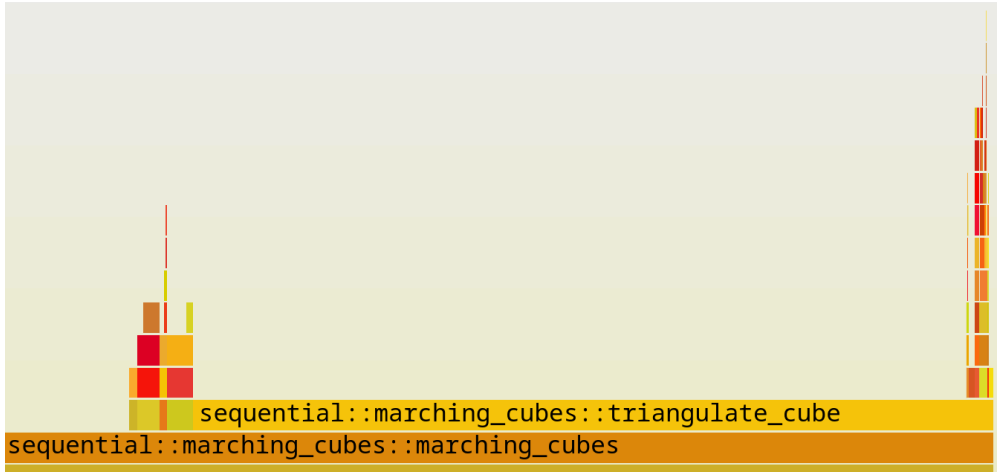
Benchmark Results: Parallel



Speedup of the parallel version

Benchmark Analaysis

- Results contradict expectations set by Rust
- Existing C debugging tools work with Rust
- Flamegraph allows analysis to some degree [18]
- Several hypotheses, no hard evidences yet

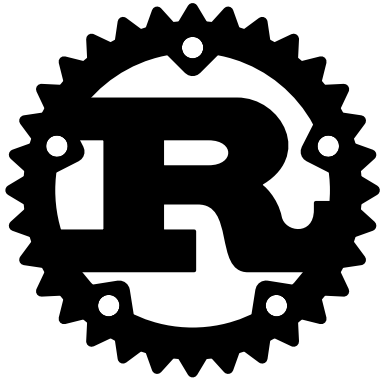


Flamegraph of the sequential version generated using [18]

Summary

Rust: Summary

- High maintainability
- High extensibility
- Memory safety without overhead
- Full control, e.g. over efficiency
- High performance, in theory



Outlook

- Implementations finished, first insights could be gained
  - But: Results are questionable
- Further work this semester:
  - Refine benchmarks and implementations
  - Research reasons for outcomes
- Future work:
  - Evaluate suitability for distributed workloads, e.g. bindings for MPI

Rust for HPC Programming?

- Decision of using Rust for an HPC poject should be based on
  - Project requirements
  - Trade-off between advantages and disadvantages
  - People
- Mature and effective for parallel computations
  - Valuable approaches and solutions that HPC may benefit from
- Possibly valuable for distributed HPC use cases in the future

References

[1] S. P. Jones, *Haskell is useless*, (Dec. 18, 2011). Accessed: Jun. 24, 2024. [OnlineVideo]. Available: https://www.youtube.com/watch?v=iSmkqocn0oQ

[2] S. P. Jones, "Teaching Creative Computer Science," *TEDxExeter 2014*. 2014. Accessed: Jun. 24, 2024. [Online]. Available: https://www.youtube.com/watch?v=Ia55clAtdMs

[3] Rezonansowy, *The C Programming Language logo*. 2013. [Online]. Available: https://commons.wikimedia.org/wiki/File:The_C_Programming_Language_logo.svg

[4] Mark Anderson, *Java programming language logo*. 2021. [Online]. Available: https://en.wikipedia.org/wiki/File:Java_programming_language_logo.svg

[5] Microsoft, *New C Sharp 2023 logo*. 2023. Accessed: Jun. 24, 2024. [Online]. Available: https://commons.wikimedia.org/wiki/File:C_Sharp_Logo_2023.svg

[6] Darrin A. Thompson and Jeff Wheeler, *The Haskell Logo*. 2009. Accessed: Jun. 24, 2024. [Online]. Available: https://commons.wikimedia.org/wiki/File:Haskell-Logo.svg

[7] T. Sterling, M. Anderson, and M. Brodowicz, *High performance computing: Modern systems and Practices*. Oxford, England: Morgan Kaufmann, 2017.

[8] J. V. Stoep and S. Hines, “Rust in the Android platform.” Accessed: May 22, 2024. [Online]. Available: https://security.googleblog.com/2021/04/rust-in-android-platform.html

[9] Jeff 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/

[10] Thomas 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/

[11] Tim Anderson, “Rusty Linux kernel draws closer with new patch adding support for Rust as second language.” Accessed: May 24, 2024. [Online]. Available: https://www.theregister.com/2021/12/07/rusty_linux_kernel_draws_closer/

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

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

[14] 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

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

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

[17] M. a. o. Fiano, “CoherentNoise.jl.” Jun. 30, 2023. Accessed: Jun. 26, 2024. [Online]. Available: https://github.com/lazarusA/CoherentNoise.jl

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