GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

HPS

Seminar Report

RUST Programming for HPC application

Yuvraj Singh

MatrNr: 21621819

Supervisor: Prof. Dr. Christian Boehme

Georg-August-Universität Göttingen
Institute of Computer Science

Summer Semester 2022

Abstract

Languages like FORTRAN, C, and C++ are usually chosen to perform High-Performance Computing (HPC) applications due to the performance they offer. But, these languages are now getting old, and are not safe. Whereas, RUST is relatively a new programming language that offers features like memory safety and user-friendly tooling.

This seminar report will first give an overview of the RUST programming language and the features it offers, to understand how it is a safe programming language. Second, How RUST compares to other programming languages. Finally, where RUST stands to perform HPC applications by covering some actively maintained RUST HPC libraries followed by some used HPC application examples.

Contents

List of Tables **iii**

List of Figures **iii**

Listings **iii**

List of Abbreviations **iv**

1 Rust Programming Language **1**
1.1 "Hello, world!" . . . 1
1.2 Variables and Mutability . . . 2
1.3 Data types . . . 2

2 RUST Features **3**
2.1 Ownership . . . 3
2.2 Borrowing . . . 4
2.3 Fearless Concurrency . . . 4
2.4 Error Handling . . . 4
2.5 Unsafe RUST . . . 4

3 General Comparison **5**
3.1 Memory Management . . . 5
3.2 Performance . . . 5
3.2.1 Test 1 . . . 6
3.2.2 Test 2 . . . 6

4 HPC with RUST **6**
4.1 Some actively maintained RUST libraries for HPC . . . 7
4.1.1 RusataCUDA, an interface to NVIDIA CUDA Driver API . . . 7
4.1.2 RUST-SmartCore, library for Machine learning . . . 7
4.1.3 RUST-Rayon, a library for Data-parallelism . . . 8
4.1.4 RUST-ArrayFire, a library for parallel computing . . . 8
4.1.5 RUST-BIO, a library for bioinformatics . . . 8
4.2 Examples of HPC applications with RUST . . . 8
4.2.1 RUST for Astrophysics; simple N-body physics simulation . . . 8
4.2.2 Transpiling Python to RUST . . . 9

5 Conclusion **11**

References **12**

A Codes Used **A1**

List of Tables

1 Overview of different possible programming languages [Bor21] 5
2 Results for Test-1 6
3 Results for N-body simulator based on leapfrog integrator [BB16] 9
4 Energy consumption for binary-trees [Rui17] 9
5 Results for Black-Scholes Model [HH20] 10

List of Figures

1 Integer types in RUST 2
2 Results for Test-2 6
3 RUST-SmartCore 8
4 RUST-py 10

Listings

1 RUST "Hello, world!" using Bash 1
2 "Hello, world!" main.rs 1
3 Cargo.toml example 1
4 immutable variable 2
5 mutable variable example 2
6 Boolean Type example 3
7 Character, Array and Tuple example 3
8 Unsafe RUST example 4
9 General comparison Test-1 RUST code A1
10 General comparison Test-1 C code A1
11 General comparison Test-1 C++ code A1
12 General comparison Test-1 Python code A1
13 General comparison Test-1 Swift code A1
14 Simple N-body physics simulator RUST code A1
15 Simple N-body physics simulator C code A2
16 Simple N-body physics simulator Go code A4
17 Simple N-body physics simulator FORTRAN code A5

List of Abbreviations

HPC High-Performance Computing

HPDA High-Performance Data Analytics

KB Kilobyte

MB Megabyte

GB Gigabyte

ms Milliseconds

s seconds

m minutes

J Joules

AST Abstract Syntax Tree

1 Rust Programming Language

RUST is a new programming language with numerous application areas. It was initially developed for network services, command line applications, web assembly, and embedded devices. RUST is a systems programming language that strives for safety and performance. It is typed, compiled, and has the property of both functional programming and object-oriented languages. RUST is a strongly statically typed language where data types of the variables are present at compile time instead of run time which boosts the performance.[Vii20; Ash22] RUST's history can be divided into four epochs. The first epoch was from 2006 to 2010 which is known as "The personal years". As it was a personal project of the inventor Graydon Hoare to develop a memory-safe programming language. The second phase lasted from 2010 to 2012, named "The Graydon years", where Graydon Hoare was still the primary maintainer, but the language was already developed as a Mozilla project. Following that, steady improvements on the core concepts led-up to "The type system years" which were from 2012 to 2014, where RUST's type system was majorly overhauled and improved. But after Graydon Hoare stepping back, RUST development became a community-driven project. Finally, the epoch from 2015 to 2016 is known as "The release year" and RUST version 1.0 was released in May of 2015. Since the release 1.0, RUST has evolved over the years, the goal is to provide a complete concurrent, safe, and system programming language. The development of RUST additionally focuses on language stability, user-friendly tooling, and ecosystem.[KN22; Bor21]

1.1 "Hello, world!"

RUST has a compiler named "rustc". "cargo" as a package manager and build system. In RUST, "cargo" is also used to make, build and run programs. One way to make "Hello, world!" program is by using "cargo" commands in shell/ terminal.

```
$ curl https://sh.rustup.rs -sSf | sh # To install RUST and Cargo in Linux/Mac
$ source "$HOME/.cargo/env"
$
$ cargo new hello # To make package "hello".
$ cd hello # Navigating to package "hello".
$ cargo build # TO compile local packages and all of the dependencies.
$ cargo run # To execute "main.rs"
Hello, world!
```

Listing 1: RUST "Hello, world!" using Bash

```
fn main() {
    println!("Hello, world!")
}
```

Listing 2: "Hello, world!" main.rs

```
[package]
name = "hello"
version = "1.61.0"
edition = "2022"

[dependencies]
log = "0.4"
ndarray = "0.10.0"
num = "0.2"
```

Listing 3: Cargo.toml example

"cargo new {package name}" command makes the package, and in it creates "Cargo.toml" and "Cargo.lock" files, with "src" and "target" folders. The "Corgo.toml" file is crucial as all

the dependencies/crates are managed in it. The "main.rs" file holds the main execution code, which is located in "src" folder.

1.2 Variables and Mutability

Variables in RUST are *immutable* by default. They are declared using the *let* keyword. A RUST variable is *immutable* when its value is bound to a name and can not be changed. The code in listing-4 would not compile by showing a compile time error: *cannot assign twice to immutable variable 'y'.*

```
fn main() {
    let y = 5; // declaring variable (immutable by default)
    println!("The value of y is: {y}"); // printing value of "y"
    y = 6; // assigning new value to "y" which will cause compile time error.
}
```

Listing 4: immutable variable

But, it is also possible to make variables in RUST *mutable* which makes it more convenient to write code. RUST's *immutable* variable can be made *mutable* by adding *mut* keyword in front of variable name as shown in listing-5. This time program would not issue any error and will compile.

```
fn main() {
    let mut y = 51; // declaring mutable variable
    println!("The value of x is: {y}"); // printing value of "y"
    y = 16; // assigning new value to "y"
    println!("The value of x is: {y}"); // printing last assigned value to "y"
}
```

Listing 5: mutable variable example

In RUST there are also *constant* value types, which are bound to a name as *immutable*. Moreover, it is not possible to use *mut* keyword with **constants** to make them mutable - as they can not be mutated. Constants are declared using *const* keyword, instead of *let* keyword.

1.3 Data types

Values in RUST are of certain data types. Which explains RUST what type of data is being specified so it knows how to work with it. RUST is *statically typed language*, meaning it should know all the variables at compile time. There are primarily four scalar types: integers, numbers, floating points, Boolean, and characters. RUST's signed integer type starts with *i*, and unsigned integer with *u*. For example, *u64* is for unsigned 64-bit integer type. Signed variants can store numbers from -(2^{n-1}) to (2^{n-1})-1 . Where the number of bits is denoted by *n*. And unsigned variants store numbers from 0 to 2^n -1. Moreover, the *isize* and *usize* types are defined with the respect to the register width of the current architecture. For example, "arch": 64 bits for the machine with 64-bit architecture. Figure-1 shows built-in integer types in RUST.[KN22; Bor21]

Integer Types

Length	Signed	Unsigned
8-bit	i8	u8
16-bit	i16	u16
32-bit	i32	u32
64-bit	i64	u64
128-bit	i128	u128
arch	isize	usize

Figure 1: Integer types in RUST [KN22]

There are two floating point size types in RUST, *f32* and *f64*. which are 32-bit and 64-bit in size, respectively.

RUST supports basic mathematics for all number types: subtraction, addition, division, multiplication, and remainder. Boolean types are also supported. With two possible values: *true* and *false*.[KN22]

```
fn main() {
    let T = true;
    let F: bool = false; // explicit type annotation
}
```

Listing 6: Boolean Type example

RUST also supports, *string type* with complete UTF-8 support, *characters*, *arrays* (compound type), *tuples* (compound type), *structs* and *enums*. Where *character* literal is specified by single quotes, and string literal by double quotes. In RUST, both arrays and tuples have fixed length/size: once declared, they can not grow/shrink.[KN22]

```
fn main() {
    // Character type example
    let char = 'A';
    let H: char = 'B';

    // Array example
    let array1 = [10, 221, 33, 434, 52, 67, 0, 1];
    let array2: [i64; 6] = [12, 231, 333, 434, 532, 12];

    // Tuple example
    let tuple: (f64, i32, u16) = ( 6.8239, 5892, 9);
    let (p, q, r) = tuple; // turning into three separate variables.
}
```

Listing 7: Character, Array and Tuple example

2 RUST Features

2.1 Ownership

The majority of computer programming languages manage memory either by garbage collector or by explicitly allocating and de-allocating the memory. Whereas in RUST, memory is managed by a system called *ownership*. Memory is only valid as long as the owner lives and it is automatically dropped/freed as soon the owner goes out of scope. Ownership is a set of rules that govern how RUST program manages memory with a set of rules that the compiler checks at compile time. The Ownership feature of RUST is one of the most important factor that enables this language to achieve high performance and memory safety without a garbage collector.

Three Ownership rules are:

1. Each value in RUST has an owner.
2. There can only be one owner at a time.
3. When the owner goes out of scope, the value will be dropped.

If any of these three rules are not obeyed, the program will not compile. [KN22]

2.2 Borrowing

Another important aspect of RUST's type system are references, which are also known as *borrowers*. In order to share the past of variables without taking ownership, the *borrowing* technique is used, which is done by using references to the owned variable instead of copying the original variable. The ownership of value is temporarily transferred to an entity and then returned to the original owner. The lifetime of the borrower is only within the function until the ownership is transferred to the original variable. Moreover, the scope of the borrower can not exceed the scope of the owner. The borrowers are immutable by default. But, RUST allows one mutable borrower per variable. This avoids the race condition when two pointers modify or access the same data without synchronization. [Ash22]

2.3 Fearless Concurrency

Splitting the program into multiple threads to run multiple tasks at the same time improves performance, but on other hand also increases complexity. As threads can run simultaneously, there is no assurance about the order in which parts of code on the different threads will run. This can cause problems like:-

- Race conditions: When threads access data in an inconsistent order.
- Deadlocks: When two threads wait for each other, as a result preventing both threads to continue.
- Bugs: These happen in certain situations that are hard to reproduce and fix.

As the majority of computers take advantage of their multiple processors. RUST's one of major goal is to handle concurrent programming safely and efficiently. RUST *fearless concurrency* addresses both concurrent and parallel programming. It is based on the concept of ownership and type system. In Concurrent programming, separate parts of a program are executed independently. In Parallel programming, different parts of the program execute at the same time. By leveraging the concept of *Fearless concurrency*, many concurrency errors are compile-time errors rather than run-time errors. In other words, incorrect code will refuse to compile and output an error at compile time. This allows to write the code that is free of subtle bugs and easy to refactor without introducing new bugs.[KN22]

2.4 Error Handling

2.5 Unsafe RUST

RUST is a safe programming language. But, RUST also has a second language mode hidden inside that does not enforce memory safety guarantees. This mode is known as *Unsafe* RUST. To switch to *Unsafe* RUST, *unsafe* keyword has to be added at beginning of the function block that holds *Unsafe* RUST code as shown in listing-8. This mode is very useful to write code for GPUs, as it is sometimes difficult to accommodate RUST safety guarantees while programming for GPUs.[KN22]

```
fn main() {
    // safe code
    unsafe fn unsafe_function() {
    // unsafe code
    }
    unsafe {
        unsafe_function();
    }
}
```

Listing 8: Unsafe RUST example

Unsafe RUST five *Superpowers* :-

- Deference a raw pointer.
- Calling an *unsafe* function.
- Implementing an *unsafe trait*
- Accessing or modifying a mutable static variable
- Accessing fields of *union*

3 General Comparison

RUST distinct itself from C++ and FORTRAN by bundling tooling with the compiler, which allows for easy dependency management, document generation, and testing. RUST, is still developing - it is not as developed as C++ and FORTRAN. Features like constant generics, constant functions, and support for GPU targets are still under development. Macros in RUST are more hygienic and ergonomic than C and C++, as identifiers declared in macros will not "leak" or collide with one declared by user.[Sud20]

General comparison between computer programming languages can be done based on their performance, how they manage memory, and how much memory they use.

3.1 Memory Management

There are three common types of memory management, The first one is, *automatic management*, where memory is automatically managed. Second, *manual management*, which is done by the programmer itself. In the third type, memory is managed by garbage collection. Where memory allocation can be either automatic or manual, and memory deallocation is done by the garbage collector. Most higher-level programming languages that ensure save memory management, come with a huge performance overhead through the use of either a garbage collector or interpreter. The Rust programming language on the other hand promises zero-cost abstractions for guaranteed memory safety. RUST's *ownership* is behind this. Table 1 provides a comparison of listed languages according to some basic criteria.

Language	Aspect	Memory management	System program	Bare-metal
RUST	Zero-cost safety	Automatic	Yes	Yes
C	Currently in use	Manual +Automatic	Yes	Yes
Ada	Safety critical apps	Manual +Automatic	Yes	Yes
Python	Interpreted	Garbage Collected	NO	NO
Java	JIT Compiler	Garbage Collected	NO	NO
Go	Modern language	Garbage Collected	Yes	NOT officially
Haskell	Functional	Garbage Collected	NO	NO

Table 1: Overview of different possible programming languages [Bor21]

3.2 Performance

Performance means a lot of different things. For example, better performance can also mean better resource utilization. To compare the performance and memory used by different computer programming languages, one of the approaches could be to write the same code in different computer programming languages, respectively.

3.2.1 Test 1

In this test, simple codes to print integers from zero to one million in C, C++, RUST, Python, and Swift were run. 9-13

Machine used - *2,3 GHz 8-Core Intel Core i9 (Mac OS)*

Language	Time taken (ms)	Memory Usage (KB)
C	1500	827
RUST	1420	913
C++	1856	860
Python	3016	8300
Swift	3848	1200

Table 2: Results for Test-1

3.2.2 Test 2

In the second test, five other programming languages than RUST were considered; C, C++, Go, Java and Python. For this test, three algorithms; 1. Bubble sort Algorithm. 2. The Monte Carlo Pi estimation and, 3. Monte Carlo pi estimation simpleRNG algorithm was considered. Code for all three algorithms in each programming language mentioned was written. Figure 2 shows the results. Bar-chart on left shows average CPU time benchmark results and the bar chart on right shows average memory usage benchmark results.[Wil22]

It can be seen RUST achieved the first position for average CPU time in bubble sort and Monte Carlo Pi Estimation algorithm, and lost by a very small margin for the third test from C and C++. The bar chart on the right-hand side shows memory usage. Here we can see RUST came at second position just after C.

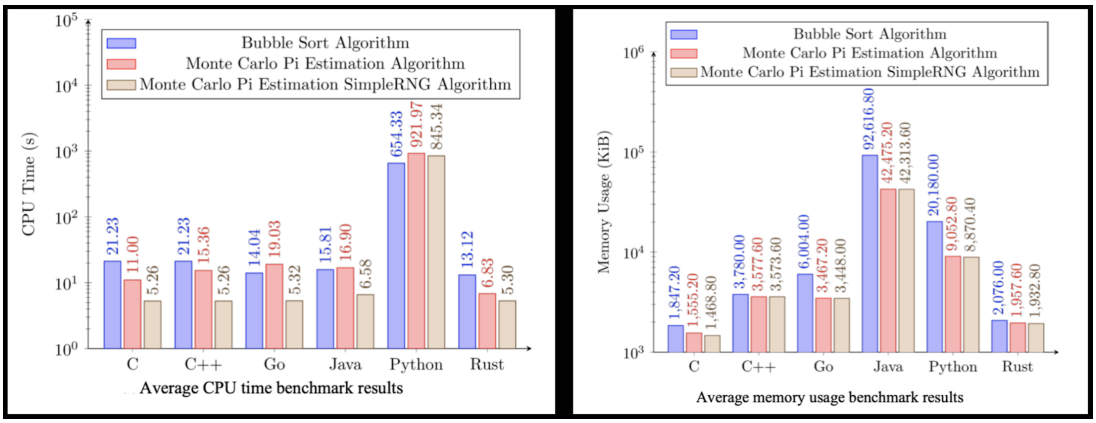

Figure 2: Results for Test-2 [Wil22]

4 HPC with RUST

It is estimated that 49% - 88% of software bugs are caused by memory unsafely. For example, double-free, use-after-free, under-flows and buffer over-, use of uninitialized memory or data races.[Sud20] Choice of programming language and technologies are important to scientific software development, which is dictated by support interface to legacy systems, architectures, convenient usage of modern technologies, and developer familiarity. It is believed that user-friendly

languages like RUST have better potential to gain traction in the developer and scientific community. Moreover, RUST is inline with modern coding standards. In RUST, *rayon* crate supports shared-memory parallelism and Multi-threading. The central package repository is located on https://crates.io and can be used to build complex projects. Some crates are still in development but the feature set is very broad. But, there is a lack of support for GPU programming. The RUST compiler back-end support for NVIDIA graphic cards. Further, RUST code written for GPU will be *unsafe* on most occasions. Crates like *rustcuda* and *accel* provide their best to complete CUDA programming experience in RUST. RUST is capable of doing HPC but of making it fully HPC capable, some environment variables need to be set. For example, *rustacuda* requires CUDA_LIBRARY_DIRECTORY to find CUDA run time libraries. [Sud20]

Lower-level programming languages like C/C++ are the most used for HPC applications. But, they do not offer much in terms of safety (by default). Therefore, it is completely the developer's responsibility to not access invalid memory regions. In the previous section, it was discussed that the RUST programming language addresses safety issues like race conditions, deadlocks, accessing invalid memory regions, and offers convenient error messaging at compile time.[Sud20]

RUST does not include in-built support for multidimensional arrays and complex numbers. But, RUST libraries like *num-complex* offers support for complex numbers, *nalgebra and ndarray* for algebra and n-dimensional array.

4.1 Some actively maintained RUST libraries for HPC

4.1.1 RusataCUDA, an interface to NVIDIA CUDA Driver API

RusataCUDA is a High-level interface NVIDIA CUDA Driver API in RUST. which helps to bring GPU acceleration by an easy-to-use interface to the CUDA toolkit. RustaCUDA makes it easy to transfer data, manage GPU memory, loading and launch compute kernels written in any language. One of the primary design goals of RusataCUDA is to make it familiar to RUST code developers. The second is to provide the safest interface even if many aspects of GPU accelerated computations are difficult to conceal with RUST safety guarantees. In order to use RusataCUDA, one is required to install CUDA version 8 or newer with a CUDA-capable GPU and modify *cargo.toml* file by adding the latest rustacuda dependencies. RusataCUDA is still under development. Moreover, it does not support Multi-GPU support, Run time linking, CUDA Graphs, and more.[Now22]

4.1.2 RUST-SmartCore, library for Machine learning

RUST-Smartcore is a machine learning library also for numerical computing. It supports a large number of well-used machine learning algorithms. Moreover, RUST-Smartcore does not have any hard dependencies on other RUST crates. Compared to other RUST libraries for machine learning it is very well documented with tutorials on RUST-Smartcore's website. This library additionally provides a set of tools for optimization, scientific computing, and linear algebra. To set up RUST-Smartcore, it is required to modify *cargo.toml* file by adding dependencies, and also need to include linear algebra library dependencies like *ndarray and nalgebra*. These libraries enable us to use n-dimensional containers for numerical and general-purpose linear algebra.[Vla22]
Figure 3 shows SmartCore library architecture represented in layers, in which linear algebra and optimization functions are at the first level, as the majority of machine learning algorithms rely on them to fit a model or to make a prediction. Due to this, machine learning algorithms are defined on top of the first level. And then, model evaluation and model selection functions are defined at the top level.

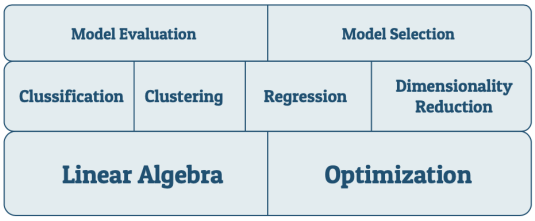

Figure 3: SmartCore's architecture represented as layers. [Vla22]

4.1.3 RUST-Rayon, a library for Data-parallelism

For data-parallelism *rayon* crate is used in RUST. It is a lightweight library that makes it easy to convert sequential computations to parallel ones. Parallel execution causes many kinds of bugs, but *rayon* crate guarantees data-race freedom.[Sud20]

4.1.4 RUST-ArrayFire, a library for parallel computing

RUST-ArrayFire is a high-performance library for parallel computing. It comes with an easy-to-use API. It enables to write portable computing code that is portable across CUDA and OpenCL. Few lines of code in RUST-ArrayFire can replace numerous lines of parallel computing code. This is possible as RUST-ArrayFire abstracts away much of the details by providing a high-level container object, which is an *array*. *Array*, represents data stored on a GPU, CPU or FPGA. This abstraction allows developers to write parallel applications in a high-level language without thinking about low-level optimization. RUST-ArrayFire also includes its own memory manager which reuses device memory whenever possible. Moreover, it also provides additional datatypes like C32 and C64 for complex single and double-precision values. B8 for 8-bit Boolean values and F16 for 16-bit float numbers. To use this library *cargo.toml* file needs to be modified by adding latest ArrayFire dependency.[Now22]

4.1.5 RUST-BIO, a library for bioinformatics

The amount of data is increasing in most sectors. It is the same for experimental data in bioinformatics. Thus, it becomes challenging for computational analysis. So far, many computational libraries for bioinformatics are written in languages like C++ and Python. Languages like C and C++, provide optimal performance but at cost of high complexity and low safety. Whereas, high-level languages like Python come with computational overhead. On the other hand, RUST has abilities of low-level, system programming languages and other features like enforcing thread safety, so that race conditions do not occur as discussed in the RUST features section.

RUST-Bio is a general-purpose bioinformatics library. It also provides a fast, safe set, and high-level API for many algorithms and data structures that are used in bioinformatics. For example, RUST-bio provides suffix arrays, pairwise alignment, an open reading frame (ORF) search algorithm, q-gram index, and major pattern matching algorithms. The performance of this RUST-bio is comparable to that of C++. This library is supported on RUST 1.53.0 or later. And to set up RUST-bio it is required to modify the *cargo.toml* file by adding the most recent RUST-bio dependencies. [Kös15]

4.2 Examples of HPC applications with RUST

4.2.1 RUST for Astrophysics; simple N-body physics simulation

The Astrophysics community generally uses lower-level programming languages like C, C++, and FORTRAN due to the performance they offer. As these languages also rely on developers for memory control and concurrency, which usually causes errors. For example, accessing an

invalid memory region, which can produce random execution behavior and can also affect the scientific results. As RUST is a safe programming language and performance is comparable to C and C++ and also addresses errors caused by accessing an invalid memory region, memory leaks, and race conditions. Therefore, RUST is a good candidate for performing algorithms used in astrophysics.[BB16]

Results - Simple N-body physics simulator
A simple N-Body dynamical simulator code based on a leapfrog integrator was written in Rust, FORTRAN, C, and Go to compare execution time for each language. 14-17 The value of N was set to 2, meaning the position of the two particles after one million years was calculated.

Machine used - *1,6 GHz Intel Core i5 (Linux)*

Language	Time taken
RUST	2m33.082s
C	2m53.504s
FORTRAN	3m16.314s
Go	4m10.233s

Table 3: Results for N-body simulator based on leapfrog integrator [BB16]

From the result in Table 3, it can be inferred that RUST is the fastest followed by C, FORTRAN, and Go respectively.

4.2.2 Transpiling Python to RUST

Python is one of the computer programming languages which is loved by the machine learning and scientific computing community. But it is difficult to achieve high-performance implementation, and it becomes even more difficult with limited computational resources. Nowadays, energy awareness in scientific and industrial computing is becoming a big concern. As the heat generated from computers is not as much responsible for carbon emissions than the resources that are used to generate energy to power computers. A large adaptation of the Python programming language by the developer community can sum up to a huge amount of energy consumption. It can be observed from the table-4 that Python is among the programming languages that use a large amount of energy, for example, to run binary-tree code.[Rui17]

Language	Energy(J)
C	39.80
C++	41.23
RUST	49.07
FORTRAN	69.82
Ada	95.02
Java	111.84
C#	189.74
JavaScript	312.14
Go	636.71
Ruby	855.12
PHP	1,397.51
Python	1,793.49
Perl	3,542.20

Table 4: Energy consumption for binary-trees [Rui17]

Moreover, the execution time and memory that the Python program consumes are not workable in constrained systems. In General, Python-like high-level language code depends on optimized libraries. The main challenge of optimizing to a target are platform-specific optimization, parallelization and cross-library parallelization. [HH20]

In transpiling Python to RUST approach, existing python code is transpiled to RUST. This is based on RUST as an intermediate source code step.

Another good thing is that source-to-source transpiled code allows optimization of code as a whole including the libraries and allows a path from a high level of abstraction to optimized machine code. As Python programs that depend on optimized libraries like SciPy or on other libraries can be transpiled to RUST semi-automatically.

The tools used in this technique are *MonkeyType and pyrs*. Transpilation consists of syntax conversion, manual refactoring, runtime types, and validation testing. First, the python program is run to get all the information about the types used by the program. This allows information required in later steps of transpilation. Second, syntax conversion is applied using a publicly available Abstract Syntax Tree (AST) to Python code. The Python AST is automatically converted into RUST code by visiting each and every AST node using the visitor pattern, which outputs RUST code. After syntax conversion, it is advised to manually check the code, as it might not compile just after syntax conversion. Once the code compiles, it should also be validated that its code functionality is equivalent to that of the original.[HH20] Further, this code can be modified or optimized for better performance. Finally, the performance of the RUST code can be measured. The outline of this approach is depicted in Figure 4.

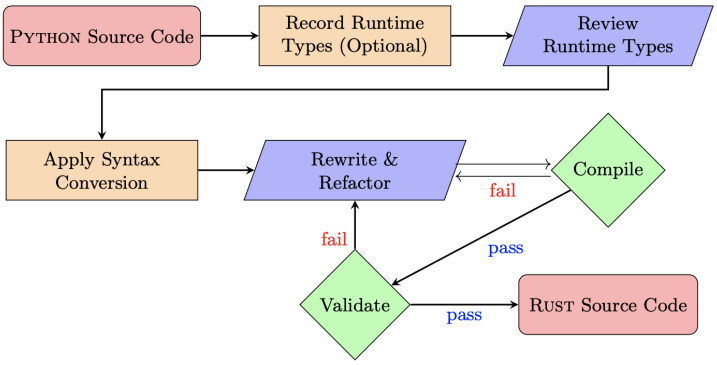

Figure 4: Transpiling Python to RUST [HH20]

Used Case - Black-Scholes Model The Black-Scholes is a model for financial market dynamics, this model was used to compare performance. First, Python code for Black-Scholes model was executed and this source code was transpiled to RUST code.[HH20]

Machine used - *1,6 GHz Intel Core i5 (Linux)*

Results

Language	Time taken	Peak memory consumed
RUST	11.70s	3.456 GB
Python	27.29ss	9.372 GB

Table 5: Results for Black-Scholes Model [HH20]

The result shows that RUST uses 2.7 times less memory resources with 2.3 times less execution time.

Using this approach optimized Python code or Python code for HPC applications can be

transpiled to RUST to get better performance, possibly with less execution time and energy consumption.

5 Conclusion

Before shifting to the RUST programming language, one should examine that the specific RUST library for HPC has all the desired capabilities, as many of RUST HPC libraries are still under development and are not fully functional. RUST is a comparatively new programming language, which is also difficult to learn due to its complex syntax with a different ecosystem; for example crate/library management system. Thus, training must be given to students, researchers, employees, and other users for better exposure. Regardless of that, RUST is very well documented with numerous tutorials and a cooperative RUST community. From the sections presented in this report, it can be concluded that RUST is a viable choice for HPC applications. This is due to its performance in comparison with languages like FORTRAN and C++. Moreover, RUST provides memory safety which hinders the possibility of errors like accessing invalid memory regions, data races, deadlocks, and compile time errors. However, so far RUST programming with GPU is not been that developed so far. And according to [Sud20] it will difficult for GPGPU programming would be hard to achieve while maintaining RUST safety guarantees. Most of the GPU code in RUST written will be *unsafe*. Nevertheless, RUST has no problems calling GPU kernels written in any other programming language. Another advantage of RUST is clean and clear error messages at compile time. There is also the possibility an OpenMP-similar, parallelization model can also be implemented currently in RUST using procedural macros. Unfortunately, the developer and the scientific community seem not that aware of this quality of RUST, and movement in this direction is quite slow.[Sud20] Last but not least, RUST's results were more promising than other programming languages in tests conducted, from performance, memory consumption, and energy consumption perspective. Because of automatic memory management in RUST, users like scientists would be able to focus more on their research work rather than thinking about allocating and freeing memory. Also, not forgetting the *fearless* concurrency of RUST which allows for parallelization with safety guarantees. **By combining all the points RUST is a serious contestant for the next big language in high-performance computing.**

The conclusion further sum-up as:-

- RUST is a **GOOD** candidate to perform HPC applications.
- RUST is getting better (inc. HPC Libraries, ecosystem, etc.).
- RUST has a variety of HPC Libraries.
- RUST is very well documented.
- RUST is safe a language.
- RUST is one of the green languages to perform HPC applications.

References

[Ash22] Gaurav Nattanmai Ganesh Ashwin Kumar Balakrishnan. "Modern C++ and Rust in embedded memory-constrained systems". In: (2022), pp. 9–11.

[BB16] Sergi Blanco-Cuaresma1 and Emeline Bolmont. "What can the programming language Rust do for astrophysics?" In: (2016), pp. 1–4.

[Bor21] Nico Borgsmüller. "The Rust Programming Language for Embedded Software Development". In: (Jan. 2021), pp. 3–6, 56–62.

[HH20] Kai Jylkkä Henri Lunnikivi and Timo Hämäläinen. "Transpiling Python to Rust for Optimized Performance". In: (Oct. 2020).

[KN22] Steve Klabnik and Carol Nichols. "The Rust Programming Language". In: (May 2022). URL: https://doc.rust-lang.org/book/title-page.html.

[Kös15] Johannes Köster. "Rust-Bio: a fast and safe bioinformatics library". In: (Oct. 2015).

[Now22] Anthony Nowell. "Are we learning yet?; A work-in-progress to catalog the state of machine learning in Rust". In: (Sept. 2022). URL: https://www.arewelearningyet.com.

[Rui17] Marco Couto Rui Pereira. "Energy Efficiency across Programming Languages". In: (2017). URL: https://greenlab.di.uminho.pt/wp-content/uploads/2017/09/paperSLE.pdf.

[Sud20] Michal Sudwoj. "Rust programming language in the high-performance computing environment". In: (Sept. 2020), pp. 5–18.

[Vii20] Rasmus Viitanen. "Evaluating Memory for Graph-Like Data Structures in the Rust Programming language: Performance and Usability". In: (2020), pp. 3–12.

[Vla22] Lorenzo Moriondo Vlad Orlov Luis Moreno. "SmartCore user guide". In: (2022). URL: https://smartcorelib.org/user_guide/quick_start.html.

[Wil22] Ayman Alahmar William Bugden. "Rust: The Programming Language for Safety and Performance". In: (June 2022), pp. 3–8.

A Codes Used

```rust
fn main() {
    for i in 0..1000000 {
        println!("{}", i);
    }
}
```

Listing 9: General comparison Test-1 RUST code

```c
#include <stdio.h>
int main(void)
{
    int count;
    for(count = 0; count <= 1000000; count++)
    {
        printf("%d \n", count);
        printf("\n");
    }
}
```

Listing 10: General comparison Test-1 C code

```cpp
#include <iostream>
int main() {
  int input;
  for (int input = 0; input <= 1000000; input++)
    {
    std::cout << "\n" << input;
  }
}
```

Listing 11: General comparison Test-1 C++ code

```python
for i in range(0, 1000000):
    print(i)
```

Listing 12: General comparison Test-1 Python code

```swift
for i in 0...1000000
    {
    print(i)
    }
```

Listing 13: General comparison Test-1 Swift code

```rust
// open source code https://github.com/marblestation/benchmark-leapfrog
const N_PARTICLES: usize = 2;

fn main() {
    let mut time: f64 = 0.;
    let time_step: f64 = 0.08;
    let half_time_step: f64 = time_step/2.;
    let time_limit: f64 = 365.25 * 1e6;
    let mut x: [[f64; 3]; N_PARTICLES] = [[0.; 3]; N_PARTICLES];
    let mut v: [[f64; 3]; N_PARTICLES] = [[0.; 3]; N_PARTICLES];
    let mut a: [[f64; 3]; N_PARTICLES] = [[0.; 3]; N_PARTICLES];
    let mut m: [f64; N_PARTICLES] = [0.; N_PARTICLES];
    m[0] = 0.08; // M_SUN
    m[1] = 3.0e-6; // M_SUN
    x[1][0] = 0.0162; // AU
    x[1][1] = 6.57192058353e-15; // AU
    x[1][2] = 5.74968548652e-16; // AU
```

```
    v[1][0] = -1.48427302304e-14;
    v[1][1] = 0.0399408809121;
    v[1][2] = 0.00349437429104;

    while time <= time_limit {
        integrator_leapfrog_part1(N_PARTICLES, &mut x, &v, half_time_step);
        time += half_time_step;
        gravity_calculate_acceleration(N_PARTICLES, &m, &x, &mut a);
        integrator_leapfrog_part2(N_PARTICLES, &mut x, &mut v, &a, time_step, half_time_step);
        time += half_time_step;
    }
    //println!("Hello, world!");
    println!("{:?}", x)
}

fn integrator_leapfrog_part1(n_particles: usize, x: &mut [[f64; 3]; N_PARTICLES], v: &[[f64; 3]; N_PARTICLES], half_time_step: f64) {
    for i in 0..n_particles {
    x[i][0]  += half_time_step * v[i][0];
    x[i][1]  += half_time_step * v[i][1];
    x[i][2]  += half_time_step * v[i][2];
    }
}

fn integrator_leapfrog_part2(n_particles: usize, x: &mut [[f64; 3]; N_PARTICLES], v: &mut [[f64; 3]; N_PARTICLES], a: &[[f64; 3]; N_PARTICLES], time_step: f64, half_time_step: f64) {
    for i in 0..n_particles {
    v[i][0] += time_step * a[i][0];
    v[i][1] += time_step * a[i][1];
    v[i][2] += time_step * a[i][2];
    x[i][0] += half_time_step * v[i][0];
    x[i][1] += half_time_step * v[i][1];
    x[i][2] += half_time_step * v[i][2];
    }
}

fn gravity_calculate_acceleration(n_particles: usize, m: &[f64; N_PARTICLES], x: &[[f64; 3]; N_PARTICLES], a: &mut [[f64; 3]; N_PARTICLES]) {
    let g = 6.6742367e-11; // m^3.kg^-1.s^-2
    for i in 0..n_particles {
    a[i][0] = 0.;
    a[i][1] = 0.;
    a[i][2] = 0.;
        for j in 0..n_particles {
      if j == i {
        continue;
            }
            let dx = x[i][0] - x[j][0];
            let dy = x[i][1] - x[j][1];
            let dz = x[i][2] - x[j][2];
            let r = (dx*dx + dy*dy + dz*dz).sqrt();
            let prefact = -g/(r*r*r) * m[j];
            a[i][0] += prefact * dx;
            a[i][1] += prefact * dy;
            a[i][2] += prefact * dz;
        }
  }
}
```

Listing 14: Simple N-body physics simulator RUST code

```
// open source code https://github.com/marblestation/benchmark-leapfrog
```

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

void integrator_leapfrog_part1(int n_particles, double x[][3], double v[][3], double half_time_step){
  for (int i=0;i<n_particles;i++){
    x[i][0]  += half_time_step * v[i][0];
    x[i][1]  += half_time_step * v[i][1];
    x[i][2]  += half_time_step * v[i][2];
  }
}
void integrator_leapfrog_part2(int n_particles, double x[][3], double v[][3], double a[][3], double time_step, double half_time_step){
  for (int i=0;i<n_particles;i++){
    v[i][0] += time_step * a[i][0];
    v[i][1] += time_step * a[i][1];
    v[i][2] += time_step * a[i][2];
    x[i][0]  += half_time_step * v[i][0];
    x[i][1]  += half_time_step * v[i][1];
    x[i][2]  += half_time_step * v[i][2];
  }
}

void gravity_calculate_acceleration(int n_particles, double m[], double x[][3], double a[][3]) {
    double G = 6.6742367e-11; // m^3.kg^-1.s^-2
    for (int i=0; i<n_particles; i++){
    a[i][0] = 0;
    a[i][1] = 0;
    a[i][2] = 0;
        for (int j=0; j<n_particles; j++){
            if (j == i) {
                continue;
            }
            double dx = x[i][0] - x[j][0];
            double dy = x[i][1] - x[j][1];
            double dz = x[i][2] - x[j][2];
            double r = sqrt(dx*dx + dy*dy + dz*dz);
            double prefact = -G/(r*r*r) * m[j];
            a[i][0] += prefact * dx;
            a[i][1] += prefact * dy;
            a[i][2] += prefact * dz;
        }
    }
}

int main(int argc, char* argv[]) {
    const int n_particles = 2;
    double time = 0;
    double time_step = 0.08;
    double half_time_step = 0.5*time_step;
    double time_limit = 365.25 * 1e6;
    double x[n_particles][3];
    double v[n_particles][3];
    double a[n_particles][3];
    double m[n_particles];

    for (int i=0; i<n_particles; i++) {
        m[i] = 0;
        x[i][0] = 0;
        x[i][1] = 0;
        x[i][2] = 0;
```

```
        v[i][0] = 0;
        v[i][1] = 0;
        v[i][2] = 0;
        a[i][0] = 0;
        a[i][1] = 0;
        a[i][2] = 0;
    }
    m[0] = 0.08; // M_SUN
    m[1] = 3.0e-6; // M_SUN
    x[1][0] = 0.0162; // AU
    x[1][1] = 6.57192058353e-15; // AU
    x[1][2] = 5.74968548652e-16; // AU
    v[1][0] = -1.48427302304e-14;
    v[1][1] = 0.0399408809121;
    v[1][2] = 0.00349437429104;

    while(time <= time_limit) {
        integrator_leapfrog_part1(n_particles, x, v, half_time_step);
        time += half_time_step;
        gravity_calculate_acceleration(n_particles, m, x, a);
        integrator_leapfrog_part2(n_particles, x, v, a, time_step,
    half_time_step);
        time += half_time_step;
    }
    printf("Position 1: %f %f %f | Position 2: %f %f %f\n", x[0][0], x[0][1], x
    [0][2], x[1][0], x[1][1], x[1][2]);
}
```

Listing 15: Simple N-body physics simulator C code

```
// open source code https://github.com/marblestation/benchmark-leapfrog
package main

import (
  "fmt"
  "math"
)

const (
  n_particles = 2
  G           = 6.6742367e-11 // m^3.kg^-1.s^-2
)

func main() {
    var time float64 = 0
    var time_step float64 = 0.08
    var half_time_step float64 = time_step/2.
    var time_limit float64 = 365.25 * 1e6

  //// Create slices and not arrays, since arrays are passed by copy to func
  x := &[n_particles][3]float64{
    {0, 0, 0},
    {0.0162, 6.57192058353e-15, 5.74968548652e-16}, // AU
  }
  v := &[n_particles][3]float64{
    {0, 0, 0},
    {-1.48427302304e-14, 0.0399408809121, 0.00349437429104},
  }
  a := &[n_particles][3]float64{}
  m := &[n_particles]float64{0.08, 3.0e-6} // M_SUN

    for time <= time_limit {
        integrator_leapfrog_part1(x, v, half_time_step)
        time += half_time_step
```

```
        gravity_calculate_acceleration(m, x, a)
        integrator_leapfrog_part2(x, v, a, time_step, half_time_step)
        time += half_time_step
    }
    fmt.Println("Positions:", x)
}

func integrator_leapfrog_part1(x, v *[n_particles][3]float64, half_time_step float64) {
  for i := 0; i<n_particles; i++ {
    x[i][0]  += half_time_step * v[i][0]
    x[i][1]  += half_time_step * v[i][1]
    x[i][2]  += half_time_step * v[i][2]
  }
}

func integrator_leapfrog_part2(x, v, a *[n_particles][3]float64, time_step, half_time_step float64) {
  for i := 0; i<n_particles; i++ {
    v[i][0] += time_step * a[i][0]
    v[i][1] += time_step * a[i][1]
    v[i][2] += time_step * a[i][2]
    x[i][0]  += half_time_step * v[i][0]
    x[i][1]  += half_time_step * v[i][1]
    x[i][2]  += half_time_step * v[i][2]
  }
}

func gravity_calculate_acceleration(m *[n_particles]float64, x, a *[n_particles][3]float64) {
  for i := 0; i<n_particles; i++ {
    a[i][0] = 0
    a[i][1] = 0
    a[i][2] = 0
    for j := 0; j<n_particles; j++ {
      if j == i {
        continue
            }
            dx := x[i][0] - x[j][0]
            dy := x[i][1] - x[j][1]
            dz := x[i][2] - x[j][2]
            r := math.Sqrt(dx*dx + dy*dy + dz*dz)
            prefact := -G/(r*r*r) * m[j]
            a[i][0] += prefact * dx
            a[i][1] += prefact * dy
            a[i][2] += prefact * dz
        }
  }
}
```

Listing 16: Simple N-body physics simulator Go code

```
! open-source code https://github.com/marblestation/benchmark-leapfrog
program leapfrog
    implicit none

    integer, parameter :: n_particles = 2
    real, dimension(n_particles) :: m
    real, dimension(3,n_particles) :: x, v, a
    real, dimension(3) :: dR
    real :: t, dt, t_end, r2
    real :: time, time_step, time_limit, half_time_step
    real, parameter :: G = 6.6742367e-11 ! m^3.kg^-1.s^-2
    integer :: i
```

```
13
14      time = 0
15      time_step = 0.08 ! time step, days
16      time_limit = 365.25e6 ! days
17      ! Set initial conditions
18      m(:) = (/0.08,3.0e-6/) ! M_SUN
19      x(:,1) = 0.0
20      x(:,2) = (/0.0162,6.57192058353e-15,5.74968548652e-16/) ! AU
21      v(:,1) = 0.0
22      v(:,2) = (/-1.48427302304e-14,0.0399408809121,0.00349437429104/)
23      a(:,:) = 0.0
24
25      half_time_step = 0.5d0*time_step
26      do while (time.le.time_limit)
27          call integrator_leapfrog_part1(n_particles,x,v,half_time_step)
28          time = time + half_time_step
29          call gravity_calculate_acceleration(n_particles,m,x,a)
30          call integrator_leapfrog_part2(n_particles,x,v,a,time_step,
    half_time_step)
31          time = time + half_time_step
32      enddo
33      write(*,*) x
34
35  end program leapfrog
36
37  subroutine integrator_leapfrog_part1(n_particles,x,v,half_time_step)
38      implicit none
39
40      ! Input/Output
41      integer,intent(in) :: n_particles
42      real,intent(out) :: x(3,n_particles)
43      real,intent(in) :: v(3,n_particles)
44      real,intent(in) :: half_time_step
45
46      ! Local
47      integer :: i
48
49      do i=1,n_particles
50          ! Positions
51          x(1,i) = x(1,i) + half_time_step * v(1,i)
52          x(2,i) = x(2,i) + half_time_step * v(2,i)
53          x(3,i) = x(3,i) + half_time_step * v(3,i)
54      enddo
55  end subroutine integrator_leapfrog_part1
56
57  subroutine integrator_leapfrog_part2(n_particles,x,v,a,time_step,half_time_step)
58      implicit none
59
60      ! Input/Output
61      integer,intent(in) :: n_particles
62      real,intent(out) :: x(3,n_particles), v(3,n_particles)
63      real,intent(in) :: a(3,n_particles)
64      real,intent(in) :: time_step, half_time_step
65
66      ! Local
67      integer :: i
68
69      do i=1,n_particles
70          ! Velocities
71          v(1,i) = v(1,i) + time_step * a(1,i)
72          v(2,i) = v(2,i) + time_step * a(2,i)
73          v(3,i) = v(3,i) + time_step * a(3,i)
74
```

```
        ! Positions
        x(1,i) = x(1,i) + half_time_step * v(1,i)
        x(2,i) = x(2,i) + half_time_step * v(2,i)
        x(3,i) = x(3,i) + half_time_step * v(3,i)
    enddo
end subroutine integrator_leapfrog_part2

subroutine gravity_calculate_acceleration(n_particles,m,x,a_grav)
    implicit none

    ! Input/Output
    integer,intent(in) :: n_particles
    real,intent(in) :: x(3,n_particles)
    real,intent(in) :: m(n_particles)
    real, intent(out) :: a_grav(3,n_particles)

    ! Local
    integer :: i,j
    real :: dx,dy,dz,rr,prefact,G
    !------------------------------------------------------------------------

    G = 6.6742367e-11 ! m^3.kg^-1.s^-2
    do i = 1,n_particles
        ! Initialization
        a_grav(1,i) = 0.0d0
        a_grav(2,i) = 0.0d0
        a_grav(3,i) = 0.0d0

        do j = 1,n_particles
            if (i.ne.j) then
                dx = x(1,i) - x(1,j)
                dy = x(2,i) - x(2,j)
                dz = x(3,i) - x(3,j)
                rr = sqrt(dx*dx + dy*dy + dz*dz)
                prefact = -G*m(j)/(rr*rr*rr)

                a_grav(1,i) =  a_grav(1,i) + prefact * dx
                a_grav(2,i) =  a_grav(2,i) + prefact * dy
                a_grav(3,i) =  a_grav(3,i) + prefact * dz
            endif
        enddo
    enddo

    !------------------------------------------------------------------------
    return
end subroutine gravity_calculate_acceleration
```

Listing 17: Simple N-body physics simulator FORTRAN code