Universiteit Leiden
The Netherlands

Opleiding Informatica

Rust in Bioinformatics: A Comparative Evaluation of Performance, Memory, and Ecosystem Trade-offs

Dylan Sabiran

Supervisors:
Prof. dr. ir. F.J. Verbeek, Prof. dr. R.V. van Nieuwpoort

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

06/06/2026

Abstract

Modern bioinformatics processes biological data at an ever-increasing scale, demanding software that is both computationally efficient and reliable under real workloads. The ecosystem has historically been divided between accessible high-level languages such as Python and R, and performant low-level languages such as C and C++. More recently, dedicated Rust bioinformatics libraries such as `rust-bio` and `phylo-rs` have emerged, though it remained unclear how Rust compares in practice to the established C++ and R ecosystems. This thesis evaluated the practical viability of Rust by benchmarking four representative tasks through each language's idiomatic harness, using real genomic and phylogenetic data. In most cases, the Rust implementations performed similarly or slightly worse than their counterparts, mostly due to immature library design. The only large exception was tree traversal, where the Rust implementation ran three orders of magnitude slower than C++ on the largest tree size because of a flaw in the upstream crate's parser. Its ecosystem also offers statistical capabilities, though less tightly integrated than R's toolkit. As such, the results were mostly driven by ecosystem maturity instead of inherent language performance. When taking heavily optimised C++ kernels or R's integrated statistical environment into consideration, that same ecosystem maturity currently stands as Rust's primary limitation in bioinformatics. Nevertheless, once its ecosystem matures, Rust has the potential to close the remaining performance and ecosystem gaps while eliminating an entire class of memory- and thread-safety errors.

Contents

1 Introduction

1.1 Background and related work

Modern bioinformatics has evolved into an ecosystem where biological data is processed at an ever-increasing scale. New technologies such as high-throughput sequencing, large public reference databases and phylogenetic resources have made computational analysis a centrepiece of the field. This has created the need for software that is both efficient enough to handle these massive inputs, and reliable enough to be used by researchers who may not have a specific expertise in software engineering [SLF+15, Per20].

When it comes to programming languages, the existing bioinformatics ecosystem has a clear divide between ease of use and low-level control. Languages such as R and Python are most commonly used, as illustrated by tools such as Bioconductor and Biopython [HCG+15, CAC+09]. These tools abstract complex operations behind high-level interfaces, which generally allows for a more interactive and accessible development experience. For instance, the R package `ape` has become a mature and widely used tool in phylogenetics for evolutionary analysis, wrapping complex tree operations in its high-level interface [PS19]. This is in contrast to compiled languages such as C and C++, which are important for performance-critical operations because they allow manual low-level optimisations, such as fine-grained control over memory. CompactTree is a significant example of this, showcasing the ability to traverse large phylogenetic trees with relatively low resource usage [Mos25]. For this reason, many of these high-level interfaces defer their expensive computations to these low-level implementations, functioning as a thin wrapper.

However, this level of performance-oriented control comes with a cost. In C and C++, the task of avoiding memory-safety errors such as buffer overflows, use-after-free bugs and data races is mostly left up to the programmer. Historically, this has been a persistent class of software vulnerabilities that continues to plague even today's software [SPWS13]. In scientific software used by bioinformatics specifically, such errors may be difficult to detect due to the sheer size and complexity of the datasets involved, posing a need for software that is not only fast, but also reliable under real workload conditions.

Rust, a systems language whose development began in 2006 and which reached its first stable release in 2015, was designed to address these issues. Its strict ownership and borrowing model enforces programmers to write code that is inherently memory- and thread-safe, without the use of a garbage collector. It aims to offer the low-level performance of C and C++, while eliminating an entire class of bugs that would otherwise fully depend on programmer fluency [BBB+17, JJKD18]. Over the years, it has seen a striking increase in popularity, which is most evident in its consistent position as the "most admired programming language" in the StackOverflow survey [Sta25]. Furthermore, many major companies and organisations (including, but not limited to Amazon, Cloudflare, Facebook and Google) deploy Rust in their infrastructure [LR23].

Following this widespread adoption, bioinformatics-oriented libraries have started to appear: Köster introduced `rust-bio`, a general-purpose library containing Rust implementations of sequence-analysis algorithms [Kös16]. More recently, `phylo-rs` has provided a Rust library for phylogenetic data structures and analysis [VAME25]. These libraries suggest that Rust can support bioinformatics workloads, but they themselves do not yet answer how Rust currently compares to the mature R and C++ ecosystems that are already widely used in practice.

For clarity, this thesis does not attempt to answer whether Rust is a "theoretical best-fit" for

bioinformatics research. Rather, it evaluates the current state of Rust for selected bioinformatics algorithms compared to C++ and R, from a practical standpoint. As end-users typically interact with bioinformatics tooling at the library level (without direct involvement in algorithmic implementation), runtime performance serves as the main evaluative criterion. However, the broader theoretical implications of adopting Rust in bioinformatics contexts are considered in discussion.
It should also be noted that comparisons against Python are deliberately excluded. It is indeed highly prevalent in bioinformatics, albeit as a high-level orchestration layer. Ultimately, its performance-critical scientific stack (NumPy, SciPy, scikit-bio, pysam, and the C-extension components of Biopython) dispatches compiled C, C++, or Fortran kernels [HMv+20, VGO+20, CAC+09]. Thus, including Python as a separate column would effectively duplicate the C++ comparison. Despite this, R is retained because of its distinctive position: it is currently the de facto environment for downstream statistical analysis in genomics, transcriptomics, and phylogenetics.
Therefore, the main research question of this thesis is:

> *What is the viability of Rust in the field of bioinformatics algorithms, and can practical benefits in runtime performance and/or software reliability be observed?*

This can be divided into three sub-questions:

1. *How does the runtime performance of Rust implementations compare to established C++ and R implementations for representative bioinformatics algorithms?*
2. *How do Rust's compile-time safety guarantees compare to C++ and R in preventing or exposing software errors?*
3. *What practical trade-offs arise when choosing Rust over more established alternatives, especially in terms of programmer experience, interoperability, and ecosystem maturity?*

Throughout, these questions are evaluated at the level of the libraries a researcher would realistically call, rather than idealised implementations written from scratch. The results therefore reflect the maturity and design of each language's available libraries as much as any intrinsic property of the language itself; this distinction is further developed in Section 4.

1.2 Thesis overview

This bachelor thesis is part of the Bioinformatics bachelor at LIACS, and was supervised by Prof. dr. ir. F.J. Verbeek and Prof. dr. R.V. van Nieuwpoort.
This section contains the background on the algorithms under study and the existing literature on Rust bioinformatics tooling; Section 2 describes the benchmark methodology; Section 3 presents the empirical results; Section 4 interprets these findings, addresses practical trade-offs, and acknowledges the limitations of the study; Section 5 answers the main research question and outlines directions for further research; Appendix A documents the full hardware, BIOS, and reproduction setup, and lastly Appendix B provides the complete runtime and memory measurements together with the empirical scaling fits.

2 Methods

2.1 Benchmark design

Four representative algorithms have been selected, for which Rust implementations are compared against established C++ and R alternatives: exact pattern matching using Knuth-Morris-Pratt, local pairwise alignment using Smith-Waterman, global pairwise alignment using Needleman-Wunsch, and post-order traversal of phylogenetic trees encoded in Newick format. They were selected because they collectively represent a range of computational properties that are relevant to bioinformatics: pattern matching stress-tests raw string-processing throughput, the alignment algorithms implement dynamic programming over potentially massive matrices, and phylogenetic tree traversal covers recursive data structure handling. Together, they provide a diverse basis for comparing performance across the three languages.
Each benchmarked algorithm is exposed as a library-style callable function, and benchmark execution is coordinated through a Python-based orchestrator. This orchestrator invokes the Rust, C++, and R benchmark harnesses, passes the prepared benchmark inputs to each implementation, collects all benchmark outputs, and normalises the results into a shared CSV and Parquet schema. This allows all the implementations to be executed through a shared workflow. An overview of this pipeline is shown in Figure 1.

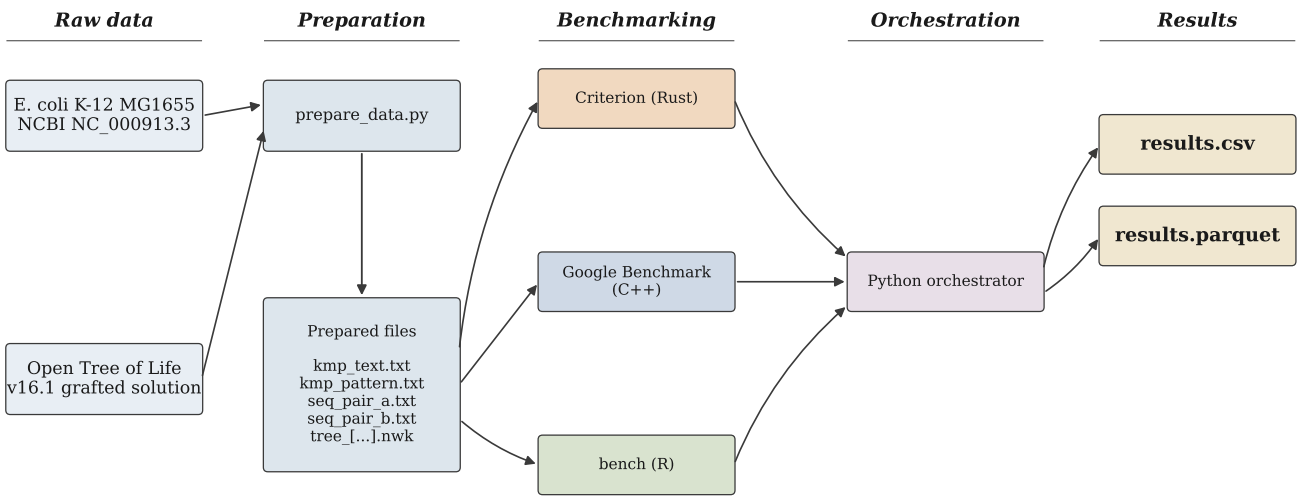

Figure 1: Overview of the benchmark pipeline. The Python orchestrator coordinates execution across the three language-specific harnesses, supplies them with shared prepared inputs, and normalises their outputs into a unified result schema.

2.2 Benchmark implementation

For each benchmarked algorithm, an implementation was selected in Rust and one or more comparison implementations were selected in C++ or R. Rust sequence algorithms use `bio` version 3.0.0, the `rust-bio` crate used for KMP, Smith-Waterman, and Needleman-Wunsch. Rust tree traversal uses `phylo` version 2.0.1, the crate published by the `phylo-rs` project [VAME25]

(the figures use the project name, "phylo-rs"). The C++ KMP benchmark uses a handwritten C++20 KMP implementation over SeqAn3 alphabets. Smith-Waterman in C++ uses SSW, and tree traversal in C++ uses CompactTree. The R Needleman-Wunsch benchmark uses Biostrings sequence containers together with `pwalign::pairwiseAlignment`, and the R tree traversal benchmark uses `ape`. Table 1 contains an overview of this:

Algorithm	Rust	Comparison implementation(s)
KMP exact matching	`rust-bio` `bio` 3.0.0	C++20 KMP using SeqAn3 alphabets 3.4.2
Smith-Waterman	`rust-bio` `bio` 3.0.0	C++ SSW, commit `a66636b`
Needleman-Wunsch	`rust-bio` `bio` 3.0.0	R Biostrings 2.78.0 and pwalign 1.6.0
Tree traversal	`phylo` 2.0.1	C++ CompactTree and R `ape` 5.8_1

Table 1: Implementations used in the benchmark comparison.

For KMP exact matching, the Rust implementation uses `bio::pattern_matching::kmp::KMP`. The C++ implementation is a handwritten KMP implementation templated over SeqAn3 alphabet ranges. For both implementations, the timed function includes construction of the failure table and scanning of the input text.
For Smith-Waterman local alignment, the Rust implementation uses the pairwise alignment API provided by `rust-bio`, while the C++ implementation uses SSW. The Rust benchmark uses match score $+1$, mismatch score -1, gap-open penalty -5, and gap-extend penalty -1. The SSW benchmark uses match score 2, mismatch penalty magnitude 2, gap-open penalty magnitude 3, and gap-extend penalty magnitude 1, which follows the parameter convention of the SSW API.
For Needleman-Wunsch global alignment, the Rust implementation uses `rust-bio`'s pairwise alignment API. The R implementation uses `Biostrings::DNAString` for sequence representation and `pwalign::pairwiseAlignment` for alignment. The Rust benchmark uses match score $+1$, mismatch score -1, gap-open penalty -5, and gap-extend penalty -1. The R benchmark constructs a nucleotide substitution matrix with match score 1 and mismatch score -3, and uses gap-opening penalty 5 and gap-extension penalty 2.
For phylogenetic tree traversal, all implementations read Newick input and perform a post-order traversal. The Rust implementation uses the upstream `phylo = "2.0.1"` crate. The C++ implementation uses CompactTree. The R implementation uses `ape`'s `phylo` representation. Each language reads the same prepared Newick files for the corresponding tree-size benchmarks. For all three languages, the timed region covers both parsing of the Newick file and the subsequent post-order traversal.

2.3 Runtime measurement

For runtime measurement, the idiomatic benchmark framework for each language is used; summarised in Table 2. Figure 2 additionally illustrates how each harness wraps the same library-style callable function and how its output is funnelled into the orchestrator.
Rust benchmarks are handled by Criterion: it performs a warmup phase, collects adaptively sized samples, and reports the median runtime together with a bootstrapped 95 % confidence interval on the median. The benchmark values are then parsed from the generated `estimates.json` file.
C++ benchmarks are handled by Google Benchmark: it executes each case with ten repetitions, and reports aggregate rows including mean, median, standard deviation, and coefficient of variation.

Language	Harness	Reported point	Interval interpretation
Rust	Criterion	median	95 % bootstrap CI on median
C++	Google Benchmark	median	95 % t-interval on mean
R	bench	median	min-to-median range, not a formal CI

Table 2: Runtime summary statistics

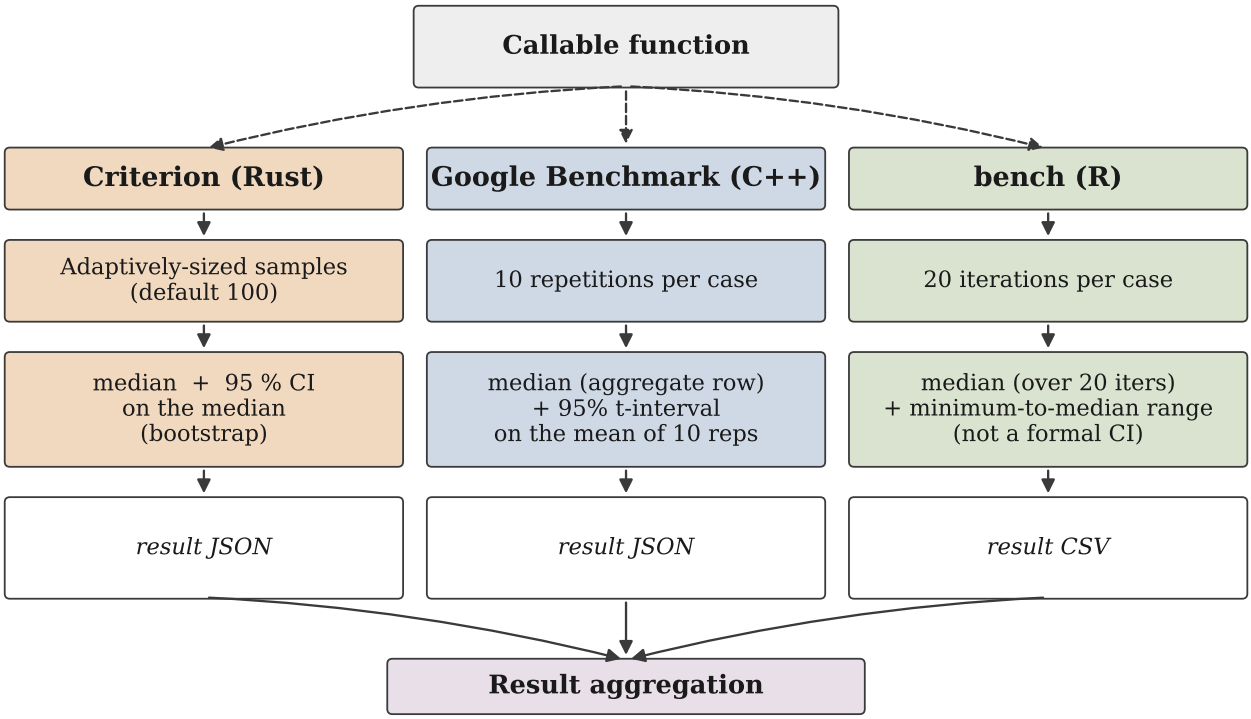

Figure 2: Runtime measurement methodology. Each harness benchmarks the respective callable function under its own idiomatic sampling regime, and reports the respective output file to the Python orchestrator.

The median aggregate as the point estimate is reported and used to compute a 95 % confidence interval on the arithmetic mean computed from the repetitions.
R benchmarks are handled by the bench package: it uses 20 iterations, after which the median elapsed time is reported. The lower interval value is the minimum observed runtime, so the R interval is a min-to-median range instead of a formal confidence interval. The raw R timing values are in seconds, so they are converted to nanoseconds before being written to the shared result file. It should be noted that the benchmark harnesses expose different statistical summaries. This asymmetry is simply a limitation of comparing idiomatic benchmarking frameworks across languages. Therefore, intervals should be treated as indicators of measurement stability; not as perfectly interchangeable statistics.

2.4 Memory measurement

Memory usage is measured only at the largest input size for each algorithm (10^6 bytes for KMP, 5000 bp for Smith-Waterman and Needleman-Wunsch, and 10^5 tips for tree traversal) because smaller inputs are more likely to be dominated by external overhead such as harness setup. Table 3 summarises the memory management strategy for each algorithm.

Language	Metric	Interpretation
Rust	`dhat::HeapStats::max_bytes`	peak concurrent Rust heap allocation
C++	custom `operator new` tracker	peak concurrent C++ allocation requests
R	`bench::mark()$mem_alloc`	cumulative R-managed allocation only

Table 3: Memory measurement mechanisms

Rust memory usage is measured with `dhat`, using `max_bytes` as peak concurrent heap allocation; C++ memory usage is measured through a custom global `operator new` override, which records peak concurrent allocation that was requested; and R memory usage is measured with `bench::mark()$mem_alloc`.
These metrics are also not exactly equivalent. `mem_alloc` reports cumulative memory usage and does not take memory used by the compiled C extensions into account. Therefore, the values from R should be interpreted as a lower bound instead of full peak resident memory usage.

2.5 Input data

Input data is derived from real biological data: the sequence benchmarks use the complete *E. coli* K-12 MG1655 reference genome, accession NC_000913.3, obtained from the NCBI Nucleotide database [NCB26]. The phylogenetic benchmark uses the Open Tree of Life synthetic supertree release OTT v16.1 [Ope26]. Using this raw data, a Python script generates plain-text prepared files, which are reused across all implementations of a given algorithm to ensure consistency. A summary of the benchmark inputs and their sizes can be found in Table 4:
For KMP, the prepared input data consists of the first 10^6 base pairs of NC_000913.3. The pattern is a 20 bp substring starting at position 0.5 Mb of the same region, which guarantees at least one true match inside the searched text. The benchmark varies the text length over four sizes: 10^3, 10^4, 10^5, and 10^6 bytes. The pattern length remains fixed at 20 bp.

Algorithm	Input sizes	Unit
KMP	1k, 10k, 100k, 1M	text length in bytes
Smith-Waterman	100, 500, 1k, 5k	sequence length in bp
Needleman-Wunsch	100, 500, 1k, 5k	sequence length in bp
Tree traversal	100, 1k, 10k, 100k	number of tips

Table 4: Benchmark input sizes.

For Smith-Waterman and Needleman-Wunsch, the two input sequences are non-overlapping 5 kbp windows from NC_000913.3, starting at positions 1.5 Mb and 3 Mb. Each benchmark slices both windows to the required length such that the aligned sequences are always equal in length. The tested sequence lengths are 10^2, $0.5 \cdot 10^3$, 10^3, and $0.5 \cdot 10^4$ bp. These sizes are intentionally smaller than the KMP text sizes, because dynamic-programming alignment has $O(n^2)$ time and memory requirements. A hypothetical $10^6 \times 10^6$ alignment matrix would require 10^{12} cell evaluations and several terabytes of memory, which is outside the scope considering the available hardware to the average consumer.
For tree traversal, the input files are Newick subtrees pruned from the Open Tree of Life v16.1 grafted solution. The prepared tree sizes are 10^2, 10^3, 10^4, and 10^5 tips. Pruning is performed by deterministic random tip selection with seed 42, followed by a cleanup of single-child internal nodes.

2.6 Execution environment

Benchmarks were executed inside a Debian KVM guest running on an Unraid host using an AMD Ryzen 7 5800X CPU, with 2 vCPUs pinned to isolated physical host cores. The benchmark process itself was pinned to one vCPU, while all non-relevant tasks were diverted to the other. Simultaneous multithreading, boost behaviour, C-states, CPPC, preferred cores, Cool&Quiet, and XMP were all disabled in BIOS. A full list of all settings and scripts is available in Appendix A.
All Rust crates are built in `--release` mode with locked dependencies. C++ benchmarks are built with `-DCMAKE_BUILD_TYPE=Release`, corresponding to optimised builds with `-O3 -DNDEBUG` on GCC. R scripts are interpreted and executed within a pinned environment.

3 Results

This section reports the benchmark results obtained on the execution environment described in Section 2.

All runtimes are wall-clock medians reported by the respective benchmark harnesses (see Section 2); the accompanying interval is the harness's native 95 % confidence interval where available, and otherwise the min-to-median range. As for the scaling behaviour summary, an empirical scaling exponent k was estimated for each (algorithm, language) series by ordinary least squares on the four $(\log n, \log t)$ pairs, with a 95 % confidence interval obtained by bootstrap resampling of the four data points.

Figure 3 shows an overview of wall time against input size on log-log axes for all four algorithms. KMP and Smith-Waterman show the expected linear and quadratic scaling, with a constant C++/Rust gap. Needleman-Wunsch shows an initial gap between the Rust and R curves, which converge as n increases. The major outlier is tree traversal: the Rust line bends upward while the C++ and R lines remain straight, which indicates a super-linear scaling for this specific implementation.

The following four subsections detail the algorithm runtimes across the four input sizes. Subsequently, the scaling behaviour and statistical reliability are evaluated, concluding with a cross-algorithm summary that incorporates peak memory usage.

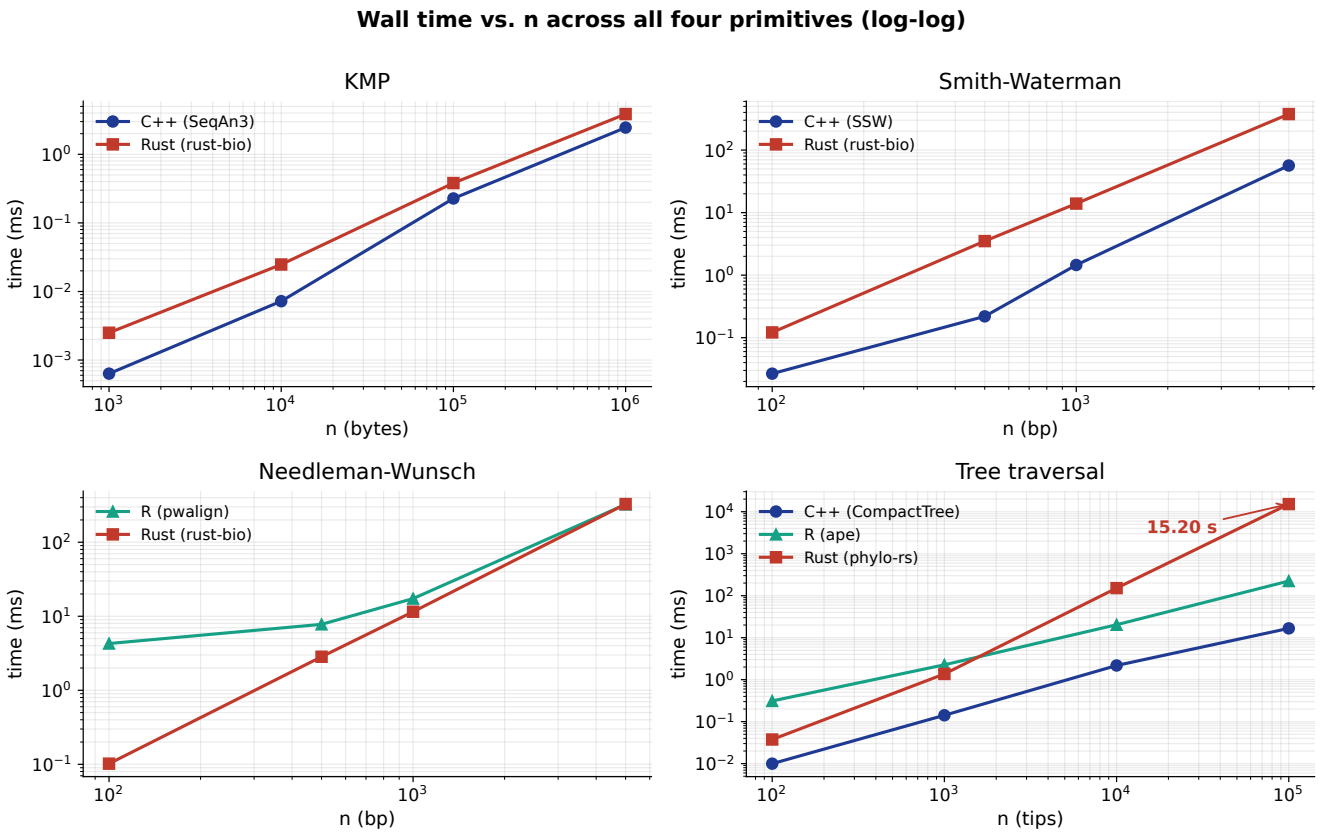

Figure 3: Wall time as a function of input size for all four benchmarked algorithms. Both axes are logarithmic, and markers indicate the median of the harness's iteration set.

3.1 Knuth-Morris-Pratt exact matching

Both KMP implementations scale similarly to the $\mathcal{O}(n+m)$ complexity of the algorithm, but the C++ implementation is consistently faster across all text lengths (Table 5). The C++/Rust ratio is not constant either: it shrinks from 3.95 at $n = 10^3$ bytes to 1.57 at $n = 10^6$ bytes (Figure 4). The empirical scaling exponents ($k_{\text{C++}} = 1.23$ and $k_{\text{Rust}} = 1.08$) are close to the theoretical $k = 1$, with the C++ value slightly inflated, likely caused by overhead from the benchmarking harness. Nonetheless, the bootstrap CIs (Section 3.5) confirm that neither deviates from linearity.

n (bytes)	C++ median	Rust median	C++/Rust
10^3	634 ns	2.50 µs	3.95×
10^4	7.22 µs	24.73 µs	3.43×
10^5	227.22 µs	381.15 µs	1.68×
10^6	2.45 ms	3.86 ms	1.57×

Table 5: KMP wall-clock medians and the respective speedup ratio.

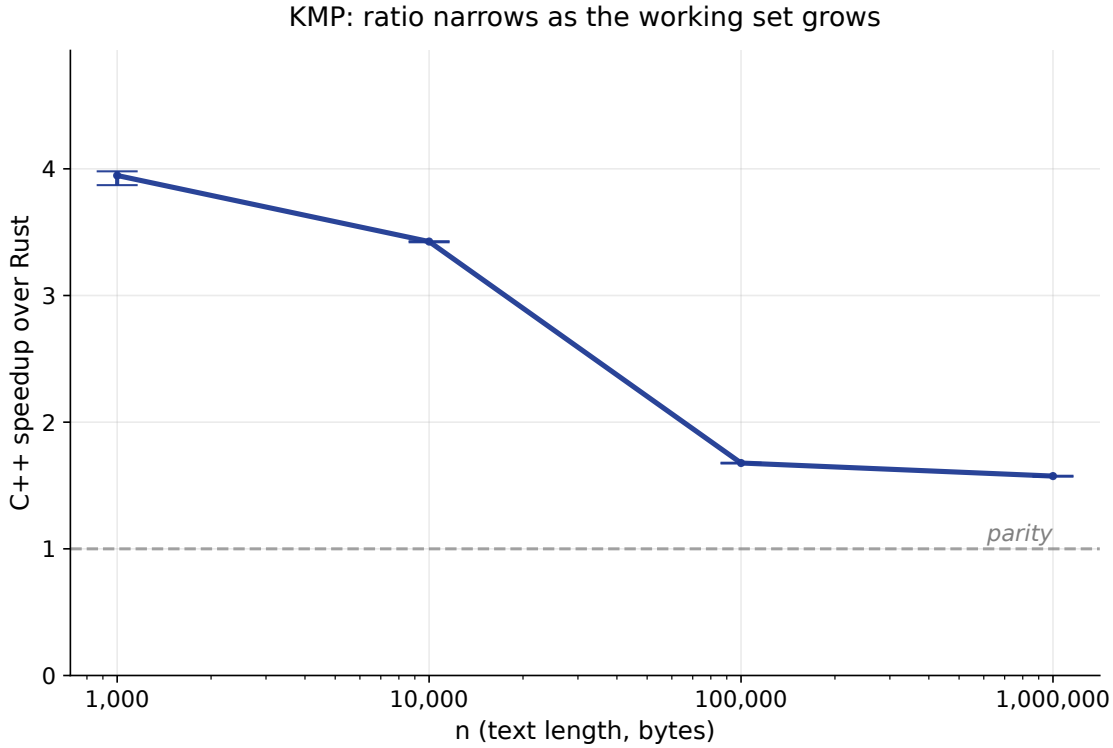

Figure 4: C++ speedup over Rust for KMP, plotted against text length on a logarithmic axis. Note that after the first marker, the error bars (derived from the 95 % intervals) are smaller than the marker size.

3.2 Smith-Waterman local alignment

For Smith-Waterman, the C++ implementation is faster than the Rust implementation at every sequence length. The gap peaks at ∼16× around $n = 500$ bp and narrows back to ∼6.7× at the largest size tested (Table 6), which is characteristic of SIMD-vs-scalar comparisons. The empirical scaling exponents ($k_{\text{C++}} = 1.98$, $k_{\text{Rust}} = 2.05$) are consistent with the theoretical $\mathcal{O}(n^2)$ complexity of the algorithm.

n (bp)	C++ median	Rust median	C++/Rust
100	26.62 µs	121.47 µs	4.56×
500	219.39 µs	3.50 ms	15.97×
1000	1.45 ms	13.94 ms	9.59×
5000	56.53 ms	376.19 ms	6.65×

Table 6: Smith-Waterman wall-clock medians together with the C++/Rust speedup ratio. The 95 % intervals are narrower than 1 % of the median at every entry and are thus omitted. Full intervals listed in Appendix B.

Shifting attention to Figure 5, the peak-memory measurement $n = 5000$ shows a much clearer result than the runtime measurement. The C++ implementation allocates 69.7 KB of peak heap, while the Rust implementation allocates 50.3 MB: a ratio of ∼722×.

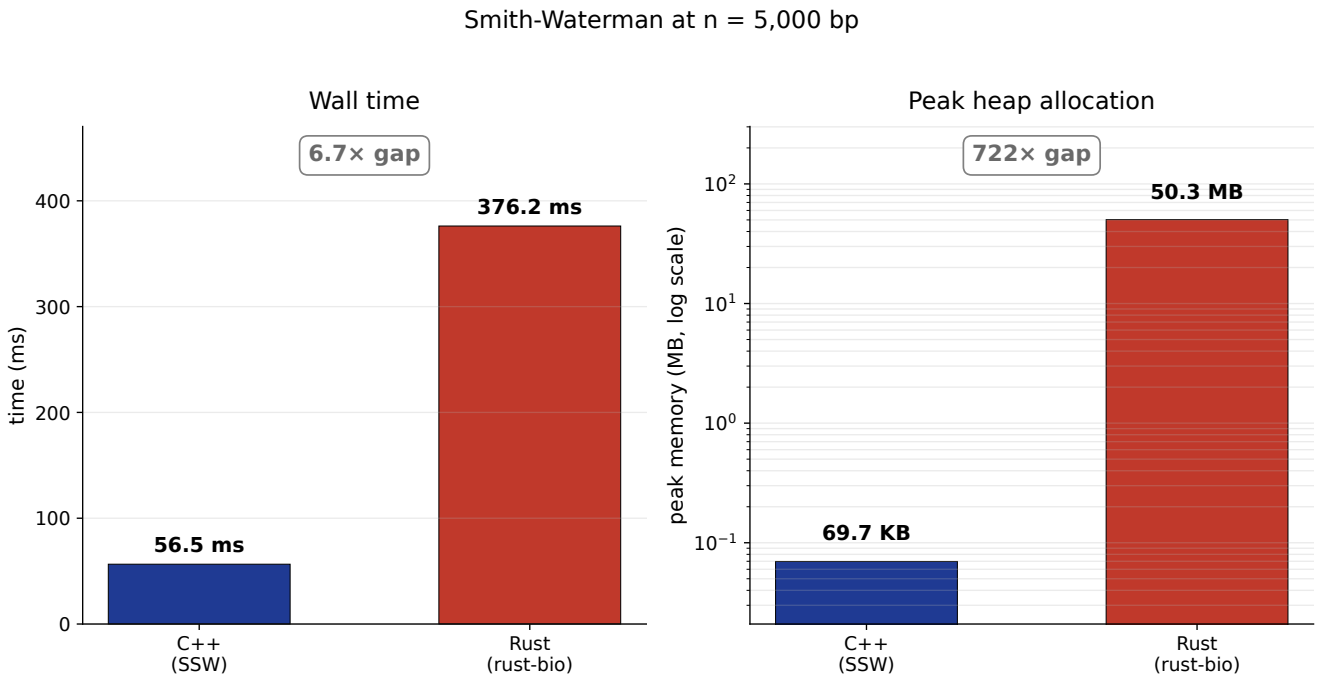

Figure 5: Smith-Waterman wall time and peak heap allocation at the largest sequence length ($n = 5000$ bp). Note that the right-hand side features a logarithmic vertical axis.

3.3 Needleman-Wunsch global alignment

The Needleman-Wunsch results, summarised in Table 7, are the most sensitive to input size of the four algorithms. The initial Rust/R speedup ratio of 41.96× at $n = 100$ bp collapses to 0.99× at $n = 5000$ bp.

n (bp)	Rust median	R median	Rust speedup
100	102.08 µs	4.28 ms	41.96×
500	2.84 ms	7.78 ms	2.74×
1000	11.52 ms	17.33 ms	1.50×
5000	327.48 ms	324.04 ms	0.99×

Table 7: Needleman-Wunsch wall-clock medians and Rust/R speedup ratio.

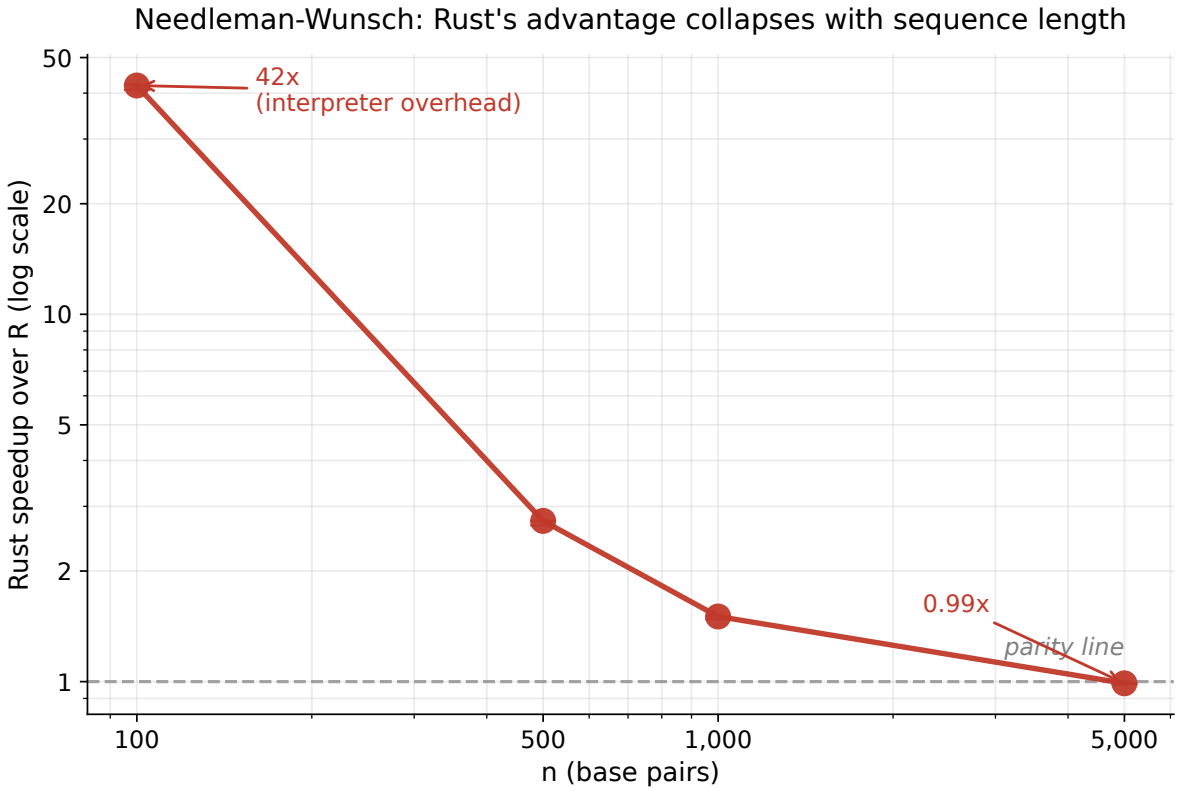

Figure 6: Rust speedup over R for Needleman-Wunsch alignment as a function of sequence length on logarithmic axes. Note that the error bars derived from the per-language intervals are smaller than the marker size at every point.

3.4 Phylogenetic tree traversal

Tree traversal is the only one of the four comparisons where the Rust implementation does not match its theoretical complexity. The measured times, summarised in Table 8, show that C++

and R both scale near-linearly with the number of tree tips, while the Rust implementation scales well beyond this: from 37.6 µs at $n = 100$ tips, to 1.37 ms at $n = 1000$, to 151.56 ms at $n = 10^4$, to 15.20 s at $n = 10^5$ (Figure 7). This run alone lasted over 20 minutes on the benchmarking system, longer than all other benchmarking runs combined.

n (tips)	C++ median	R median	Rust median	Rust/C++	Rust/R
100	9.98 µs	311.13 µs	37.56 µs	3.76× slower	8.28× faster
1000	141.88 µs	2.26 ms	1.37 ms	9.64× slower	1.65× faster
10 000	2.18 ms	20.34 ms	151.56 ms	69.67× slower	7.45× slower
100 000	16.67 ms	223.56 ms	15.20 s	912× slower	68.01× slower

Table 8: Tree traversal wall-clock medians across all three languages.

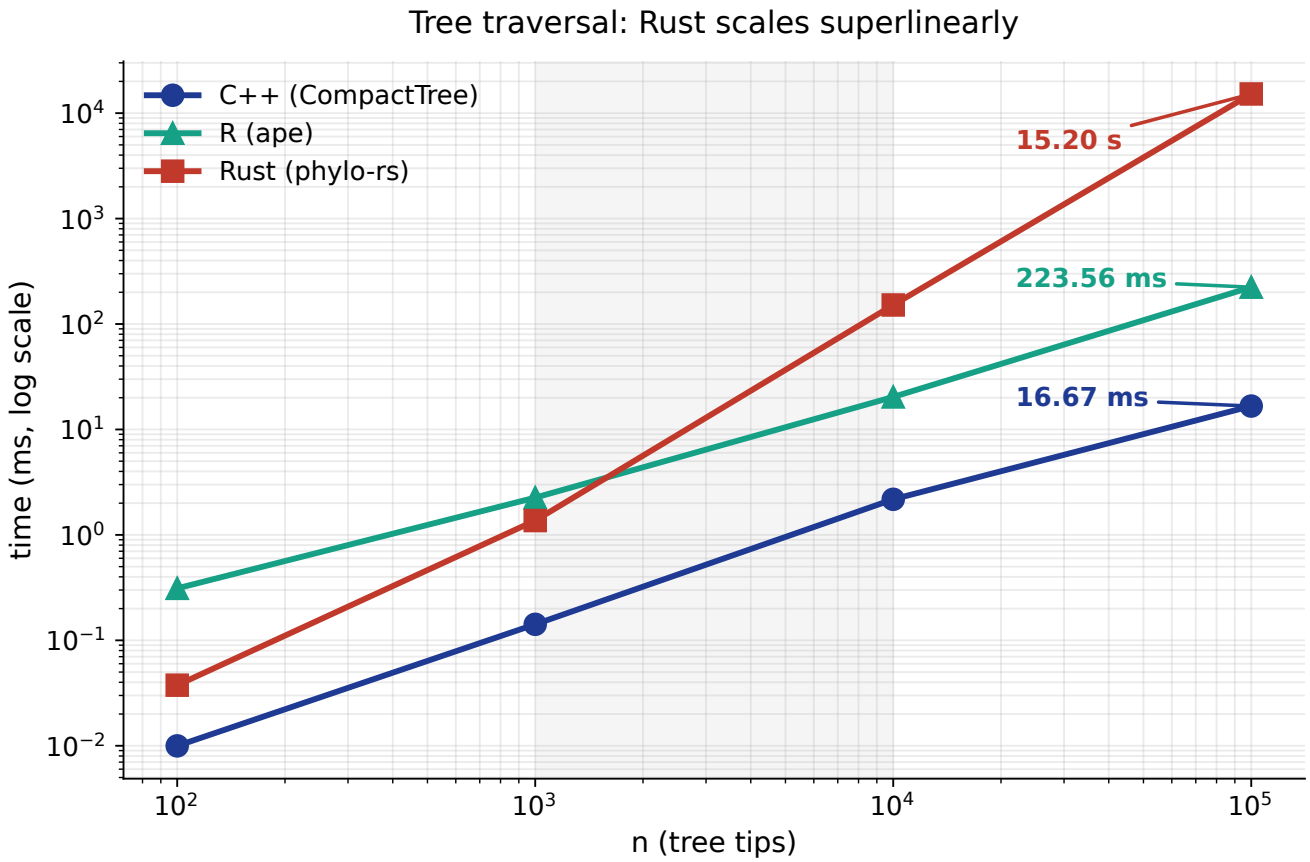

Figure 7: Tree traversal wall time against tree size for all three languages, both axes logarithmic.

3.5 Scaling exponents and statistical reliability

Figure 8 consolidates the (algorithm, language) scaling exponents reported throughout the previous sections. Each point is the slope of an ordinary least-squares fit of $\log t$ against $\log n$ over the

four input sizes, with a 95 % confidence interval obtained by 5000 bootstrap resamples of the four data points. Two reference lines mark the expected exponents for $\mathcal{O}(n)$ and $\mathcal{O}(n^2)$ algorithms. Seven of the nine series sit in their expected complexity class. Two deviate by a full class. The first is the Rust tree traversal point, whose CI lies between the two reference lines—a super-linear deviation from the expected linear value. The second is the R Needleman-Wunsch point, whose fitted exponent of $k = 1.11$ falls short of its expected quadratic value; this reflects a crossover over the tested range from interface-overhead-bound runtime at small inputs to compiled-loop-bound runtime at large inputs (Section 4). In contrast to this, the within-harness measurement variability is very low across all tests: 30 of the 36 (algorithm, language, n) combinations have a 95 % interval whose half-width is below 1 % of the median, with no combination exceeding 2.5 %.

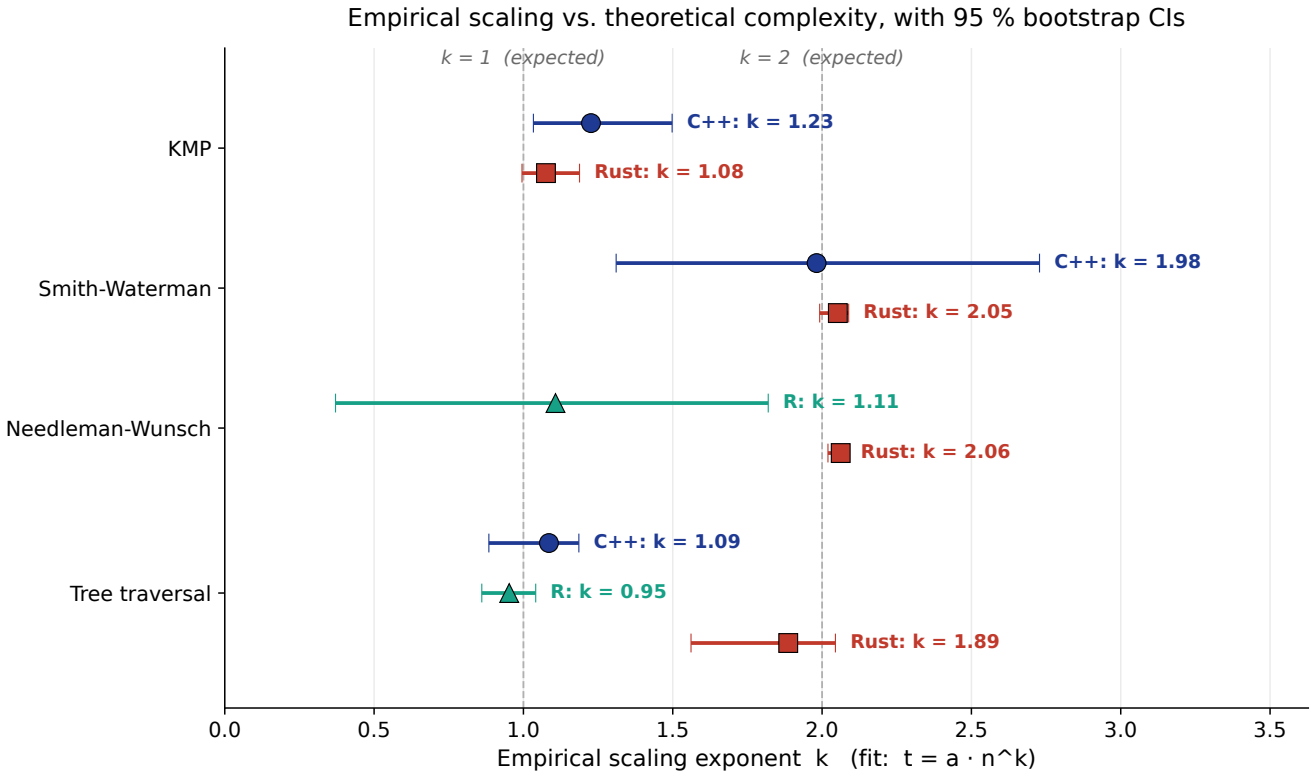

Figure 8: Empirical scaling exponents per (algorithm, language) series with 95 % bootstrap confidence intervals. The two dashed reference lines mark the main relevant theoretical complexities $k = 1$ (linear) and $k = 2$ (quadratic).

3.6 Peak memory usage

Peak memory at the largest input size is represented in Figure 9, on a logarithmic vertical axis so that the roughly five orders of magnitude range from KMP (∼200 B) to Needleman-Wunsch R (∼75 MB) fits on the same panel. Note that R bars are shown hatched: as mentioned in Section 2, the `bench::mark()` memory metric reports only cumulative R-managed allocation and excludes any memory allocated by the C kernels invoked from R. Therefore, the R bars should be read as a lower bound.

Overall, the memory comparisons are not aligned with the runtime comparisons: memory usage with KMP is virtually negligible in both tested languages ($< 200\,\mathrm{B}$); the Smith-Waterman memory gap is almost three orders of magnitude despite the runtime gap being a single order of magnitude; for Needleman-Wunsch and tree traversal, the Rust footprint is within a small factor of the comparison implementation.

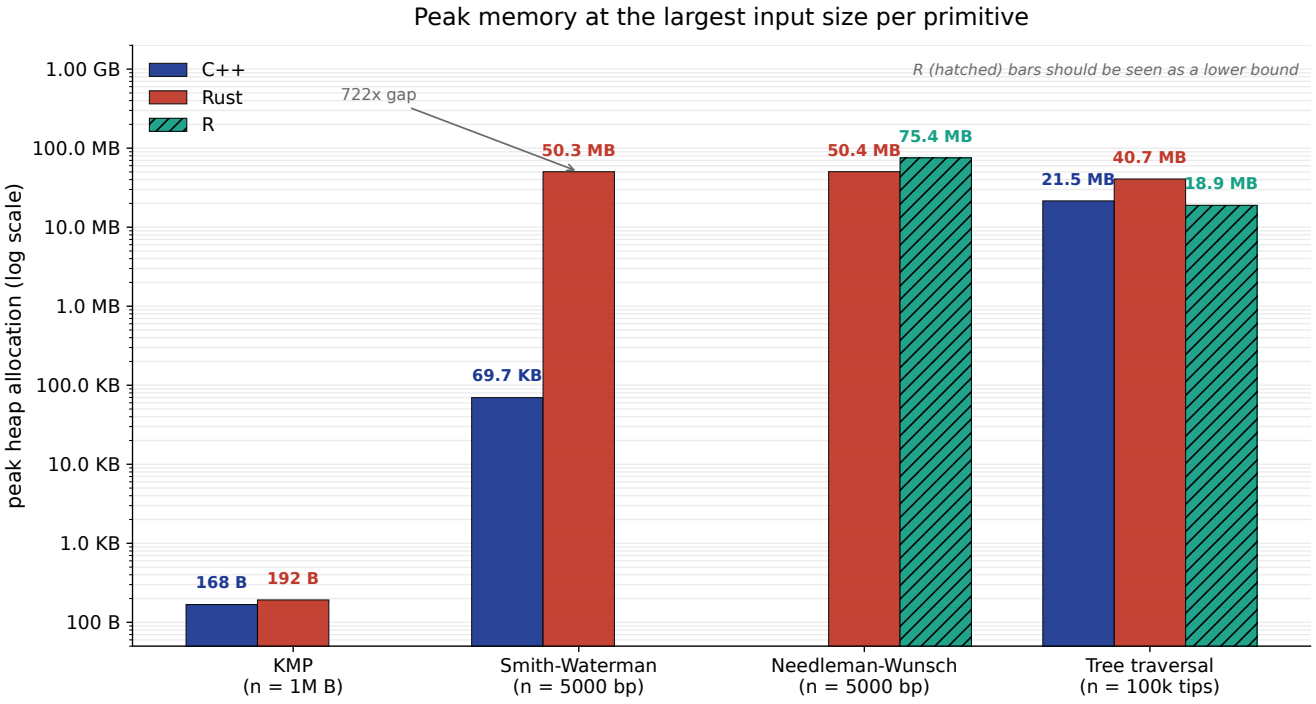

Figure 9: Peak heap allocation at the largest input size measured for each algorithm, grouped by algorithm and coloured by language. The vertical axis is logarithmic. R bars are hatched to flag the measurement asymmetry.

3.7 Cross-algorithm summary

Taken together, the four benchmarks reveal a complex collection of patterns: for algorithms with mature reference implementations in both Rust and C++ (KMP and Smith-Waterman), the C++ baseline retains a constant-factor runtime advantage at every input size, and in the Smith-Waterman case is accompanied by a three-orders-of-magnitude gap in peak memory. For Needleman-Wunsch, the Rust advantage is large at small input sizes but collapses to practical parity at the largest size tested. Tree traversal stands as the major exception: the Rust implementation does not exhibit its expected linear complexity, and at the largest input size it is roughly three orders of magnitude slower than the C++ baseline despite operating on the same Newick input. These three regimes form the empirical basis for the trade-offs analysed in Section 4.

4 Discussion

4.1 Interpreting the performance results as an ecosystem comparison

The benchmark compares the library functions a researcher would realistically call over idealised implementations written from scratch. This means the results reflect ecosystem maturity as much as raw language performance, though for a good reason: a researcher likely rarely picks a language in isolation; they evaluate the available libraries for their use case and conventions, and most will likely favour something that works as-is over a stronger alternative that demands deep technical expertise. Therefore, this benchmark captures what each ecosystem currently offers its users, which is the comparison this thesis cares about most.
This reframes the speed gaps reported in the benchmark results, which will be further explored in the next subsections: raw throughput was never the language's central claim; performing in the same range as mature, heavily optimised C++ libraries, while adding compile-time safety, is the bar that matters here.

4.2 Knuth-Morris-Pratt: sequence representation

The KMP benchmark is the closest of the four experiments to a direct comparison, because both implementations use the same asymptotic algorithm and both include construction of the failure table in the timed call. The C++ implementation is faster at all input sizes, with the speedup decreasing from approximately 4× at 10^3 bytes to 1.57× at 10^6 bytes (Table 5, Figure 4). Both implementations allocate almost no memory at the largest input size.
Both implementations utilise a one-byte-per-symbol representation: `rust-bio` scans a byte slice (`&[u8]`) directly, while the C++ harness stores the text as a `std::vector<seqan3::dna4>`, in which each `dna4` symbol also occupies a single byte. Therefore, the two working sets have the same size, which the peak-memory measurement confirms: both allocate only the failure table and hit list (around 200 B), as the input text is inserted outside the measured region. Thus, possible contributors to the performance gap are the bounds-checked slice indexing and the iterator-and-`collect` layer of `rust-bio`'s `find_all`, against the hand-rolled C++ scan over unchecked `operator[]` accesses. The narrowing of the performance gap as the text grows is consistent with this fixed per-call overhead becoming less relevant for longer scans. Therefore, the KMP result should be interpreted as a comparison between `rust-bio`'s KMP API and a hand-written SeqAn3-typed scan, with no evidence that the Rust language is inherently worse at string search.

4.3 Smith-Waterman: SIMD versus scalar implementation

The Smith-Waterman result is another example of the difference in library maturity. The C++ implementation uses SSW, a SIMD-oriented striped Smith-Waterman implementation, while the Rust implementation uses `rust-bio`'s scalar pairwise aligner [ZLGM13, Kös16]. The C++ implementation is between 4.56× and 15.97× faster across the tested input sizes, with the largest relative gap at $n = 500$ bp and a smaller, though still substantial, gap at $n = 5000$ bp (Table 6). This pattern is consistent with the expected performance benefits of an SIMD implementation, whose advantage is largest when vector lanes are well utilised and smaller when the problem becomes increasingly memory-bound.

The gap in memory utilisation is even more substantial than the wall time gap: at $n = 5000$, the Rust implementation allocates approximately 50.3 MB, while SSW allocates only 69.7 KB (Figure 5). This follows directly from the algorithmic data structures used by the libraries: `rust-bio` materialises an $\mathcal{O}(n^2)$ dynamic-programming matrix, while SSW uses a striped representation that keeps only a much smaller amount of state live during the sweep. The Rust figure closely matches the analytic expectation for a 5000×5000 matrix of two-byte scoring cells, with a small remainder attributable to other external overhead.

4.4 Needleman-Wunsch: interface overhead and compiled kernels

Needleman-Wunsch gives the clearest counterexample to the assumption that R is always slow in bioinformatics. At $n = 100$ bp, Rust is approximately 42× faster than R. However, this advantage collapses as input size increases. At $n = 5000$ bp, the two implementations are essentially tied (Table 7, Figure 6). The interpretation of this is that the small-input result mostly measures the overhead of R's interface layer, which seemingly becomes negligible in a large-input scenario.

This behaviour follows from the structure of the R implementation: the benchmark does not run a nested dynamic-programming loop in the R interpreter. Instead, `pwalign::pairwiseAlignment` fully delegates the expensive alignment computation loop to compiled code. At $n = 100$, the 100×100 matrix is so small that overhead such as dispatch, S4 object handling, and data conversion play a sizeable part in runtime. At $n = 5000$ however, the 5000×5000 dynamic-programming matrix dominates the runtime instead, and both Rust and R spend most of their time in the respective compiled loops.

The memory comparison for Needleman-Wunsch is less significant, since the R metric is a lower bound on true peak memory usage. As stated previously, this is because `bench::mark()$mem_alloc` only captures R-managed allocation, which does not necessarily include all memory allocated by compiled C code. Therefore, the R memory number should not be interpreted as evidence that R is more memory-efficient than Rust in this benchmark.

The practical takeaway is that R can be entirely adequate for heavy computations when the expensive part is implemented below the R layer. Conversely, Rust's advantage is largest when it avoids high-level dispatch overhead on small or repeated calls.

4.5 Tree traversal: the upstream parser and ecosystem maturity

Tree traversal stands as the most striking result in the benchmark. At small sizes, Rust performs reasonably: at $n = 100$ tips it is slower than C++ but faster than R. Between $n = 1000$ and $n = 10000$, however, Rust becomes slower than R. At $n = 100000$, the Rust implementation takes 15.20 seconds, compared with 16.67 ms for C++ and 223.56 ms for R (Table 8, Figure 7). This is the only benchmark in which the Rust implementation significantly deviates from its expected complexity class.

Examining the underlying Rust implementation, the upstream `phylo` crate's tree operation includes Newick parsing as well as traversal, the former of which is the main bottleneck. In `phylo` 2.0.1, node insertion calls a helper that scans the internal node vector to find reusable empty slots. This is a reasonable design for mutation-heavy tree workloads because it allows deleted node identifiers to be reused. In this benchmark, however, the tree is only appended during parsing and then traversed.

The scan therefore repeats work whose result is always the same, producing an $\mathcal{O}(n^2)$ complexity class.
A local patch that replaces the scan with append-only allocation would likely close the gap for this benchmark. However, applying such a patch would disregard the most important ecosystem-maturity signal in this thesis.
Collectively, the results support the notion that young ecosystems contain more risks than mature ecosystems. C++ has several mature tree and sequence libraries, and R has decades of phylogenetics infrastructure around packages such as `ape` [PS19, Mos25]. More recently, credible Rust bioinformatics libraries such as `rust-bio` and `phylo-rs` have appeared, but the ecosystem remains in its infancy [Kös16, VAME25]. If the main Rust implementation for a workload is discovered to contain a hot path that is poorly matched to that use case, a researcher has fewer mature alternatives to switch to without migrating their pipeline to a different language. Therefore, this result is the strongest indication that Rust's bioinformatics ecosystem remains in an exploratory phase.

4.6 Static safety guarantees and compiler-level differences

The performance results alone do not capture the full picture; the remaining research questions aim to address this gap. Rust's most distinctive advantage in software development has never been about performance, but about its static safety guarantees over existing low-level languages that move entire classes of errors from runtime behaviour into compile-time rejections. Rust's ownership and borrowing rules are one of the most fundamental examples of this, designed to provide memory safety without a garbage collector. Formal work on Rust has examined how guarantees such as this hold even in the presence of unsafe abstractions [BBB+17, JJKD18, The26a]. This directly addresses the reliability question raised in the second research question, since scientific software is often run in environments where memory errors may be difficult to diagnose [SPWS13].
R occupies the least significant position out of the three, since it is an interpreted language by default. In other words, it does not expose manual memory management to the user in the same way as the low-level languages. However, much of R's demonstrated performance in this thesis comes from pre-compiled extensions (including the Bioconductor packages), and relies on these extensions to perform reliably. Therefore, many of R's bioinformatics workloads theoretically inherit the safety risks of the underlying lower-level implementations, largely dominated by C/C++. In practice, this inherited risk is lowered by the maturity of the existing kernels.
A closer analysis of C++ demonstrates that out-of-bounds accesses, use-after-frees, invalid pointer dereferences, and data races can be undefined behaviour, and programmers are generally not required to diagnose such behaviour before execution [cpp26]. In Rust, many of these issues are tackled by its fundamental safety guarantees: a use-after-free is rejected by the borrow checker, data races are constrained through ownership and thread-safety rules, and safe slice indexing prevents the reading of arbitrary memory.

```cpp
std::vector<int> nodes{0, 1, 2};
int& first = nodes[0];   // reference into the buffer
nodes.push_back(3);      // may reallocate, invalidating ‘first‘
std::cout << first;      // undefined behaviour
```

```rust
fn main() {
    let mut nodes = vec![0, 1, 2];
    let first = &nodes[0];   // borrows ‘nodes‘
    nodes.push(3);           // compile error: cannot mutate while borrowed
    println!("{first}");
}
```

Figure 10: The same aliasing implementation in both C++ (top) and Rust (bottom). The Rust version is rejected at compile time, but the C++ version compiles without error; if `push_back` triggers a reallocation, then `first` is left dangling and the subsequent read is undefined behaviour.

Figure 10 illustrates this contrast on an identical aliasing pattern implemented in both languages. The C++ fragment compiles without issue, and will invoke undefined behaviour if the `push_back` call reallocates the underlying buffer. In contrast, the equivalent Rust implementation is rejected at compile time: the borrow checker detects that `push` would require a mutable borrow of `nodes` while an immutable borrow into the same buffer is still live, and rejects the program with error `E0502`. This showcases Rust's ability to report faults of this class at compile time, instead of relying on runtime analysis and programmer fluency.

```
error[E0502]: cannot borrow ‘nodes‘ as mutable because it is also borrowed as immutable
--> main.rs:4:1
  |
3 | let first = &nodes[0];
  |             ----- immutable borrow occurs here
4 | nodes.push(3);
  | ^^^^^^^^^^^^^ mutable borrow occurs here
5 | println!("{first}");
  |          ------- immutable borrow later used here
```

Guarantees such as this do not make Rust absolutely immune to memory errors, since it exposes functionality such as `unsafe` code blocks [The26b, The26c] to bypass its strict static analysis. Furthermore, it does not contain any fundamental protections against statistics or bioinformatics domain-specific errors.
Therefore, the main practical advantage is that Rust converts an entire class of memory faults into compile-time rejections, and explicitly marks boundaries where a programmer must manually guarantee memory safety themselves.

4.7 Ecosystem maturity, interoperability, and developer experience

The third and final research question concerns practical trade-offs, and here R remains the strongest of the three ecosystems. This is mainly because R fundamentally contains a statistical environment with a large collection of domain-specific packages, code conventions, and workflows. For instance,

CRAN Task Views organise packages by statistical and scientific topic, including but not limited to machine learning, omics, and phylogenetics [CRA26]. Furthermore, Bioconductor adds a specialised layer for genomics and high-throughput biological data [HCG+15]. Currently, R retains its position as the strongest candidate for workloads that mainly contain exploratory analysis or statistical modelling/visualisations.

Rust does have useful building blocks, but it does not yet replicate R's statistical toolkit as an integrated environment. For numerical arrays, Rust has `ndarray`; for probability distributions and statistical functions, it has `statrs`; `linfa` aims to provide a toolkit for machine learning; and for dataframe-style data manipulation, Polars provides a Rust-based query engine [nda26, sta26, lin26, Pol26]. These libraries suggest that the infrastructure for statistical work in Rust is still taking shape. However, these currently exist as scattered infrastructure components, compared to R's fully integrated statistical environment. Consequently, a Rust implementation makes the most sense when the task is purely performance-critical and the input-output contract is well defined. R remains more attractive when the task involves a more exploratory phase.

Another consideration is language interoperability: how readily an implementation in one language can be embedded into a pipeline built around another. This is a critical requirement in bioinformatics specifically, since real pipelines are typically built around multiple languages, pairing performance-critical kernels with higher-level orchestration and analysis. The benchmark setup of this thesis itself matches that design: a Python orchestrator invokes the Rust, C++ and R harnesses and parses their outputs into a shared CSV and Parquet schema (Section 2). More specifically, this is a *coarse-grained* setup, in which each language runs as its own monolithic entity. Finer, in-process forms of interop also exist: `extern "C"` and `cbindgen` [Moz26] represent the same kind of C-level interfaces that R and Python already use to call compiled C and C++ kernels. Furthermore, higher-level bindings such as `PyO3` [PyO26] and `extendr` [ext26] (for Python and R respectively) enable Rust functions to be called directly from within those languages.

Rust documentation provided by `rustdoc` is consistent, type-linked, and generally strong at the API level. It is useful for understanding function signatures, trait bounds, and module structure. However, the Rust bioinformatics ecosystem has fewer long-form tutorials, worked biological examples, and domain-level vignettes than mature R packages. R documentation can occasionally be uneven, but high-quality Bioconductor packages commonly include vignettes that explain not only how to call a function, but how the function fits into an analysis. For domain scientists, that difference may matter the most.

The implementation work in this thesis generally reflected the expected programmer experience, though these observations reflect the experiences of only a single developer new to Rust: the ownership model introduced some initial learning overhead, but the strict compiler caught critical issues and automatically provided a solution in most cases. Additionally, Cargo provided a coherent build and dependency workflow in contrast to C++'s manual CMake targets, dependency pins and API details. R was highly flexible at the user-interface level, but several package migrations and measurement semantics resulted in some friction: for example, `Biostrings::pairwiseAlignment` moved to `pwalign`, and `bench::mark()$mem_alloc` does not measure the same quantity as the Rust and C++ memory trackers.

4.8 Limitations and threats to validity

Several limitations constrain the interpretation of the benchmark results:

- **Output equivalence is not enforced:** The largest caveat of the benchmark is that it assumes each library is mature enough to be relied upon for accurate results. A dedicated pre-benchmark equivalence check would strengthen the methodology. This is especially important for the alignment algorithms, where different scoring conventions can yield different optimal alignments. The runtime comparison itself is largely unaffected, since both full-matrix dynamic-programming implementations evaluate the same $\mathcal{O}(nm)$ cells regardless of the penalty values; the asymmetry therefore threatens output equivalence more than the timing results.
- **The alignment scoring schemes are not fully identical.** The Smith-Waterman and Needleman-Wunsch implementations use the conventions exposed by their respective libraries. Therefore, exact alignment-score equivalence cannot be inferred from the benchmark alone.
- **The statistical summaries differ by harness.** Criterion (Rust) reports a bootstrapped confidence interval on the median, Google Benchmark (C++) reports a median point estimate with a confidence interval on the mean, and bench (R) reports a min-to-median range instead of a formal confidence interval. The intervals are adequate indicators of measurement stability, but cannot be used as identical statistical objects.
- **Memory measurements are not fully comparable across languages.** As previously mentioned, R only reports its own managed allocation and misses some memory allocated by compiled code. These values should therefore be treated as lower bounds.
- **The study is single-machine and single-threaded.** The benchmark was run on one controlled x86-64 environment. The methodology does not consider ARM behaviour, GPU acceleration, multi-threaded algorithms, which may be present on distributed research systems.
- **The safety comparison is qualitative.** The safety argument is based on language semantics and compiler behaviour, not on a large empirical study of bugs in production bioinformatics tools.

As a consequence, the results cannot be interpreted as a general expectation for all Rust/C++ implementations and R workflows. However, they do provide clues about the current state of the Rust bioinformatics ecosystem specifically.

5 Conclusion

5.1 Answers to the research questions

5.1.1 RQ1: performance

The benchmark results show that Rust delivers a solid compiled speed, but every outcome depended far more on representation choices and library maturity than on the language itself. In KMP, C++ was faster at every input size, with the gap narrowing as the input size grew. The cause is a constant-factor difference between the two libraries rather than any inherent weakness of Rust at string search. Smith-Waterman was a similar case: SSW ran faster than `rust-bio` because the former is implemented as a SIMD striped aligner, against the latter's scalar full-matrix one. The resulting large memory gap is consistent with this difference in data structure. For the Needleman-Wunsch comparison, the Rust implementation started out faster at very small inputs, but this advantage collapsed into parity at larger sizes once R's interface overhead reduced, relative to the computation by the compiled kernel. Lastly, tree traversal showcased the clearest gap in ecosystem maturity: Rust's poor large-n behaviour originated from the parsing path in the upstream `phylo` crate, not from any inefficiency inherent to the language's tree-walking. Therefore, RQ1 can be answered positively, but with a condition: Rust is fast enough to be viable, yet its real-world performance is ultimately limited by the maturity and design of whichever crate/library a user chooses to use.

5.1.2 RQ2: static safety guarantees and compiler behaviour

Modern C++ offers strong countermeasures of its own (including RAII, smart pointers, standard containers, and the sanitizer and static-analysis toolchain), yet its default semantics still offer weaker protection against undefined behaviour. Notably, Rust does not provide an *absolute* guarantee of correctness either; `unsafe` blocks, FFI boundaries, panics, and ordinary logic errors are all still possible, and the Rust Reference explicitly states that undefined behaviour can reappear once an unsafe boundary is violated. In safe Rust, however, ownership, borrowing, and the type system eliminate an entire class of use-after-free, aliasing, and data-race errors before the program runs, while safe indexing turns out-of-bounds access into a defined check. R sits somewhere in the middle: it is memory-managed and safe at the user level, but its bioinformatics speed comes largely from compiled extensions, and those extensions do not acquire Rust-style static guarantees merely by being called from R. Therefore, to answer RQ2, Rust provides stronger static safety than compiled languages such as C or C++, which many user-level pipelines (such as those in R) typically depend upon.

5.1.3 RQ3: statistical toolkit, ecosystem maturity, and documentation

Rust has recently gained a credible bioinformatics foundation with the appearance of libraries such as `rust-bio` and `phylo-rs`, and its build tooling and API documentation are consistent. However, it cannot yet be compared to the most established scientific ecosystems. This also applies to its bioinformatics ecosystem, which is not yet as integrated as R's. This remains the more mature choice for workloads such as downstream modelling, visualisations, and domain-specific biological analysis. Furthermore, its ecosystem is still thin enough that one weak crate can become an unavoidable practical cost, as the tree-traversal result demonstrated. Interoperability is in a better state: Rust's

C ABI and bindings such as `PyO3` and `extendr` let a Rust kernel drop into an existing R or Python pipeline, which theoretically makes incremental adoption currently possible. Therefore, the answer to RQ3 is that Rust can properly take over the task of an individual computational kernel, but not a complete exploratory and statistical workflow such as R.

5.2 Practical recommendations

For a researcher or programmer choosing among the three, the practical approach is to match each language to the part of the pipeline where it fits best. Rust's stricter compiler and ownership model carry a learning cost, so its reliability gains are most effective when a team already has Rust familiarity or is willing to expand into it. In cases such as this, Rust is the right tool for compiled performance together with strong safety guarantees. C++ is the right tool when the workload rests on one or more long-established and heavily optimised libraries, or when it must integrate with existing HPC and systems code. R remains the right tool for a complete package that can handle downstream analysis, statistical modelling, exploratory work, and established Bioconductor workflows. In practice, the most realistic architecture for many projects is hybrid: Rust and/or C++ for the performance-critical kernels, and R for analysis, diagnostics, and interpretation.

5.3 Future work

Several extensions could strengthen the benchmarking pipeline. The first is to add automated output-equivalence checks before timing, since an output validation step would introduce an extra reliability metric into the benchmark result. The second is to reduce the impact of library design choices, by adding a SIMD Rust Smith-Waterman implementation, or a packed-alphabet Rust KMP. These additions would ensure that the comparison is no longer scalar against SIMD or byte-slice against packed. The third is to rerun the tree benchmark against an implementation better suited to append-only, traverse-once workloads. The fourth is to move beyond a single x86-64, single-threaded setting to cover other architectures and parallel variants utilising a GPU. Finally, a dedicated study could measure the practical benefits in hybrid workflows, in which a Rust kernel is dispatched from an R or Python pipeline, and compared against C or C++ kernels. None of these would likely overturn the present conclusion, but together they can further clarify exactly which limitations belong to the Rust language itself, and which belong to the current state of its ecosystem.

The main case for Rust in bioinformatics is the combination of compiled performance with added static safety guarantees that the established low-level alternatives do not provide by default. When a mature crate exists, Rust is already a realistic drop-in replacement for a computation-heavy kernel. This thesis has found, however, that this is not always the case, and suitable alternatives may not be readily available either. Nevertheless, as the ecosystem continues to grow, that balance may keep shifting in Rust's favour in the near future.

References

[BBB+17] Abhiram Balasubramanian, Marek S. Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamaric, and Leonid Ryzhyk. System programming in Rust: Beyond safety. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS XVI)*, 2017.

[CAC+09] Peter J. A. Cock, Tiago Antao, Jeffrey T. Chang, Brad A. Chapman, Cymon J. Cox, Andrew Dalke, Iddo Friedberg, Thomas Hamelryck, Frank Kauff, Bartek Wilczynski, and Michiel J. L. de Hoon. Biopython: freely available python tools for computational molecular biology and bioinformatics. *Bioinformatics*, 25(11):1422–1423, 2009.

[cpp26] cppreference.com contributors. C++ undefined behavior. https://en.cppreference.com/w/cpp/language/ub, 2026. Accessed: 2026-05-22.

[CRA26] CRAN Task View Initiative. CRAN Task Views. https://cran.r-project.org/web/views/, 2026. Accessed: 2026-05-22.

[ext26] extendr Authors. extendr: R extension library for Rust. https://github.com/extendr/extendr, 2026. Accessed: 2026-06-03.

[HCG+15] Wolfgang Huber, Vincent J. Carey, Robert Gentleman, Simon Anders, Marc Carlson, Benilton S. Carvalho, Hector Corrada Bravo, Sean Davis, Laurent Gatto, Thomas Girke, Raphael Gottardo, Florian Hahne, Kasper D. Hansen, Rafael A. Irizarry, Michael Lawrence, Michael I. Love, James MacDonald, Valerie Obenchain, Andrzej K. Oleś, Hervé Pagès, Alejandro Reyes, Paul Shannon, Gordon K. Smyth, Dan Tenenbaum, Levi Waldron, and Martin Morgan. Orchestrating high-throughput genomic analysis with Bioconductor. *Nature Methods*, 12:115–121, 2015.

[HMv+20] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, 2020.

[JJKD18] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages*, 2(POPL):66:1–66:34, 2018.

[Kös16] Johannes Köster. Rust-Bio: a fast and safe bioinformatics library. *Bioinformatics*, 32(3):444–446, 2016.

[lin26] linfa contributors. linfa: A machine learning toolkit for rust. https://docs.rs/linfa/latest/linfa/, 2026. Accessed: 2026-05-22.

[LR23] Shing Lyu and Andrew Rzeznik. *Welcome to the World of Rust*, pages 1–8. Apress, Berkeley, CA, 2023.

[Mos25] Niema Moshiri. CompactTree: a lightweight header-only C++ library and Python wrapper for ultra-large phylogenetics. *Gigabyte*, 2025.

[Moz26] Mozilla. cbindgen: A project for generating C bindings from Rust code. https://github.com/mozilla/cbindgen, 2026. Accessed: 2026-06-03.

[NCB26] NCBI Nucleotide Database. Escherichia coli str. k-12 substr. mg1655 complete genome. https://www.ncbi.nlm.nih.gov/nuccore/NC_000913.3, 2026. Accessed: 2026-04-13.

[nda26] ndarray contributors. ndarray: An n-dimensional container for general elements and numerics. https://docs.rs/ndarray/latest/ndarray/, 2026. Accessed: 2026-05-22.

[Ope26] Open Tree of Life. Open tree of life synthetic tree release v16.1. https://tree.opentreeoflife.org/about/synthesis-release/v16.1, 2026. Accessed: 2026-04-13.

[Per20] Jeffrey M. Perkel. Why scientists are turning to Rust. *Nature*, 588(7836):185–186, 2020.

[Pol26] Polars contributors. Polars: Dataframe library user guide. https://docs.pola.rs/, 2026. Accessed: 2026-05-22.

[PS19] Emmanuel Paradis and Klaus Schliep. ape 5.0: an environment for modern phylogenetics and evolutionary analyses in R. *Bioinformatics*, 35(3):526–528, 2019.

[PyO26] PyO3 Project Authors. PyO3: Rust bindings for the Python interpreter. https://github.com/PyO3/pyo3, 2026. Accessed: 2026-06-03.

[SLF+15] Zachary D. Stephens, Skylar Y. Lee, Faraz Faghri, Roy H. Campbell, Chengxiang Zhai, Miles J. Efron, Ravishankar Iyer, Michael C. Schatz, Saurabh Sinha, and Gene E. Robinson. Big data: Astronomical or genomical? *PLOS Biology*, 13(7):e1002195, July 2015.

[SPWS13] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal war in memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, pages 48–62. IEEE, 2013.

[Sta25] Stack Overflow. Stack overflow developer survey 2025, 2025. Accessed: 2026-05-17.

[sta26] statrs contributors. statrs: Statistical computing library for rust. https://docs.rs/statrs/latest/statrs/, 2026. Accessed: 2026-05-22.

[The26a] The Rust Project Developers. The Rust Programming Language: Understanding ownership. https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html, 2026. Accessed: 2026-05-22.

[The26b] The Rust Project Developers. The Rust Reference: Behavior considered undefined. https://doc.rust-lang.org/reference/behavior-considered-undefined.html, 2026. Accessed: 2026-05-22.

[The26c] The Rust Project Developers. The Rustonomicon: What unsafe rust can do. https://doc.rust-lang.org/nomicon/what-unsafe-does.html, 2026. Accessed: 2026-05-22.

[VAME25] Sriram Vijendran, Tavis K. Anderson, Alexey Markin, and Oliver Eulenstein. Phylo-rs: an extensible phylogenetic analysis library in Rust. *BMC Bioinformatics*, 26:197, 2025.

[VGO+20] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: fundamental algorithms for scientific computing in Python. *Nature Methods*, 17(3):261–272, 2020.

[ZLGM13] Mengyao Zhao, Wan-Ping Lee, Erik P. Garrison, and Gabor T. Marth. SSW Library: An SIMD Smith-Waterman C/C++ library for use in genomic applications. *PLOS ONE*, 8(12):e82138, 2013.

A Environment and Reproducibility

A.1 Hardware and execution environment

Component	Configuration
Host CPU	AMD Ryzen 7 5800X, Zen 3, AM4, 8 physical cores. SMT was disabled for benchmarking, leaving 8 active physical cores. The CPU has a 3.8 GHz base clock and can boost up to 4.7 GHz, but all boost behaviour was disabled, such that all measurements were taken at a static 3.8 GHz.
Memory	32 GiB RAM, running at JEDEC DDR4-2133.
Host OS	Unraid 7.2.4, acting as the KVM/QEMU hypervisor.
Guest OS	Debian minimal installation running inside the KVM guest. No GUI, no `unattended-upgrades`, and no `irqbalance`.
Guest vCPUs	2 vCPUs pinned 1:1 to host physical cores using libvirt `vcpupin`. The QEMU emulator process was pinned away from those cores using `emulatorpin`.
Guest memory	8 GiB initial / 8 GiB maximum, swap disabled.
Kernel and libc	`6.12.74+deb13+1-amd64`, `Debian GLIBC 2.41-12+deb13u2`

Table 9: Host and guest benchmark environment

Setting	Value
Simultaneous Multi-Threading	Disabled
Turbo Boost	Disabled
Precision Boost Overdrive	Disabled
Global C-State Control	Disabled
CPPC	Disabled
CPPC Preferred Cores	Disabled
AMD Cool&Quiet	Disabled
Memory XMP profile	Disabled

Table 10: Host BIOS settings used during benchmarking.

A number of extra steps were taken to ensure maximum consistency between benchmark runs. Firstly, the Unraid host was booted with `isolcpus=6-7`, which removes these cores from the host scheduler's normal balancing domains. Secondly, the guest was configured to minimise scheduler, kernel-housekeeping, and page-cache noise.

A.2 Code repository & full reproduction recipe

All related code (including the benchmarking harness), scripts, and the full reproduction recipe can be found in the public GitHub repository: `https://github.com/medylme/lu-bachelor-thesis` Table 11 lists the resolved versions of all toolchains and libraries used for the reported measurements.

Component	**Version**
Rust toolchain	rustc 1.94.0 (4a4ef493e 2026-03-02), Cargo 1.94.0
C++ compiler	GCC 14.2.0 (Debian 14.2.0-19)
CMake	4.3.2
Python (orchestrator)	CPython 3.14.4
R	4.5.3 (2026-03-11)
r-bench	1.1.4
r-ape	5.8_1
bioconductor-biostrings	2.78.0
bioconductor-pwalign	1.6.0
pandas	3.0.2
pyarrow	24.0.0

Table 11: Resolved software toolchain and library versions used during benchmarking, as produced by `pixi run versions`.

B Additional Plots and Raw Data

B.1 Complete runtime and memory measurements

Algorithm	Language	n	Median	95 % interval	Peak memory
KMP	C++	1,000	634 ns	[628 ns, 646 ns]	—
KMP	C++	10,000	7.22 µs	[7.21 µs, 7.23 µs]	—
KMP	C++	100,000	227.22 µs	[226.93 µs, 227.74 µs]	—
KMP	C++	1,000,000	2.45 ms	[2.45 ms, 2.46 ms]	168 B
KMP	Rust	1,000	2.50 µs	[2.50 µs, 2.50 µs]	—
KMP	Rust	10,000	24.73 µs	[24.73 µs, 24.74 µs]	—
KMP	Rust	100,000	381.15 µs	[381.11 µs, 381.23 µs]	—
KMP	Rust	1,000,000	3.86 ms	[3.86 ms, 3.86 ms]	192 B

(continued on next page)

(continued from previous page)

Algorithm	Language	n	Median	95 % interval	Peak memory
Smith-Waterman	C++	100	26.62 µs	[26.52 µs, 26.72 µs]	—
Smith-Waterman	C++	500	219.39 µs	[219.15 µs, 219.91 µs]	—
Smith-Waterman	C++	1,000	1.45 ms	[1.45 ms, 1.45 ms]	—
Smith-Waterman	C++	5,000	56.53 ms	[56.46 ms, 56.62 ms]	69.7 KB
Smith-Waterman	Rust	100	121.47 µs	[121.30 µs, 121.73 µs]	—
Smith-Waterman	Rust	500	3.50 ms	[3.50 ms, 3.51 ms]	—
Smith-Waterman	Rust	1,000	13.94 ms	[13.94 ms, 13.95 ms]	—
Smith-Waterman	Rust	5,000	376.19 ms	[376.09 ms, 376.37 ms]	50.3 MB
Needleman-Wunsch	Rust	100	102.08 µs	[101.55 µs, 103.00 µs]	—
Needleman-Wunsch	Rust	500	2.84 ms	[2.84 ms, 2.84 ms]	—
Needleman-Wunsch	Rust	1,000	11.52 ms	[11.51 ms, 11.53 ms]	—
Needleman-Wunsch	Rust	5,000	327.48 ms	[327.16 ms, 327.69 ms]	50.4 MB
Needleman-Wunsch	R	100	4.28 ms	[4.20 ms, 4.28 ms]	—
Needleman-Wunsch	R	500	7.78 ms	[7.57 ms, 7.78 ms]	—
Needleman-Wunsch	R	1,000	17.33 ms	[17.05 ms, 17.33 ms]	—
Needleman-Wunsch	R	5,000	324.04 ms	[322.23 ms, 324.04 ms]	75.4 MB
Tree traversal	C++	100	9.98 µs	[9.93 µs, 10.01 µs]	—
Tree traversal	C++	1,000	141.88 µs	[140.83 µs, 142.62 µs]	—
Tree traversal	C++	10,000	2.18 ms	[2.17 ms, 2.18 ms]	—
Tree traversal	C++	100,000	16.67 ms	[16.08 ms, 16.87 ms]	21.5 MB
Tree traversal	Rust	100	37.56 µs	[37.52 µs, 37.64 µs]	—
Tree traversal	Rust	1,000	1.37 ms	[1.37 ms, 1.37 ms]	—
Tree traversal	Rust	10,000	151.56 ms	[151.47 ms, 151.74 ms]	—
Tree traversal	Rust	100,000	15.20 s	[15.18 s, 15.20 s]	40.7 MB
Tree traversal	R	100	311.13 µs	[298.77 µs, 311.13 µs]	—
Tree traversal	R	1,000	2.26 ms	[2.19 ms, 2.26 ms]	—
Tree traversal	R	10,000	20.34 ms	[20.08 ms, 20.34 ms]	—
Tree traversal	R	100,000	223.56 ms	[215.17 ms, 223.56 ms]	18.9 MB

Table 12: Complete runtime and memory measurements at the four input sizes per algorithm.

B.2 Empirical scaling fits

Table 13 extends the scaling-exponent results shown in Figure 8 with the corresponding intercept terms. Each row is the result of an ordinary least-squares fit of $\log t$ against $\log n$ over the four input sizes, where t is the median runtime in milliseconds. The bracketed interval on k is a 95 % bootstrap CI obtained from 5000 resamples with replacement of the four data points. The intercept column gives $\log_{10}(a)$, where a is the coefficient in the fit $t = a \cdot n^k$; the implied runtime in milliseconds for

an input size n can therefore be approximated as $t \approx 10^{\log_{10}(a)} \cdot n^k$.

Algorithm	Language	Expected k	Fitted k [CI]	Intercept $\log_{10}(a)$ (t in ms)
KMP	C++	1	1.23 [1.03, 1.50]	-6.92
KMP	Rust	1	1.08 [1.00, 1.19]	-5.85
Smith-Waterman	C++	2	1.98 [1.31, 2.73]	-5.73
Smith-Waterman	Rust	2	2.05 [1.99, 2.09]	-5.01
Needleman-Wunsch	R	2	1.11 [0.37, 1.82]	-1.84
Needleman-Wunsch	Rust	2	2.06 [2.02, 2.08]	-5.12
Tree traversal	C++	1	1.09 [0.88, 1.19]	-4.12
Tree traversal	R	1	0.95 [0.86, 1.04]	-2.46
Tree traversal	Rust	1	1.89 [1.56, 2.04]	-5.33

Table 13: Scaling fits per (algorithm, language) series.