Evaluation of the Development Effort and Scalability Performance of Graph Algorithms in Rust

Master Project (12 CP)

University of Basel
Faculty of Science
Department of Mathematics and Computer Science
High Performance Computing Group

Advisor and Examiner: Prof. Dr. Florina M. Ciorba
Supervisor: Reto Krummenacher

Author: Sylvain Rousselle
Email: sylvain.rousselle@unibas.ch

July 24, 2025

University of Basel

Contents

List of Acronyms

HPC	High-Performance Computing
miniHPC	HPC cluster of the HPC Group at the Department of Mathematics and Computer Science at the University of Basel
I/O	Input/Output
MPI	Message Passing Interface
RMA	Remote Memory Access
SLURM	Simple Linux Utility for Resource Management
BLAS	Basic Linear Algebra Subprograms
API	Application Programming Interface
RNG	Random Number Generator
FFI	Foreign Function Interface
STD	Standard Library
CPU	Central Processing Unit
RAII	Resource Acquisition Is Initialisation
MMD	Maximum Mean Discrepancy
EMD	Earth Mover's Distance
NUMA	Non-Unified Memory Access
AWS	Amazon Web Services

Chapter 1

Introduction

In today's world, performance and productivity are often all that counts. Most programming languages focus on either performance (i.e. shorter execution times) or productivity (i.e. low development effort). For example, C and C++ provide high performance but at the cost of productivity, whereas Python sacrifices performance for high productivity. The Julia language positioned itself as a trade-off between performance and development effort [1], although it comes with its own approach to distributed computing [2]. When adapting applications to today's increasingly distributed architectures, users need to account for an increased development cost, independently of the language, to integrate with inter-process communication libraries like MPI.

1.1 Motivation

A preliminary study [3] investigated and compared the performance, development effort, and scalability of graph benchmarking algorithms from the GAP benchmarking suite [4] across multiple execution paradigms. This study investigated implementations in C++, Julia, Python, and DAPHNE [5]. Rust is another programming language worthy of comparison, bringing a balance between performance and productivity along with a focus on safety and concurrency. As such, Rust becomes particularly relevant for scalable distributed applications.

1.2 Objectives

In this project, we originally aimed to expand on the findings of the preliminary study [3] by exploring Rust as a potential candidate for the study's comparison due to its potential for performance and productivity, and its focus on safe concurrency. However, parts of Rust's ecosystem are still in an early stage, leading us to redefine our goals for this project. We originally planned to implement four of the graph-based kernels described by the GAP benchmarking suite, namely Connected Components, Breadth-First Search, Triangle Counting, and Page-Rank, in Rust to compare its performance and the development effort against the results from the preliminary study [3]. Due to a lack of required functionalities in Rust's sparse linear algebra ecosystem, we have revised our objectives to focus on the general suitability of Rust for High-Performance Computing (HPC). For this, we evaluate the performance of Rust on two computational workloads: k-means clustering and the Mandelbrot set.

1.3 Contribution

We make two contributions. First, we successfully parallelise k-means clustering and the Mandelbrot set in Rust. Additionally, we notice that k-means clustering is a questionable choice as a benchmarking workload as its performance is highly sensitive on the initial parameters and the dataset used. Our second contribution is in establishing that, due to its non-standard MPI syntax, immature sparse linear algebra support and different approach to shared memory parallelism, Rust is not yet mature enough for wide-spread adoption in HPC communities, albeit showing potential for HPC applications due to its focus on safe concurrency, which would help reduce the number of concurrency-related bugs.

1.4 Outline

We begin by presenting the Rust programming language, highlighting its quirks and features that are relevant for distributed and parallel computing in Chapter 2. We further present the current state of its library ecosystem on the topics of shared memory parallelism, inter-process message-passing, and (sparse) linear algebra. We continue with the shift in objectives in Section 2.3 and describe the computational workloads we use on a theoretical level. In Chapter 3, we introduce the algorithms we use, our approaches to their parallelisation, and their implementation details. In Chapter 4, we cover the metrics we collect, how we evaluate them, and the measures we take to control the environment of our experiments. We continue by presenting our results and their analysis in Chapter 4 and conclude with Chapter 5.

Chapter 2

Background

In this chapter, we cover the origins of Rust and some of its design choices and paradigms, uses in the industry, and the current state of Rust's HPC ecosystem. Further, we discuss the state of (sparse) linear algebra support in Rust and why we redefined our objectives. Finally, we go over the updated project plan.

2.1 The Rust programming language

Rust is a general purpose programming language, designed in 2006 by Mozilla Research employee Graydon Hoare [6]. Rust was sponsored by Mozilla in 2009, with the first release appearing in 2012, and the first stable release in 2015. In 2021, ownership of all trademarks and domain names were transferred to the Rust Foundation, newly founded by Amazon Web Services (AWS), Google, Huawei, Microsoft, and Mozilla. Rust emphasizes performance, type safety, and concurrency, while enforcing memory safety. Instead of a conventional garbage collector, Rust enforces the Resource Acquisition Is Initialisation (RAII) idiom, meaning that variables are automatically freed when they go out of scope. Moreover, Rust ensures memory-safety and thread-safety with its ownership system and so-called *borrow checker*, which tracks the lifetime of references at compile time [7]. Rust does not enforce a particular programming paradigm, but takes ideas from both functional programming and object-oriented programming. The following sections are meant to highlight some key features of Rust that we use in this project. For those interested in learning more about the language, a good starting point is the official book [8], which is also freely available online[1].

2.1.1 Functional programming

We begin by discussing some functional programming aspects in Rust. As an example, let us take a look at a simple *for* loop, in which we iterate from 0 to 10 (exclusive). This loop can be written in an imperative programming style, shown in Listing 2.1, or in a functional programming style by converting the range 0..10 into an *iterator*, as shown in Listing 2.2.

[1]https://doc.rust-lang.org/book/, accessed June 15, 2025.

Listing 2.1: Imperative-style *for* loop in Rust, iterating from 0 to 10 (exclusive)

```rust
for i in 0..10 {
    // loop body
}
```

Listing 2.2: Same *for* loop as Listing 2.1, written in functional programming style

```rust
(0..10) // Create a range from 0 to 10 (exclusive)
    .into_iter() // Convert the range into an iterator
    .for_each(|i| { /* loop body */ }) // Apply closure to each item
```

The functional programming side of Rust comprises two major concepts, closures and iterators, which we discuss in more details next.

Closures

In Listing 2.2, the *for_each* call takes a closure as input. Closures are anonymous functions that can be defined in one place and evaluated in another context. Unlike functions, closures can capture their environment, meaning they can access variables from the scope in which they are defined. This is important in the context of shared-memory parallelism, which we will cover in a later section. In Listing 2.3, the closure *equal_to_x* accesses the variable x, which is not defined inside the closure, but within the same scope as the closure.

Listing 2.3: A closure that captures the variable x from its environment

```rust
fn main() {
    let x = 0;
    let equal_to_x = |k| k == x; // equal_to_x captures x
    println!("{}", equal_to_x(0)); // prints "true"
}
```

Iterators

To write the for loop in Listing 2.2 in a functional style, we first convert the range into an iterator. An iterator is a pattern that allows us to perform a task on a sequence of items, one element at a time. Iterators abstract away the logic of iterating over each item and determining when the end of the sequence is reached. For example, say we have an array of integers and we want to double the value of each element in the array. Instead of manually defining and updating an index variable, we can create an iterator over the array, as shown in Listing 2.4.

Listing 2.4: Using an iterator to double the value of each element in an array

```rust
let mut arr = [1, 2, 3, 4]; // Define a mutable array of integers
let arr_iter = arr.iter_mut(); //  Create an iterator over arr
for value in arr_iter { // Iterate over each value
    *value *= 2; // Double the value
}
```

Diving further into the functional side of Rust, we can abstract the for loop away by using specific functions, known as *iterator adaptors*, that take the iterator and produce another iterator. In our example, we can use the adaptor *map*, which takes a closure, applies it to each element in the iterator, and returns the resulting iterator:

Listing 2.5: Using iterator adaptors to double the value of each element in an array

```rust
let mut arr = [1, 2, 3, 4];
arr = arr.iter_mut() // Create an iterator over arr
            .map(|x| *x * 2) // Apply the closure to each element
            .collect(); // Consume the iterator to get the result
```

Since Rust's iterators are lazy, we need to call a consuming adaptor to get the result from the previous calls to iterator adaptors. In Listing 2.5, we call the *collect* adaptor to aggregate the result back into an array.

2.1.2 Shared memory parallelism in Rust

When it comes to concurrency, Rust follows the philosophy *Share memory by communicating* instead of *Communicate by sharing memory*. Concurrent access to a shared variable is strictly controlled. At any time, there can be either multiple immutable references to a variable, or a single mutable reference to that variable. This is enforced by the *borrow checker* at compile time, effectively preventing race conditions.

When a thread spawns, it takes a closure which it then executes. We discussed in Section 2.1.1 how closures can capture their environment. The closures passed to threads are no different, as illustrated in Listing 2.6.

Listing 2.6: Spawning a thread to execute code concurrently, which captures the variable x by reference

```rust
fn main() {
    let x = 0;
    // Closures passed to threads take no arguments
    let handle = std::thread::spawn(|| {
        let y = 0;
        let is_equal = y == x; // x is captured here
        println!("{}", is_equal); // Prints "true"
    });

    // Join the thread to ensure it finishes before the program exits.
    // unwrap() used for simplicity instead of proper error handling
    handle.join().unwrap();
}
```

In Listing 2.6, the thread's closure captures the variable x from the global context. Since x is immutable and thus only read from, the closure can capture x by reference. If the closure captured a mutable variable by reference instead, we would have a mutable reference within the thread and another in the global context where the variable lives. This is not allowed and the compiler will throw an error. Instead, the closure can take ownership of the variable, invalidating the variable in the global context. We say that the variable has been *moved*.

Listing 2.7: Spawning a thread that takes ownership of a mutable variable

```
fn main() {
    let mut x = 0;
    // Closures passed to threads take no arguments
    let handle = std::thread::spawn(|| {
        x += 1; // x is moved here
        let y = 0;
        let is_equal = y == x;
        println!("{}", is_equal); // Prints "false"
    }); // x is no longer valid after this line
    // Attempting to access x results in a compilation error

    // unwrap() used for simplicity instead of proper error handling
    handle.join().unwrap();
}
```

If the global context needs to retain access to a variable, we have to share the value in a way that prevents race conditions. For example, we could wrap the value in a *mutex*, requiring a lock to be acquired before accessing the value. An exception to this is integers, for which atomic variants exist. The other option is to use message-passing to communicate with the thread through channels.

Listing 2.8: Using channels to communicate with a thread

```
fn main() {
    let x = 0;

    // Creating transmitters (tx) and receivers (rx)
    let (tx1, rx1) = std::sync::mpsc::channel();
    let (tx2, rx2) = std::sync::mpsc::channel();

    // Explicitely tell the closure to move all captured variables
    let handle = std::thread::spawn(move || {

        // Receive a message through the first channel
        let x = rx1.recv().unwrap(); // rx1 is moved here

        let y = 0;
        let is_equal = y == x;

        // Send the result through the second channel
        tx2.send(is_equal).unwrap(); // tx2 is moved here
    });

    // Send x through first channel
    tx1.send(x).unwrap();

    // Receive the result through the second channel
    let is_equal = tx2.recv().unwrap();

    println!("{}", is_equal); // Prints "true"
    handle.join().unwrap();
}
```

In Listing 2.8, we create two channels to send and receive messages to and from the thread. Rust's channels are *multiple sender, single receiver*, hence the need to create two channels in our example. The main thread creates a variable x and sends it through the first channel. The thread reads from the channel, compares the received value to an internal variable y, and sends the result back through the second channel.

Rayon

Rayon [9] is a shared memory parallelism crate (in Rust's ecosystem, libraries are called crates) for Rust, which abstracts away the manual management of threads by providing parallel iterators as a high-level interface. Rayon also provides task parallelism, but recommends using the iterator approach first as it is more efficient. In both cases, Rayon leverages work-stealing [10] to load-balance the work within its threadpool.

Parallelising the example in Listing 2.5 is simple. The only change required is to call *par_iter_mut()* instead of *iter_mut()*, as shown in Listing 2.9. Rayon provides a parallel version for most methods in the Standard Library (STD) that return an iterator over a collection, prepending the prefix *par_* to the original name of the STD function. There are some exceptions to this naming scheme where *_par_* is inserted in the middle of the function's name instead. For example, the parallel version of *into_iter()* is *into_par_iter()*.

Listing 2.9: Parallelising the example in Listing 2.5 by leveraging Rayon's parallel iterators

```
let mut arr = [1, 2, 3, 4];
arr = arr.par_iter_mut() // Create a parallel iterator over arr
            .map(|x| *x * 2)
            .collect();
```

2.1.3 MPI bindings for Rust

The Message Passing Interface (MPI) [11] is a specification for a message-passing style concurrency library, which describes bindings for the C/C++ and Fortran programming languages. Currently, there is no Rust implementation of MPI. However, the rsmpi [12] crate aims to bridge the gap by providing bindings to MPI 3.1 implementations in C through Rust's Foreign Function Interface (FFI). Rsmpi is tested to work with MPICH version 3.3.2 and OpenMPI version 4.0.3, but, according to the developers of rsmpi, users also reported success with Spectrum MPI version 10.3.0.1 and Cray MPI version 8.1.16 [12].

The bindings are very usable in their current state but do not yet cover the entire MPI 3.1 specification. Most notably, one-sided communication, i.e. Remote Memory Access (RMA), and MPI parallel I/O are not yet supported. Additionally, the syntax of MPI calls in Rust differs from the traditional C syntax, as shown in Listing 2.10.

Listing 2.10: Example of the MPI syntax in Rust

```rust
fn main() {
    // Equivalent to MPI_Init in C
    let universe = mpi::initialize().unwrap();

    // Get the MPI_COMM_WORLD communicator
    let world = universe.world();

    // Get the size of the communicator and
    // the rank of this process
    let size = world.size();
    let rank = world.rank();

    // Simple point-to-point communication example
    if rank == 0 {
        // Send 42 to rank 1. Equivalent to MPI_Send in C.
        world.process_at_rank(1).send(&42i32);
    } else {
        // Receive a message from rank 0. Equivalent to MPI_Recv in C.
        let (msg, status) = world.process_at_rank(0).receive::<i32>();
    }
} // Implicit MPI_Finalize happens here
  // This is due to the 'universe' variable going
  // out of scope, at which point it is freed
```

2.1.4 Rust in the industry

Examples of software using Rust are, but not limited to [6]:

- Firefox, specifically its underlying browser engine Gecko [13].
- The Linux kernel, with the first drivers written in Rust being accepted and released in version 6.8 [14].
- Firecracker, an open-source virtualisation software by AWS [15].
- OpenDNS, a DNS resolution service from Cisco, did use Rust at some point, but seems to have moved away from it since [16].
- The npm package manager [17].

2.2 State of sparse linear algebra support in Rust

Initially, our plan was to work with the kernels described by the GAP suite [4], which is an effort to standardise graph processing evaluations. A good way to represent sparsely connected graphs is to store their adjacency matrix, as sparse matrices can be efficiently stored in memory. This further allows graph algorithms to be specified as a succession of matrix operations. To this end, we investigated Rust's sparse linear algebra ecosystem by looking at multiple crates. However, we quickly observed that the support for sparse linear algebra in Rust was not yet mature enough for this project. We discuss the crates we have investigated and their limitations in more details in the following sections.

2.2.1 nalgeblra

The first option we considered is *nalgebra* [18], which is a general linear algebra crate with support for sparse linear algebra. We ruled out nalgebra for performance reasons, as is reflected in their documentation:

> "The library is in an early, but usable state. [...] the focus so far has been on correctness and API design, with little focus on performance." [19]

2.2.2 faer

Like nalgebra, *faer* [20] is a general linear algebra crate with support for sparse linear algebra. It is a fast evolving project that already offers good performance, but is, at the time of writing, poorly documented. Since faer does not provide some of the matrix operations required for this project such as broadcast multiplication, we looked into the possibility of grafting a custom implementation on top of faer. Doing so requires access to the data buffers underlying faer's data structures. Unfortunately, those data structures do not expose their underlying buffers, which makes this idea unworkable. In addition, this also makes it harder to send the data to another process with MPI, as one would need to serialise and deserialise the data structure before and after sending, respectively. Thus, we ruled out faer as a candidate for this project, but we will keep an eye on its development for future endeavours.

2.2.3 sprs

Like faer and nalgebra, *sprs* [21] is a work in progress. It however focuses solely on sparse linear algebra, and unlike faer, it provides access to the data structures' underlying buffers, allowing for custom implementations of the missing matrix operations. Unfortunately, sprs currently only provides matrix multiplication, addition, subtraction, and in-place division, which only covers a fraction of the matrix operations required by the kernels from the GAP suite. Since one of our original goals was to compare Rust to other languages, having multiple custom implementations of basic subroutines makes the different programming languages harder to compare, if not incomparable.

2.2.4 cblas

Due to the difficulties encountered with Rust's own linear algebra ecosystem, we looked at the *cblas* [22] crate, which provides bindings to the C BLAS API [23]. Unfortunately, the underlying BLAS implementation is provided by OpenBLAS [24], which does not support sparse linear algebra:

> "OpenBLAS implements only the standard (dense) BLAS and LAPACK functions with a select few extensions popularized by Intel's MKL." [25]

2.3 Redefining the project's goals

The issues mentioned above rendered our project goals unworkable, at which point we decided to redefine the project's goals. Instead of comparing the performance and development effort of kernels written in Rust, we now only investigate Rust's suitability for HPC. This allows us to move away from graph-based kernels, which are currently difficult to implement in Rust, and focus on other workloads. We chose k-means clustering from the Rodinia benchmark suite [26] and the Mandelbrot set, as both workloads do not rely on sparse linear algebra. We describe the two workloads in more details in the next sections.

2.3.1 k-means clustering

k-means clustering [27], which is part of the Rodinia benchmark suite [26], is an algorithm for partitioning n data points into k disjoint subsets $S_j \in \{S_1, \ldots, S_k\}$, each containing n_j data points so as to minimize the sum-of-squares criterion

$$J = \sum_{j=1}^{k} \sum_{n \in S_j} |\mathbf{x_n} - \mu_j|^2$$

where $\mathbf{x_n}$ is a vector representing the n-th data point and μ_j is the geometric centroid [28] of the data points in S_j. The result is a partition of the data space into Voronoi cells, as can be seen in Figure 2.1.

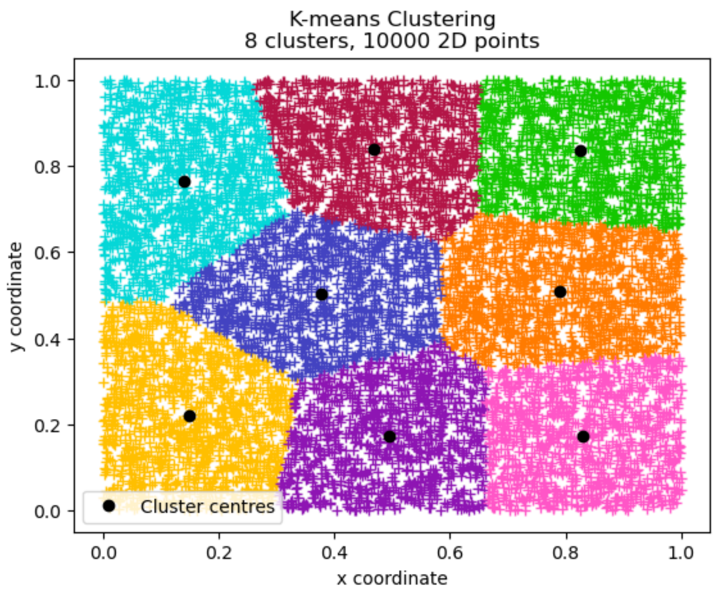


Figure 2.1: Partitioning of 10'000 points into 8 clusters, with each centre being the arithmetic mean of the points that are part of that cluster.

Finding optimal solutions for k-means is a computationally difficult (NP-hard) problem [29, 30], leading to heuristic algorithms to be generally preferred. One such method is Lloyd's algorithm [27, 31], which comprises two steps. Initially the cluster centres are set to some value. In the first step, every data point is assigned to the cluster whose centre is closest to that point w.r.t the squared Euclidean distance. Then in the second step, the cluster centres are re-computed according to the the points contained in each cluster after the first step. These two steps are then repeated until a stopping criterion is met, i.e. there is no further change in the assignment of the data points.

When performed until convergence, the result may be counter-intuitive, as the input parameter k may not match the number of clusters in the dataset. This is further amplified by the assumption of spherical clusters of roughly equal size that are separable, which may not reflect the dataset's structure. Moreover, the number of iterations until convergence is heavily dependent on the data points and the initial cluster centres' positions.

2.3.2 The Mandelbrot set

The Mandelbrot set is perhaps one of the most famous fractal curves. It is defined in the complex plane as the set of complex numbers $c = x + iy$ for which the series

$$z_{n+1} = z_n^2 + c \tag{2.1}$$

with the starting point $z_0 = 0 + 0i$ remains bounded in absolute value under a maximal number of iterations n_{max}. The set features an infinitely complex boundary with increasingly finer details at higher magnifications, as can be seen in Figure 2.2.

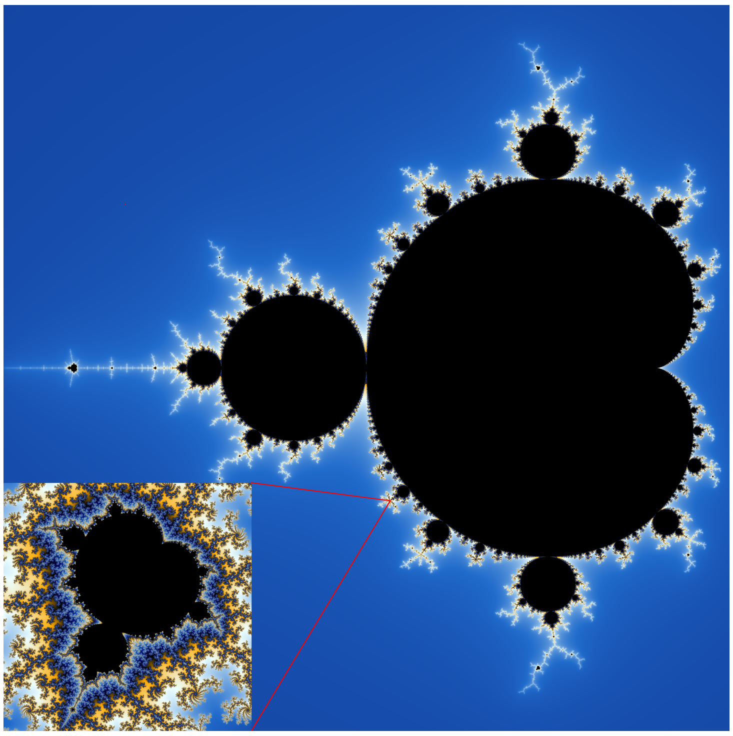


Figure 2.2: Visualisation of the Mandelbrot set using a cyclic, continuously coloured space. The frontier of the set features increasingly intricate details at higher magnifications.
Bottom left: 4000× zoom showing the area around the point $c = -0.673085 - 0.357678i$.

The visualisation of the Mandelbrot set is done by colouring the pixels in the complex plane, with the x-axis and y-axis respectively representing the real and imaginary components of the number c. The set itself is traditionally coloured black, while its complement is coloured based on how quickly equation 2.1 diverges. To do so, an integer value n_{div} is defined so that it represents the largest value n for which $|z_n| \leq 2$ still holds. Each colour represents a different value of n_{div}.

This approach has the drawback of creating bands of colour, which as a type of aliasing, is not visually appealing (see Figure 2.3). This can however be solved by using the *normalised iteration count* algorithm [32, 33] to get a smooth transition of colours. The algorithm associates a real number ν to each value of z by using the connection between n_{div} and the potential function $\Phi(z) = \lim\limits_{n\to\infty} \frac{|z_n|}{2^n}$, where z_n is the value after n iterations.

We redefine n_{div} to be the largest value of n for which $|z_n| \leq r$ still holds, with the escape radius $r \geq 2$. For large r, we have

$$\frac{\log|z_n|}{2^n} = \frac{\log r}{2^{\nu(z)}}$$
$$\nu(z) = n_{div} + 1 - \log_2(\frac{\log|z_{n_{div}}|}{\log r})$$

The value $\nu(z)$ is then mapped on a cyclic scale containing h distinct colours, numbered from 0 to $h-1$. The colour of the pixel is defined as $h_k \in \{0, 1, \dots, h-1\}$ with

$$k = \lfloor \nu(z) * d \rfloor \mod h$$

where d is the colour density in the picture (e.g. $d = 256$ for 8-bit RGB) and h_k is the k-th colour in the scale.

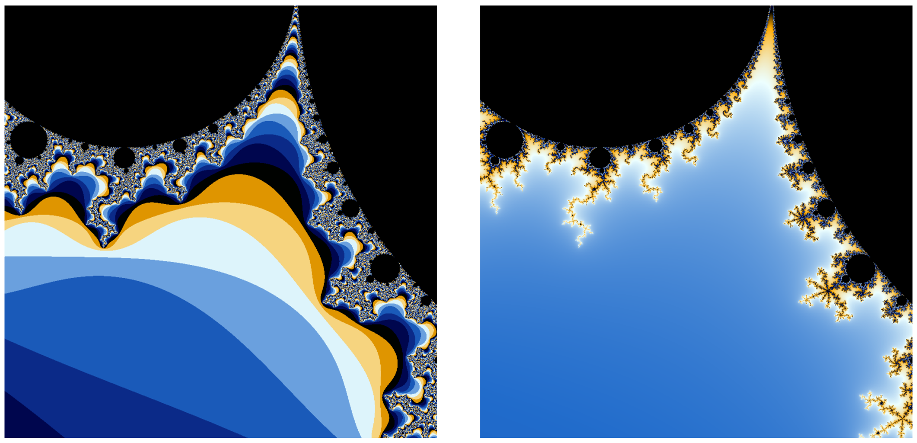


Figure 2.3: Example of the aliasing artifacts (banding) that appear in the Mandelbrot set when using the iteration count directly (left). On the right, we use the normalized iteration count algorithm with an escape radius of $r = 1'000$, which results in smooth colour transitions.

Chapter 3

Methodology

In this chapter, we introduce the algorithms we use, along with their implementation details, as well as our approach to their parallelisation. The metrics we collect, how we evaluate them, and the measures we take to control the environment of our experiments are covered at the start of next chapter.

3.1 k-means clustering

The datasets we use contain 1 million, 2 million, 4 million, and 8 million points respectively. Each point is represented using two coordinates ranging between 0 and 1. The 4 million points dataset is obtained by keeping the first half of the 8 million points dataset. Similarly, the 2 million and 1 million points dataset are obtained by keeping the first half of the next bigger dataset. We generate the 8 million points dataset with the utility tool provided by the Rodinia benchmark suite [26] on Dropbox[1].

3.1.1 Sequential implementation

We base our sequential implementation on the one provided by the Rodinia benchmark suite [26] on GitHub[2]. After initialising the cluster centres' starting values, we iterate over the dataset as follows: For each point in the dataset, we find the closest cluster centre w.r.t. the squared euclidean distance and update the point's membership. We then add the point's coordinates to the corresponding centre's running sums (two per cluster, one for the x-axis and one for the y-axis) and keep track of the cluster's new size. Once all the points have been processed, we go through the cluster centres' sums and divide each coordinate by the size of the respective cluster, which gives us the mean position of the points in that cluster. This corresponds to the new position of the cluster centre, which is used in the next iteration of the algorithm. The algorithm terminates when the number of points that change membership in the current iteration is less or equal to a chosen input parameter threshold t. In our experiments, we partition the dataset in $k = 8$ clusters and use $t = 0$ for the termination criterion.

[1]https://www.dropbox.com/s/cc6cozpboht3mtu/rodinia-3.1-data.tar.gz, accessed March 11, 2025.

[2]https://github.com/yuhc/gpu-rodinia/tree/master/openmp/kmeans/kmeans_serial, accessed May 28, 2025.

Our implementation differs slightly from the reference implementation in the choice of the initial cluster centres. The version provided in Rodinia uses the first k points in the dataset as initial cluster centres, whereas we choose the centres randomly using a Random Number Generator (RNG) derived from the ChaCha family of stream ciphers [34]. For reproducibility purposes, we use the same seed for the RNG in all of our experiments. The pseudocode for our sequential implementation is provided in Algorithm 1.

Algorithm 1 Pseudocode for the sequential k-means clustering algorithm in 2D with k clusters, n points and termination threshold t. The *min_dist* function returns the index of the nearest centre with respect to the squared euclidean distance.

```
Require: $points = [p_1, \ldots, p_n]$, $k > 1$, $t >= 0$
  function K_MEANS($points$, $k$, $t$)
      $memberships \leftarrow [m_0, \ldots, m_n]$, $m_i = 0\ \forall i$
      $centres \leftarrow [c_1, \ldots, c_k]$, $c_i \leftarrow (x, y)$, $x, y \in [0, 1]$
      $new\_centres \leftarrow [nc_1, \ldots, nc_k]$, $nc_i = (0, 0)\ \forall i$
      $sizes \leftarrow [s_1, \ldots, s_k]$, $s_i = 0\ \forall i$
      $delta \leftarrow 0$
      while $delta > t$ do
          for $p_i \in points,\ \ m_i \in memberships,\ \ i \in 1, 2, \ldots, n$ do
              $delta \leftarrow 0$
              $nearest \leftarrow min\_dist(p_i, centres) \in [1, k]$
              if $nearest \neq m_i$ then
                  $delta \leftarrow delta + 1$
              end if
              $m_i \leftarrow nearest$
              $new\_centres_{nearest} \leftarrow new\_centres_{nearest} + p_i$
              $sizes_{nearest} \leftarrow sizes_{nearest} + 1$
          end for
          for $c_i \in centres,\ nc_i \in new\_centres,\ s_i \in sizes,\ i \in 0, 1, \ldots, k$ do
              $c_i \leftarrow nc_i / s_i$
              $nc_i \leftarrow 0$
              $s_i \leftarrow 0$
          end for
      end while
      return $memberships,\ centres$
  end function
```

3.1.2 Parallel implementation using Rayon

To distribute the work among threads, we partition the dataset in chunks containing a roughly equal number of points. More precisely, we divide the total number of points n by the number of workers w, rounded up. This gives us the workload per worker $l = \lceil \frac{n}{w} \rceil$. If n is not divisible by l, the size last worker's chunk is $l_{last} = n \mod l$. These chunks are processed in parallel by keeping track of the chunk's contribution to the new clusters' sizes and coordinates in each thread. After each iteration, we synchronize the threads to combine the contributions together and update the cluster centres on the main thread before proceeding to the next iteration.

3.1.3 Distributed implementation using MPI

When using MPI, the distribution of work is done in a similar way. As before, the dataset is partitioned in chunks of roughly equal size, which are sent to their respective MPI ranks. The root rank also initialises and broadcasts the initial cluster centres to all ranks. Each rank then computes the contribution of its share of work to the new cluster centres, after which we call *MPI_Reduce* to merge the individual contributions into the root rank. The root rank computes the new cluster centres and broadcasts them to all ranks, allowing the next iteration to start.

3.1.4 Hybrid implementation using Rayon+MPI

The hybrid implementation, as its name suggests, combines the two previously mentioned approaches together. The main difference is that the dataset is now partitioned twice. The chunks are first distributed among the MPI ranks, which each subdivide their chunk to process it in parallel using Rayon.

3.2 Mandelbrot

Our sequential implementation for computing the Mandelbrot set is based on John Burkardt's implementation[3]. For each pixel in the image, we compute the number of iterations required for the series $z_n = z_{n-1}^2 + c$ with $z_1 = 0$ to diverge, where $c = x + iy$ describes the pixel's corresponding position in the complex plane. If the series has not diverged after a given number of iterations, we colour the pixel black, else we colour it based on the number of iterations it required to reach the divergence criterium and the final value of z_n.

The main difference to the reference implementation lies in how we colour the set. The reference implementation uses a two-tone colouring, while we use a cyclic, continuous colour space instead. To avoid the occurrence of aliasing artifacts in the resulting image (refer to Figure 2.3 in Section 2.3.2), we define the criterium for divergence as when the series' absolute value exceeds the radius $r \geq 1'000$, instead of the traditional $r \geq 2$. The pseudocode for the sequential implementation is provided in Algorithm 2 and 3. In the following sections, we discuss the parallelisation of Algorithm 3.

3.2.1 Parallel implementation using Rayon

Since each pixel's colour can be determined independently of the other pixels, the complex plane can be partitioned in an arbitrary manner. In our case, we split the space into vertical bands of roughly equal width (see Figure 3.1), which are then distributed among the threads.

[3]https://people.sc.fsu.edu/~jburkardt/c_src/mandelbrot/mandelbrot.html, accessed May 28, 2025

3.2.2 Distributed implementation using MPI

The same principle applies when distributing the computation with MPI. We again split the complex plane into vertical bands of roughly equal width and each rank computes the pixels contained within one such band. After the computation is complete, we call *MPI_Gatherv* to collect the partial images from all ranks into the root.

Algorithm 2 Pseudocode for assigning a smooth colour to a pixel based on the number of iteration performed until the divergence criterion has been reached. *colours* is a precomputed array of 8-bit RGB colours.

```
function GET_COLOUR($l$, $z$, $iters_{max}$)
    if $l \geq iters_{max}$ then
        return $(0, 0, 0)$
    else
        $smoothed \leftarrow \log_2(\frac{\log_2 |z|^2}{2})$
        $colour\_index \leftarrow (\sqrt{l + 10 - smoothed} \cdot 256) \mod n\_colors$
        return $colours[colour\_index]$
    end if
end function
```

Algorithm 3 Pseudocode for computing an image of size $n \times n$ pixels of the Mandelbrot set given the maximum number of iterations $iters_{max}$, the divergence threshold radius r, and the portion of the complex plane depicted in the image is delimited by x_{max}, x_{min}, y_{max}, y_{min}.

```
Require: $n > 0$, $iters_{max} > 0$, $r \geq 2$, $x_{max}$, $x_{min}$, $y_{max}$, $y_{min}$
    function MANDELBROT($p$, $iters_{max}$, $r$)
        $pixels \leftarrow [p_1, \ldots, p_{n^2}]$, $p_k = (R,\ G,\ B)$ where $R, G, B \in [0, 255]$
        for $0 \leq j < n$ do
            $x \leftarrow (j \cdot x_{max} + (n - j - 1) \cdot x_{min}) \div (n - 1)$
            for $0 \leq k < p$ do
                $y \leftarrow (k \cdot y_{max} + (n - k - 1) \cdot y_{min}) \div (n - 1)$
                $c \leftarrow x + i \cdot y$
                $z = 0$
                for $0 \leq l < iters_{max}$ do
                    $z \leftarrow z^2 + c$
                    if $|z| > r$ then
                        break
                    end if
                end for
                $pixels[j \cdot n + k] \leftarrow get\_colour(l,\ z,\ iters_{max})$
            end for
        end for
        return $pixels$
    end function
```

3.2.3 Hybrid implementation using Rayon+MPI

The hybrid implementation combines the two previous approaches together. However, we split the space a second time inside the ranks, but this time horizontally. This partitioning is illustrated on the right in Figure 3.1.

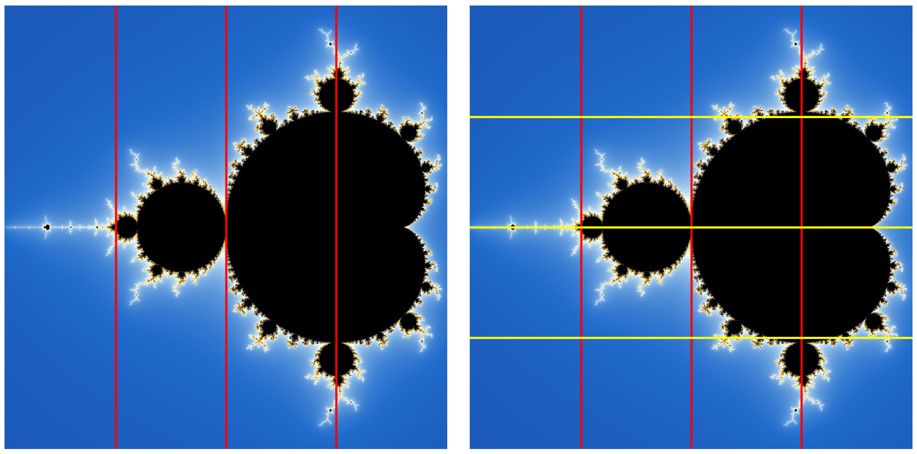


Figure 3.1: Depiction of how we distribute the Mandelbrot set among workers. On the left, we divide the set in vertical bands. On the right, we show our partitioning scheme for when we use both MPI and Rayon together. First we distribute the vertical bands among the MPI ranks (red) and each band is then divided horizontally for Rayon (yellow).

Chapter 4

Results and Discussion

We first discuss how we measure our experiments and what metrics we use for analysis. In our experiments, we collect the execution time in two different forms, shown in Listing 4.1. First is the effective computing time, which includes workload distribution among workers, the computations, and the aggregation of results. The second is the end-to-end execution time, which encompasses the computing time, file I/O, library initialisation, and memory allocation. We do not include the time required to parse command line arguments in the end-to-end execution time. For k-means clustering, we also collect the number of iterations performed. We use this data to analyse the strong and weak scaling of the implemented kernels. The specific details are listed in Table 4.1.

Listing 4.1: Definition of what our various time measurements include

```
Begin of program
    Argument parsing
    Memory allocation                  <------------  Begin of end-to-end time
    Library initialisation (Rayon and/or MPI)
    Read input file (k-means only)
    Split work among workers           <------------  Begin of computing time
    Computations
    Collection/reduction of results    <------------  End of computing time
    Write output file
End of program                         <------------  End of end-to-end time
```

We run our experiments on miniHPC [35], which is a small HPC cluster at the University of Basel (refer to Table 4.1 for details). To control the environment, we set an environment variable to limit the number of threads that Rayon creates and we force MPI to place at most one process per socket, as shown in Listing 4.2. Regarding Rayon, we are unfortunately not aware of any means to control the mapping of threads to CPU cores. Moreover, Rayon's lack of NUMA-awareness is still an open issue [36].

Listing 4.2: Environment setup and MPI workload placement

```
RAYON_NUM_THREADS=SLURM_CPUS_PER_TAKS
srun --cpu-bind=sockets [...]
```

A further consideration we take is that, due to k-means' sensitivity to the initial cluster centres, we seed the RNG to get the same initial conditions for every run. Finally, the source code of the project, as well as additional material, are available on BitBucket[1] and the compile command we use to build the executable is shown in Listing 4.3. Both building and executing the binary on miniHPC requires that the modules *OpenMPI/2.0.2-GCC-6.3.0-2.27* and *Clang/17.0.6-GCCcore-13.2.0* are loaded.

Listing 4.3: Compile command with flags used to build the executable

```
ml OpenMPI Clang #Loading the required modules
cargo build --release --bin rusthpc
```

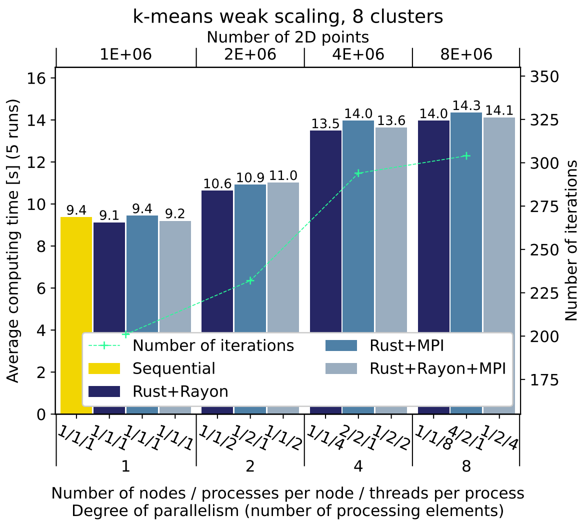
[1] https://bitbucket.org/unibasdmihpc/sylvain-rousselle-msc-project

Table 4.1: Design of factorial experiments

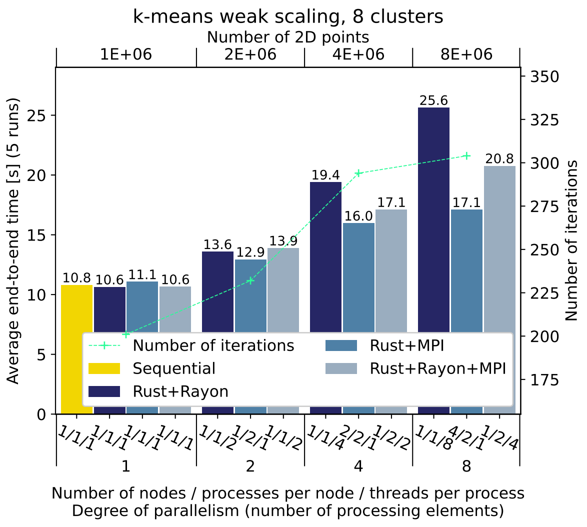
Factor	**Value**	**Properties**
k-means clustering	Strong Scaling	Problem size: 2 millions 2D points
	Weak Scaling	Problem size: 1, 2, 4, 8 millions 2D points
	Number of clusters	$k = 8$
	Termination threshold	$t = 0$
Mandelbrot	Strong Scaling	Dataset size: 5'000x5'000 pixels
	Weak Scaling	Problem size: 5'000x5'000, 7'071x7'071, 10'000x10'000, 14'142x14'142 pixels
	Max iterations	1'000
Tools	Programming language	Rust version 1.85.1
	Shared memory library	Rayon version 1.10.0
	Message-passing library	rsmpi version 0.8.0 links to OpenMPI 2.0.2
Degree of Parallelism	Rust	Sequential (1 thread)
	Rust+Rayon	1, 2, 4, 8 threads
	Rust+MPI	1, 2, 4, 8 ranks, 2 ranks per node
	Rust+Rayon+MPI	1, 2 ranks 1, 2, 4 threads per rank
Metrics	Execution time [s]	Computing time, end-to-end time
Validity	Repetitions	5
Computing nodes	miniHPC-Broadwell	2x Intel Xeon E5-2640 v4 10 cores each, 2.4 GHz 25 MB L3 cache, 64 GB RAM
Network	Node interconnectivity	100Gbit/s Intel Omni-Path links

4.1 k-means clustering

As shown in Figure 4.1 the weak scaling performance of k-means is consistent across the tested parallelisation methods, with the exception of all versions using Rayon displaying a higher overhead in end-to-end measurements (Figures 4.1b, 4.2b, and 4.4b). Since only the end-to-end time is affected, this limits the possible causes to either file I/O, memory allocation, or library initialisation. File I/O can be ruled out, as all versions perform this sequentially. Finally, we can see that the MPI+Rayon hybrid version's overhead is only roughly half as much as that of the Rayon version. This correlates directly with the number of threads created by Rayon: we only see a difference in time in the two right-most groups of columns, which are also the two groups where the Rayon version creates twice as many threads per process as the hybrid version. Thus, we suspect that this phenomenon is due to the library initialisation of Rayon, but are not able to confirm it.



(a) Computing time

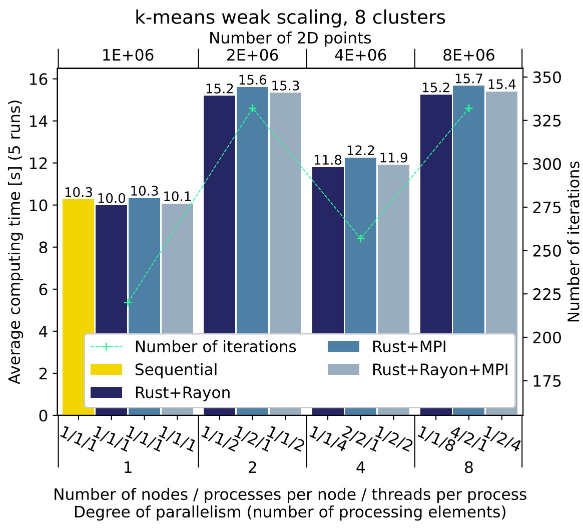


(b) End-to-end execution time

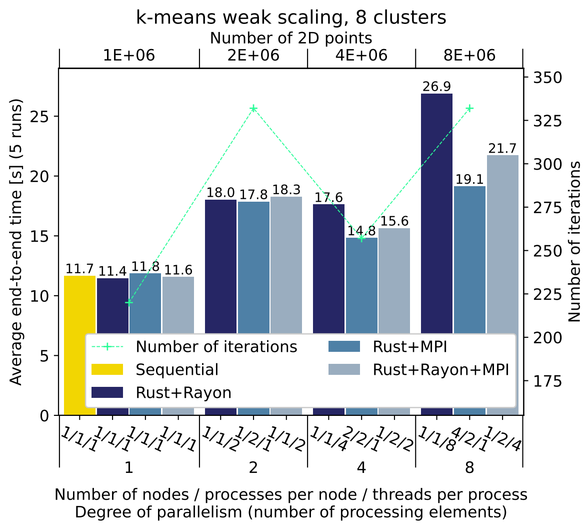
Figure 4.1: Weak scaling performance (averaged over 5 runs) of k-means clustering. There are 4 column groups, each with its own dataset size (top x-axis) and degree of parallelism (lower x-axis). The parallelism is characterised by a triple *nodes/processes per node/threads per process*. The total degree of parallelism is included as a single number below the parallelism triples. We also report the number of iterations performed before reaching convergence for each column group on the right-hand y-axis.

Additionally, we have generated a second set of datasets to analyse the influence of the dataset on performance. To this end, we have repeated the weak scaling experiment on the second set of datasets. As can be seen in Figures 4.2, the execution time of k-means unexpectedly seems disconnected from the dataset size. This is reflected by the number of iterations required to reach convergence, which is a property of the dataset. On the 4 million points dataset, k-means reached convergence after only 257 iterations, in contrast to the 332 iterations performed on the 2 million points dataset.

We have generated this second set of datasets in the same manner as the first one (see Section 3.1). To further ensure that both sets of datasets are comparable, we compute the dissimilarity of the datasets using the Maximum Mean Discrepancy (MMD) [37] and the Earth Mover's Distance (EMD) [38] metrics at each dataset size. MMD is sensitive to global structure and higher-order statistical differences between distributions, whereas EMD measures the cost of transforming one point cloud's distribution into the other and is sensitive to spatial shifts and rearrangements. While EMD produces values in the range $[0, \sqrt{2}]$ on the space $[0, 1]^2$, MMD has a lower bound of 0, but no upper bound. As shown in Figure 4.3, both metrics return low values across the board, indicating that the datasets are similar.

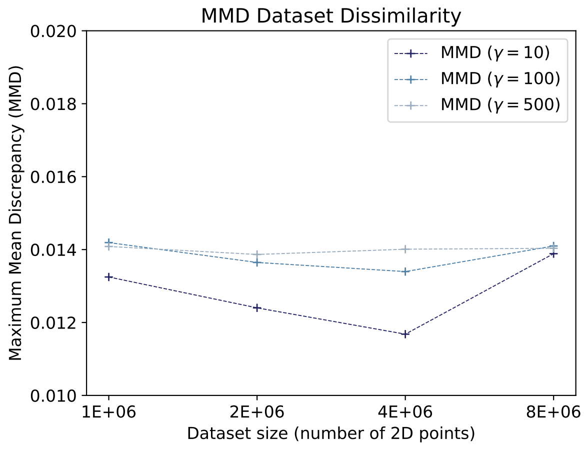


(a) Computing time

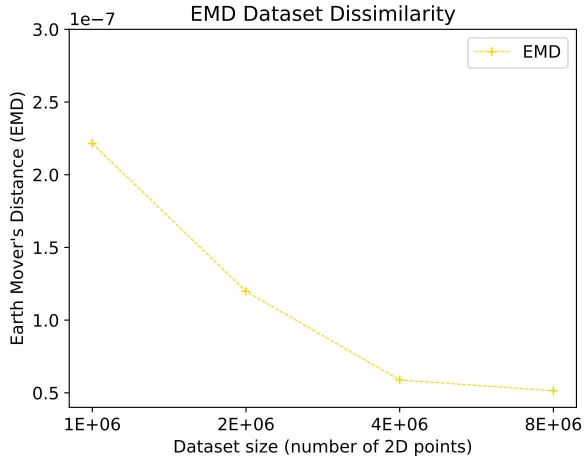


(b) End-to-end execution time

Figure 4.2: Weak scaling performance (averaged over 5 runs) of k-means clustering on a second set of datasets. There are 4 column groups, each with its own dataset size (top x-axis) and degree of parallelism (lower x-axis). The parallelism is characterised by a triple *nodes/processes per node/threads per process*. The total degree of parallelism is also included as a single number below the parallelism triples. We also report the number of iterations performed before reaching convergence for each column group on the right-hand y-axis.



(a) Maximum Mean Discrepancy (MMD)



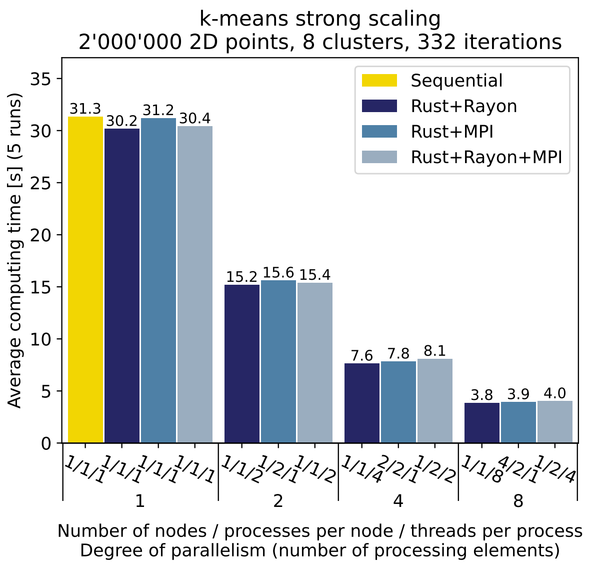
(b) Earth Mover's Distance (EMD)

Figure 4.3: MMD (left) and EMD (right) values between our two sets of datasets at each dataset size. The overall low values indicate that the two datasets are similar. Note the difference in scale between Figure 4.3a and Figure 4.3b.

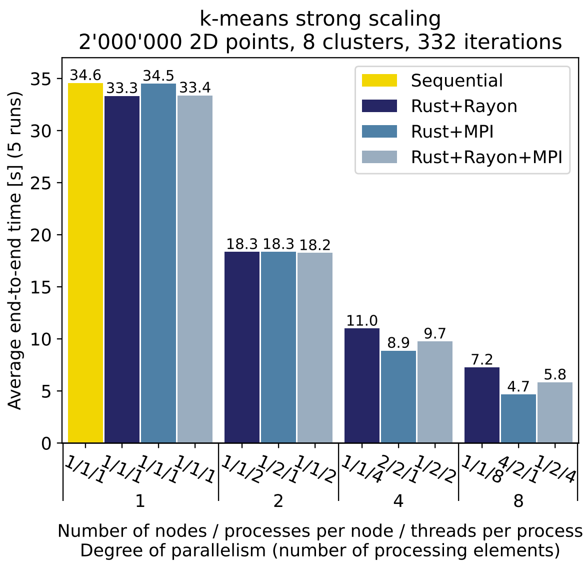
We thus conclude that the cause for the better performance on the 4 million points dataset of the second experiment is due to an advantageous distribution of the points within the dataset, which leads to faster convergence. The same effect can be observed in the end-to-end time measurements shown in Figure 4.2b, although not as pronounced as in Figure 4.2a. The dependence of k-means' performance not only on the initial cluster centres' positions, but also on the chosen dataset raises questions about the usefulness of

k-means as a reliable, general benchmark tool.

When measuring the strong scaling, the influence of the dataset on the performance is less of an issue as it is kept constant across all runs. In Figure 4.4a, we can see that the execution time almost halves each time the computing resources are doubled (e.g. 30s - 31s with one processing element vs. approx. 15.5s with two processing elements). While the trend is the same in Figure 4.4b, the performance gains are lower, which is expected as the I/O operations to read in the dataset and writing the results to disk cannot be parallelized.



(a) Computing time



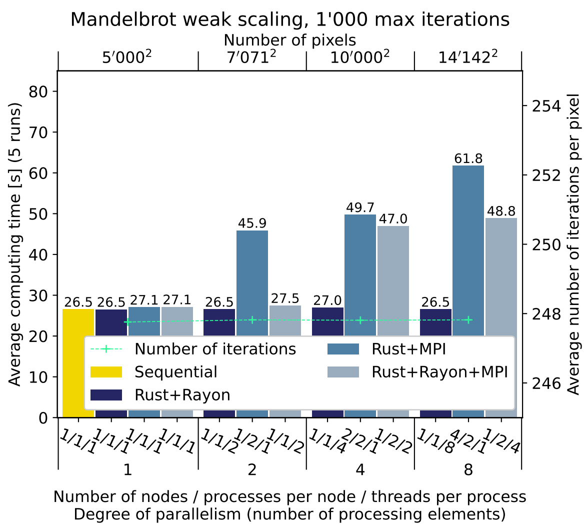
(b) End-to-end time

Figure 4.4: Strong scaling performance of k-means clustering on the 2 million points dataset, averaged over 5 runs. There are 4 column groups, each with its own degree of parallelism. The parallelism is characterised by a triple *nodes/processes per node/threads per process*. The total degree of parallelism is also included as a single number below the parallelism triples.

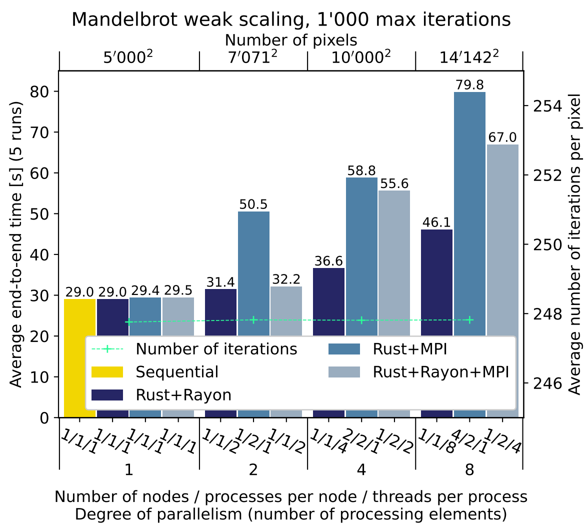
4.2 Mandelbrot

In the weak and strong scaling experiments for Mandelbrot, shown if Figures 4.5 and 4.6, all versions using MPI show lower performance gains than expected when increasing the number of MPI ranks. This is caused by our vertical partitioning of the complex plane (recall Figure 3.1). The pixels coloured in black all reach the given maximum iteration threshold and the majority of black pixels are concentrated on the right side of the picture. The other colours indicate points that diverge before reaching the iteration threshold and are as such less compute-intensive. Thus, our partitioning scheme causes high load imbalance, which could be improved upon by switching to a horizontal partitioning instead to reduce the load imbalance.

The keen reader will have noticed that, despite using the same partitioning scheme, the version parallelized with Rayon does not suffer from such performance losses. We impute this phenomenon to Rayon's work-stealing scheduling (refer to Section 2.1.2), which is able to compensate for the load imbalance.

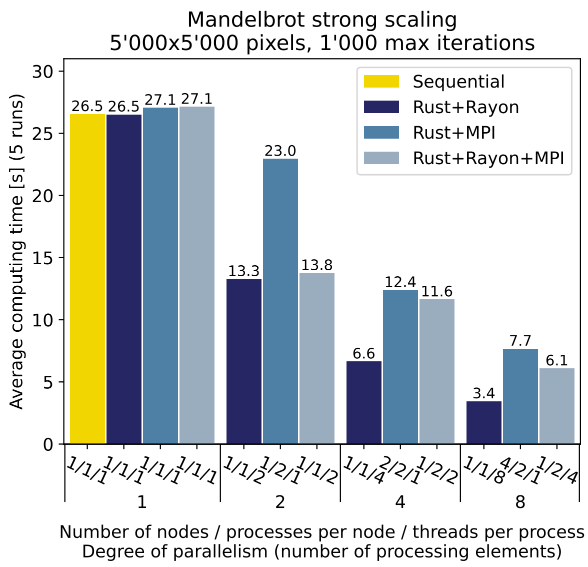


(a) Computing time

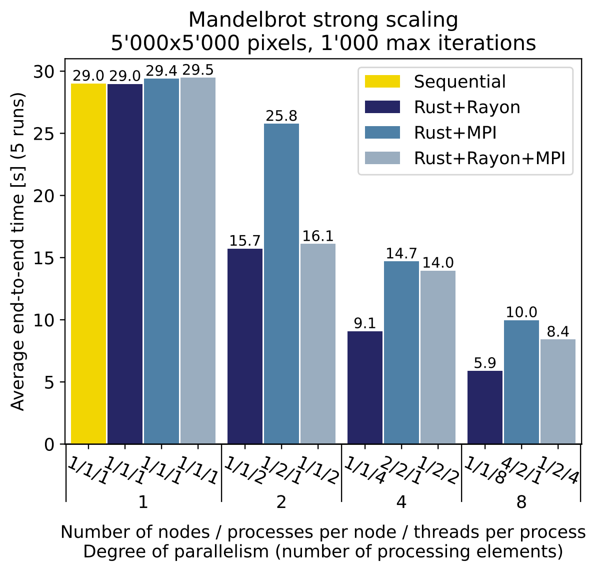


(b) End-to-end time

Figure 4.5: Weak scaling performance of Mandelbrot, averaged over 5 runs. For each column group, we report the size of the computed image and the degree of parallelism. The parallelism is characterised by a triple nodes/processes per node/threads per process. The total degree of parallelism is included as a single number below the parallelism triples.



(a) Computing time



(b) End-to-end time

Figure 4.6: Strong scaling performance of Mandelbrot, averaged over 5 runs. For each column group, we report the size of the computed image and the degree of parallelism. The parallelism is characterised by a triple nodes/processes per node/threads per process. The total degree of parallelism is included as a single number below the parallelism triples.

Chapter 5

Conclusion

What started as a comparison of performance and productivity between Rust and other programming languages, based on the implementation of graph algorithms from the GAP benchmark suite, ended up with completely different goals due to limitations in Rust's sparse linear algebra ecosystem.

In accordance with our revised goals, we have evaluated k-means clustering and the Mandelbrot set, as both workloads lend themselves well to parallelisation yet do not require sparse linear algebra. In our analysis of the Mandelbrot set, we found that the partitioning scheme is crucial in managing load-imbalance, which can be mitigated with more dynamic types of scheduling such as Rayon's work-stealing. Regarding k-means clustering, we found its performance outcomes to be consistent across the tested parallelisation methods. However, we also notice that k-means clustering's performance is heavily dependent not only on the initialisation of the cluster centres but also on the dataset used. This finding challenges the algorithm's suitability as a general benchmark tool, as the dependency on the dataset weakens reproducibility and comparability of results. We also observed a non-negligible overhead in the end-to-end performance when using Rayon, which we suspect is due to the initialisation of its thread pool.

Overall, Rust shows potential for HPC applications, especially in reducing the number of concurrency-related bugs. However, with its non-standard MPI syntax, immature sparse linear algebra ecosystem, and different approach to shared memory parallelism, we found that Rust is not yet mature enough to be considered an HPC-ready programming language.

Use of AI tools

For this report, I used DeepL [39] to translate single words and short partial sentences. I also sparingly used ChatGPT [40] to point out what could be improved upon. The prompt template is shown in Listing 5.1. No AI tools were used to generate new content. I have checked all the texts and take full responsibility for the result.

Listing 5.1: ChatGPT prompt for pointing out passages that can be improved upon

```
Is this good as a part of the [section name] in a paper? Do not
rephrase or modify the text, only point out what can be improved upon.

[text goes here]
```

Bibliography

[1] A. Claster, "'why we created julia' turns ten years old." [Online]. Available: https://info.juliahub.com/blog/why-we-created-julia-turns-ten-years-old, accessed July 13, 2025.

[2] T. J. P. Language, "Multi-processing and distributed computing." [Online]. Available: https://docs.julialang.org/en/v1/manual/distributed-computing/, accessed July 13, 2025.

[3] Q. Guilloteau, J. H. M. Korndörfer, and F. M. Ciorba, "Seamlessly scaling applications with daphne," in *COMPAS 2024-Conférence francophone d'informatique en Parallélisme, Architecture et Système*, 2024. Available: https://hal.science/hal-04637841.

[4] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite," *arXiv preprint arXiv:1508.03619*, 2015. doi: 10.48550/arXiv.1508.03619.

[5] DAPHNE, "Daphne." [Online]. Available: https://daphne-eu.eu/, accessed May 28, 2025.

[6] Wikipedia, "Rust (programming language)." [Online]. Available: https://en.wikipedia.org/wiki/Rust_(programming_language), accessed April 28, 2025.

[7] T. R. Team, "Rust programming language." [Online]. Available: https://www.rust-lang.org/, accessed June 15, 2025.

[8] S. Klabnik and C. Nichols, *The Rust programming language*. No Starch Press, 2023.

[9] J. Stone and N. Matsakis, "Rayon," 2015. [Online]. Available: https://crates.io/crates/rayon, accessed May 6, 2025.

[10] J. Stone and N. Matsakis, "Rayon faq," 2024. [Online]. Available: https://github.com/rayon-rs/rayon/blob/main/FAQ.md, accessed July 1, 2025.

[11] M. Forum, "Mpi." [Online]. Available: https://www.mpi-forum.org/, accessed June 15, 2025.

[12] A. G. Benedikt Steinbush and J. Brown, "Mpi bindings for rust," 2015. [Online]. Available: https://crates.io/crates/mpi, accessed May 6, 2025.

[13] Wikipedia, "Gecko (software)." [Online]. Available: https://en.wikipedia.org/wiki/Gecko_(software), accessed June 15, 2025.

[14] Wikipedia, "Rust for linux." [Online]. Available: https://en.wikipedia.org/wiki/Rust_for_Linux, accessed April 28, 2025.

[15] A. W. Services, "Firecracker." [Online]. Available: https://firecracker-microvm.github.io/, accessed June 15, 2025.

[16] advancedwebdeveloper, "Cisco, rust and zeromq." [Online]. Available: https://users.rust-lang.org/t/cisco-rust-and-zeromq/14397, accessed June 15, 2025.

[17] S. D. Simone, "Npm adopted rust to remove performance bottlenecks." [Online]. Available: https://www.infoq.com/news/2019/03/rust-npm-performance/, accessed June 15, 2025.

[18] S. Crozet and E. Bopp, "nalgebra," 2015. [Online]. Available: https://crates.io/crates/nalgebra, accessed May 6, 2025.

[19] S. Crozet and E. Bopp, "Current state of nalgebra," Year undefined. [Online]. Available: https://docs.rs/nalgebra-sparse/latest/nalgebra_sparse/#current-state, accessed April 29, 2025.

[20] S. Quiñones, "faer," 2022. [Online]. Available: https://crates.io/crates/faer, accessed May 6, 2025.

[21] V. Barielle and M. Ulimoen, "sprs," 2015. [Online]. Available: https://crates.io/crates/sprs, accessed May 6, 2025.

[22] I. Ukhov, "cblas," 2017. [Online]. Available: https://crates.io/crates/cblas, accessed May 6, 2025.

[23] Wikipedia, "Basic linear algebra subprograms." [Online]. Available: https://en.wikipedia.org/wiki/Basic_Linear_Algebra_Subprograms, accessed April 28, 2025.

[24] OpenMathLib, "OpenBLAS," 2011. [Online]. Available: http://www.openmathlib.org/OpenBLAS/docs/, accessed 29.04.2025.

[25] OpenMathLib, "Sparse linear algebra support in openblas," Year undefined. [Online]. Available: http://www.openmathlib.org/OpenBLAS/docs/faq/#does-openblas-support-sparse-matrices-andor-vectors, accessed April 29, 2025.

[26] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE international symposium on workload characterization (IISWC)*, pp. 44–54, Ieee, 2009. doi: 10.1109/IISWC.2009.5306797.

[27] Wolfram, "K-means clustering algorithm." [Online]. Available: https://mathworld.wolfram.com/K-MeansClusteringAlgorithm.html, accessed June 19, 2025.

[28] Wolfram, "Geometric centroid." [Online]. Available: https://mathworld.wolfram.com/GeometricCentroid.html, accessed April 29, 2025.

[29] D. Aloise, A. Deshpande, P. Hansen, and P. Popat, “Np-hardness of euclidean sum-of-squares clustering,” *Machine learning*, vol. 75, pp. 245–248, 2009.

[30] M. Mahajan, P. Nimbhorkar, and K. Varadarajan, “The planar k-means problem is np-hard,” *Theoretical computer science*, vol. 442, pp. 13–21, 2012.

[31] S. Lloyd, “Least squares quantization in pcm,” *IEEE transactions on information theory*, vol. 28, no. 2, pp. 129–137, 1982.

[32] F. Garcia, A. Fernandez, J. Barrallo, and L. Martin, “Coloring dynamical systems in the complex plane,” *The University of the Basque Country, Plaza de O˜ nati*, vol. 2, 2009.

[33] L. Vepstas, “Renormalizing the mandelbrot escape,” 1997.

[34] D. J. Bernstein *et al.*, “Chacha, a variant of salsa20,” in *Workshop record of SASC*, vol. 8, pp. 3–5, Citeseer, 2008.

[35] HPC Group at University of Basel, “miniHPC.” [Online]. Available: https://hpc.dmi.unibas.ch/research/minihpc/, accessed June 2, 2025.

[36] J. Stone and N. Matsakis, “Rayon: Scheduling should be topology/numa aware,” 2017. [Online]. Available: https://github.com/rayon-rs/rayon/issues/319, accessed June 2, 2025.

[37] L. Song, “Learning via hilbert space embedding of distributions,” *University of Sydney (2008)*, vol. 17, 2008.

[38] Y. Rubner, C. Tomasi, and L. J. Guibas, “A metric for distributions with applications to image databases,” in *Sixth international conference on computer vision (IEEE Cat. No. 98CH36271)*, pp. 59–66, IEEE, 1998.

[39] DeepL SE, “Deepl,” 2017. [Online]. Available: https://deepl.com, accessed May 6, 2025.

[40] OpenAI, “Chatgpt,” 2022. [Online]. Available: https://chatgpt.com/, accessed July 2, 2025.

University of Basel

Faculty of Science

Declaration on Scientific Integrity

(including a Declaration on Plagiarism and Fraud)
Translation from German original

Title of Thesis:

Name Assessor:	Florina M. Ciorba
Name Student:	Sylvain Rousselle
Matriculation No.:	20-050-936

I attest with my signature that I have written this work independently and without outside help. I also attest that the information concerning the sources used in this work is true and complete in every respect. All sources that have been quoted or paraphrased have been marked accordingly.

Additionally, I affirm that any text passages written with the help of AI-supported technology are marked as such, including a reference to the AI-supported program used. This paper may be checked for plagiarism and use of AI-supported technology using the appropriate software. I understand that unethical conduct may lead to a grade of 1 or “fail” or expulsion from the study program.

Place, Date: Eiken, 23.07.2025 Student: S. Rousselle

Will this work, or parts of it, be published?

☐ No

☒ Yes. With my signature I confirm that I agree to a publication of the work (print/digital) in the library, on the research database of the University of Basel and/or on the document server of the department. Likewise, I agree to the bibliographic reference in the catalog SLSP (Swiss Library Service Platform). (cross out as applicable)

Publication as of: __________

Place, Date: Eiken, 23.07.2025 Student: S. Rousselle

Place, Date: Basel, 25.07.2025 Assessor: [illegible]

Please enclose a completed and signed copy of this declaration in your Bachelor’s or Master’s thesis.

September 2023