Rust vs C++: Performance in an extended sieve of Eratosthenes, and basic implementation of Chudnovsky algorithm

Zachary Meyner

I. Introduction

Rust is a programming language maintained by Mozilla that is designed to maximize speed—in terms of time spent coding and execution—as well as safety. Rather than a garbage collector or manually managing memory safety of Rust is achieved through its system of ownership—a set of rules checked at compile time to ensure safety [1]. The rules of ownership are:

- Each value has an owner,
- Each value can only have one owner, and
- When the owner goes out of scope the variable is deleted [1].

Rust also employs a system of borrowing for referencing. When a variable needs to be used in a different part of code, say a function,

- The variable is borrowed by that function via a reference and does not own that variable,
- The function cannot return that reference, and
- The reference is deleted at the end of the function call because of ownership [1].

Rust also includes a very robust package manager and database, and build tool called Cargo [2]. Cargo's package manager and crates.io makes community support for any features not included in the standard library to be maintained. Rust's technology has allowed it to rank as the most popular language on Stack Overflow developer survey for seven years [3]. The language has also amassed a large dedicated fanbase, including a discord server with 40,000+ users [4].

Rust's potential in ease of writing in helped raise the popularity of the language and brought about speed tests of the language. It has been run against C in the embedded software and determined to be a viable alternative [5]. The safety benefits in systems programming has been explored [6]. Rust has also been compared to C, Fortran, and Java for speed and effort in parallel architectures, and found to be as fast as the fastest languages with the lowest amount of effort when programming [7] [8].

The Sieve of Eratosthenes finds all prime numbers $p < n$ by starting at the first prime (normally 3) p_0 and multiplying by $k \leq \sqrt{n}$ where $k \in \mathbb{N}$ and marking resultant numbers as not prime. When all k have been multiplied by p_0 the steps are repeated on the next unmarked number p_1. The extended Sieve of Eratosthenes divides the range of the sieve into segments of size $\sqrt{n} + 1$ and mark prime numbers in the respective ranges one segment at a time. Dave Plummer has run a Sieve of Eratosthenes race on many programming languages [9]. Plummer's results show a basic Sieve as opposed to an extended sieve, and rather than testing speed, Plummer tested how many times the sieve can go up to a number in a certain amount of time. The sieve of Eratosthenes gives an excellent test of the languages standard library vector implementations, and ease of writing array code in each langauge.

Approximating π has been one of the longest problems in Mathematics, dating back to Ancient Egypt [10]. Methods for approximating π have become quite computationally efficient since then and eventually lead to a Ramunujan-type approximation by the Chudnovsky brotherswhich was used to break the the world record for computing π [11]. This numerical approximation provides a great way to compare the efficiency of Rust and C++, and external supported packages in each langauges.

II. Methodology

All tests were run 10 times on a local machine with [INSERT MACHINE SPECS HERE (CPU, MEM, OS, COMPILER VERSION)]. All C++ builds will be using the `-Ofast` compiler flag, and all rust builds will be done with the command `cargo build --release` which by default uses the highest rustc speed optimization level [2]. For each program timing will be done via the linux `/usr/bin/time` command to get the time at execution and once execution is done. Each program will be run 10 times and have their execution times averaged together.

A. Extended Sieve of Eratosthenes

Input will determine what number for the sieve to go up to, and will start at 100,000,000. After each programs has its time averaged for the given input, the input will increment by 500,000 and repeat the same process up to 500,000,000. In both langauges the standard library vector and its associated functions were used. For C++ that gives us `std::vector<T>` and in Rust it is `std::vec::Vec<T>` Rust's vector performs bound checking on calls, so it is better to use the langauges built in iterators in many places in the code. So code that would traditionally look like:

```
for (ull i = 4; i <= limit; i += 2) {
    mark[i] = false;
}
```

in C++ would now be typed as.

```
mark.iter_mut()
    .skip(4)
    .step_by(2)
    .for_each(|p| *p = false);
```

This was done wherever possible to maximize the performance in Rust.

B. Chudnovsky Algorithm

Input size will determine how many digits of π the programs will calculate, and will be starting at 50,000. After each programs time is averaged for the given input, the input will increment by 50,000 up to 250,000. Chudnovsky algorithm will quickly exaust the precision possible by a 64-bit floating point number, as well as how large 64-bit integers can be, so big number packages will need to be used. In both langauges the libraries gmp [12] and mpfr [13] will be used for big numbers and big floats repsectively. Since gmp and mpfr are C libraries they will work natively in C++, but for Rust we will need to use a library called "rug" from crates.io that acts as a wrapper to the C function calls [14]. To get the rug library into the rust project we simply add `rug = "1.19.1"` into the Cargo.toml file below the `[dependencies]` line. The use of community made wrapper for gmp leads to many changes in the code between the two, for example in C++ we have

```
// Iteratre the multinomial
// Numerator
mpz_add_ui(kth, kth, 12);
// k^3
mpz_init_set(monNum, kth);
mpz_mul(monNum, monNum, kth);
mpz_mul(monNum, monNum, kth);
// -16k
mpz_submul_ui(monNum, kth, 16);
// Denominator
mpz_init_set_ui(monDen, n + 1);
mpz_mul_ui(monDen, monDen, n + 1);
mpz_mul_ui(monDen, monDen, n + 1);
// The final bit
mpq_init(monAdd);
mpq_set_num(monAdd, monNum);
mpq_set_den(monAdd, monDen);
mpq_canonicalize(monAdd);
mpq_mul(multinomial, multinomial, monAdd);
```

which has the Rust equivalent of

```
nth_val[3] += 12;
let multnom_numer =
    &nth_val[3] *
    Rational::from(&nth_val[3] *
    &nth_val[3]) -
    Rational::from(16 * &nth_val[3]);
let multnom_denom = Rational::from(
    (n + 1) *
    (n + 1) * (n + 1));
nth_val[2] *= multnom_numer /
              multnom_denom;
```

becuase of the rug library wrapper for gmp and mpfr.

III. Results

A. Extended Sieve of Eratosthenes

Input	Rust	C++
100000000	1.515	2.192
150000000	3.515	5.146
200000000	5.952	8.568
250000000	8.900	12.804
300000000	12.309	17.66
350000000	15.942	23.146
400000000	19.957	29.142
450000000	24.324	35.784
500000000	29.136	43.072

TABLE I
TABLE COMPARISON BETWEEN RUST AND C++ IN AN EXTENDED SIEVE OF ERATOSTHENES

As seen in Table I and Figure 2 as input grows larger Rust becomes increasingly faster than C++.

B. Chudnovsky Algorithm

As seen in Figure 1 the speed of Chudnovsky is really close between Rust and C++, and Table II shows that Rust is slighly faster than C++.

Digits	Rust	C++
50000	41.525	55.47
100000	103.368	117.545
150000	265.303	280.494
200000	587.227	604.184
250000	1133.657	1153.602

TABLE II
TABLE COMPARISON BETWEEN RUST AND C++ IN CHUDNOVSKY ALGORITHM

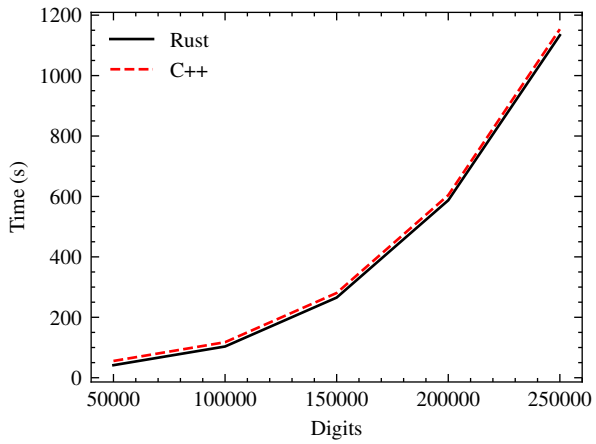

Fig. 1. Graph comparison between Rust and C++ in Chudnovsky Algorithm

IV. Discussion

The results clearly show that Rust is able to calculate primes via the extended sieve significantly faster than C++, and is approximately equal to C++ in calculating digits of π in the Chudnovsky algorithm. We examined a calltree for the extended sieve program and were unable to determine what lead to the significant difference in execution time between the Rust and C++.

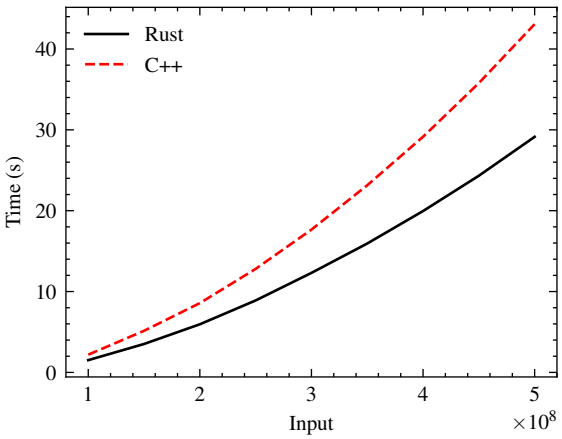

Fig. 2. Graph comparison between Rust and C++ in an Extended Sieve of Eratosthenes

V. Future Works

Future works could look into what leads to the performance difference in extended sieve algorithm between the two langauges. Packages similar to GMP and MPFR could also be developed natively in Rust and tested against the speed of GMP and MPFR. Addionally future works could look into the multithreaded performance of these algorithms in both languages.

VI. Conclusion

In this study we showed that Rust is a potential alternative in terms of speed for numerical calculations-specifically the extended sieve and Chudnovsky Algorithm. Rust performed significantly faster in an extended sieve of Eratosthenes and slighly faster than Chudnovsky algorithm in C++.

References

[1] S. Klabnik and C. Nichols, *The rust programming language*. No Starch Press, 2023. [Online]. Available: https://doc.rust-lang.org/book/title-page.html

[2] "The cargo book." [Online]. Available: https://doc.rust-lang.org/cargo/index.html

[3] "Stack overflow developer survey 2022," Jun 2022. [Online]. Available: https://survey.stackoverflow.co/2022/

[4] "Rust programming language community server discord server." [Online]. Available: https://discord.com/invite/rust-lang-community

[5] N. Borgsmüller, "The rust programming language for embedded software development," Ph.D. dissertation, Technische Hochschule Ingolstadt, 2021.

[6] A. Balasubramanian, M. S. Baranowski, A. Burtsev, A. Panda, Z. Rakamarić, and L. Ryzhyk, "System programming in rust: Beyond safety," in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, ser. HotOS '17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 156–161. [Online]. Available: https://doi.org/10.1145/3102980.3103006

[7] M. Costanzo, E. Rucci, M. Naiouf, and A. D. Giusti, "Performance vs programming effort between rust and c on multicore architectures: Case study in n-body," in *2021 XLVII Latin American Computing Conference (CLEI)*, 2021, pp. 1–10. [Online]. Available: https://doi.org/10.48550/arXiv.2107.11912

[8] H. Heyman and L. Brandefelt, "A comparison of performance & implementation complexity of multithreaded applications in rust, java and c++," B.S. thesis, KTH Royal Institute of Technology, 2020. [Online]. Available: http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-280110

[9] D. Plummer, "Primes," 2023. [Online]. Available: https://github.com/PlummersSoftwareLLC/Primes

[10] D. M. Burton, *Egyptian Geometry*. McGraw Hill, 2011, pp. 55–55.

[11] B. Lynn, "Ramanujan's formula for pi." [Online]. Available: https://crypto.stanford.edu/pbc/notes/pi/ramanujan.html

[12] "The gnu mp bignum library." [Online]. Available: https://gmplib.org/

[13] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, "Mpfr: A multiple-precision binary floating-point library with correct rounding," *ACM Trans. Math. Softw.*, vol. 33, no. 2, pp. 13–es, jun 2007. [Online]. Available: https://doi.org/10.1145/1236463.1236468

[14] T. Spiteri, "rug." [Online]. Available: https://gitlab.com/tspiteri/rug