The Performances of Sorting Algorithms by Using Assembly Language

Victor Deian Balutoiu [1], Liviu Octavian Mafteiu-Scai [1]

[1] *West University of Timisoara, Timisoara, Romania*

***Abstract* – In many programs the data have to be sorted according to one or more criteria. Algorithmics offers us specific algorithms classified according to the order of complexity. But from this point to the realization of a concrete and efficient program for a specific set of data there is a long way. The choice between performance and flexibility of an application is not easy. This paper presents an experimental study that tracks the time performance of assembly language implementations in relation to C and Rust in cases of Bubble Sort, Insertion Sort and Quick Sort algorithms. Conclusions as well as future research directions are also included.**

***Keywords* – bubble sort, insertion sort, quick sort, assembly language, C/C++, rust, runtime.**

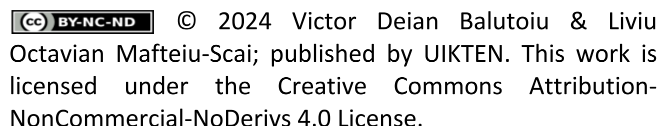
1. Introduction

In many programs the data have to be sorted according to one or more criteria. Algorithmics offers us specific algorithms classified according to their order of complexity.

DOI: 10.18421/SAR72-02
https://doi.org/10.18421/SAR72-02

Corresponding author: Liviu Octavian Mafteiu-Scai,
West University of Timisoara, Timisoara, Romania
Email: liviu.mafteiu@e-uvt.ro

Received: 10 May 2024.
Revised: 14 June 2024.
Accepted: 20 June 2024.
Published: 28 June 2024.



© 2024 Victor Deian Balutoiu & Liviu Octavian Mafteiu-Scai; published by UIKTEN. This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 4.0 License.

The article is published with Open Access at https://www.sarjournal.com/

A selection made on the basis of the complexity value is the first step, but it is not enough to have a performant implementation. Depending on the selected language for implementation, it is possible to have big differences between programs, and this is not all. There are many factors like language configuration, OS type and version, hardware configuration and many others that can have a major impact in program performance. The dilemma of Performance vs. Flexibility of the code also has to be taken into consideration.

This experimental study is based on the question "Can Assembly language get the best performance due to its suitability to the processor?" In the past, Assembly language was more used than it is today. The C language has dominated the programming world for many years now, as it is much easier to use and maintain than Assembly. A few years ago, the Rust programming language was beginning to be taken into consideration as a safer alternative to C. Compared to C and Rust, Assembly programs are too hard to maintain and require a lot of work to implement on big projects. However, to streamline critical (generally short) sequences in high-level languages, Assembly language is still used. Assembly language is more used to write drivers and operating systems, but these are not the subjects of this work. The elements used in our study are presented in the following subsections.

A. Sorting

In computer science the sorting process refers to rearrangement of a given array or list of elements/objects according to one or more comparison criteria. The criteria comparison is used to decide the new order of elements in the data structure. There are only two possibilities: ascending or descending order.

Sorting algorithms are important in computer science because they reduce the complexity of real problems like searching algorithms, database algorithms, divide and conquer methods, data structure algorithms and many more.

Bubble sort is the simplest sorting algorithm that works by repeatedly comparing and swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets because its average and worst-case complexity is quite high. The worst case complexity is $O(n^2)$ and the best-case complexity is O(n), which occurs when the input list is already sorted.

Insertion Sort algorithm works very close to the way of sorting playing cards in hands. The array is split into two parts: a sorted part and an unsorted part. The values from the unsorted part are picked and placed at the correct position in the sorted part. The worst case complexity is $O(n^2)$ and the best case is O(n) when the array is already sorted.

Quick sort algorithm is based on divide- and-conquer method, where an array is divided into subarrays by selecting an element from the array and uses it as pivot element. While dividing the array, the pivot should be positioned in such a way that elements less than the pivot are kept on the left side, and elements greater than the pivot is on the right side of the pivot in the case of an ascending order. The left and right subarrays are also divided using the same approach. This process continues until each subarray contains a single element. At this point, elements are already sorted. Finally, these elements are combined to form a sorted array of initial array. The average-case time complexity is O(n*log(n)) and the worst-case complexity is O(n^2) when the pivot choice consistently results in unbalanced partitions [6].

B. Programming Languages

In this subsection, the basic characteristics of the programming languages used in this study will be briefly mentioned.

Machine code is the lowest level of programming and the only language that a processor can understand. It is the oldest but at the same time the only one currently used in binary computers as a language of processors. Due to its specification for each type of processor and its difficulty to use it to implement algorithms, it is extremely rarely used now.

Assembly language, which originated in 1947 as a programming language for automatic relay computers [1], is a low-level programming language with a very strong correspondence between its instructions (mnemonics) and machine code instructions. The assembly depends on the machine code instructions, so the code is not portable i.e. for each type of processor there is a particular assembly programming language, even than in some cases are very close.

Anyway, the assembly code is translated into machine code using a specific assembler program. We can say that a program written in assembly is a humanized form of its equivalent in machine code. Assembly language requires less memory, has shorter execution time, and it is most suitable for writing interrupt service routines, drivers or memory resident programs as high level programming languages [8]. Most high-level programming languages allow the insertion of code sequences in assembly, with the aim of improving the efficiency of critical processing as well as execution time, and this possibility is still used by programmers in our days.

C language was created by Dennis Ritchie at Bell Labs in the 1970s with the aim of having a straightforward and effective programming language that could be used to create system software, low-level applications or other common programs [2]. Its low-level memory access through the use of pointers, efficient memory management, bits manipulation, preprocessor directives, numerous and extensive libraries, efficient performance and relatively basic syntax are just a few features that make the C language still a preferred choice for many programmers.

Rust language has its origins in a personal project of Graydon Hoare while he was working at Mozilla Research in 2006 [7]. Rust supports functional programming (due to the influence of other functional programming languages like OCaml), but is not a functional programming language. Its syntax is very close to C/C++ syntax and this may be a reason why it was quickly adopted by many programmers. Other reasons why Rust is preferred by programmers are: performance, safety, memory management and built-in support for concurrency.

C. Implementations Comparison

This subsection does not refer the comparison of two or more programming languages. There are several comparison criteria and many published works and books that deal with the subject in detail for two or more languages (out of the several thousand that exist).

In this paper, the comparison of codes written in different programming languages is of interest. But not only in the narrow sense of "equivalence" i.e. two codes are equivalent if and only if for the same input they generate the same output, even if this criterion is the first to be fulfilled in any comparative analysis. Even if our interest in this study is the execution times obtained with the implementation of the same sort algorithm using different programming languages, in our opinion the problem is not quite as simple as it seems at first sight.

We consider that such a comparison has a small degree of relativity i.e. depends on the different ways of implementing data structures, of read or write operations, of the particular way in which algorithm is implemented in each language, in general, the execution times for the two programs could differ by orders of magnitude.

The differences in execution time when implementing the same algorithm in different programming languages can come from the way these languages implement certain data structures, the way they can perform bitwise operations, the way they can pass objects by value, references or pointers, if we have an interpreter or a compiler, etc.

In fact, these differences in execution determine the choice of a certain programming language to solve a specific problem.

2. State of Art

This section reviews some comparative studies related to Assembly, C and Rust programming languages.

The paper [5] provides a comparative study between the C and Rust languages in terms of performance and programming effort in case of HPC (High Performance Computing). The algorithm used was an N-computational-bodies (N-Body) problem. From a purely performance focused outlook, when using the most optimized version of the algorithm in each language, C is faster than Rust when doing calculations in single precision. In the case of double precision, the performances are very close, but still with a slight advantage of the C language. In terms of programming effort, implementing the algorithm takes less lines of code in Rust, while also providing more easily maintainable code. Rust also allowed creating parallel code easily, while C was much harder to set up and use.

In paper [9] are compared the performance of C and Assembly programming languages using for tests two Kalman Filter algorithms. The author concludes that the programs created in Assembly are more efficient.

Seven high level programming languages (C, C++, Java, Perl, Python, Rexx, and Tcl) are compared in work [4] by measuring different metrics such as program length, programming effort, runtime efficiency, memory consumption, and reliability. Taking into consideration the results reported by the author, C and C++ languages are better than others languages in terms of runtime time and required memory. The length of the required code as well as the time required to write the code are not elements that give an advantage to the C and C++ languages.

A comparison between C, C++, C#, Java, Perl and Python programming languages regarding memory usage and runtimes in case of using these to implement specific algorithms in bioinformatics is made in paper [3]. After experiments, the conclusion is that the implementations in C and C++ is fastest and use less memory even if the necessary code is longer than in other languages.

3. Algorithms Implementation

The implemented sorting algorithms in this study were Bubble Sort, Insertion Sort and Quick Sort, discussed in section I.

Each of these two algorithms has three implementations: Rust, C and Assembly. For each implementation in part, we tried to find the fastest implementation (as well as execution time) by modifying the parameters or data structures specific to each programming language, a fact that brings us closer to a more correct comparison of the implementations.

To guarantee that the measurements are as close and precise as possible, the RDTSCP processor instruction was used. This is native in Assembly, and written in inline Assembly for both C and Rust. For the same purpose we checked the output for all three implementations with the same input.

4. Description of the Experiments

The purpose of this experimental study was to see how the use of assembly language in the implementation of sorting algorithms improves the execution times, knowing that these algorithms need significant time for large datasets].

Hardware and software configuration used in our experiments was:

- hardware:

Processor: Intel(R) Celeron(R) CPU N2940 1.83GHz

RAM: 4 GB

- software:

OS: - Debian GNU/Linux 12

Compilers/Assembler:

- gcc (Debian 12.2.0-14) 12.2.0
- Rust 1.75.0 (82e1608df 2023-12-21)
- NASM version 2.16.01

In our experiments, generated datasets were used. The datasets were generated by a program generator.

Finaly, we had three types of data (arrays) to be sorted: reverse, sorted and random. For each type, there are different arrays with different size (number of elements to be sorted): 100, 1000 and 10000. For each size and type, there were three differently generated datasets.

A. Measuring Execution Time

For all experiments, CPU clocks were used to express execution times. To determine the execution time, the current count of the internal CPU clock values was read, at the beginning and at the end of processing for each individual implementation. To do this, the RDTSCP assembly instruction was used.

Each individual implementation, for each dataset, was ran 2000 times, in order to be able to calculate an average value of the execution times as close as possible to reality. On a computer where the processing cost is stable, the runtime for the same program and the same input data, should remain the same for however many runs. Such a computer can be considered a theoretical one these days.

In modern computers it is not correct to consider that a program execution is s deterministic. First, in modern computers, it is normal for runtime to vary, because the computer probably runs many programs (and parts of the OS) at a time and these matters. In these computers, most of all architectural components are designed to optimize the average runtime, based on a statistical analysis. These architectural features are based on statistics and which depend a lot on the initial conditions can lead to different runtimes, such as mostly the contents of the cache memory or branch predictors. Thus, it is normal to have differences in runtime around 20-30%. It is simply that the "startup time" for an application is "indeterminate" as the latency for displaying "printf()" text to screen. Those factors could account for any discrepancy that appears, as can be seen in the subsection V-C.

B. Checking the Output of Implementations

These experiments are only preliminary experiments, to see if all implementation give us for a given dataset- the same outputs.

5. Experimental Results

Before experiment we tested for each algorithm and for each programming language several implementations and we selected the best ones. At the same time, taking into account that the data structures and the operation on them are different in different programming languages, we aimed for the implementations in each programming language to be as close as possible to the implemented algorithm.

A. Runtime Experiments

These experiments were the main goal of our study, i.e. to see the advantage of using assembly language in sorting algorithms such as Bubble Sort, Insertion Sort and Quick Sort.

A graphic representation of the 2000 runs results can be seen in Figure 1 in the case of each sorting algorithm, for random dataset with 10000 size of array. Figure 2 and Figure 3 are also representations of 2000 runs for reverse respectively for sorted data.

As can be seen in Figure 1, Figure 2 and Figure 3 there are small differences between the execution times of the same implementation in the same programming language with the same dataset, the reasons for these differences being already discussed in section I-C. However, converting CPU clocks to seconds makes these differences insignificant for current processors.

A comparison of the results in relation to random, reverse and sorted shows that the execution time of the programs is in accordance with the complexities (worst, best and average) of the implemented algorithms.

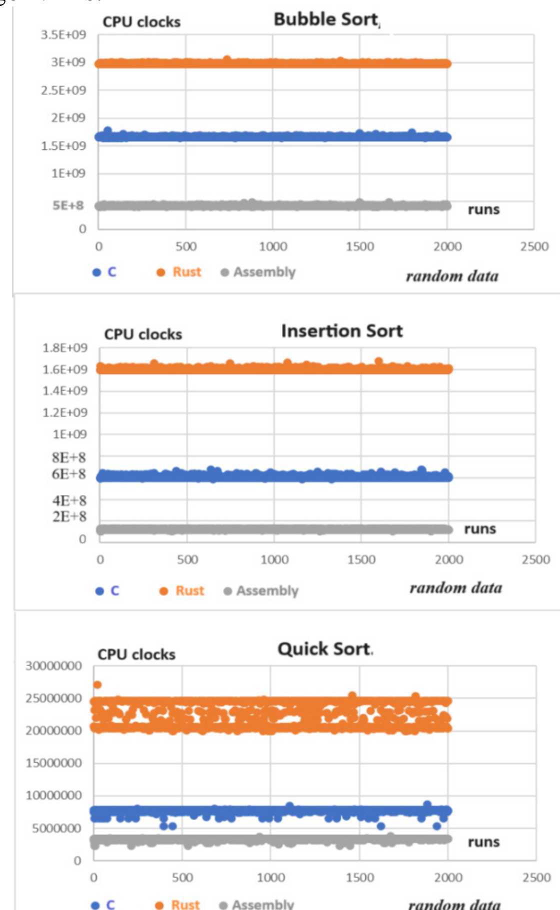


Figure 1. Experiments: random-10000 size of array

Table 1 shows the average experimental results for all experiments we made. The values in the table represent the average values calculated for 2000 runs of each implementation with the same data set, and are expressed in CPU clocks.

Taking into account the 54000 tests performed for the previously mentioned data sets, on sorting data, we can conclude that assembly language implementations are on average about 5 times faster than those in the C programming language and about 15 times faster than those in the Rust programming language (see Global overall in Table 1)

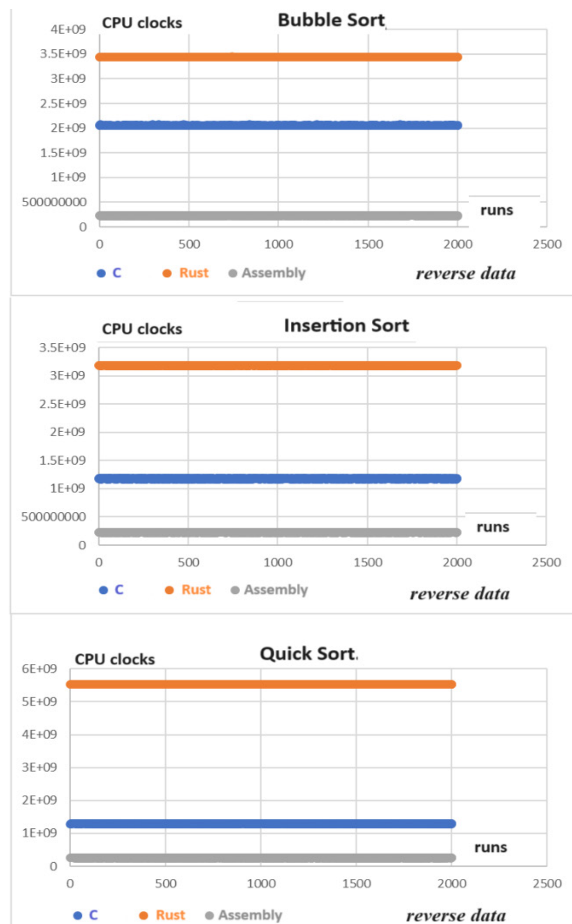


Figure 2. Experiments: reverse-10000 size of array

The difference between C and Rust seems a bit big, but such differences were also mentioned in other works such as in the paper [10] where the possible causes of lower performances of the Rust language are mentioned.

6. Conclusions and Future Work

The main purpose of this study was to see to what extent assembly programming language can improve the performance of sorting algorithms.

The experiments showed a clear advantage of the assembly language in relation to C or Rust in terms of execution times. And even if it is not about orders of magnitude as differences in execution times, it is clear that when we operate in the field of "big data", even double can mean a lot in sorting data. At the same time, it is clear that when flexibility and/or productivity in programming are more important, programming languages such as C or Rust are preferable.

It is also verified that there is a correlation between algorithms' complexity and time complexity of their implementation.

A hybridization of these, i.e. implementations in C/Rust with the critical sequences written in assembly will be part of a future study, this being in our opinion the only viable solution to solve the dilemma Performance-Flexibility. Another extension of this study will be the parallelization of some sorting algorithms on multicore systems using big dataset.

Figure 3. Experiments: sorted-10000 size of array

Table 1. Runtime-Average values for 2000 tests/implementation for all 100, 1000 and 10000 size of arrays

Data type	**Algorithm**	**Programming language**	**Average runtime (CPU clocks)**
random	Bubble		
		C	560791016
		Rust	1006112558
		Assembly	141391368
	Insertion		
		C	205734177
		Rust	540852148
		Assembly	41031319
	Quick		
		C	2803112
		Rust	8124516
		Assembly	1219761
	Overall		
		C	*256442768*
		Rust	*518363074*
		Assembly	*61214149*
reverse	Bubble		
		C	693727668
		Rust	1159107933
		Assembly	77799825
	Insertion		
		C	398757285
		Rust	1073553117
		Assembly	75131253
	Quick		
		C	437983280
		Rust	1863798441
		Assembly	86041831
	Overall		
		C	*510156077*
		Rust	*1365486497*
		Assembly	*79657636*
sorted	Bubble		
		C	292083444
		Rust	703611178
		Assembly	60128945
	Insertion		
		C	165825
		Rust	176577
		Assembly	31969
	Quick		
		C	638998216
		Rust	2855053888
		Assembly	107350895
	Overall		
		C	*310415828*
		Rust	*1186280548*
		Assembly	55837269
Global Overall *(54000 tests)*			
		C	***359004891***
		Rust	***1023376706***
		Assembly	***65569684***

References:

[1]. Booth, A. D., & Britten, K. H. (1947). *Coding for ARC*. IAS.
Retrieved from: https://albert.ias.edu/20.500.12111/7941[accesed: 14 March 2024]

[2]. Kernighan, B. W., & Ritchie, D. M. (1988). *The C programming Language.* Prentice-Hall.Retrieved from:https://www.cs.princeton.edu/~bwk/cbook.html [accesed: 15 March 2024]

[3]. Fourment, M., & Gillings, M. R. (2008). A comparison of common programming languages used in bioinformatics. *BMC bioinformatics*, *9*, 1-9.
Doi: 10.1186/1471-2105-9-82

[4]. Prechelt, L. (2000). An empirical comparison of seven programming languages. *Computer*, *33*(10), 23-29.
Doi: 10.1109/2.876288

[5]. Costanzo, M., Rucci, E., Naiouf, M., & De Giusti, A. (2021). Performance vs programming effort between rust and c on multicore architectures: Case study in n-body. In *2021 XLVII Latin American Computing Conference (CLEI)*, 1-10. IEEE.
Doi: 10.1109/CLEI53233.2021.9640225

[6]. Skiena, S. S. (1998). *The algorithm design manual, 2*. New York: Springer

[7]. Sharma, R., Kaihlavirta, V., & Matzinger, C. (2019). *The Complete Rust Programming Reference Guide: Design, develop, and deploy effective software systems using the advanced constructs of Rust*. Packt Publishing Ltd.

[8]. Kusswurm, D. (2014). *Modern X86 Assembly Language Programming.* Berlin: Springer.

[9]. Yeh, H. G. (1991). Processing performance of two Kalman filter algorithms with a DSP32C by using assembly and C languages. *IEEE transactions on industrial electronics, 38*(4), 298-302.
Doi: 10.1109/41.84024

[10]. Zhang, Y., Zhang, Y., Portokalidis, G., & Xu, J. (2022, October). Towards understanding the runtime performance of rust. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 1-6.
Doi: 10.1145/3551349.3559494