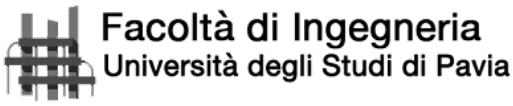

Advanced Computer Architecture

Genomics and proteomics substring matching: a parallel study on Boyer-Moore with hash functions

Marco Bernazzani (562312)
Michele Di Frisco Ramirez (557426)

Course Instructors: Marco Ferretti, Luigi Santangelo

July 1, 2025

Contents

1 Introduction **3**

2 Boyer-Moore Algorithm **3**

3 Hash Functions **4**

4 Performance of the Serial Algorithm and A-priori Study of Parallelism **5**

5 MPI Strategy and Implementation **6**

6 Cluster Results **7**

7 Scalability Study **10**

8 Hash Collision Study **11**

9 Conclusion **12**

10 Individual Contributions **12**

11 References **12**

1 Introduction

This project explores substring matching in genomic sequences using the Boyer-Moore algorithm enhanced with various hash functions. We assess the performance and scalability of our MPI-based parallel implementation across multiple Google Cloud Platform (GCP) clusters.
All source code and raw experimental results are available at:
https://github.com/MrOverflow99/ACA—MPI_Ricerca_Genoma-

2 Boyer-Moore Algorithm

The Boyer-Moore algorithm is one of the most efficient string-searching algorithms, particularly effective when the pattern is much shorter than the text. It compares characters from right to left and, by preprocessing the pattern, it uses heuristics to skip sections of the text, achieving sublinear time in many practical scenarios.

The algorithm is based on two main heuristics:

- **Bad Character Heuristic**: When a mismatch occurs, the algorithm shifts the pattern so that the mismatched character in the text aligns with its last occurrence in the pattern. If the character is not present in the pattern, the pattern is shifted beyond the mismatched character entirely.
- **Good Suffix Heuristic**: When a mismatch occurs after some suffix of the pattern has matched, the pattern is shifted so that another occurrence of this suffix is aligned with the text. If the suffix does not appear again in the pattern, the shift is made past the entire matched suffix.

These heuristics make the Boyer-Moore algorithm particularly efficient, often skipping large portions of text and reducing unnecessary comparisons.

Illustrative Example

Figure 1: Boyer-Moore pattern search example (source: GeeksforGeeks)

In this example, we are searching for the pattern `AABA` in the text `AABAACAADAABAABA`.

- Initially, the pattern is aligned at index 0. It fully matches the text from position 0 to 3.
- Then, the pattern is shifted using the bad character and good suffix rules to skip unnecessary comparisons.
- A second match is found at index 9, and a third at index 12.

This illustrates the power of the algorithm: instead of checking each position one by one, Boyer-Moore makes intelligent jumps, drastically reducing the number of character comparisons.

Time Complexity

- **Preprocessing Time:** $\mathcal{O}(m + \sigma)$ for the bad character table, and $\mathcal{O}(m)$ for the good suffix table (where m is the length of the pattern and σ is the alphabet size).
- **Search Time:** Best case is sublinear (often $\mathcal{O}(n/m)$), average case is efficient in practice, and worst case is $\mathcal{O}(n \cdot m)$.

Because of its efficiency and robustness in real-world scenarios, Boyer-Moore is widely used in applications like search engines and DNA sequence analysis. It serves as the core of our implementation for genomic substring matching.

3 Hash Functions

We integrated 11 different hash functions into our Boyer-Moore implementation to evaluate their impact on performance and collision behavior. Among them, the most representative in terms of behavior and distribution characteristics are the following:

- **FNV-1a 64-bit**: This hash processes each byte by XORing it with the current hash value and then multiplying by a prime number. It ensures a good distribution with minimal collisions and is efficient for fixed-length substrings.
- **MurmurHash2**: It performs a sequence of multiplications, bit shifts, and XORs on blocks of input to quickly produce highly mixed outputs. This helps reduce patterns and improve distribution in substring hashing.
- **PolyHash**: Computes the hash of a string by treating it as a polynomial, where each character is a coefficient and a fixed base is used as the variable. It enables rolling hash capabilities, useful for overlapping substrings.
- **DJB2**: This function iteratively multiplies the hash by 33 and adds the ASCII value of each character. Its simplicity makes it fast, and its pattern-independent behavior makes it decent for simple substring hashing.
- **AddShift-hash**: A minimalistic hash that adds the character value and left-shifts the result at each step. It allows quick computation but suffers from more frequent collisions.

For the experiments and performance analysis in this report, we focused on the functions listed above. The remaining functions implemented were FNV-1a (32-bit), xxHash32, and CRC32. Additionally, we designed two basic hash functions — `xor_h` and `better_xor` — to evaluate their behavior in terms of collisions, which is discussed in detail in the final section of this document. All the hash functions used in this project are declared in `hash_funct.h` and implemented in `hash_funct.c`.

4 Performance of the Serial Algorithm and A-priori Study of Parallelism

To estimate the degree of parallelism achievable, we measured the time taken by I/O and by the core algorithm. Four serial executions were performed on a ∼2.8GB genome file:

- File read time: 2.108742s, 1.624497s, 1.497589s and 1.642331s.
- Function execution time: 10.871110s, 10.308316s, 10.161705s and 10.450701s

Averaging these runs, file I/O takes ∼1.718s while Boyer-Moore function time is ∼10.447s.

Function	% Time	Cumulative sec	Self sec	Calls
`boyer_moore_search`	85.93%	12.157	10.447	1
`readFile`	14.06%	12.157	1.71	1

Table 1: Profiling summary of main functions

Thus, the portion of code we aim to parallelize (i.e., the Boyer-Moore algorithm) constitutes approximately:

$$p = 1 - \frac{1.71}{12.157} \approx 0.859 \tag{1}$$

We estimate that ∼85.9% of the workload is parallelizable.

According to Amdahl's Law:

$$\text{Speedup}(n) = \frac{n}{n + p(1-n)} \tag{2}$$

Fixing $p = 0.859$, the theoretical speedups for n cores can be plotted and compared with empirical performance to assess the efficiency of our parallel implementation.

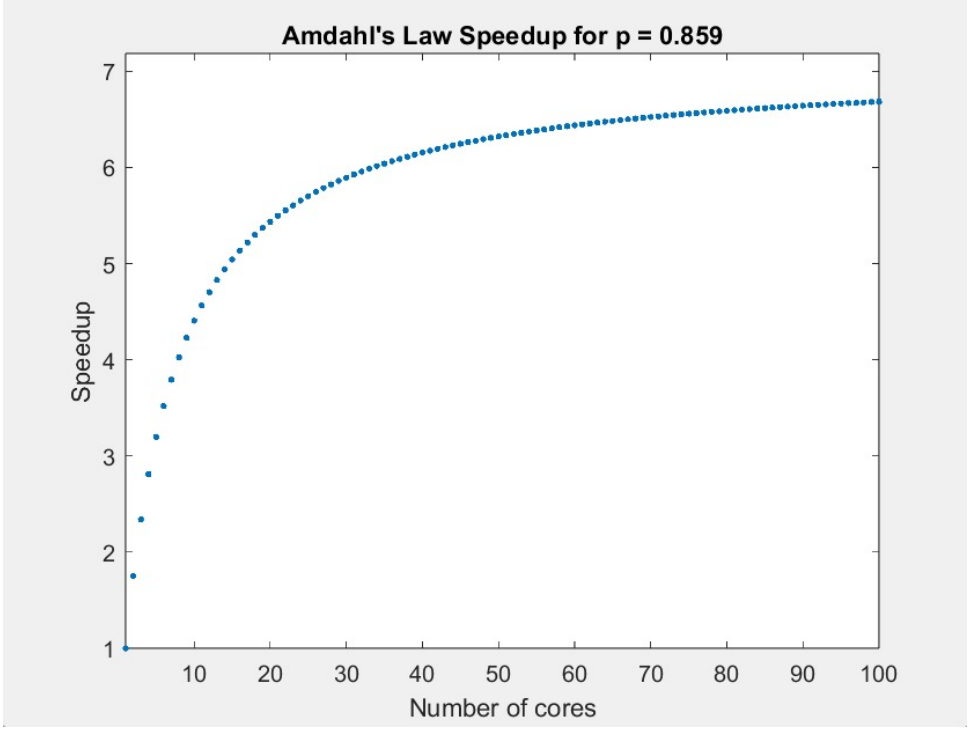

Figure 2: Theoretical speedup curve for $p = 0.859$ according to Amdahl's Law

5 MPI Strategy and Implementation

The strategy adopted to parallelize the code consists in splitting the text among all processes. Each process receives from the master a substring of the original text whose length is given by the following formula:

$$\text{substring length} = \frac{text\_length}{n\_processes} + pattern\_length - 1 \tag{3}$$

Only the last process receives a longer substring given by the next formula:

$$\text{substring length} = \frac{text\_length}{n\_processes} + pattern\_length - 1 + text\_length \mod n\_cores \tag{4}$$

The **master process** is responsible for reading the input text and pattern, and for distributing the necessary data to all slave processes.

Each slave process receives a chunk of the text and the full pattern and proceeds independently with the substring matching using the Boyer-Moore algorithm enhanced with the chosen hash function. After computation, results are sent back to the master, which aggregates them to compute the global number of matches and collisions.

Parallel implementation with Open MPI

Before performing the actual matching, the master process must distribute key information to the other processes. This is done through the following MPI primitives:

- `MPI_Bcast()` is used to broadcast the number of processes, the length of the text, and the length of the pattern to all processes.
- `MPI_Scatter()` distributes the activation flags vector so that each process knows whether it is active or not.

```
66    MPI_Bcast(&executors, 1, MPI_INT, 0, MPI_COMM_WORLD);
67    MPI_Scatter(flag, 1, MPI_INT, &isActive, 1, MPI_INT, 0, MPI_COMM_WORLD);
68    MPI_Bcast(&txtlen, 1, MPI_LONG_LONG, 0, MPI_COMM_WORLD);
69    MPI_Bcast(&patlen, 1, MPI_LONG_LONG, 0, MPI_COMM_WORLD);
70    MPI_Bcast(&choice, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

Figure 3: MPI functions used to communicate between processes

The function `who_is_active()` computes the set of active processes.

Then, the master splits the text into chunks using `split_dataset()` and sends each chunk using `MPI_Send()` to the respective process.

Each slave process calls `receive_dataset()` (which internally uses `MPI_Recv()`) to get its portion of text.

Finally, the occurrences found by each process are reduced back to the master using `MPI_Reduce()`, which collects and sums all the partial results.

6 Cluster Results

We tested different cluster configurations on GCPs Computer engine and measured the execution time. We executed the program on E2 VMs for the Light-Clusters and N2 VMs for the Fat-Cluster. Here are the configurations used:

- **Light Cluster (1 zone, 4 VMs, 4 cores)**
- **Light Multi-Region Cluster (4 zones, 4 VMs, 4 cores)**
- **Light Multi-Region 16-core Cluster(4 zones, 16 VMs, 16 cores)**
- **Fat Cluster (1 zone, 2 VMs, 8 cores each)**

Status	Name ↑	Zone	Recommendations	In use by	Internal IP	External IP
✓	v1-aca	europe-west1-b			10.132.0.3 (nic0)	35.187.7.116 (nic0)
✓	v10-aca	europe-west4-a			10.164.0.4 (nic0)	34.91.117.13 (nic0)
✓	v11-aca	europe-west4-a			10.164.0.5 (nic0)	34.34.108.244 (nic0)
✓	v12-aca	europe-west10-a			10.214.0.4 (nic0)	34.32.36.27 (nic0)
✓	v13-aca	europe-west10-a			10.214.0.5 (nic0)	34.32.86.49 (nic0)
✓	v14-aca	europe-west10-a			10.214.0.6 (nic0)	34.32.91.217 (nic0)
✓	v15-aca	europe-central2-a			10.186.0.4 (nic0)	34.118.98.53 (nic0)
✓	v16-aca	europe-central2-a			10.186.0.5 (nic0)	34.116.194.38 (nic0)
✓	v2-aca	europe-west1-b			10.132.0.4 (nic0)	34.34.145.157 (nic0)
✓	v3-aca	europe-west1-b			10.132.0.7 (nic0)	104.199.81.87 (nic0)
✓	v4-aca	europe-west1-b			10.132.0.6 (nic0)	34.140.53.193 (nic0)
✓	v5-aca	europe-central2-a			10.186.0.2 (nic0)	34.118.29.112 (nic0)
✓	v6-aca	europe-west4-a			10.164.0.2 (nic0)	34.13.141.199 (nic0)
✓	v7-aca	europe-west10-a			10.214.0.3 (nic0)	34.32.112.86 (nic0)
✓	v8-aca	europe-central2-a			10.186.0.3 (nic0)	34.116.191.106 (nic0)
✓	v9-aca	europe-west4-a			10.164.0.3 (nic0)	34.147.81.56 (nic0)

Figure 4: Light-Cluster VMs with v1-aca used as master and the others as slaves

VM instances

Filter Enter property name or value

Status	Name ↑	Zone	Recommendations	In use by	Internal IP	External IP
✓	v17-fat-aca	europe-west1-b			10.132.0.8 (nic0)	104.199.81.87 (nic0)
✓	v18-fat-aca	europe-west1-b			10.132.0.9 (nic0)	35.233.6.30 (nic0)

Figure 5: Fat-Cluster VMs with v17-fat-aca used as master and v18-fat-aca as slave

Time performance for each configuration using selected hash functions is summarized in the corresponding figures. We took, as mentioned before, the most relevant results for the goal of this research, you can find all the results in our GitHub repo.
File analyzed: `sequenza_pulita.txt` ∼2.8GB.
Pattern researched: `'TACCTAA'`

Here are the results:

Light Cluster (1 zone, 4 VMs, 4 cores)

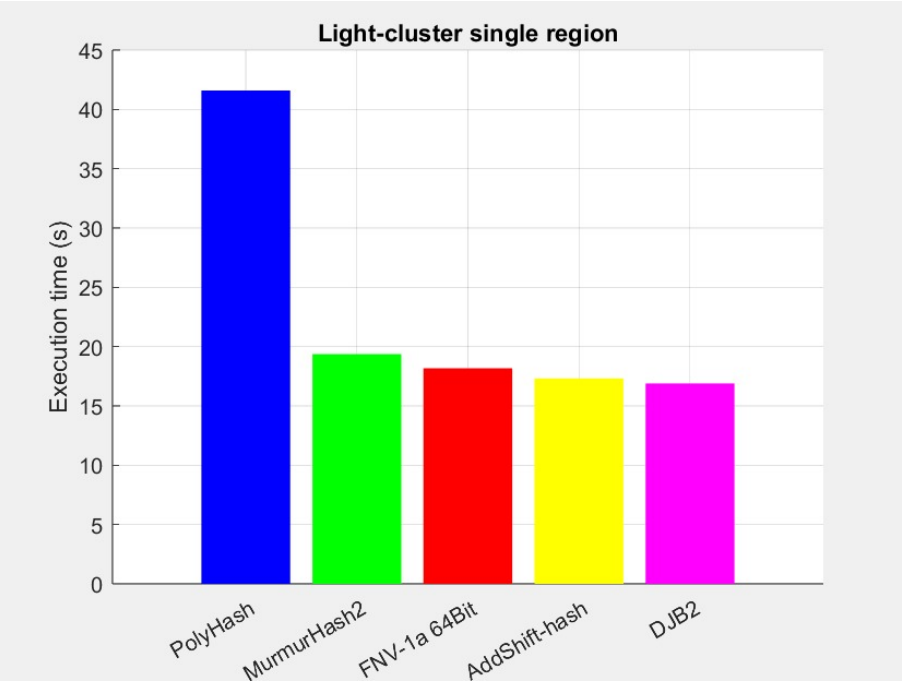

Figure 6: Execution time Light-Cluster single region (europe-west1-b).

Light Multi-Region Cluster (4 zones, 4 VMs, 4 cores)

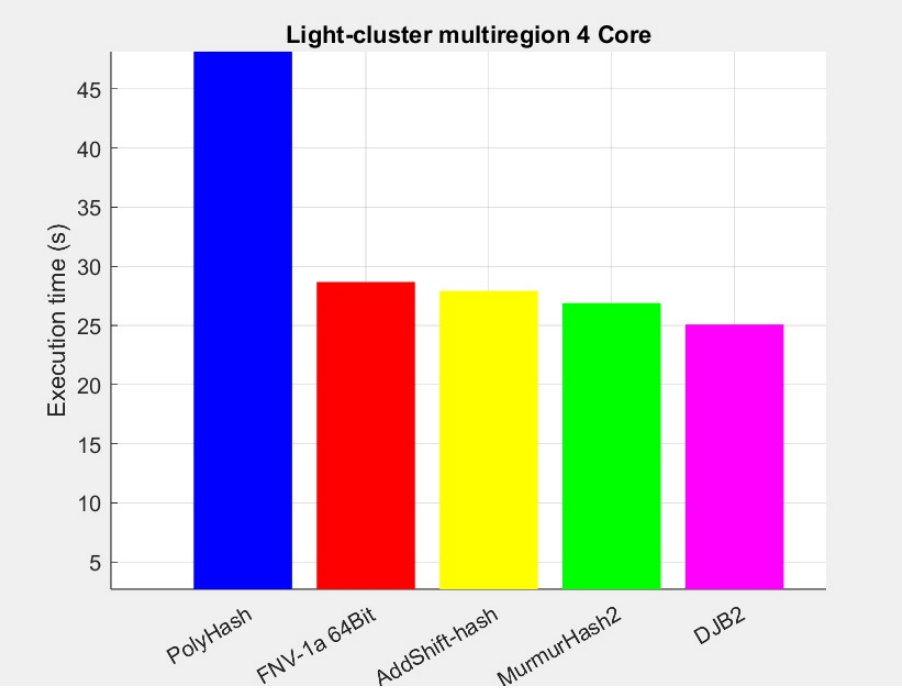

Figure 7: Execution time Light-Cluster multi region (europe-west1-b; europe-central2-a; europe-west10-a; europe-west4-a).

Light Multi-Region 16-core Cluster(4 zones, 16 VMs, 16 cores)

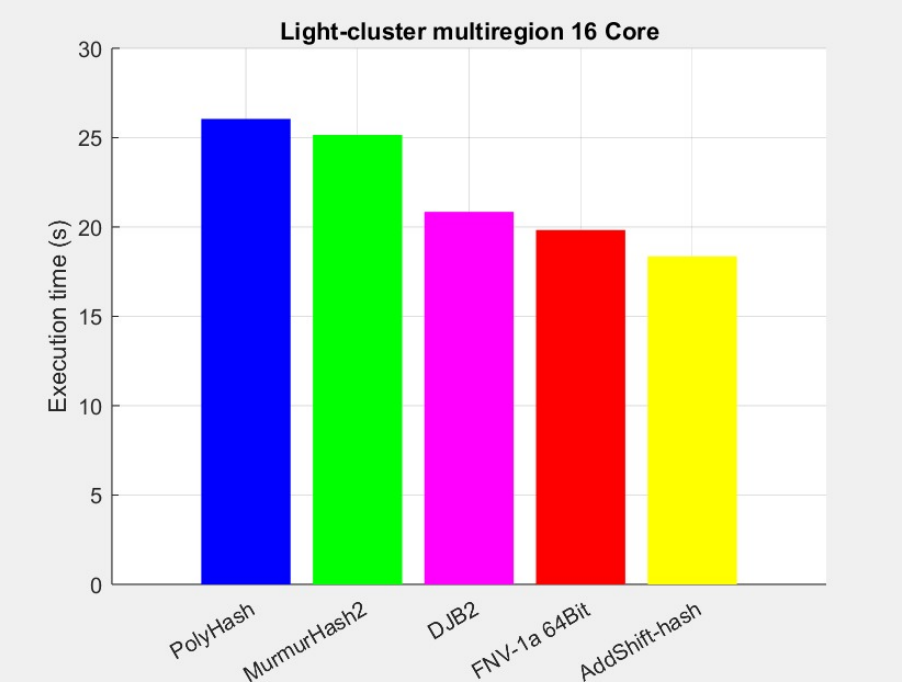

Figure 8: Execution time Light-Cluster multi region (europe-west1-b; europe-central2-a; europe-west10-a; europe-west4-a)

Fat Cluster (1 zone, 2 VMs, 8 cores each)

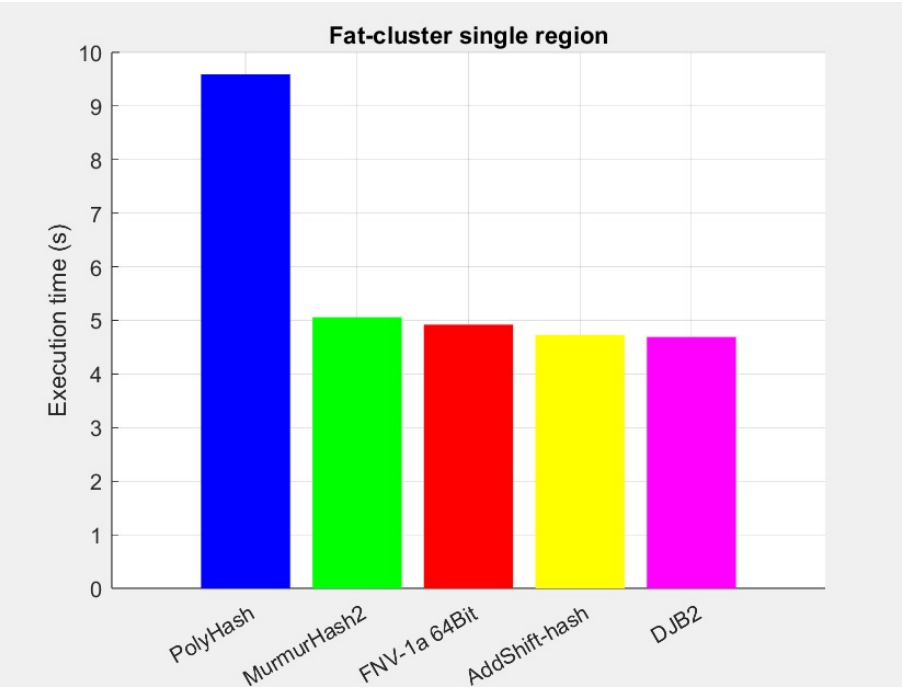

Figure 9: Execution time Fat-Cluster single region (europe-west1-b)

The serial program runs in 12.165 seconds, the best performance on parallel algorithms is of 3.392 seconds. The estimated speedup is ∼3.6 which is very similar to the one proposed by Amdahl's law. (We took in consideration only the execution of `boyer_moore_search` function, so no hash involved.)

7 Scalability Study

- **Weak scalability** We analyzed serial performance on inputs of increasing size (250MB, 500MB, 1GB, 2GB). The results are summarized in Table 2.

Input Size	Time (seconds)	Cores
250MB	0.99	1
500MB	1.26	2
1GB	1.74	4
2GB	2.70	8

Table 2: Serial performance with increasing file size and cores involved

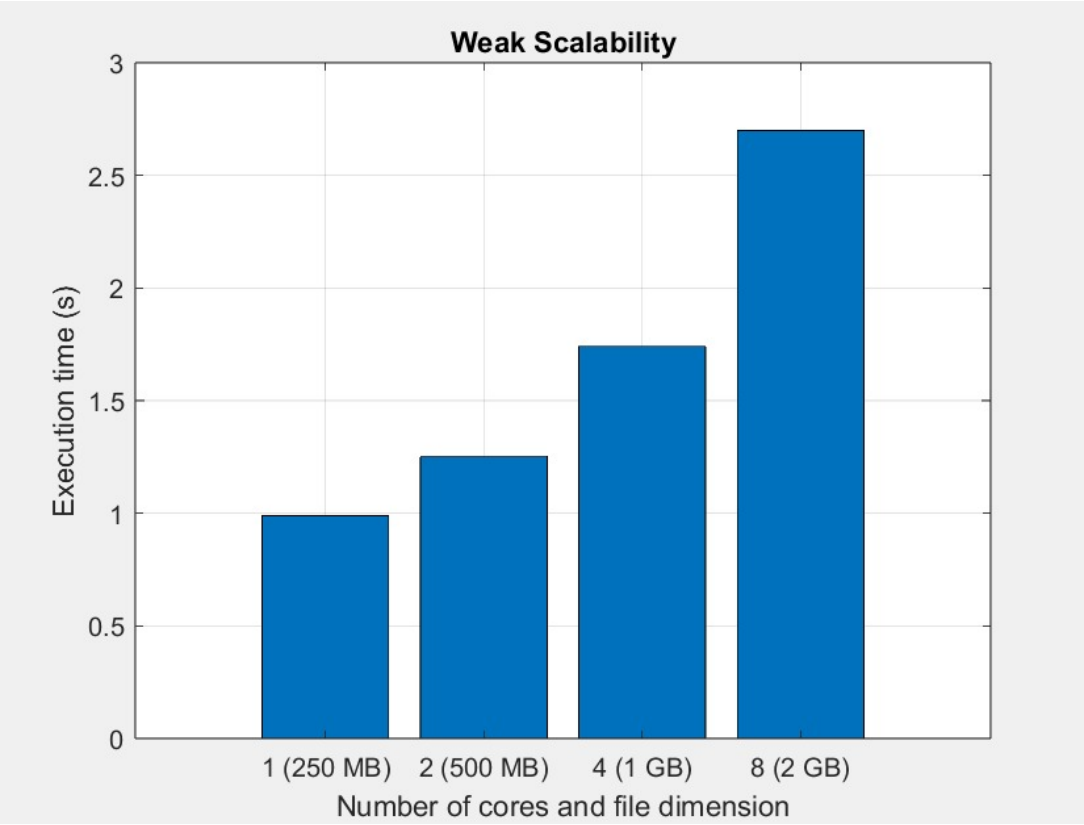

Figure 10: We can observe a likely-exponential growth of the execution time.

- **Strong scalability** The tests were done using always a file with the same dimention (1 GB) and encreasing the number of cores involved.

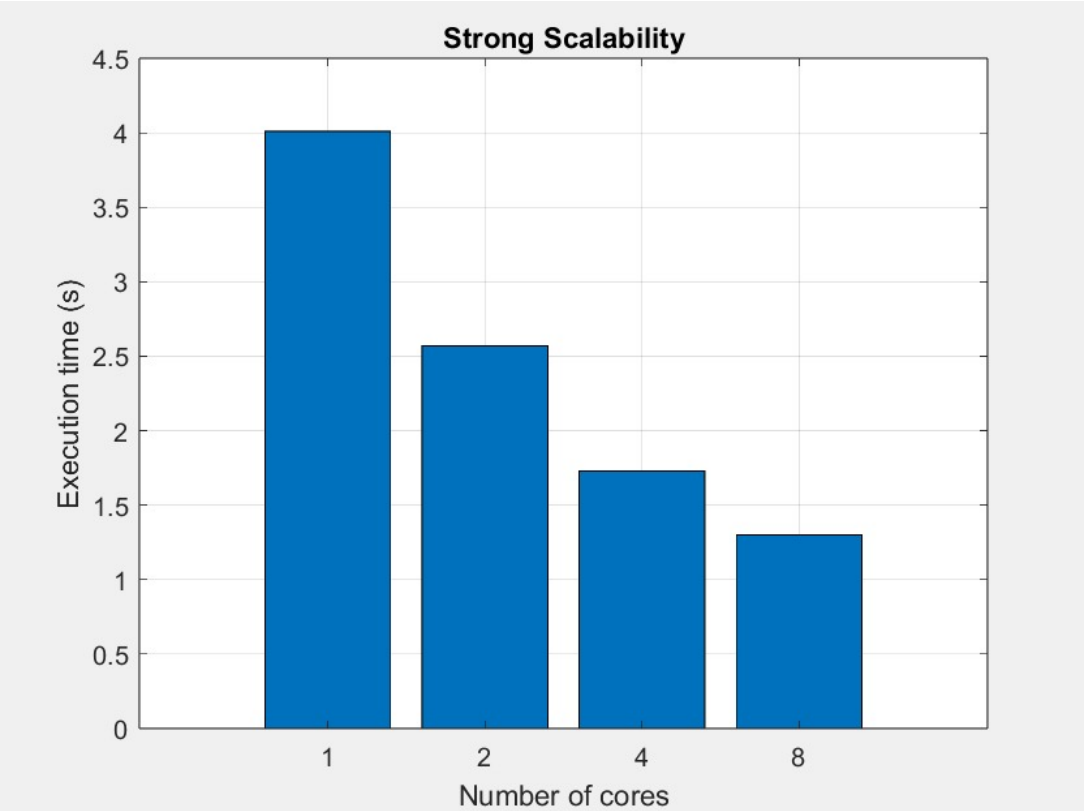

Figure 11: As expected we can see a good reduction of the execution time.

8 Hash Collision Study

We evaluate the collision rates of selected hash functions: DJB2, AddShift-hash, XOR, Better XOR. These are analyzed in terms of number of collisions detected per rank and total. All the others functions apart from the mentioned above had as result no collisions.

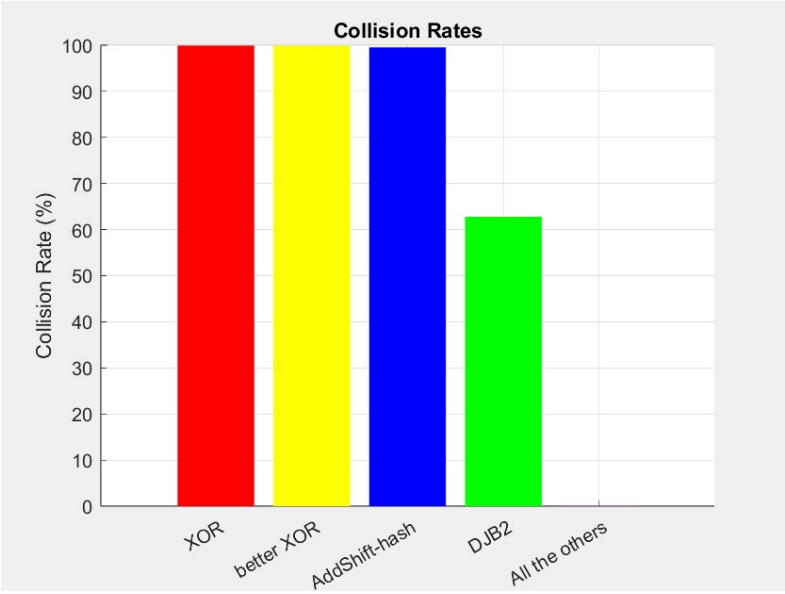

Figure 12: Hash collision analysis per hash function

9 Conclusion

We developed an highly parallel genomic substring matcher using the Boyer-Moore algorithm enhanced with multiple hash functions. Through testing across different cluster architectures on Google Cloud Platform, our solution demonstrated strong scalability, that is, consistent execution time reduction with increased computational resources, and accurate pattern matching. However, the program does not possess weak scalability, since increasing number of cores and size of the genome file proportionally also increases the execution time.

Our comparative study revealed that:

- Hash functions significantly impact collision rates and thus runtime stability.
- The MPI-based parallelization leads to a theoretical speedup close to Amdahl's law prediction.

10 Individual Contributions

While the entire project was developed collaboratively, each member took particular responsibility for different aspects:

Marco Bernazzani (562312): Focused on the implementation and optimization of the MPI-based parallel algorithm, conducted the empirical experiments on GCP clusters, and managed the analysis and visualization of performance data.

Michele Di Frisco Ramirez (557426): Worked on the design and integration of the various hash functions within the Boyer-Moore framework, handled the serial implementation and profiling, and was responsible for the LaTeX documentation and report formatting.

Additionally, as stated in our GitHub README, we would like to express our sincere thanks to **Vito Giacalone** for his suggestions on the project structure and the Utilities library.

11 References

https://www.geeksforgeeks.org/boyer-moore-algorithm-for-pattern-searching/
https://github.com/GigioMagno