Università di Pavia

Substring Matching with Application to Genomics/Proteomics

Advanced Computer Architecture Report

Authors:
Luca Iacopino — Matricola: 564278
Calogero Bugia — Matricola: 558614

Date: 03/07/2025

Contents

Problem Statement and Objectives

Searching for a short nucleotide sequence within the entire human genome is the fundamental objective of this study. In our case, the text represents the genome as a long sequence of nucleotide bases. Each base is encoded as a character, and the target pattern (substring) is itself a shorter sequence of these bases. Our goal is to implement and compare two approaches for solving this problem: a serial implementation and a parallel one, in order to evaluate differences in performance and scalability.

Analysis of the Serial Algorithm

The Rabin-Karp algorithm is a hash-based pattern matching technique used to search for a substring of length m within a longer text of length n. The core idea is to calculate the hash of the pattern and compare it against the hash of each substring of length m in the text.

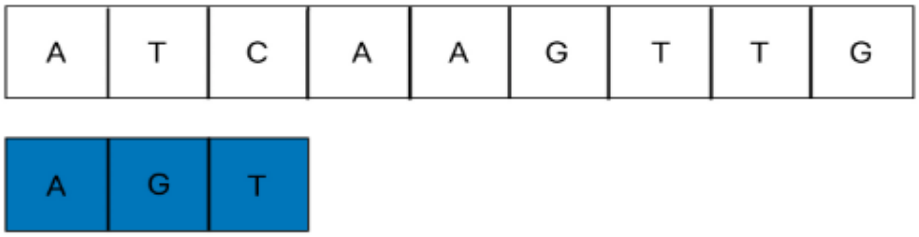

Figure 1: Check between pattern and text

If the hash values match, a character-by-character comparison is performed to confirm the actual match.

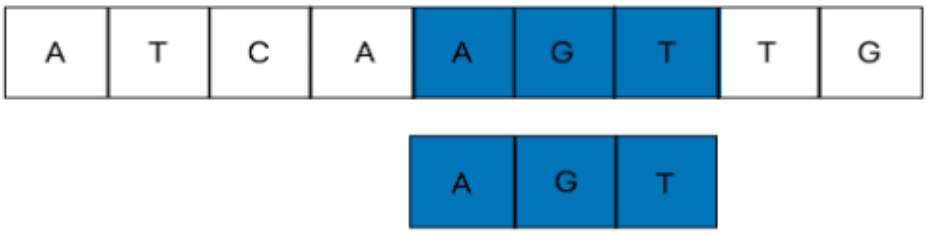

Figure 2: Pattern occurrence found in the text

The best-case computational complexity of the algorithm is $\mathcal{O}(n + m)$, while in the worst case—when hash collisions occur frequently—it can degrade to $\mathcal{O}(n \cdot m)$. To mitigate this, the algorithm leverages a **hash function** that converts a string into a numeric value, reducing the number of required comparisons.

The implementation of the Rabin-Karp algorithm is included in our custom library and can be found at:

`repo/Libraries/rabinkarp.c`

A relevant excerpt of the core matching logic is shown below:

```c
void rabin_karp2(char *txt, char *pattern, const size_t lentxt, const size_t lenpat, long long int *occurrences){
  size_t i,j;
  long long int pat_hash = 0;
  long long int txt_hash = 0;
  long long int h = 1;
  *occurrences = 0;

  for (i = 0; i < lenpat - 1; i++)
      h = (h * P) % M;

  for(i = 0; i < lenpat; i++){
      pat_hash = (P * pat_hash + *(pattern + i)) % M;
      txt_hash = (P * txt_hash + *(txt + i)) % M;
    }

  for(i = 0; i <= lentxt - lenpat; i++){

    if(pat_hash == txt_hash){
      for(j = 0; j < lenpat; j++){
        if(*(txt + i + j) != *(pattern + j)){
          break;
        }
      }
      if(j == lenpat){
        (*occurrences)++;
      }
    }

    if(i < lentxt - lenpat){
      txt_hash = (P * (txt_hash - *(txt + i) * h) + *(txt + i + lenpat)) % M;

      if(txt_hash < 0)
        txt_hash = txt_hash + M;
    }
  }
}
```

Figure 3: Rabin-Karp serial implementation (code excerpt)

The complete implementation is available in the public repository for further reference.

Performance of the Serial Algorithm and a Priori Study of Parallelism

To evaluate the potential benefits of parallelization, it is important to first determine which portions of the code are suitable for parallel execution.

We conducted a performance profiling using the tool `gprof`, applying it to our serial implementation with an input file of approximately 3GB. The analysis was carried out with a pattern of 8 characters, using the rolling hash function within the Rabin-Karp algorithm. The results are summarized below:

%time	cumul. sec	self sec.	calls	self s/call	tot. s/call	name
95.97	23.87	22.91	1	22.91	22.91	rabin_karp2
4.02	23.87	0.96	1	0.96	0.96	readFile

Table 1: Function-level profiling results using `gprof`.

From the profiling results, it is evident that the program is composed of two main functions:

- **readFile**: responsible for loading the input text and the pattern into memory.

- **rabin_karp2**: implements the core logic of the Rabin-Karp substring matching algorithm.

The execution time of each function was measured to compute the percentage of the total runtime that is parallelizable. In our case, the Rabin-Karp algorithm accounts for approximately 96% of the execution time, while the remaining 4% corresponds to file reading operations. This breakdown allows us to estimate the theoretical upper bound of performance improvement through parallelization.

To quantify this, we applied Amdahl's Law, which models the expected speedup as:

$$\text{speedup} = \frac{n}{n + p(1 - n)}$$

where n is the number of processing units and p is the fraction of the program that cannot be parallelized. With $p = 0.04$, the theoretical speedup is shown in the graph below:

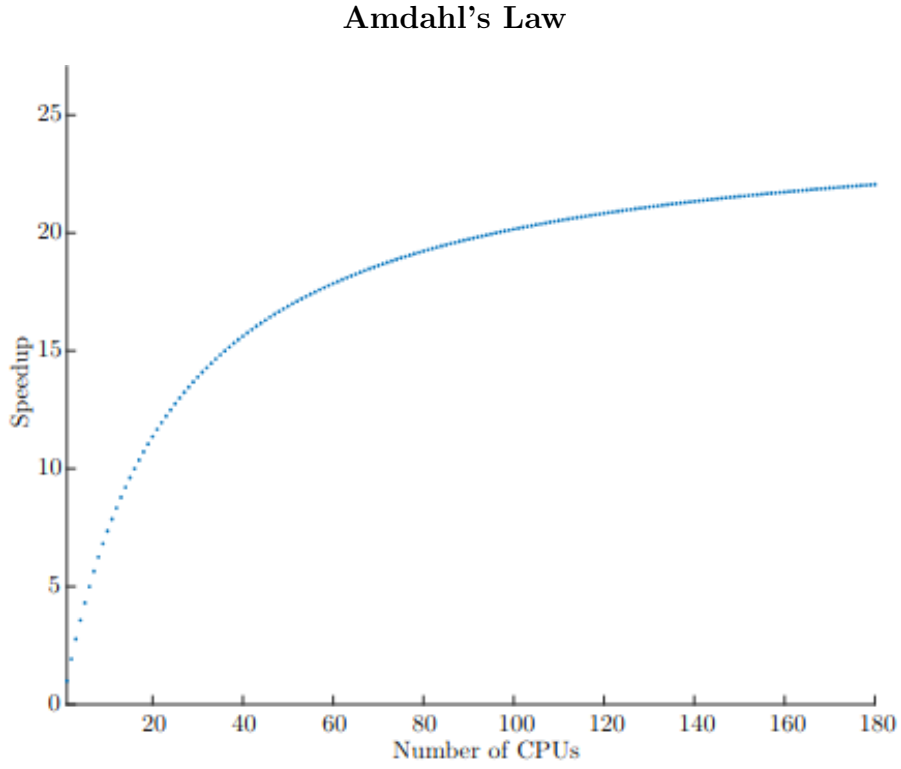

Figure 4: Theoretical Speedup vs Number of CPUs

Text Splitting Strategy for Parallel Processing

To enable parallel substring search, the input text is divided into multiple segments, each assigned to a different process. The master process handles the distribution of data, while the slave processes perform the actual pattern matching on their respective portions.

Each process, except the last one, receives a number of characters calculated as:

$$\frac{\text{text length}}{\text{number of processes}} + \text{pattern length} - 1$$

This additional *pattern length - 1* characters ensures that potential matches spanning across the boundaries of adjacent chunks are not missed.

The final process receives a slightly larger segment to accommodate the remainder of the division, as defined by:

$$\frac{\text{text length}}{\text{number of processes}} + \text{pattern length} - 1 + \text{text length mod \#cores}$$

This approach guarantees complete coverage of the input text and correct detection of patterns, even when they are located at the borders between chunks.

As illustrated in the diagram below, the master process is responsible for distributing the work. It partitions the input text into multiple chunks and assigns each chunk to a slave process. This ensures that the pattern matching can be performed in parallel on independent portions of the text.

Each slave receives a chunk and executes the Rabin-Karp algorithm to detect pattern occurrences locally. The diagram emphasizes the structured coordination between the master and the slaves during the parallel execution phase.

Master-slave Communication Scheme

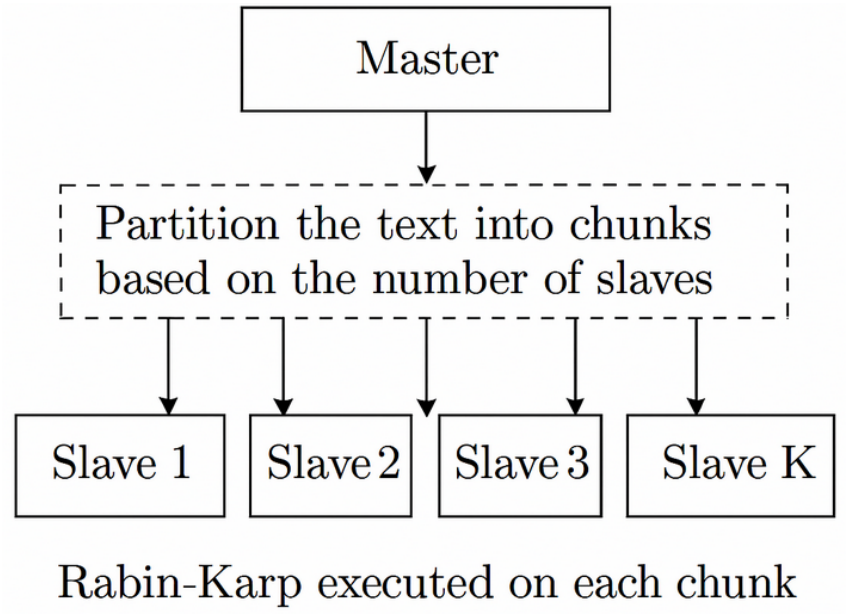

Figure 5: The master partitions the text into chunks and assigns them to the slaves, each of which executes the Rabin-Karp algorithm independently.

MPI Parallel Implementation

Before executing the algorithm, each process needs to receive critical input data. The master process handles this communication phase by broadcasting and scattering the necessary variables.

First, the number of executor cores is shared with all processes using `MPI_Bcast()`, along with the length of the input text and the pattern. A vector of flags is scattered via

`MPI_Scatter()` to indicate which processes are active or inactive, depending on the size of the chunk assigned to each.

```
50      //Send to each core the number of cores that will perform the search
51      MPI_Bcast(&executors, 1, MPI_INT, 0, MPI_COMM_WORLD);
52
53      //Send to each core a value that specify if the core will be and active or not
54      MPI_Scatter(flag, 1, MPI_INT, &isActive, 1, MPI_INT, 0, MPI_COMM_WORLD);
55
56      //Send to each core the length of the text
57      MPI_Bcast(&txtlen, 1, MPI_LONG_LONG, 0, MPI_COMM_WORLD);
58
59      //Send to each core the length of the pattern
60      MPI_Bcast(&patlen, 1, MPI_LONG_LONG, 0, MPI_COMM_WORLD);
```

Figure 6: Preliminary MPI variable distribution to all processes

The splitting of the input text is computed on the master node using the function `split_dataset()`, which partitions the data into balanced chunks. Each chunk is then sent to a slave process using `MPI_Send()`, while the slaves invoke `receive_dataset()` (internally using `MPI_Recv()`) to receive their portion.

Each process then performs Rabin-Karp independently on its chunk. Once completed, the partial results are collected and aggregated using `MPI_Reduce()`, which sends the final count of occurrences back to the master.

Performance and Scalability Analysis

Numerous tests have been carried out on the Google Cloud Platform (GCP) using its Compute Engine service. To measure the performance of the algorithm, the `clock_t()` function was used. This timing function was invoked at both the beginning and the end of the program's execution.

By computing the difference between the end and start timestamps and dividing the result by the system constant `CLOCKS_PER_SEC`, we obtained an estimate of the execution time. Repeating the measurements multiple times allowed us to derive a more reliable and consistent estimate.

Cross-Cluster Performance Analysis

- **Light Cluster – Single Region (8 nodes, 2 vCPU each):** In this configuration, all nodes are deployed within the same data center. This physical proximity results in low network latency and efficient communication. Although each node has limited computational power, the combined parallel execution achieves good performance.

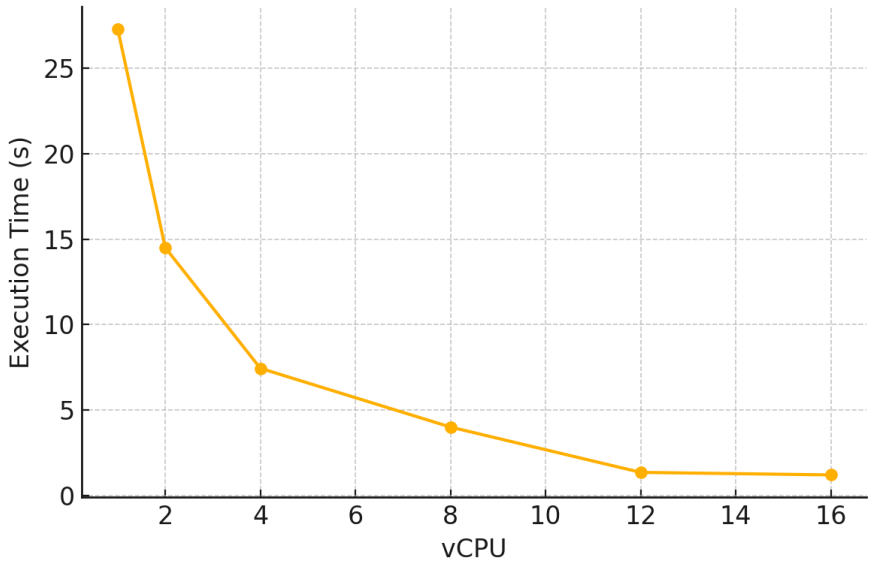

Figure 7. Execution time in a light single-region cluster

- **Fat Cluster – Single Region (2 nodes, 8 vCPU each):** In this case, the parallel computation is concentrated on fewer but more powerful nodes. Thanks to local memory access and fewer inter-node communications, the performance is significantly better compared to the light configuration.

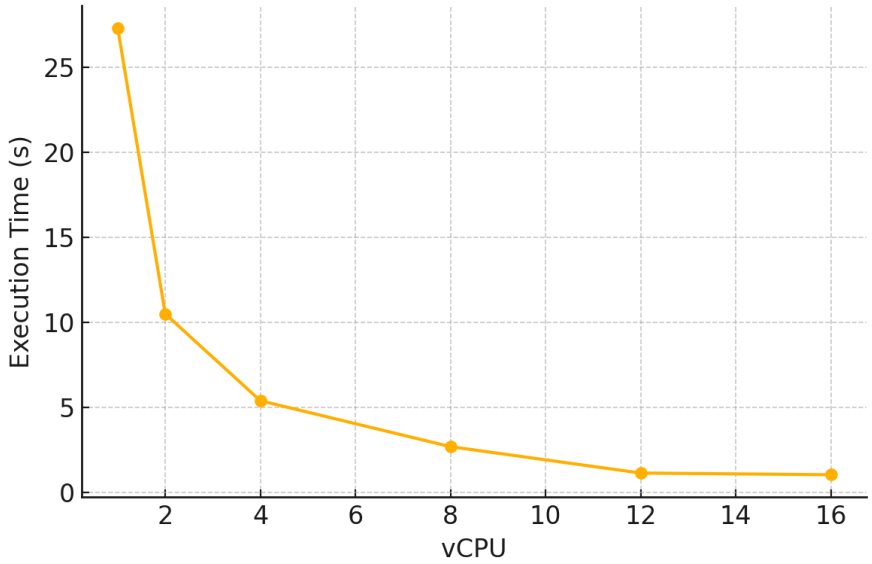

Figure 8. Execution time in a fat single-region cluster

- **Light Cluster – Multi Region (8 nodes, 2 vCPU each):** Nodes are distributed across multiple geographic regions. This setup increases network latency and synchronization times, degrading overall performance compared to the single-region counterpart.

Name	Location
node11-light	northamerica-northeast2-b
node12-light	australia-southeast1-b
node13-light	europe-central2-a
node14-light	asia-northeast3-c
node15-light	europe-north2-b
node16-light	africa-south1-c
node17-light	us-south1-a
node18-light	southamerica-east1-c

Table 2: Node allocation for the light multi-region cluster

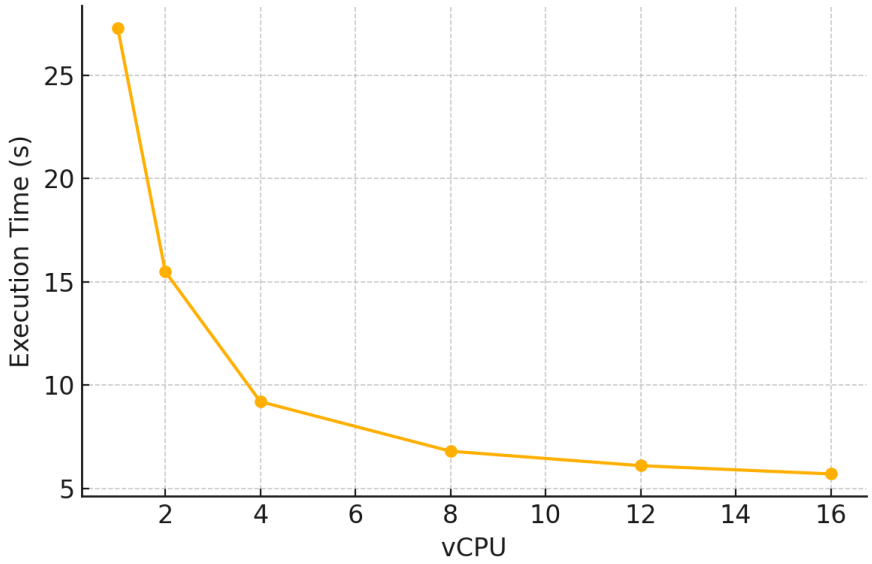

Figure 9. Execution time in a light multi-region cluster.

- **Fat Cluster – Multi Region (2 nodes, 8 vCPU each):** This configuration uses two powerful nodes, each located in a different region. Although inter-region communication adds some overhead, the low number of nodes reduces the total synchronization cost.

Name	Location
node19-fat	europe-west3-a
node20-fat	us-east1-b

Table 3: Node allocation for the fat multi-region cluster

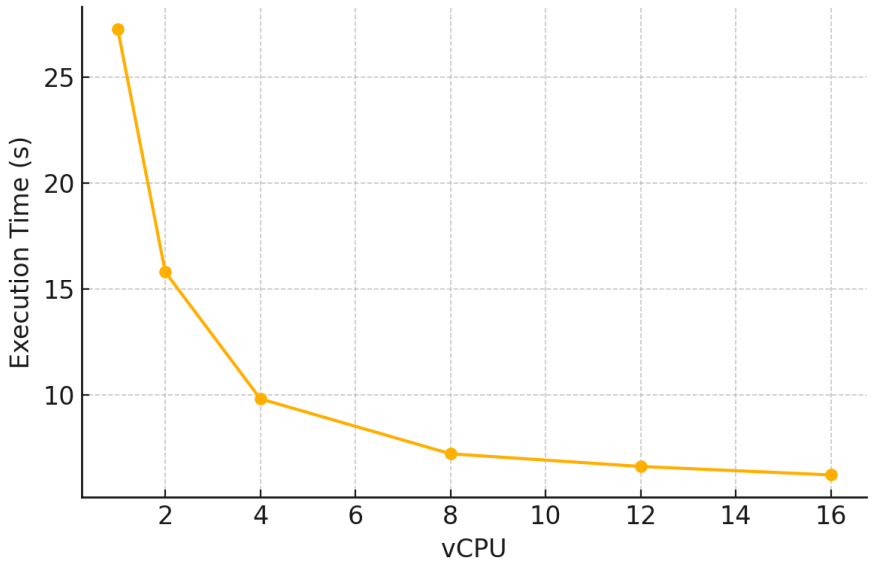

Figure 10. Execution time in a fat multi-region cluster.

Speedup Analysis

Despite the variability introduced by the hardware configuration, number of nodes, and geographical distribution, our experiments show a substantial performance improvement over the serial implementation.

In the best case scenario, the serial version of the algorithm required approximately 27 seconds to complete the computation. In contrast, the parallel implementation, under optimal cluster conditions, achieved a runtime as low as 0.9 seconds. This corresponds to a speedup of approximately 30×.

Compared to the theoretical predictions based on Amdahl's Law (Figure 4), our results are consistent with the expected trend, although slightly below the theoretical maximum due to real-world communication overhead and synchronization delays.

Scalability Analysis

- **Weak Scalability** describes how the solution time varies with the number of processors for a fixed problem size per processor. As the number of processors increases, the total size of the problem increases proportionally. Ideally, the execution time should remain constant.

Text size	# cores	Execution time (s)
250MB	1	$\sim$ 26.0
500MB	2	$\sim$ 26.5
1GB	4	$\sim$ 27.0
2GB	8	$\sim$ 27.3
3GB	12	$\sim$ 27.6

Tabella 4: Weak scalability test with increasing problem size

Figure 11: Weak scalability trend with increasing text size and number of cores

- **Strong Scalability** evaluates how the solution time varies with the number of processors for a fixed total size problem. The goal is to reduce execution time as more processors are used.

Text size	**# cores**	**Execution time (s)**
3GB	1	$\sim$ 27.0
3GB	2	$\sim$ 14.5
3GB	4	$\sim$ 7.2
3GB	8	$\sim$ 3.8
3GB	12	$\sim$ 1.4
3GB	16	$\sim$ 0.9

Tabella 5: Strong scalability test with fixed problem size

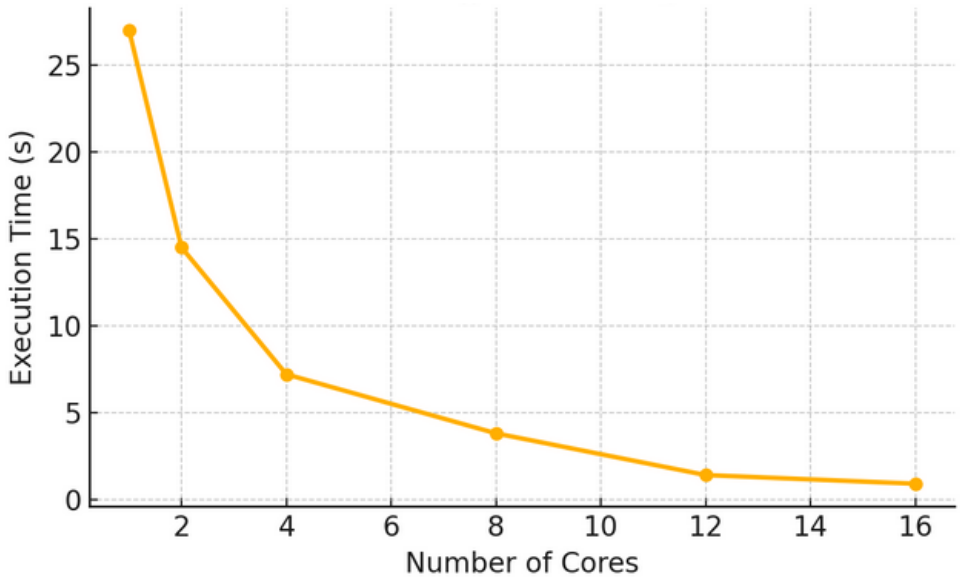

Figure 12: Strong scalability trend with increasing number of cores on a fixed text size

Individual Contribution

This project was a collaborative effort, and both team members contributed equally to all major stages, including serial algorithm design, MPI parallelization, testing, and report writing.