ISSN: 2834-7706

Research Article

Journal of Mathematical Techniques and Computational Mathematics

Rust vs C++, a Battle of Speed and Efficiency

Vincent Ng*

Computer Science St Joseph International School Kuala Lumpur, 57000, Malaysia.

***Corresponding Author**
Vincent Ng, Computer Science St Joseph International School Kuala Lumpur, 57000, Malaysia.

Submitted: 2023, May 18; **Accepted:** 2023, Jun 17; **Published:** 2023, June 21

Citation: Vincent, Ng. (2023). Rust vs C++, a Battle of Speed and Efficiency. *J Math Techniques Comput Math, 2*(6), 216-220.

Abstract
This study compares the performance of two excellent options for system-level development, the programming languages C++ and Rust. Through a series of tests and experiments using socket servers and various algorithms, this experiment analyses the speed and efficiency of code written in each language by looking at variables like memory management, and compilation times. The findings reveal that Rust has a number of advantages over C++, including quicker compilation times, improved memory safety, and in many situations equivalent or better performance. C++ still performs exceptionally well in several fields, nevertheless, such as low-level hardware programming and backward compatibility. Overall, the results indicate that Rust is a strong candidate for systems programming jobs, especially for new projects or those requiring a high level of performance and security.

Keywords: Rust, C++, Algorithms, Benchmarking, Compiler, Memory Safety.

1. Introduction
Rust and C++ differ from each other primarily in terms of efficiency and speed. Rust has established itself as a rival to C++ over the past few years. These factors are particularly crucial for system-level programmes like socket servers, encryption algorithms, and even hardware, where even the smallest performance improvements can have a significant impact on overall system performance. Understanding the performance disparities between Rust and C++ and how these languages compare in terms of their capacity to generate quick and effective code is therefore of great importance.

1.1 Purpose
This project's main purpose is to compare how well Rust performs in direct comparison to its peers such as C++ by developing similar applications and pushing Rust and C++ to their limits with sorting algorithms.

1.2 Problem Statement
The aim of this piece of research is to answer the question: Is Rust or C++ better for performance-based applications?

1.3 Context
My machine is a 2020 Macbook Air with the M1 Chip (8GB ram), I will be using Rust 1.66.0 and C++ 17 to do the tests. The M1 chip has 8 cores which are sufficient for experimenting.

1.4 Project Scope
Performance measurements (execution time, compilation time) and implementation complexity (lines of code) will be used to assess which language is most appropriate. In this project, I'll write an identical programme in both C++ and Rust as a case study. I'll quantify software complexity using LOC (lines of code).

1.5 Outline
In Chapter 2 'Method' I detail the exact implementations of our programs and how the results are produced. Chapter 3 'Results' presents these results which are then discussed in Chapter 5 'The Analysis'. Chapter 6 'The Conclusion' presents my conclusion.

2. Experiment
2.1 Method
To compare the two languages, testing sorting algorithms like Counting Sort and Bubble Sort against large inputs will be one of our methods to measure software complexity and performance. To avoid advantages/disadvantages given by certain external libraries in each respective language, all code will be written using their default built-in libraries.

2.2 Measurements
In this experiment, I will be using the time unix command built-in Linux with some flags to get the compilation time, as for CPU/memory consumption measurements I'll be using Gtime on Mac (GNU-time). As for LOC, I'll be just simply counting the lines of code used to create the program (not including blank lines and comments). The exact command used can be seen in Figure 1.

To ensure maximum performance and disadvantages of using system libraries for benchmarking, I decided not to include benchmarking the program in the code itself and instead use an external program to benchmark the program as I believe it is more accurate and fair that way. Other than that, to get an accurate representation the same programs will be run multiple times and an average will be taken to compare. To summarise, measurements will be separated into two categories, Category 1, Software Complexity (LOC), and Category 2, Software Performance (Compilation Time, Execution Time).

```
> gtime -f "real %es\nuser %Us \nsys  %Ss \nMemory Used:%MKB \ncpu %P" g+
 -std=c++17 main.cpp
real 0.59s
user 0.23s
sys  0.15s
Memory Used:67760KB
cpu 64%
```

Figure 1: Command Used for Measurements

2.3 Code Used

2.3.1 Counting Sort

I will first explain Counting Sort before showing the code. Counting Sort has a time-space complexity of O(n+k). Counting Sort works by counting unique elements in an array; the algorithm makes sure that each element is in a specified range, it then creates a count array to store the number of occurrences of each element in the input array. The count array is then modified such that each element at index I stores the sum of the previous elements, giving the starting index for each element in the sorted output array. A separate output array is created to store the sorted elements, the input array is iterated over and over resulting in a sorted result array, refer to the pseudocode to see how it should look in code. For my Rust code for this implementation refer to Figure 2. As for my C++ implementation look at Figure 3. The way I came out with these pieces of code is I would write my own version of the sorting algorithm and consider other developers' implementation of a similar algorithm and make changes to my code to fully optimise the program.

```
procedure countingSort(array: array of integers, k: integer)
    let counts be an array of k+1 zeros
    let output be an array of the same size as array
    for i := 0 to length(array)-1 do
        counts[array[i]] := counts[array[i]] + 1
    end for
    for i := 1 to k do
        counts[i] := counts[i] + counts[i-1]
    end for
    for i := length(array)-1 down to 0 do
        output[counts[array[i]]-1] := array[i]
        counts[array[i]] := counts[array[i]] - 1
    end for
    copy output to array
end procedure
```

Figure 2: Pseudocode for Counting Sort

```
pub fn counting_sort(arr: &mut [u32], maxval: usize) {
    let mut occurences: Vec<usize> = vec![0; maxval + 1];

    for &data in arr.iter() {
        occurences[data as usize] += 1;
    }

    let mut i = 0;
    for (data, &number) in occurences.iter().enumerate() {
        for _ in 0..number {
            arr[i] = data as u32;
            i += 1;
        }
    }
}
```

Figure 3: Code for Rust Implementation of Counting Sort

```cpp
int *Counting_Sort(int Arr[], int N) {
    int max = Max(Arr, N);
    int min = Min(Arr, N);
    int *Sorted_Arr = new int[N];

    int *Count = new int[max - min + 1];

    for (int i = 0; i < N; i++) Count[Arr[i] - min]++;

    for (int i = 1; i < (max - min + 1); i++) Count[i] += Count[i - 1];

    for (int i = N - 1; i >= 0; i--) {
        Sorted_Arr[Count[Arr[i] - min] - 1] = Arr[i];
        Count[Arr[i] - min]--;
    }

    return Sorted_Arr;
}
```

Figure 4: Code for C++ Implementation of Counting Sort

2.4 Bubble Sort

Like previously I will explain the Bubble Sort algorithm before I show the code; Bubble sort works via comparing adjacent elements in an array and swapping them if they are not in the correct order. The algorithm iterates over every index with the same checks until in the end there is a sorted list left. Refer to the pseudocode below to get a better view of how it should look in code. The code for Rust and the C++ implementation can be seen below in the pseudocode

```
procedure bubbleSort(list: array of elements)
    n := length(list)
    do
        swapped := false
        for i := 0 to n-2 do
            if list[i] > list[i+1] then
                swap(list[i], list[i+1])
                swapped := true
            end if
        end for
        n := n - 1
    while swapped is true and n > 1
end procedure
```

Figure 5: Pseudocode for Bubble Sort

```rust
pub fn bubble_sort<T: Ord>(arr: &mut [T]) {
    if arr.is_empty() {
        return;
    }
    let mut sorted = false;
    let mut n = arr.len();
    while !sorted {
        sorted = true;
        for i in 0..n - 1 {
            if arr[i] > arr[i + 1] {
                arr.swap(i, i + 1);
                sorted = false;
            }
        }
        n -= 1;
    }
}
```

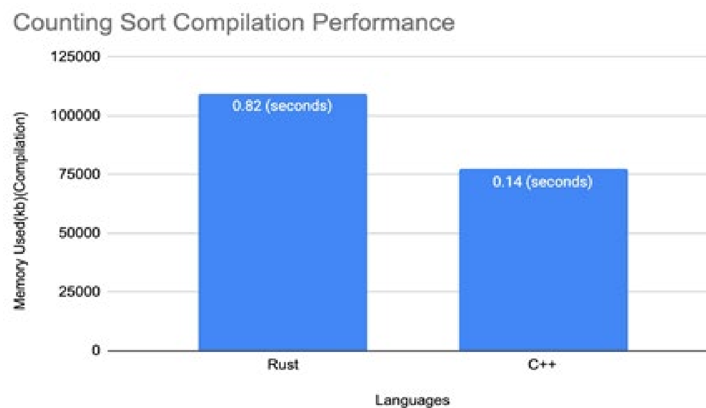
```cpp
void bubbleSortArray(int array[], unsigned int length)  {

    bool changed;

    length -= 1;

    for (int i = 0; i < length; i++) {

        changed = false;

        for (int j = 0; j < length - i; j++) {

            if (array[j] > array[j + 1]) {

                std::swap(array[j], array[j+1]);
                changed = true;
            }
        }

        if (changed == false)
            return;
    }
}
```

2.5 Results/Discussion

Below are the results of the experiment, it is evident that C++ is slightly more efficient than Rust in terms of Compilation Performance and Execution Performance. Though the difference is not as much, it can make a massive difference in performance for programs with larger scales like a search engine or a machine learning algorithm. In every single algorithm, C++ managed to outperform Rust even when the C++ code is not as optimised as the Rust code. This can be regarded as the fact that Rust implements a lot of memory safety features such as bound checking, ownership, mutability etc. These functions can lead to slightly slower execution times compared to C++ with less memory safety, compilation times are also affected by the same factors.

Languages	Memory Used (kb) (Compilation)	Memory Used (kb) (Execution)	Compilation Time (seconds)	Execution Time (seconds)	LOC (Lines of Code)
Rust	109392	1621	0.82s	0.011s	26
C++	77301	1504	0.14s	0.005s	49

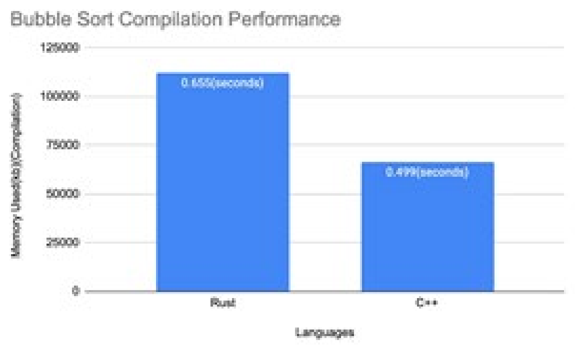
Table 1: Results for Counting Sort Algorithm

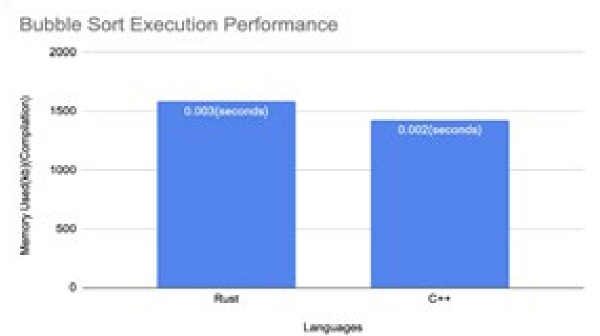


As seen in terms of performance for the Counting Sort algorithm C++ is outperforming Rust by a little bit, this difference expands more though with bigger volumes of inputs example 1,000,000 numbers. Below is the table for the Bubble Sort algorithm. Once again C++ is outperforming Rust in terms of performance but in this case, the difference is actually

pretty small which shocked me as I expected more difference in terms of execution time, but turns out they're pretty close. In terms of code complexity, Rust wins against C++ because Rust has so many built-in functions in the standard library that just makes writing code for these types of programs easier.

Languages	Memory Used (kb) (Compilation)	Memory Used (kb) (Execution)	Compilation Time (seconds)	Execution Time (seconds)	LOC (Lines of Code)
Rust	112384	1584	0.655	0.003	27
C++	66576	1424	0.499	0.002	38





To justify why Rust always uses significantly more memory when compiling, it's because when Rust compiles to an executable it optimises the program for runtime execution which is why it uses more system resources compared to C++ whose compiler doesn't optimise its executable as much as Rust. Other than that, as mentioned before Rust implements many memory safety functions which is one of the main factors into why it takes slightly longer to compile and execute.

3. Summary

To answer the question of which language is better for performance-based applications, both Rust and C++ are fantastic options for performance-based applications, but each language has advantages and disadvantages. I believe that C++ is better suited for low-level performance-based applications like firmware for washing machines, whereas Rust is better suited for high-level sophisticated performance-based applications like search engines. Rust is simpler to develop complicated systems with than C++ because it has a better "crate" environment and more memory safety features. Rust's simplicity of implementation, which may make code straightforward and simple to manage, is one of its major advantages. For example, when I was making the program for Counting Sort, I tried to translate the C++ code directly into Rust, I looked around the system library and realised that there was another way to implement the same algorithm shorter and easier with built-in functions.

However, as the Rust compiler compels programmers to write "safe" code, and more debugging and programme modifications result, Rust's emphasis on safety can occasionally have an adverse effect on development productivity [1]. Moreover, multi-threading is difficult in Rust because of ownership limitations and mutability. As there are fewer memory safety mechanisms in C++, it enables greater flexibility and gives programmers more control over memory management and optimisation. In low-level applications, where programmers must optimise code for certain hardware, this flexibility might be helpful.

Rust's "cargo" tool, which comes pre-packaged with building, testing, and benchmarking, makes development simpler and is another benefit for high-level complicated systems [2]. The greater selection of libraries, tools, and frameworks offered by C++, however, makes it more appropriate for particular applications. In the end, the demands of the project and development team will determine whether Rust or C++ should be used. Developers ought to take into account elements like performance, safety, adaptability, simplicity of use and accessible information and tools before selecting the language for their project. In general, C++ may be a better option for those who value flexibility and control over memory flow, whereas Rust may be a better option for those who place a higher priority on safety and simplicity of implementation, but as Rust improves as a language, there might be a day where Rust can overtake C++ completely and become the head of programming as Rust is mostly criticised for its lack of maturity.

References

1. Bugden, W., & Alahmar, A. (2022). Rust: The programming language for safety and performance. arXiv preprint arXiv:2206.05503.
2. MCFALLS, D. (2017). Concurrent algorithms and data structures in c++ and rust. Technical report, Stanford University.

Copyright: *©2023 Vincent Ng. This is an open-access article distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.*