UNIVERSITÀ DEGLI STUDI DI SALERNO

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE ED ELETTRICA E MATEMATICA APPLICATA

Corso di Laurea Magistrale in Ingegneria Informatica [A-H]

HPC Project report.

Prof:
Francesco Moscato
fmoscato@unisa.it

Giuseppe D'Aniello
gidaniello@unisa.it

Student:
Giosuè Ciaravola
Mat. 0612705043
g.ciaravola3@studenti.unisa.it

ANNO ACCADEMICO 2023/2024

Indice

Capitolo 1

Single-machine parallelization (Moscato)

Provide a parallel version of the **Edit distance algorithm** to find a distance between two strings. Implementation MUST use an hybrid message passing / shared memory paradigm and has to be implemented by using **MPI and OpenMP**. Students MUST provide parallel processes on different nodes, and each process has to be parallelized by using OpenMP (i.e.: MPI will spawn OpenMP-compiled processes). Furthermore, an implementation with **OpenMP and CUDA** must be provided.
The parallel algorithm used in "OpenMP + MPI" solution could not be the same of the "OpenMP + CUDA" approach.

1.1 Introduction to the problem

The **Levenshtein Distance algorithm**, also known as edit distance, is used to measure the difference between two strings. The main objective is to determine the minimum number of insertions, deletions, or substitutions of a single character required to transform one string into the other.

Levenshtein distance is often used in the field of computer science, especially in the context of automatic correction of spelling errors, text comparison, or recognition of similar patterns.

1.2 Solution

We'll start from the sequential version and after that we'll consider the OMP+MPI implementation and only eventually the OMP+CUDA approach.

The most efficient solution for this algorithm is implemented through dynamic programming.

The algorithm operates on a matrix (**D**), where the rows represent the characters of the first string (**str1**), and the columns represent the characters of the second string (**str2**). In each cell of the matrix, the minimum cost is stored to transform the substring represented by the first i literals of the first string into the substring represented by the first j literals of the second string. Before populating the matrix, an additional row and column are added, initialized with the row and column indices, respectively. The algorithm iteratively fills the matrix cells taking the minimum between the following transition rules:

- **Insertion:** D[i, j] = D[i, j-1] + 1;
  (occurs when there is an empty space in str1 and a character in str2.)
- **Deletion:** D[i, j] = D[i-1, j] + 1;
  (occurs when there is a character in str1 and an empty space in str2.)
- **Substitution:** D[i, j] = D[i-1, j-1] + t($str1_i$, $str2_j$);
  (where t($str1_i$, $str2_j$) is 0 if the i-th character of the first string is equal to the j-th character of the second string and 1 otherwise.)

The last cell **(D[m, n])** of the matrix contains the Levenshtein distance between the two strings, where m and n are the respective lengths of the two strings.

Its computational complexity is **O(m * n)**, where m and n are the lengths of the two involved strings.

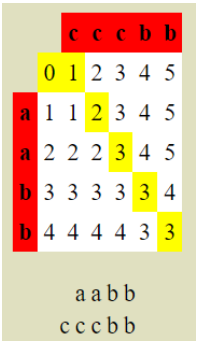
Analyzing the algorithm, parallelization could prove to be complex for the following reasons:

- **Data dependency**: In Levenshtein Distance, each element of the matrix depends on the results of calculations from the previous column and the data from the previous row. Parallelizing the algorithm requires careful management of dependencies between matrix cells. Concurrent access to shared data must be synchronized to ensure that intermediate results are consistent and available to all parallel processes;
- **Parallelization overhead**: Introducing parallelism incurs additional costs, such as creating threads or processes, managing synchronization, and communication between work units. If the problem to be solved is relatively small, the overhead of parallelization may outweigh the benefits gained from workload division. Carefully evaluating the size of the problem is essential to determine whether parallelization is advantageous;
- **Difficulties in workload partitioning**: The Levenshtein Distance algorithm may encounter challenges in dividing the workload into independent subproblems, as each matrix cell depends on preceding cells. Workload partitioning could be complex, and the classic approach of dividing the entire structure into smaller parts might not be immediately applicable.

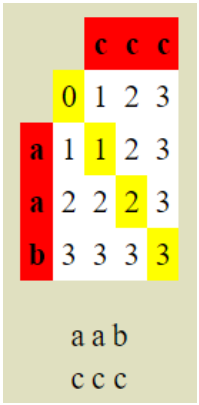
The classic idea for parallelizing an algorithm is to **divide the entire work structure** into smaller parts and assign each part to a different process.

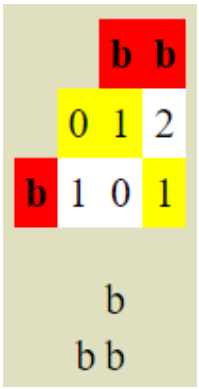
Due to the mentioned issues, **dividing the construction of the matrix** would be futile, as each element relies on the previous ones, leading to synchronization that is challenging to implement and would essentially make the algorithm sequential.

So we decide to **split the strings** and execute the algorithm separately on parts of the strings, obtaining their distances and summing them to retrieve the total distance. However, partitioning the strings and running the algorithm each time on the i^{th} pair of strings will cause the algorithm to lose a **fundamental characteristic**: if we consider the strings "**aabb**" and "**cccbb**" and compare them character by character, their distance is **4**. However, by applying the dynamic programming algorithm, it is able to take into account the fact that by **adding a space at the beginning** of the first string, the distance between " **aabb**" and "**cccbb**" has decreased to **3**:



However, if we were to divide the strings, comparing "**aab**" with "'**ccc**" and "**b**" with "**bb**", we would obtain two distances of **3** and **1**, which sum to **4**:



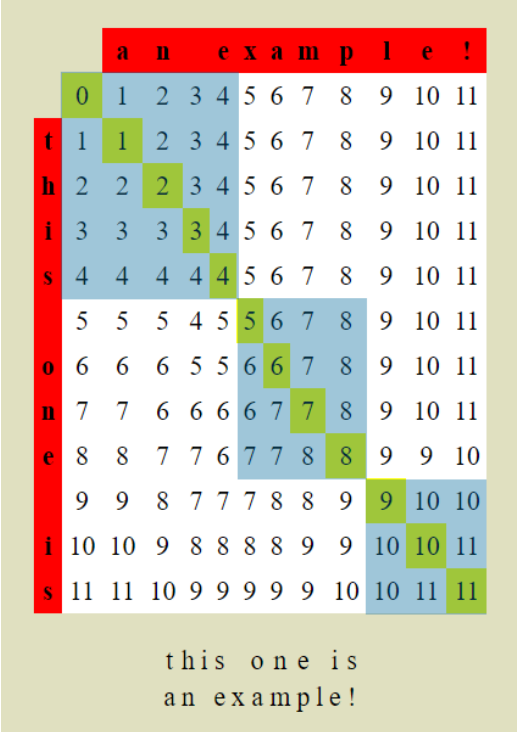


Therefore, we understand that dividing the strings and running the algorithm on data partitions before reuniting the results could lead to an **error**. We will see, however, that for **large datasets with few partitions** applied, this error is **tolerable given the speedup achieved**.

Our dynamic programming algorithm thus becomes an **approximation algorithm**: algorithms designed to provide **close or acceptable solutions** to computational problems when, for some reason, **obtaining an exact solution is not possible**. These algorithms are widely used in situations where searching for a precise solution would require **too much time or computational resources**.

By applying this approximation of string division, even in the sequential version, with an equal number of characters, the approximation algorithm proves to be **proportionally faster** than the dynamic one. This is because, when comparing two strings of length **n**, dividing it into **m** comparisons of two strings of length **n/m**, instead of constructing **a matrix of size [(n+1) x (n+1)]**, results in constructing **m matrices of size [(n+1)/(m+1) x (n+1)/(m+1)]**. This leads to a reduction in the size of the problem **from n to n/m** (although the complexity remains unchanged).

In the image below, we see an example: in the dynamic programming algorithm, we would build **the entire table**, whereas in the approximation algorithm, with, for example, three partitions, only the **three small tables along the diagonal in blue** are constructed (each with the addition of the initial row and column and locally computed values).



1.3 OMP+MPI Parallelization

MPI, widely recognized as **Message Passing Interface**, functions as a extensively adopted library for parallel computing on distributed memory systems. It provides a set of functions that facilitate **inter-process communication**, enabling multiple processes to collaborate and enhance problem-solving efficiency by distributing the workload among them. As a standard, MPI is **not tied to any particular system or architecture**, making it portable and applicable across a diverse range of computers, ranging from small clusters to extensive supercomputers.

One of the primary strengths of MPI lies in its ability to offer significant **flexibility** in designing parallel algorithms. The library provides various communication operations, including **point-to-point communication** (exchange of messages between two processes), **collective communication** (message exchange among all processes), and **non-blocking communication** (allowing processes to continue execution while awaiting a message).

Despite these advantages, MPI has its **constraints**. Notably, it does not provide support for **shared memory programming**, a valuable feature in certain algorithmic scenarios. In contrast, **OpenMP** presents a **useful solution for parallelizing** existing code originally designed for single-core execution. However, achieving benefits from OpenMP is **not universally guaranteed**, as several factors may limit its suitability or the advantages derived from parallelization.

One situation where OpenMP might be **less suitable** is when the code is already **optimized for single-core execution**. In such cases, parallelizing the code may not result in a **significant performance boost**. Additionally, OpenMP is primarily designed for parallelizing code **heavily reliant on loops**, such as code involved in solving problems within scientific computing. If the code has a lower dependence on loops, deriving benefits from parallelization might prove **challenging**.

Another scenario where OpenMP may not be the most fitting choice is when the code requires **extensive communication between cores**. In cases demanding a high level of inter-core communication, it might be more appropriate to utilize another parallel library, such as MPI, specifically crafted to manage heightened communication requirements between cores.

1.3.1 OMP+MPI Approach

In the OMP+MPI approach, each MPI process randomly generates the two strings, and subsequently creates its own **partition based on its rank and the total number of MPI processes**.

The partition is generated in a way to **distribute the workload** as evenly as possible.

At this point, each MPI process creates a vector with a number of positions equal to the number of its OMP threads. Each thread inserts the result of the edit distance calculated on its own substring partition, at the [tid] index of the vector. These substring partitions are obtained from the **partition of the MPI process string partition, based on its thread id (tid) and the total number of OMP threads of that MPI process**.

At the end of the execution of all threads, all elements of the vector are summed so that each MPI process has its own **local Edit Distance value**.

Once all MPI process have their local results, a **MPI function** will be used to gather all the local results from all processes and consolidate them into a globally defined result.

The useful function for our purpose is precisely **MPI_Reduce**.
Below is the signature of the function.

```
int MPI_Reduce( void* sendbuf, void* recvbuffer, int count,
                MPI_Datatype datatype, MPI_Op op,
                int root, MPI_Comm communicator)
```

What MPI_Reduce does is to reduce all the results of all the processes according to an **MPI_Op** op that can be sum, difference, maximum, etc... To get a single result and put it in the global result, the operation that comes in handy is just the sum. So by making the sum of all these local results, the sum will be the total Edit Distance betweem the two strings and so it will end up in a variable that we will call "**ED**."

1.3.2 Experimental setup

Now it's time to begin the experiments, starting by first presenting the hardware characteristics of the CPU.

Hardware configuration

Socket 1	: ID = 0
Number of cores	: 6
Number of threads	: 12
Manufacturer	: GenuineIntel
Name	: Intel Core i7 10750H
Codename	: Comet Lake
Specification	: Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz
Package (platform ID)	: Socket 1440 FCBGA (0x5)
CPUID	: 6.5.2
Extended CPUID	: 6.A5
Core Stepping	: R0/R1
Technology	: 14 nm

At the top all the CPU features are displayed, but only a portion is shown. In particular, our focus will be on the **number of cores** and **logical threads**, as they will be crucial for conducting the experiments correctly to avoid undesired results.

Softwares

- **VisualStudio** used for coding
- **Python** used for plots
- **Makefile** used for creating the script

Time Measures

Understanding the chosen approach for parallelization with OMP and MPI, what is useful for comprehending the effectiveness or lack thereof of the carried out parallelization is precisely **time**, as there is nothing more immediately visible than to see whether the time elapsed for the execution of the parallel algorithm is greater or less than the well-known serial version.
We will carry out a total of four different time measurements, considering the following times:

- **String_generation_time**: Time elapsed between the start and end of the creation of random String.
- **Communication_time**: Communication time among MPI processes.
- **Edit_distance_time**: Execution time of the edit distance algorithm.
- **Program_execution_time**: Total program execution time.

Error, Performance, Speed-Up & Efficiency

In the realm of high-performance computing, the correlation between **speed-up** and **efficiency** holds significant significance.

Speed-up gauges the **enhanced speed** achieved through the utilization of multiple processors in comparison to a solitary processor.

On the other hand, efficiency assesses the actual **amount of work accomplished by the processors** relative to the maximum theoretical workload.

Traditionally, speed-up is quantified as the ratio of the execution time on a single processor to the execution time on multiple processors:

$$S_n = \frac{T_s}{T_p(n)}$$

In the context where T_s represents the duration of the most efficient known serial algorithm, and $T_p(n)$ denotes the execution time of the parallel algorithm using n processors. For instance, if a particular application takes **100 seconds to execute on a single processor** and **50 seconds on 4 processors**, the resulting **speed-up is 2**. In theory, the utilization of multiple processors should ideally yield **linear** speed-up concerning the number of processors employed. Nonetheless, practical implementation faces challenges related to **communication** and **synchronization** among processors, making achieving perfect linearity not always feasible.

Efficiency, particularly relative efficiency, is given by the ratio of the time to execute an algorithm on a single processor to n multiplied by the time to execute a parallel algorithm on n processors:

$$E_r = \frac{T_s}{n \cdot T_n}$$

Using the previous example, relative efficiency would be $\mathbf{100/(4 \cdot 50)}$ on 4 processors. In theory, efficiency should always be greater than or equal to 1, as more processors are used to perform the work. However, in practice there may be losses in efficiency due to issues of communication and synchronization between processors.

Just to analyze the actual utility of the approximation algorithm, we will consider the **approximation error** as an additional metric.

Analysis

For the analysis of error, timings, speedup, and efficiency, a well-defined criterion will be followed: the **serial version of the dynamic algorithm** (just for the sample for the error analysis), the **serial version of the approximate algorithm** (where the number of partitions of the string is done according to the number of OMP threads and MPI processes of the parallel version with which it will be compared) and the **parallel versions implemented with OMP and MPI** will be considered.

Analyses will be conducted for all **four optimization levels** (**-O0, -O1, -O2, -O3**).

For each optimization level, **three analyses** will be performed based on the problem size, specifically, the number of characters of the compared strings.

In particular, **three case studies** will be examined:

1. The sequential and parallel programs compiled and executed with two strings of **10000 characters**;
2. The sequential and parallel programs compiled and executed with two strings of **20000 characters**;
3. The sequential and parallel programs compiled and executed with two strings of **40000 characters**.

This selection is made to observe the trends of erros, timings, speedup, and efficiency across different size of the strings.

For each individual case study, considering the number of cores and logical threads discussed in the previous sections, analyses will be conducted following this pattern:

- 0 MPI Processes - No OpenMP threads (Sequential dynamic version);
- 1 MPI Process - 1, 2, 4, 8 OpenMP threads (Parallel and Approximate (just for the strings partitioning) versions);
- 2 MPI Processes - 1, 2, 4, 8 OpenMP threads (Parallel and Approximate (just for the strings partitioning) versions);
- 4 MPI Processes - 1, 2, 4, 8 OpenMP threads (Parallel and Approximate (just for the strings partitioning) versions);

All cases were executed **7 times** to mitigate any fluctuations due to system overhead. Consequently, the presented data (except the error) is the result of **averages**.

1.4 OMP+MPI Analysis of the plots

This chapter will show the anlysis of the approximation error and the plots made by analyzing the sequential, approximate and parallel version of the Edit Distance algorithm carried out with OMP and MPI. We will start from the error analysis and go on with the optimization 0 (no optimization) level, discussing about the three case studies for each optimization level showing both tables and graphs related to them.

1.4.1 Error analysis

We will analyze the error **independently** of the optimization as it does **not depend on it**.

10000 Characters

Modality	OMP	MPI	Edit Distance
Sequential	0	0	8758
Approximate	1	1	8758
OMP+MPI	1	1	8758
Approximate	1	2	8764
OMP+MPI	1	2	8764
Approximate	1	4	8784
OMP+MPI	1	4	8784
Approximate	2	1	8764
OMP+MPI	2	1	8764
Approximate	2	2	8784
OMP+MPI	2	2	8784
Approximate	2	4	8812
OMP+MPI	2	4	8812
Approximate	4	1	8784
OMP+MPI	4	1	8784
Approximate	4	2	8812
OMP+MPI	4	2	8812
Approximate	4	4	8836
OMP+MPI	4	4	8836
Approximate	8	1	8812
OMP+MPI	8	1	8812
Approximate	8	2	8836
OMP+MPI	8	2	8836
Approximate	8	4	8881
OMP+MPI	8	4	8881

Edit Distance table for 10000 characters.

Firstly, we notice that the result of the parallel versions has **the same** outcome as the sequential version approximated with the corresponding string partitions.

From these data, which represent the **distances** calculated by various programs with different combinations of MPI processes and OMP threads, we can observe a clear **trend**: as the **number of partitions increases**, the **error also increases**. In particular, we can calculate the error obtaining **the difference between distances by subtracting the corrected distance of the sequential version from the approximated distance of the parallel version and dividing it by the number of characters**.

In this case, with the maximum number of partitions **(32 = 8 threads * 4 processes)**, we have an error of **1.23%**. Naturally, this **decreases as the number of partitions decreases**. We expect that for **larger-sized case studies** in the future, this **error will be lower**.

20000 Characters

Modality	OMP	MPI	Edit Distance
Sequential	0	0	17497
Approximate	1	1	17497
OMP+MPI	1	1	17497
Approximate	1	2	17512
OMP+MPI	1	2	17512
Approximate	1	4	17528
OMP+MPI	1	4	17528
Approximate	2	1	17512
OMP+MPI	2	1	17512
Approximate	2	2	17528
OMP+MPI	2	2	17528
Approximate	2	4	17567
OMP+MPI	2	4	17567
Approximate	4	1	17528
OMP+MPI	4	1	17528
Approximate	4	2	17567
OMP+MPI	4	2	17567
Approximate	4	4	17634
OMP+MPI	4	4	17634
Approximate	8	1	17567
OMP+MPI	8	1	17567
Approximate	8	2	17634
OMP+MPI	8	2	17634
Approximate	8	4	17697
OMP+MPI	8	4	17697

Edit Distance table for 20000 characters.

Firstly, we notice that the result of the parallel versions has **the same** outcome as the sequential version approximated with the corresponding string partitions.

In this case, with the maximum number of partitions **(32 = 8 threads * 4 processes)**, we have an error of **1%**. As expected, the error, with the same number of partitions, has **decreased** as the number of characters increased. This is because **each partition has more characters**, and therefore, there is a **higher probability of finding possible shifts**.

40000 Characters

Modality	OMP	MPI	Edit Distance
Sequential	0	0	34979
Approximate	1	1	34979
OMP+MPI	1	1	34979
Approximate	1	2	35000
OMP+MPI	1	2	35000
Approximate	1	4	35032
OMP+MPI	1	4	35032
Approximate	2	1	35000
OMP+MPI	2	1	35000
Approximate	2	2	35032
OMP+MPI	2	2	35032
Approximate	2	4	35066
OMP+MPI	2	4	35066
Approximate	4	1	35032
OMP+MPI	4	1	35032
Approximate	4	2	35066
OMP+MPI	4	2	35066
Approximate	4	4	35127
OMP+MPI	4	4	35127
Approximate	8	1	35066
OMP+MPI	8	1	35066
Approximate	8	2	35127
OMP+MPI	8	2	35127
Approximate	8	4	35214
OMP+MPI	8	4	35214

Edit Distance table for 40000 characters.

Firstly, we notice that the result of the parallel versions has **the same** outcome as the sequential version approximated with the corresponding string partitions.

In this case, with the maximum number of partitions **(32 = 8 threads * 4 processes)**, we have an error of **0.5%**. As expected, the error, with the same number of partitions, has **decreased even further** as the number of characters increased.

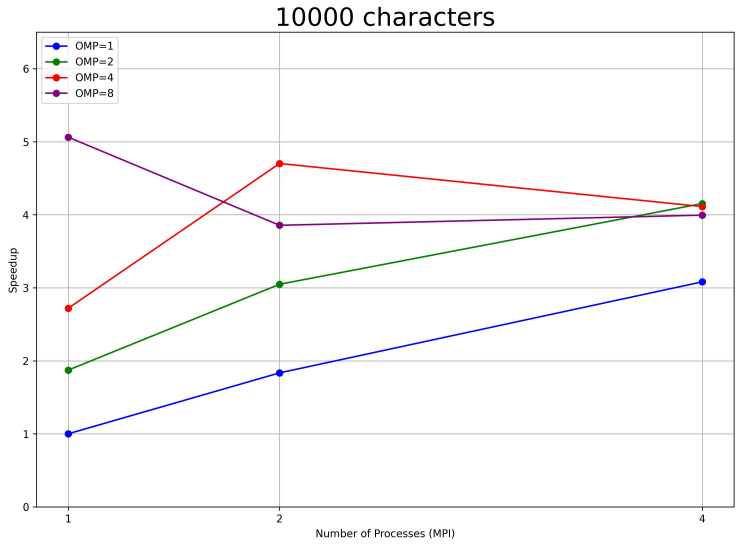
The **trend** is clear: as the **output increases**, the **error decreases** more and more, while as the number of **partitions increases**, the **error increases**. Ultimately, since we parallelize the algorithm precisely to address problems with **massive datasets**, we can conclude that parallelizing the algorithm with this approximation, **from the error prospective, is advantageous**.

1.4.2 Optimization 0

All considerations made from now on regarding parallel executions are made in reference to the **approximated sequential reference version, in relation to the number of partitions**. This is because, as we will see, with an **increase in the number of partitions**, the **approximated sequential version will also speed up**. As explained earlier, the **size of the problem decreases** with the increase in partitions.

Modality	OMP	MPI	String_generation_time	Communication_time	Edit_distance_time	Program_execution_time	speedup	efficiency
Approximate	1	1	0.00032186	0	1.04351986	1.04384186	1	100
Approximate	1	2	0.000298	0	0.520327	0.52062543	1	100
Approximate	1	4	0.00038829	0	0.26126057	0.26164957	1	100
Approximate	2	1	0.00029157	0	0.52425171	0.52454371	1	100
Approximate	2	2	0.00028786	0	0.25922343	0.25951143	1	100
Approximate	2	4	0.00028014	0	0.12869871	0.128979	1	100
Approximate	4	1	0.00028286	0	0.25176357	0.252047	1	100
Approximate	4	2	0.00028329	0	0.13230843	0.13259214	1	100
Approximate	4	4	0.00027457	0	0.06312629	0.06340086	1	100
Approximate	8	1	0.00027229	0	0.12985671	0.13012914	1	100
Approximate	8	2	0.00030829	0	0.064747	0.06505571	1	100
Approximate	8	4	0.00029829	0	0.03177357	0.03207229	1	100
OMP+MPI	1	1	0.00028743	0.00001114	1.04291643	1.043204	1.0006114	100.0611
OMP+MPI	1	2	0.00028843	0.00232586	0.28348286	0.28377143	1.8346647	91.7332
OMP+MPI	1	4	0.00031829	0.00322386	0.08460143	0.08491986	3.0811353	77.0284
OMP+MPI	2	1	0.00026786	0.00001971	0.27988443	0.28015257	1.8723502	93.6175
OMP+MPI	2	2	0.00027543	0.00105443	0.084877	0.08515286	3.0475951	76.1899
OMP+MPI	2	4	0.00032814	0.00598029	0.03072843	0.03105671	4.153015	51.9127
OMP+MPI	4	1	0.00031871	0.000019	0.09237414	0.09269286	2.7191631	67.9791
OMP+MPI	4	2	0.00025914	0.00033429	0.02794457	0.02820371	4.7012298	58.7654
OMP+MPI	4	4	0.00030857	0.00253614	0.01511371	0.01542229	4.1109897	25.6937
OMP+MPI	8	1	0.00027257	0.00002743	0.025437	0.02570957	5.0615057	63.2688
OMP+MPI	8	2	0.000278	0.00073843	0.01659714	0.01687514	3.8551208	24.0945
OMP+MPI	8	4	0.00030543	0.00061443	0.00772414	0.00802971	3.9942001	12.4819

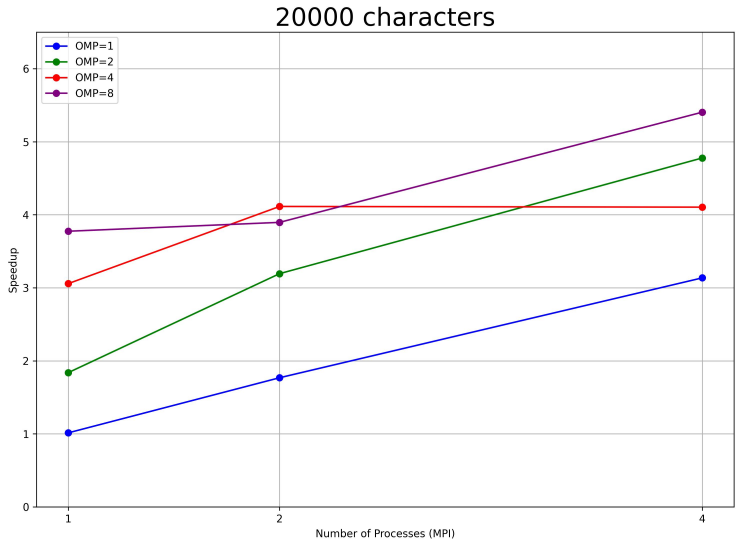
Timing table for 10000 characters.



Speedup plot for 10000 characters.

Modality	OMP	MPI	String_generation_time	Communication_time	Edit_distance_time	Program_execution_time	speedup	efficiency
Approximate	1	1	0.000571	0	4.06732857	4.06789986	1	100
Approximate	1	2	0.00058729	0	2.02560529	2.02619257	1	100
Approximate	1	4	0.00057443	0	1.018953	1.01952771	1	100
Approximate	2	1	0.00059986	0	2.04855329	2.04915343	1	100
Approximate	2	2	0.00059529	0	1.03048771	1.03108314	1	100
Approximate	2	4	0.00056829	0	0.51539157	0.51596071	1	100
Approximate	4	1	0.000523	0	1.02778086	1.02830386	1	100
Approximate	4	2	0.00054257	0	0.515141	0.51568414	1	100
Approximate	4	4	0.00054857	0	0.25804871	0.25859729	1	100
Approximate	8	1	0.000601	0	0.51535129	0.51595257	1	100
Approximate	8	2	0.00057343	0	0.25721929	0.25779329	1	100
Approximate	8	4	0.00060271	0	0.12725029	0.12785343	1	100
OMP+MPI	1	1	0.000561	0.00001386	4.01293529	4.01349657	1.01355508	101.3555
OMP+MPI	1	2	0.00057714	0.01306914	1.14540957	1.145987	1.7680764	88.4038
OMP+MPI	1	4	0.00065557	0.02635029	0.32452543	0.32518143	3.13525812	78.3815
OMP+MPI	2	1	0.000538	0.00002314	1.11455686	1.11509514	1.83764896	91.8824
OMP+MPI	2	2	0.00056	0.005649	0.322358	0.322918	3.19301848	79.8255
OMP+MPI	2	4	0.00062514	0.00211071	0.10738543	0.10801086	4.7769338	59.7117
OMP+MPI	4	1	0.00051357	0.00001786	0.33590986	0.33642386	3.05657234	76.4143
OMP+MPI	4	2	0.00058871	0.00337171	0.12478871	0.12537757	4.11304938	51.4131
OMP+MPI	4	4	0.00060014	0.00888043	0.06241829	0.06301843	4.10351847	25.647
OMP+MPI	8	1	0.00055471	0.00002914	0.13616286	0.136718	3.77384522	47.1731
OMP+MPI	8	2	0.00054243	0.00589857	0.06563986	0.06618257	3.89518389	24.3449
OMP+MPI	8	4	0.00062314	0.00163471	0.02303629	0.02365971	5.40384499	16.887

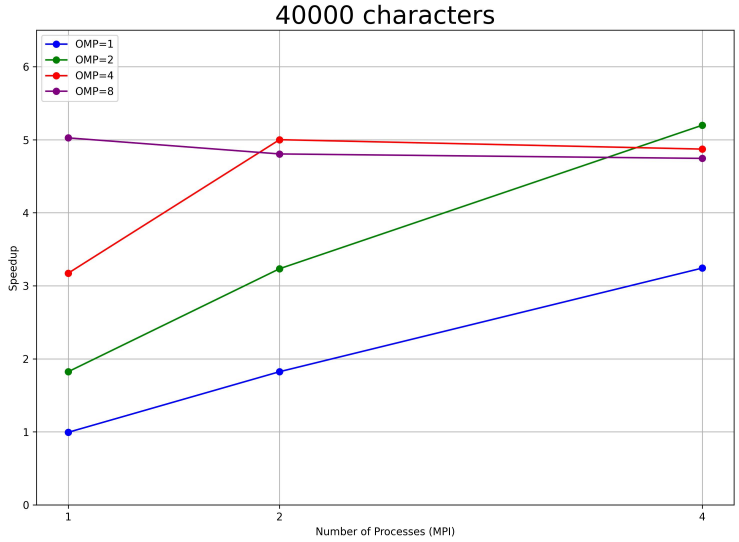
Timing table for 20000 characters.



Speedup plot for 20000 characters.

Modality	OMP	MPI	String_generation_time	Communication_time	Edit_distance_time	Program_execution_time	speedup	efficiency
Approximate	1	1	0.00109957	0	16.37222986	16.37332971	1	100
Approximate	1	2	0.001128	0	8.28118014	8.28230843	1	100
Approximate	1	4	0.00135057	0	4.12007157	4.12142229	1	100
Approximate	2	1	0.00103943	0	8.241308	8.242348	1	100
Approximate	2	2	0.00117657	0	4.12628671	4.12746343	1	100
Approximate	2	4	0.00115514	0	2.05595529	2.05711057	1	100
Approximate	4	1	0.00106857	0	4.09826814	4.09933714	1	100
Approximate	4	2	0.00108771	0	2.04187543	2.04296343	1	100
Approximate	4	4	0.00109957	0	1.03384229	1.034942	1	100
Approximate	8	1	0.00110171	0	2.06563871	2.06674086	1	100
Approximate	8	2	0.00104843	0	1.03706071	1.03810914	1	100
Approximate	8	4	0.00110129	0	0.51927414	0.52037557	1	100
OMP+MPI	1	1	0.00105057	0.00001229	16.477388	16.478439	0.9936214	99.3621
OMP+MPI	1	2	0.00119829	0.00910743	4.54065771	4.54185629	1.8235514	91.1776
OMP+MPI	1	4	0.00136271	0.02428314	1.26984314	1.27120614	3.2421353	81.0534
OMP+MPI	2	1	0.00105029	0.00002071	4.51964243	4.52069286	1.8232488	91.1624
OMP+MPI	2	2	0.00122986	0.01666629	1.27533971	1.27657	3.2332449	80.8311
OMP+MPI	2	4	0.00133757	0.00667657	0.39443414	0.39577171	5.19772	64.9715
OMP+MPI	4	1	0.00104214	0.00001529	1.29128129	1.29232357	3.1720672	79.3017
OMP+MPI	4	2	0.00119414	0.00278914	0.40741371	0.40860814	4.9998109	62.4976
OMP+MPI	4	4	0.00133686	0.01252	0.21113486	0.21247171	4.8709637	30.4435
OMP+MPI	8	1	0.00116543	0.00002214	0.41014657	0.41131271	5.0247434	62.8093
OMP+MPI	8	2	0.00117671	0.00280786	0.21489043	0.21606771	4.8045547	30.0285
OMP+MPI	8	4	0.00118986	0.01164871	0.108511	0.109701	4.7435809	14.8237

Timing table for 40000 characters.



Speedup plot for 40000 characters.

For this initial level of optimization, we begin to notice a few things. Since we applied parallelization by **dividing the workload among MPI processes and then among OMP threads**, we observe that the speedups, with the **same number of partitions (#MPI processes * #OMP threads)**, appear to be **similar**.

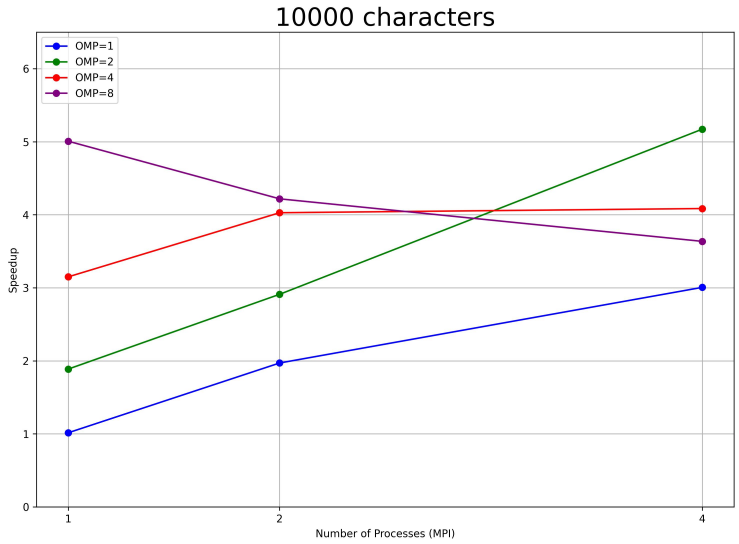
We also notice that as the size of the problem **decreases proportionally with the increase in partitions**, the speedup **increases almost proportionally with the increase in partitions**. It's **not perfectly proportional** because there are still other times (String_generation_time, Communication_time, etc.) that do not depend on partitions, which **lowers the slope of the curve**.

Finally, however, we notice that for **high partitions**, the speedup tends to **stabilize**. This is probably because the **problem becomes so small** that the sequential algorithm **improves more or less linearly**, as the **communication times** of the parallel algorithm increase.

1.4.3 Optimization 1

Modality	OMP	MPI	String_generation_time	Communication_time	Edit_distance_time	Program_execution_time	speedup	efficiency
Approximate	1	1	0.00024286	0	0.711541	0.71178457	1	100
Approximate	1	2	0.00027914	0	0.36428414	0.36456371	1	100
Approximate	1	4	0.00021543	0	0.17604057	0.17625671	1	100
Approximate	2	1	0.00022129	0	0.35903014	0.35925157	1	100
Approximate	2	2	0.00024914	0	0.175871	0.17612057	1	100
Approximate	2	4	0.00022429	0	0.08787757	0.08810243	1	100
Approximate	4	1	0.00022029	0	0.17213757	0.172358	1	100
Approximate	4	2	0.00024	0	0.087621	0.08786157	1	100
Approximate	4	4	0.00022814	0	0.04441029	0.04463857	1	100
Approximate	8	1	0.00023557	0	0.08624829	0.08648429	1	100
Approximate	8	2	0.00023443	0	0.04650629	0.04674071	1	100
Approximate	8	4	0.00022986	0	0.01970229	0.01993229	1	100
OMP+MPI	1	1	0.00023157	0.00001514	0.70132914	0.70156386	1.0145685	101.4568
OMP+MPI	1	2	0.00022386	0.00180571	0.184852	0.18507786	1.9697857	98.4893
OMP+MPI	1	4	0.00029114	0.00365529	0.05835371	0.05864643	3.0054126	75.1353
OMP+MPI	2	1	0.00021271	0.000018	0.19021829	0.190433	1.8864985	94.3249
OMP+MPI	2	2	0.00025843	0.00220057	0.06025986	0.06052029	2.9101081	72.7527
OMP+MPI	2	4	0.000229	0.00105229	0.01680614	0.01703729	5.171154	64.6394
OMP+MPI	4	1	0.00021486	0.00002186	0.05451343	0.05473071	3.1492006	78.73
OMP+MPI	4	2	0.000246	0.000136	0.02156957	0.02181743	4.0271277	50.3391
OMP+MPI	4	4	0.00026243	0.00256714	0.01066486	0.01092871	4.0845218	25.5283
OMP+MPI	8	1	0.00021771	0.00002643	0.01705429	0.01727471	5.006409	62.5801
OMP+MPI	8	2	0.00021857	0.00063986	0.01086186	0.01108257	4.2174972	26.3594
OMP+MPI	8	4	0.00023629	0.00101514	0.00524514	0.005483	3.6352883	11.3603

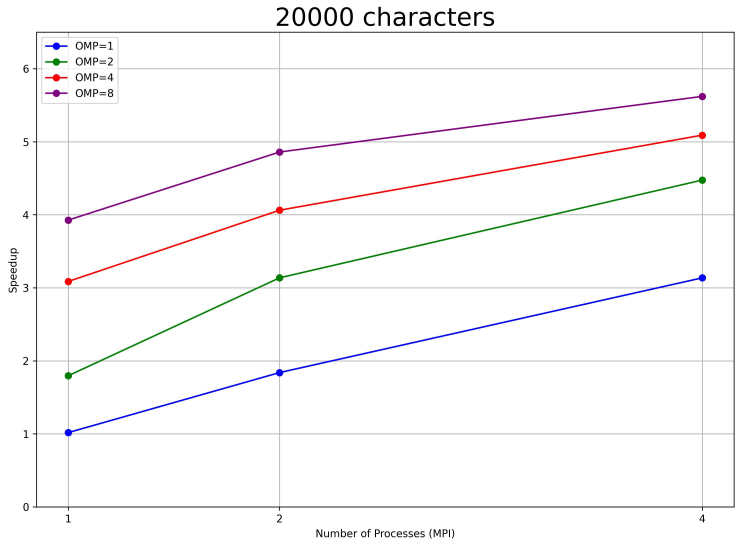
Timing table for 10000 characters.



Speedup plot for 10000 characters.

Modality	OMP	MPI	String_generation_time	Communication_time	Edit_distance_time	Program_execution_time	speedup	efficiency
Approximate	1	1	0.00044729	0	2.78654243	2.78699014	1	100
Approximate	1	2	0.00050157	0	1.39782157	1.39832329	1	100
Approximate	1	4	0.00043329	0	0.692282	0.69271586	1	100
Approximate	2	1	0.00043657	0	1.38896943	1.38940629	1	100
Approximate	2	2	0.00046386	0	0.69387986	0.69434386	1	100
Approximate	2	4	0.00044114	0	0.35463314	0.355075	1	100
Approximate	4	1	0.00044486	0	0.70146743	0.70191229	1	100
Approximate	4	2	0.00043729	0	0.35155957	0.35199714	1	100
Approximate	4	4	0.00050343	0	0.17167457	0.17217843	1	100
Approximate	8	1	0.00047543	0	0.35622586	0.35670157	1	100
Approximate	8	2	0.000473	0	0.17439171	0.17486471	1	100
Approximate	8	4	0.00045786	0	0.08611686	0.08657514	1	100
OMP+MPI	1	1	0.00043071	0.000014	2.74002271	2.74045486	1.0169809	101.6981
OMP+MPI	1	2	0.00041929	0.001252	0.760068	0.76048943	1.838715	91.9357
OMP+MPI	1	4	0.00056657	0.01113186	0.22035429	0.22092257	3.1355595	78.389
OMP+MPI	2	1	0.00040871	0.00001729	0.77338214	0.773793	1.7955788	89.7789
OMP+MPI	2	2	0.00048	0.000272	0.22091957	0.22140157	3.1361289	78.4032
OMP+MPI	2	4	0.00055086	0.00811843	0.07880757	0.07936	4.4742314	55.9279
OMP+MPI	4	1	0.000546	0.00001671	0.22687643	0.227425	3.0863462	77.1587
OMP+MPI	4	2	0.00045271	0.00396043	0.08621357	0.08666814	4.0614363	50.768
OMP+MPI	4	4	0.00044557	0.00338614	0.03338386	0.03383171	5.0892611	31.8079
OMP+MPI	8	1	0.00042229	0.00002657	0.09042657	0.09085057	3.9262447	49.0781
OMP+MPI	8	2	0.00050071	0.00057257	0.03549186	0.03599429	4.8581243	30.3633
OMP+MPI	8	4	0.00046657	0.000958	0.014936	0.01540443	5.6201463	17.563

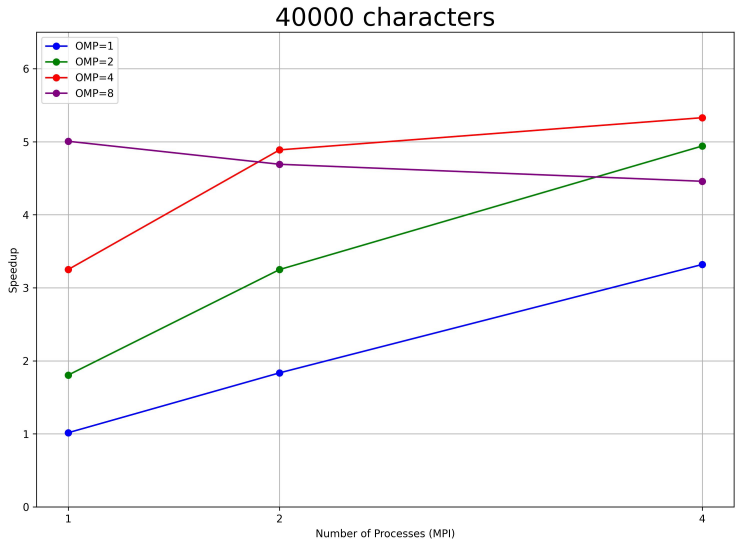
Timing table for 20000 characters.



Speedup plot for 20000 characters.

Modality	OMP	MPI	String_generation_time	Communication_time	Edit_distance_time	Program_execution_time	speedup	efficiency
Approximate	1	1	0.00090271	0	11.18323714	11.18414057	1	100
Approximate	1	2	0.00092243	0	5.55147014	5.55239286	1	100
Approximate	1	4	0.00104686	0	2.82334871	2.82439571	1	100
Approximate	2	1	0.00092943	0	5.49735171	5.49828143	1	100
Approximate	2	2	0.00097543	0	2.75579414	2.75676971	1	100
Approximate	2	4	0.00086943	0	1.39223171	1.39310486	1	100
Approximate	4	1	0.00088086	0	2.74495629	2.74583743	1	100
Approximate	4	2	0.00088829	0	1.39114629	1.392035	1	100
Approximate	4	4	0.00088157	0	0.705406	0.706288	1	100
Approximate	8	1	0.00097071	0	1.40708114	1.408052	1	100
Approximate	8	2	0.000926	0	0.697671	0.69859714	1	100
Approximate	8	4	0.00097357	0	0.34832129	0.34929514	1	100
OMP+MPI	1	1	0.00085957	0.00001214	11.013763	11.01462471	1.0153901	101.539
OMP+MPI	1	2	0.00099243	0.00658257	3.02458814	3.02558286	1.8351482	91.7574
OMP+MPI	1	4	0.00103714	0.01434643	0.84997286	0.85101186	3.3188676	82.9717
OMP+MPI	2	1	0.000927	0.00001629	3.04580429	3.04673314	1.8046482	90.2324
OMP+MPI	2	2	0.00102586	0.01323	0.84737071	0.84839886	3.2493793	81.2345
OMP+MPI	2	4	0.00109643	0.01349514	0.28086343	0.28196171	4.9407589	61.7595
OMP+MPI	4	1	0.000889	0.00001943	0.84405671	0.84494743	3.2497139	81.2428
OMP+MPI	4	2	0.00087386	0.003488	0.28394143	0.28481714	4.8874692	61.0934
OMP+MPI	4	4	0.00104057	0.002423	0.13151357	0.13255586	5.3282293	33.3014
OMP+MPI	8	1	0.00083629	0.00001943	0.28045971	0.28129757	5.0055605	62.5695
OMP+MPI	8	2	0.000978	0.00170186	0.14795543	0.148937	4.6905547	29.316
OMP+MPI	8	4	0.00113929	0.00958714	0.07721786	0.07835871	4.4576426	13.9301

Timing table for 40000 characters.

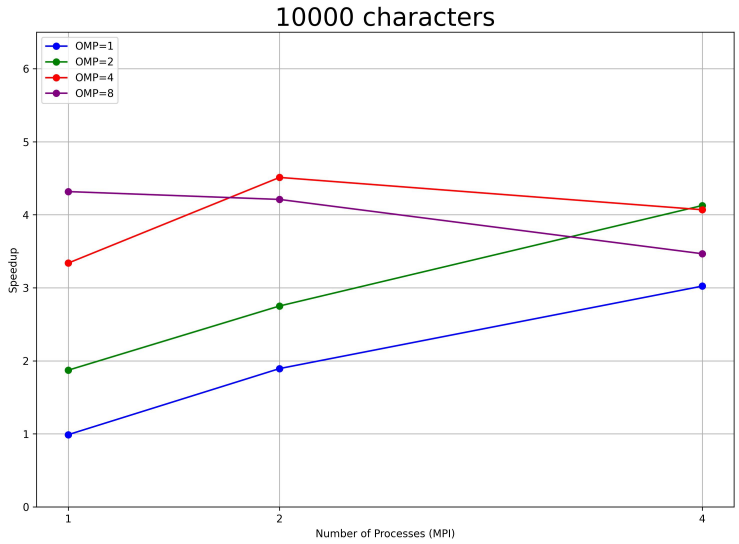


Speedup plot for 40000 characters.

1.4.4 Optimization 2

Modality	OMP	MPI	String_generation_time	Communication_time	Edit_distance_time	Program_execution_time	speedup	efficiency
Approximate	1	1	0.00026029	0	0.62356371	0.62382443	1	100
Approximate	1	2	0.00024443	0	0.31988214	0.32012657	1	100
Approximate	1	4	0.000226	0	0.154749	0.154975	1	100
Approximate	2	1	0.00024443	0	0.31466629	0.31491071	1	100
Approximate	2	2	0.00026986	0	0.15336371	0.15363357	1	100
Approximate	2	4	0.00022	0	0.07742457	0.07764457	1	100
Approximate	4	1	0.00025686	0	0.15318529	0.15344243	1	100
Approximate	4	2	0.00024786	0	0.07863943	0.07888743	1	100
Approximate	4	4	0.00023943	0	0.03940329	0.03964314	1	100
Approximate	8	1	0.00023571	0	0.07818471	0.07842043	1	100
Approximate	8	2	0.00025643	0	0.04024671	0.04050314	1	100
Approximate	8	4	0.00024786	0	0.017572	0.01782	1	100
OMP+MPI	1	1	0.00021329	0.00001586	0.63076143	0.63097486	0.9886677	98.8668
OMP+MPI	1	2	0.00024429	0.00077943	0.16880057	0.169045	1.8937358	94.6868
OMP+MPI	1	4	0.00028986	0.00346686	0.050986	0.05127586	3.0223776	75.5594
OMP+MPI	2	1	0.000268	0.00001429	0.167981	0.16824914	1.8716928	93.5846
OMP+MPI	2	2	0.000255	0.00221529	0.05560114	0.05585643	2.7505083	68.7627
OMP+MPI	2	4	0.000226	0.00379243	0.01859371	0.01881971	4.1257041	51.5713
OMP+MPI	4	1	0.00022743	0.00002043	0.04574357	0.04597129	3.3377885	83.4447
OMP+MPI	4	2	0.000228	0.00156314	0.01726129	0.01748943	4.5105778	56.3822
OMP+MPI	4	4	0.00025514	0.00126671	0.009487	0.00974457	4.0682285	25.4264
OMP+MPI	8	1	0.00023071	0.00003186	0.01793586	0.01816686	4.3166756	53.9584
OMP+MPI	8	2	0.00022271	0.00051414	0.00939957	0.00962257	4.2091808	26.3074
OMP+MPI	8	4	0.00023457	0.00122586	0.00490671	0.00514171	3.4657702	10.8305

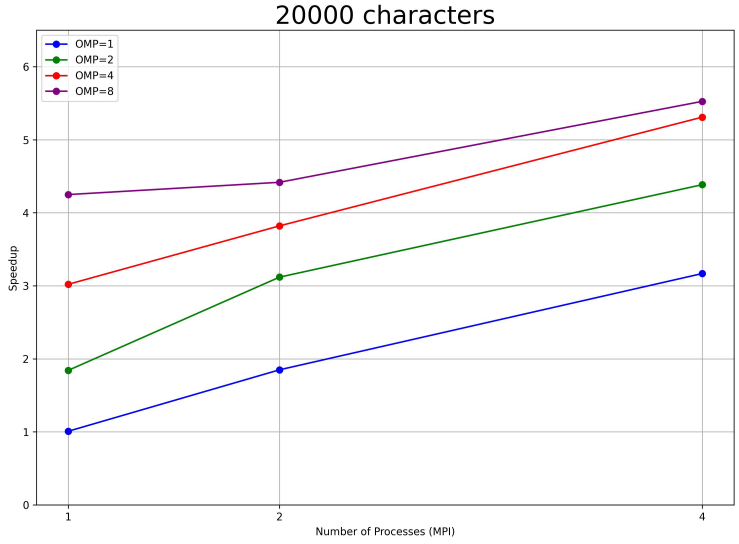
Timing table for 10000 characters.



Speedup plot for 10000 characters.

Modality	OMP	MPI	String_generation_time	Communication_time	Edit_distance_time	Program_execution_time	speedup	efficiency
Approximate	1	1	0.00048814	0	2.51579857	2.51628686	1	100
Approximate	1	2	0.00051886	0	1.259125	1.25964386	1	100
Approximate	1	4	0.00053514	0	0.62161314	0.62214871	1	100
Approximate	2	1	0.00049086	0	1.24840843	1.24889957	1	100
Approximate	2	2	0.000476	0	0.632564	0.63304	1	100
Approximate	2	4	0.000505	0	0.31705314	0.31755843	1	100
Approximate	4	1	0.00043757	0	0.62413686	0.62457443	1	100
Approximate	4	2	0.00044243	0	0.30965971	0.31010214	1	100
Approximate	4	4	0.00045943	0	0.15786886	0.15832843	1	100
Approximate	8	1	0.00048043	0	0.32211457	0.32259514	1	100
Approximate	8	2	0.00047643	0	0.15447514	0.15495157	1	100
Approximate	8	4	0.000434	0	0.07758743	0.07802157	1	100
OMP+MPI	1	1	0.00044957	0.00001114	2.49574529	2.49619486	1.0080491	100.8049
OMP+MPI	1	2	0.00043	0.00192043	0.68063643	0.68106657	1.8495165	92.4758
OMP+MPI	1	4	0.00059043	0.01198057	0.19583929	0.19642971	3.1672841	79.1821
OMP+MPI	2	1	0.00042	0.00001986	0.677754	0.678174	1.8415622	92.0781
OMP+MPI	2	2	0.00044857	0.00355114	0.20257014	0.20301871	3.1181362	77.9534
OMP+MPI	2	4	0.00058371	0.00537129	0.07185457	0.07243857	4.3838306	54.7979
OMP+MPI	4	1	0.00045729	0.00001557	0.20637557	0.206833	3.019704	75.4926
OMP+MPI	4	2	0.00050143	0.00469871	0.08067557	0.081177	3.820074	47.7509
OMP+MPI	4	4	0.00046771	0.00268557	0.029359	0.02982671	5.3082759	33.1767
OMP+MPI	8	1	0.00043343	0.00002257	0.07550214	0.075936	4.2482504	53.1031
OMP+MPI	8	2	0.00047071	0.00070743	0.034614	0.03508486	4.41648	27.603
OMP+MPI	8	4	0.00048543	0.00109457	0.01363714	0.01412271	5.5245451	17.2642

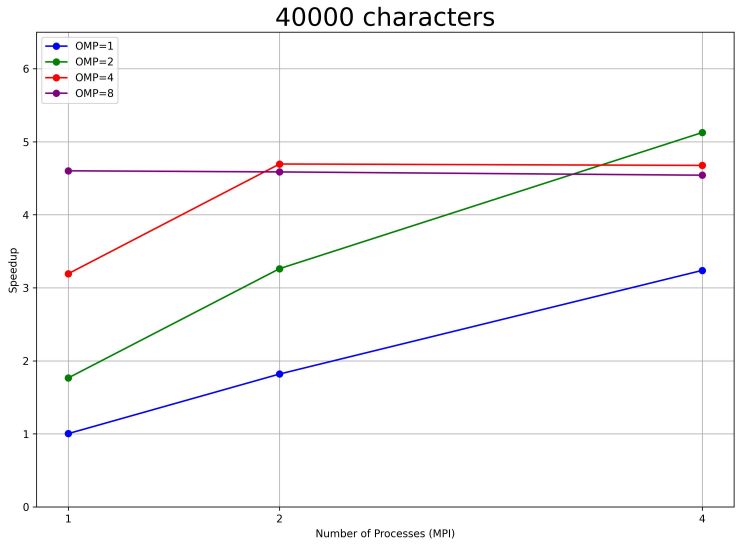
Timing table for 20000 characters.



Speedup plot for 20000 characters.

Modality	OMP	MPI	String_generation_time	Communication_time	Edit_distance_time	Program_execution_time	speedup	efficiency
Approximate	1	1	0.00088971	0	10.00369571	10.00458557	1	100
Approximate	1	2	0.00096329	0	4.96913957	4.97010286	1	100
Approximate	1	4	0.00118343	0	2.47851286	2.47969671	1	100
Approximate	2	1	0.00091843	0	4.92314886	4.92406743	1	100
Approximate	2	2	0.00085543	0	2.48664486	2.48750043	1	100
Approximate	2	4	0.00106343	0	1.272213	1.27327643	1	100
Approximate	4	1	0.00087343	0	2.50954243	2.510416	1	100
Approximate	4	2	0.00090471	0	1.24834171	1.24924643	1	100
Approximate	4	4	0.00087571	0	0.62507314	0.625949	1	100
Approximate	8	1	0.00084857	0	1.24787614	1.248725	1	100
Approximate	8	2	0.00086686	0	0.63318443	0.63405129	1	100
Approximate	8	4	0.00086843	0	0.31045457	0.31132314	1	100
OMP+MPI	1	1	0.00085671	0.00001471	9.97317271	9.97402943	1.0030636	100.3064
OMP+MPI	1	2	0.000945	0.00162714	2.731277	2.73222214	1.8190698	90.9535
OMP+MPI	1	4	0.00113371	0.02476129	0.76471329	0.76584714	3.2378481	80.9462
OMP+MPI	2	1	0.00087014	0.000028	2.78837743	2.78924757	1.7653748	88.2687
OMP+MPI	2	2	0.00087029	0.01873314	0.761798	0.76266843	3.2615752	81.5394
OMP+MPI	2	4	0.001092	0.00636771	0.24734286	0.248435	5.1251894	64.0649
OMP+MPI	4	1	0.00087386	0.00003529	0.78558371	0.78645771	3.1920547	79.8014
OMP+MPI	4	2	0.00094443	0.00651371	0.265177	0.26612171	4.6942672	58.6783
OMP+MPI	4	4	0.00118514	0.014195	0.13270657	0.13389186	4.6750341	29.219
OMP+MPI	8	1	0.00090829	0.00001986	0.27048543	0.27139386	4.6011543	57.5144
OMP+MPI	8	2	0.00089329	0.00349771	0.13737143	0.13826471	4.585778	28.6611
OMP+MPI	8	4	0.00116914	0.01379843	0.06739086	0.06856029	4.5408671	14.1902

Timing table for 40000 characters.

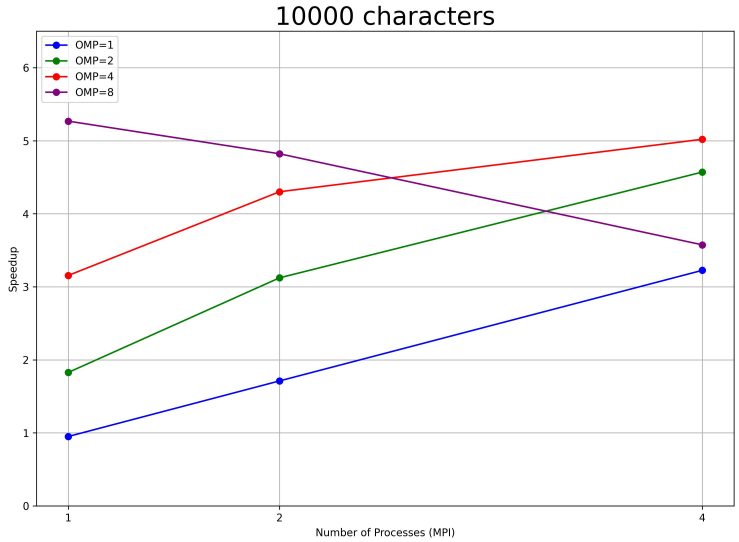


Speedup plot for 40000 characters.

1.4.5 Optimization 3

Modality	OMP	MPI	String_generation_time	Communication_time	Edit_distance_time	Program_execution_time	speedup	efficiency
Approximate	1	1	0.00026343	0	0.57900186	0.57926543	1	100
Approximate	1	2	0.00021643	0	0.29128743	0.291504	1	100
Approximate	1	4	0.00023614	0	0.14713886	0.14737514	1	100
Approximate	2	1	0.000242	0	0.29189114	0.29213329	1	100
Approximate	2	2	0.00024929	0	0.14989114	0.15014057	1	100
Approximate	2	4	0.00022986	0	0.07280129	0.07303114	1	100
Approximate	4	1	0.00022857	0	0.14447386	0.14470271	1	100
Approximate	4	2	0.00021014	0	0.07171871	0.071929	1	100
Approximate	4	4	0.00025214	0	0.04170486	0.041957	1	100
Approximate	8	1	0.00022629	0	0.07398029	0.07420671	1	100
Approximate	8	2	0.00023943	0	0.03699557	0.03723543	1	100
Approximate	8	4	0.00024057	0	0.01666243	0.016903	1	100
OMP+MPI	1	1	0.00022657	0.00001429	0.60953657	0.60976329	0.9499841	94.9984
OMP+MPI	1	2	0.00022286	0.00121029	0.17013443	0.17035786	1.7111274	85.5564
OMP+MPI	1	4	0.00033471	0.00086986	0.045366	0.04570114	3.2247584	80.619
OMP+MPI	2	1	0.00024714	0.00002671	0.15968557	0.15993286	1.8265996	91.33
OMP+MPI	2	2	0.00021371	0.002419	0.047872	0.04808586	3.1223437	78.0586
OMP+MPI	2	4	0.00026171	0.00261614	0.015719	0.01598071	4.5699549	57.1244
OMP+MPI	4	1	0.00021986	0.00003914	0.04565114	0.045871	3.1545577	78.8639
OMP+MPI	4	2	0.00023071	0.00053357	0.016494	0.01672471	4.3007611	53.7595
OMP+MPI	4	4	0.000267	0.00174171	0.00809214	0.00835943	5.019123	31.3695
OMP+MPI	8	1	0.00023414	0.00001957	0.01385371	0.01408786	5.2674238	65.8428
OMP+MPI	8	2	0.00022114	0.000148	0.00750343	0.00772457	4.8203876	30.1274
OMP+MPI	8	4	0.00025871	0.00029543	0.00447071	0.00472957	3.5738968	11.1684

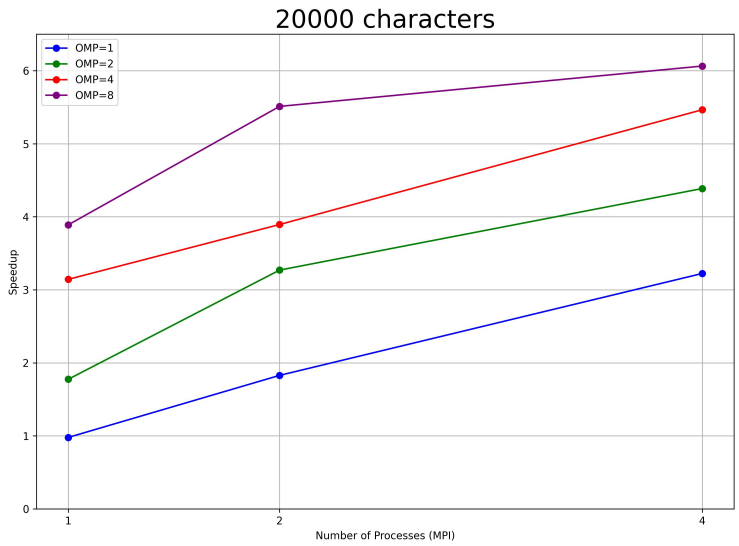
Timing table for 10000 characters.



Speedup plot for 10000 characters.

Modality	OMP	MPI	String_generation_time	Communication_time	Edit_distance_time	Program_execution_time	speedup	efficiency
Approximate	1	1	0.00048643	0	2.31495129	2.31543771	1	100
Approximate	1	2	0.000519	0	1.16186757	1.16238671	1	100
Approximate	1	4	0.00054314	0	0.5867	0.58724314	1	100
Approximate	2	1	0.00044014	0	1.14578943	1.14623014	1	100
Approximate	2	2	0.00045886	0	0.58233129	0.58279043	1	100
Approximate	2	4	0.00048543	0	0.29396043	0.29444586	1	100
Approximate	4	1	0.000487	0	0.57863486	0.57912214	1	100
Approximate	4	2	0.00043486	0	0.29656629	0.29700143	1	100
Approximate	4	4	0.00046371	0	0.14698129	0.147445	1	100
Approximate	8	1	0.00043314	0	0.291118	0.29155143	1	100
Approximate	8	2	0.000451	0	0.14907243	0.14952343	1	100
Approximate	8	4	0.00045414	0	0.07457314	0.07502743	1	100
OMP+MPI	1	1	0.00041929	0.00001471	2.36658029	2.36699971	0.9782163	97.8216
OMP+MPI	1	2	0.00045571	0.00261486	0.63552343	0.63597914	1.827712	91.3856
OMP+MPI	1	4	0.000571	0.00172771	0.18163743	0.18220857	3.2229172	80.5729
OMP+MPI	2	1	0.00051829	0.00001614	0.64547171	0.64599043	1.7743764	88.7188
OMP+MPI	2	2	0.00045671	0.00054729	0.17781757	0.17827443	3.2690635	81.7266
OMP+MPI	2	4	0.00058914	0.007844	0.06654471	0.067134	4.3859424	54.8243
OMP+MPI	4	1	0.00044586	0.00001943	0.18384657	0.18429243	3.1424088	78.5602
OMP+MPI	4	2	0.00045871	0.00850029	0.07583071	0.07628943	3.8930876	48.6636
OMP+MPI	4	4	0.00045586	0.00174457	0.02652643	0.02698229	5.4645111	34.1532
OMP+MPI	8	1	0.00042471	0.00002571	0.074554	0.07497929	3.8884263	48.6053
OMP+MPI	8	2	0.00042343	0.00032557	0.02671114	0.02713471	5.510411	34.4401
OMP+MPI	8	4	0.00045943	0.00180257	0.011916	0.01237543	6.0626126	18.9457

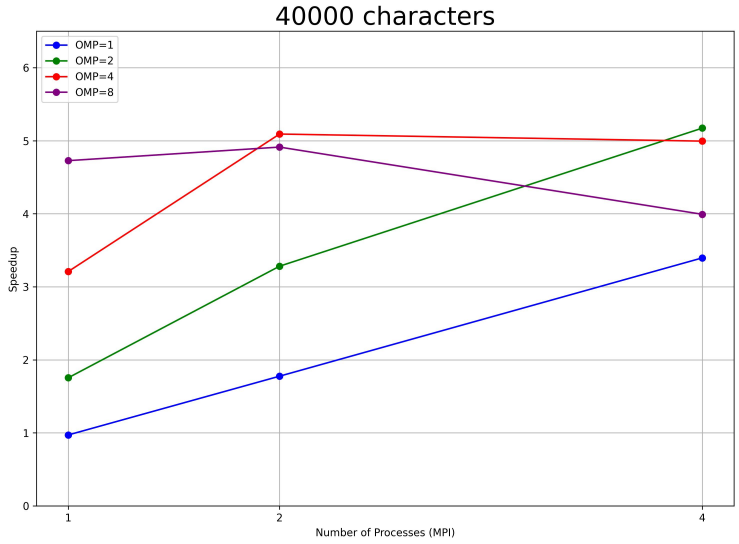
Timing table for 20000 characters.



Speedup plot for 20000 characters.

Modality	OMP	MPI	String_generation_time	Communication_time	Edit_distance_time	Program_execution_time	speedup	efficiency
Approximate	1	1	0.00084943	0	9.25390543	9.25475514	1	100
Approximate	1	2	0.00099686	0	4.58611414	4.587111	1	100
Approximate	1	4	0.000925	0	2.33192443	2.33285	1	100
Approximate	2	1	0.00088729	0	4.55759871	4.55848629	1	100
Approximate	2	2	0.00098329	0	2.32804943	2.32903271	1	100
Approximate	2	4	0.00093014	0	1.17033929	1.17126971	1	100
Approximate	4	1	0.00088414	0	2.31267386	2.31355829	1	100
Approximate	4	2	0.00085443	0	1.17493357	1.175788	1	100
Approximate	4	4	0.00093814	0	0.58289929	0.58383743	1	100
Approximate	8	1	0.00094314	0	1.15418029	1.15512357	1	100
Approximate	8	2	0.00086843	0	0.583638	0.58450671	1	100
Approximate	8	4	0.00089743	0	0.29164071	0.29253829	1	100
OMP+MPI	1	1	0.00090214	0.00001443	9.53036643	9.53126871	0.9709888	97.0989
OMP+MPI	1	2	0.00089114	0.00847714	2.58231043	2.58320171	1.7757463	88.7873
OMP+MPI	1	4	0.00108386	0.00788971	0.68603743	0.68712143	3.3951059	84.8776
OMP+MPI	2	1	0.00088971	0.00001786	2.59852529	2.59941557	1.7536581	87.6829
OMP+MPI	2	2	0.00085943	0.002601	0.70888814	0.70974757	3.2814945	82.0374
OMP+MPI	2	4	0.00120243	0.00916429	0.22527657	0.226479	5.1716482	64.6456
OMP+MPI	4	1	0.00083757	0.00002557	0.72031671	0.72115714	3.2081195	80.203
OMP+MPI	4	2	0.00098814	0.00268357	0.22998471	0.23097286	5.0905895	63.6324
OMP+MPI	4	4	0.00109571	0.01219443	0.11581714	0.116913	4.9937768	31.2111
OMP+MPI	8	1	0.00084314	0.00001571	0.243557	0.24440029	4.7263593	59.0795
OMP+MPI	8	2	0.00087029	0.00275957	0.11812229	0.11899329	4.9120983	30.7006
OMP+MPI	8	4	0.00124414	0.01509443	0.07203657	0.07328114	3.9919995	12.475

Timing table for 40000 characters.



Speedup plot for 40000 characters.

1.4.6 Final Considerations

After a careful analysis of all optimizations, we can observe that, in general, the **trends** identified for optimization 0 have more or less **remained unchanged**.

However, it is important to note that with an **increase in optimizations**, the overall **times have significantly improved**. Having **increased for both** the sequential and parallel versions, in the end, the **speedups are quite comparable**. With more iterations, they would likely have been even closer.

1.5 OMP+CUDA Parallelization

CUDA, or **Compute Unified Device Architecture**, is a parallel computing platform and application programming interface model created by NVIDIA. It is designed to harness the power of Graphics Processing Units (GPUs) for general-purpose processing, enabling significant acceleration of various computational workloads.

At its core, CUDA provides a programming model that allows developers to offload **parallelizable tasks to the GPU**, taking advantage of the parallel processing capabilities inherent in these devices. This is particularly useful for computationally intensive applications such as scientific simulations, image and signal processing, machine learning, and more. One of the main features of CUDA is that it allows developers to divide their tasks into parallel threads that can be executed concurrently on the GPU. This is achieved through the use of CUDA kernels, which are functions that are executed in parallel on the GPU.

The GPU consists of numerous **CUDA cores**, which are small processing units that work in parallel to execute the tasks specified in the CUDA kernels. This parallelism significantly accelerates computation compared to traditional sequential execution on CPUs.

1.5.1 GPU Specifications

Max regs number/SM	: 65536
GPU Name	: NVIDIA GeForce RTX 2060 with Max-Q Design
CUDA Cores	: 30
Global Memory	: 6.00 GB
Shared Memory/block	: 48.00 KB
Max Threads/block	: 1024
Max Blocks/SM	: 16
Max Blocks Dim.	: 1024 x 1024 x 64
Max Grid Dim.	: 2147483647 x 65535 x 65535
Max SMs	: 30
Max Thread Grid Dim.	: 2147483647 x 65535 x 65535
Max Thread Block Dim.	: 1024 x 1024 x 64
Kernel Exec. Capacity	: 7.5

1.5.2 OMP+CUDA approach

It might be thought that using OMP and CUDA together is **not possible**, even though OMP is designed primarily for parallelizing code on **multicore CPUs**, and CUDA is designed for parallelizing on the **GPU**. However, it is precisely on these distinctive characteristics that the **combined approach will be developed**.

The idea is to use the **same approach** designed for OMP+MPI but implemented **slightly differently**. The size of the problem, in this case, the two strings, will be **divided in two**: the **first half will remain on the CPU**, and OMP threads will handle parallelization on the host, while the **second half will be sent to the device**, which will work in parallel exploiting the capabilities of the kernel on the GPU.

Unlike the OMP+MPI approach, there won't be any process dividing the workload here, but it will be divided before the GPU and CPU work independently and operate the **edit distance on the assigned halves**. To achieve this, we need to use CUDA APIs such as **cudaMalloc()**, which, when executed on the host, **allocates a portion of memory** based on the working data structure, depending on the quantity of elements that will be addressed to the GPU.

Here is the function signature:

```
cudaError_t cudaMalloc(void** devPtr, size_t size);
```

Additionally, we require **cudaMemcpy()** to copy half of the elements from the host to a certain data structure, in another structure allocated with cudaMalloc().

Here's the function signature:

```
cudaError_t cudaMemcpy(void* dst, const void* src,size_t count,
    cudaMemcpyKind kind);
```

The situation here, however, is a bit more **delicate** compared to the OMP+MPI approach. While with OMP, we can apply the same methodology used for combined parallelization with MPI, with CUDA, we need to slightly modify the approach because **CUDA threads** come into play, participating in parallelization on the GPU.

The CUDA threads are organized in blocks executed by **Streaming multiprocessors**. Each GPU has its configuration and its maximum streaming multiprocessors simultaneously active on GPU and consequently the maximum amount of blocks assignable for each streaming multiprocessor.

Since we want to assign to each thread a portion of elements to perform the edit distance we need to know the max number of thread for each block assignable for a single streaming multiprocessor as well known as **BlockSize**.

The block size, denoted by **blockDim** represents the number of threads within a block. Threads within a block can cooperate and share data using fast on-chip shared memory. A block

is the basic unit of execution on a GPU, and all threads within a block can synchronize and communicate with each other.

Associated with the block size, there is the **GridSize** denoted by **gridDim** represents the number of blocks in the grid. The grid is the overall set of blocks that execute a kernel on the GPU. The combination of block size and grid size determines the total number of threads launched on the GPU.

The gridsize, along with the block size, infleunces the overall parallelism and scalability of a CUDA program.
Both blocks and grid can work up to three dimensions specifying the **dim3** type when declaring blockSize and gridSize.
We can determine the gridSize as follows:

```
dim gridSize((N-1)/blockSize.x + 1)
```

Since the gridSize depends on the blockSize, it is more important understand **how many threads we can set into a block**, so it is important to set the blockSize.

In my GPU the maximum number of registers per SM is **65536** so this means that the threads within the block in the SM can use a maximum of this number of registers. To determine the maximum number of registers that a single thread uses to execute the kernel, we need to use a compilation option, namely **–ptxas-options=v**.

Analyzing the kernel, each individual thread uses a maximum of **20 registers**. Therefore, if we calculate a simple ratio, namely **65536/20**, we would find that there should be approximately **3277** threads in each block. However, this is not feasible because the maximum number of threads is indeed **1024**, and it is also not possible to use more blocks per SM because there can be at most 1 block per SM. As a result, we choose the maximum size for the blockSize, so **blockSize = 1024**.

Furthermore, by using this calculatorand inputting the correct information derived from the characteristics of the GPU mentioned above and the number of registers each thread uses to execute the kernel, we see that the GPU is fully utilized, reaching 100% of its potential.

1.5.3 CUDA Kernel

After determining the maximum number of threads that can work simultaneously on a single streaming multiprocessor to execute the kernel, let's return to the discussion of fair element distribution.

Since each thread should not work on a single character of the strings but on multiple characters to avoid too uncontrollable an error, it is crucial to determine how many elements each thread will handle. Before delving into the actual division and before calling the kernel, it is necessary, therefore, to use a scaling factor to reduce the grid size, ensuring that all 1024 threads receive a significant portion of the string. This is because, without introducing this scaling factor, each thread would operate on only one character, making the error uncontrollable. If we increase this factor, the error decreases, but we would use fewer threads, while decreasing it increases the error but increases the number of threads used. What we will use is a balance between the two.

After this, we can call the kernel, passing a pointer to contain the result (also allocated in GPU memory using cudaMalloc()) since the kernel, by definition, returns VOID. The kernel has a section where, taking the thread index from the **int i = blockIdx.x * blockDim.x + threadIdx.x;** relationship, a certain number of elements, a starting index, and an ending index are assigned to this thread. Additionally, there are additional lines of code that further distribute the workload more evenly, as the last thread might work not only on the elements assigned to it but also on another large set of elements.

By doing this, each thread, once the portion of the string passed to the GPU is taken, its partition can be computed, and the edit distance can be calculated, then summing up all the local results.

1.5.4 OMP approach

Since the kernel invocation is asynchronous, it means that while the GPU is being called, the **CPU continues to work** and execute the subsequent instructions. This situation can be leveraged by simultaneously working on the host, conducting the search on the first half of the array thanks to the parallelization achieved using OMP. Even with OMP, we perform the same data division based on the number of available OMP threads set using the **omp_set_num_threads()** function. Thus, there will be a period during which both the host and the device work in parallel by invoking the same function defined as **__host__ __device__**.

This allows it to be called by both the CPU and the GPU since they need to execute the edi distance algorithm. Additionally, there are no concurrency issues because the elements are read-only, and a specific quantity of elements will be seen by one and only one thread.

When both the host and the device have completed their executions, all that remains is to s**um up all the local results** to obtain the total distance.

1.6 Analysis

1.6.1 OMP Threads and L1 Cache

Essentially, we perform the same analyses as done for the OMP+MPI version. In this case as well, we have a sequential program that divides the data in the same way we will use for this version.

Additionally, we will leverage a CUDA function, namely **cudaFuncSetCacheConfig()**:

```
cudaError_t cudaFuncSetCacheConfig(const void* func, enum cudaFuncCache
    cacheConfig);
```

This function, if the cacheConfig parameter is specified with **cudaFuncCachePreferL1** allows assigning more memory to the L1 cache present in global memory. We tell the CUDA system to prefer L1 cache as memory to optimize data access during the execution of the associated CUDA kernel. This cache is a level 1 cache (as the name suggests) and is located directly on the GPU chip. Since this cache is small, it is significantly faster, and this function allows telling the CUDA compiler to try to maximize the efficiency of using the L1 cache to store and retrieve data. It is clear, however, that the effect of cache preference depends on the specific GPU in use and the nature of the kernel.

In this case, the only thing that will vary is the number of OMP threads.

1.6.2 Error, Performance, Speed-Up & Efficiency

The indicators we will use are essentially the same, as in this case as well, we have a reference sequential version that partitions the data in the same way.

1.7 OMP+CUDA Analysis of the plots

1.7.1 Error

10000 Characters

Modality	OMP	Cuda	ED
Sequential	0	0	8758
Approximate	1	1024	9169
OMP+CUDA	1	1024	9169
OMP+CUDA_L1	1	1024	9169
Approximate	2	1024	9182
OMP+CUDA	2	1024	9182
OMP+CUDA_L1	2	1024	9182
Approximate	4	1024	9197
OMP+CUDA	4	1024	9197
OMP+CUDA_L1	4	1024	9197
Approximate	8	1024	9208
OMP+CUDA	8	1024	9208
OMP+CUDA_L1	8	1024	9208

Error for 10000 characters.

20000 Characters

Modality	OMP	Cuda	ED
Sequential	0	0	17497
Approximate	1	1024	18277
OMP+CUDA	1	1024	18277
OMP+CUDA_L1	1	1024	18277
Approximate	2	1024	18287
OMP+CUDA	2	1024	18287
OMP+CUDA_L1	2	1024	18287
Approximate	4	1024	18306
OMP+CUDA	4	1024	18306
OMP+CUDA_L1	4	1024	18306
Approximate	8	1024	18345
OMP+CUDA	8	1024	18345
OMP+CUDA_L1	8	1024	18345

Error for 2000 characters.

40000 Characters

Modality	OMP	Cuda	ED
Sequential	0	0	34979
Approximate	1	1024	36288
OMP+CUDA	1	1024	36288
OMP+CUDA_L1	1	1024	36288
Approximate	2	1024	36305
OMP+CUDA	2	1024	36305
OMP+CUDA_L1	2	1024	36305
Approximate	4	1024	36323
OMP+CUDA	4	1024	36323
OMP+CUDA_L1	4	1024	36323
Approximate	8	1024	36345
OMP+CUDA	8	1024	36345
OMP+CUDA_L1	8	1024	36345

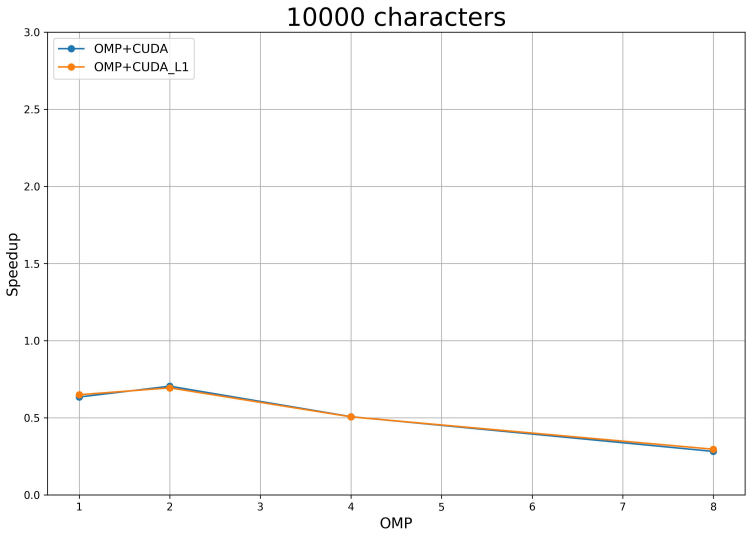
Error for 4000 characters.

Like before, we notice that the results between the parallel version and the approximate sequential version are **the same**. As easy to imagine, the **error increases** a bit since the number of **partitions is very high**. CUDA is advantageous to use only for **truly large workloads**, so the error certainly decreases in such cases.

1.7.2 Optimization 0

Modality	OMP	BlockSize	String_generation_time	Kernel_edit_distance_time	Program_execution_time	speedup	efficiency
Approximate	1	1024	0.00028671	0.25414014	0.25444086	1	100
Approximate	2	1024	0.00028386	0.12876571	0.12906443	1	100
Approximate	4	1024	0.00026529	0.06570043	0.06597957	1	100
Approximate	8	1024	0.00027043	0.03354057	0.033825	1	100
OMP+CUDA	1	1024	0	0.00429714	0.401	0.6345159	63.4516
OMP+CUDA	2	1024	0	0.00440943	0.18314286	0.70472	35.236
OMP+CUDA	4	1024	0	0.00439271	0.13014286	0.5069781	12.6745
OMP+CUDA	8	1024	0	0.00439414	0.12	0.281875	3.5234
OMP+CUDA_L1	1	1024	0	0.00434929	0.39185714	0.6493205	64.932
OMP+CUDA_L1	2	1024	0	0.00428	0.186	0.6938948	34.6947
OMP+CUDA_L1	4	1024	0	0.00431157	0.13042857	0.5058675	12.6467
OMP+CUDA_L1	8	1024	0	0.00433457	0.11428571	0.2959688	3.6996

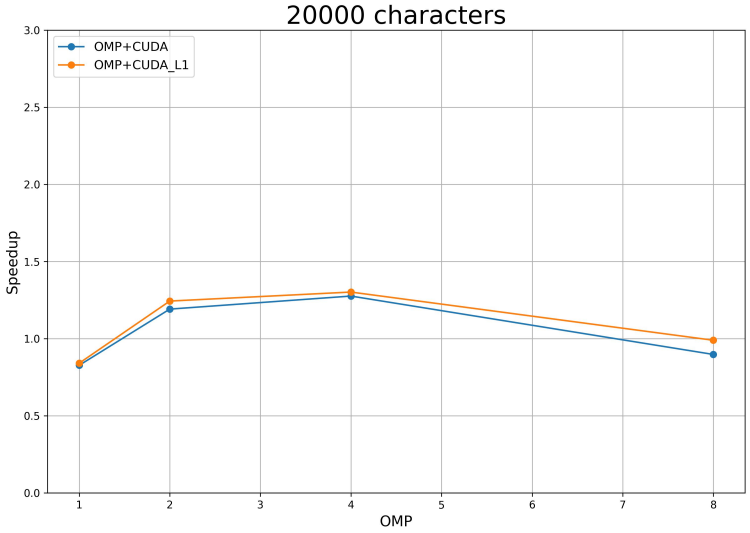
Timing table for 10000 characters.



Speedup plot for 10000 characters.

Modality	OMP	BlockSize	String_generation_time	Kernel_edit_distance_time	Program_execution_time	speedup	efficiency
Approximate	1	1024	0.00060971	1.03452571	1.03515743	1	100
Approximate	2	1024	0.00053314	0.51124614	0.51179843	1	100
Approximate	4	1024	0.00054357	0.25553886	0.25610529	1	100
Approximate	8	1024	0.00053443	0.13189671	0.132453	1	100
OMP+CUDA	1	1024	0.00014286	0.00871157	1.25142857	0.8271806	82.7181
OMP+CUDA	2	1024	0.00042857	0.00816029	0.42957143	1.1914164	59.5708
OMP+CUDA	4	1024	0	0.00847271	0.20071429	1.2759694	31.8992
OMP+CUDA	8	1024	0.00014286	0.00844186	0.14757143	0.8975518	11.2194
OMP+CUDA_L1	1	1024	0	0.00847171	1.23114286	0.8408102	84.081
OMP+CUDA_L1	2	1024	0	0.00868386	0.41157143	1.2435227	62.1761
OMP+CUDA_L1	4	1024	0	0.00875471	0.19671429	1.301915	32.5479
OMP+CUDA_L1	8	1024	0	0.008624	0.13385714	0.9895101	12.3689

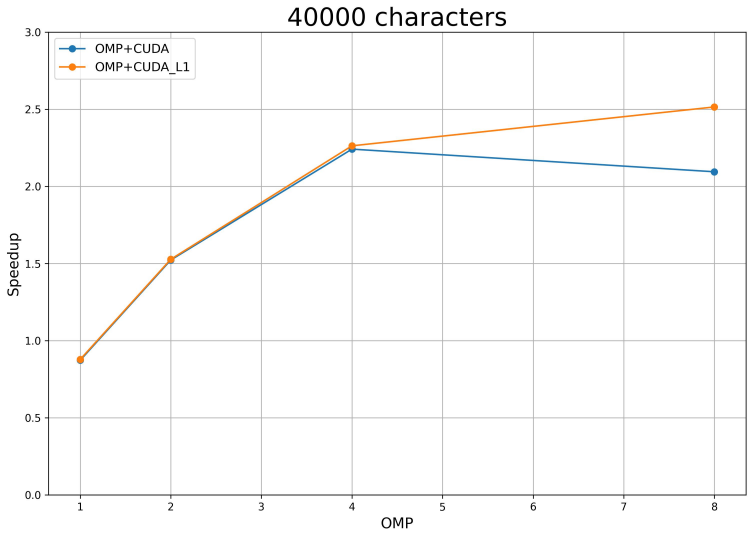
Timing table for 20000 characters.



Speedup plot for 20000 characters.

Modality	OMP	BlockSize	String_generation_time	Kernel_edit_distance_time	Program_execution_time	speedup	efficiency
Approximate	1	1024	0.00107857	4.09047671	4.091615	1	100
Approximate	2	1024	0.00111914	2.07013971	2.07131243	1	100
Approximate	4	1024	0.00104114	1.04984886	1.05093071	1	100
Approximate	8	1024	0.00106186	0.52377329	0.52487657	1	100
OMP+CUDA	1	1024	0.00114286	0.039596	4.68871429	0.8726518	87.2652
OMP+CUDA	2	1024	0.001	0.03938157	1.36028571	1.522704	76.1352
OMP+CUDA	4	1024	0.00114286	0.03911157	0.46885714	2.2414732	56.0368
OMP+CUDA	8	1024	0.001	0.038715	0.25057143	2.0947184	26.184
OMP+CUDA_L1	1	1024	0.001	0.03953529	4.65857143	0.8782982	87.8298
OMP+CUDA_L1	2	1024	0.001	0.03766657	1.356	1.5275165	76.3758
OMP+CUDA_L1	4	1024	0.001	0.03933443	0.46442857	2.2628468	56.5712
OMP+CUDA_L1	8	1024	0.001	0.03917386	0.20871429	2.514809	31.4351

Timing table for 40000 characters.

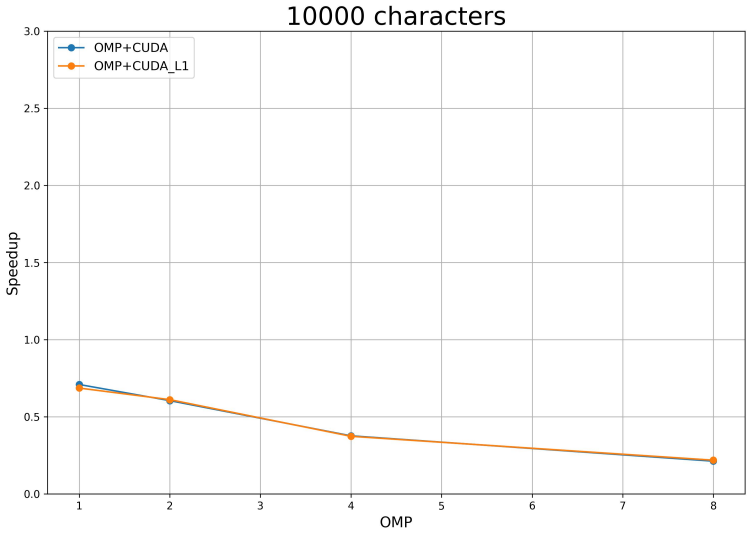


Speedup plot for 40000 characters.

1.7.3 Optimization 1

Modality	OMP	BlockSize	String_generation_time	Kernel_edit_distance_time	Program_execution_time	speedup	efficiency
Approximate	1	1024	0.00023357	0.17959714	0.17984386	1	100
Approximate	2	1024	0.00021129	0.08890086	0.089124	1	100
Approximate	4	1024	0.00025914	0.04339529	0.04367129	1	100
Approximate	8	1024	0.00023971	0.02459343	0.02484671	1	100
OMP+CUDA	1	1024	0	0.00454671	0.25357143	0.7092434	70.9243
OMP+CUDA	2	1024	0	0.00438943	0.14742857	0.6045233	30.2262
OMP+CUDA	4	1024	0	0.00440486	0.116	0.3764766	9.4119
OMP+CUDA	8	1024	0	0.00438771	0.11714286	0.2121061	2.6513
OMP+CUDA_L1	1	1024	0	0.00427086	0.26214286	0.6860529	68.6053
OMP+CUDA_L1	2	1024	0	0.00430371	0.14585714	0.6110362	30.5518
OMP+CUDA_L1	4	1024	0	0.004291	0.117	0.3732589	9.3315
OMP+CUDA_L1	8	1024	0	0.004322	0.11371429	0.2185013	2.7313

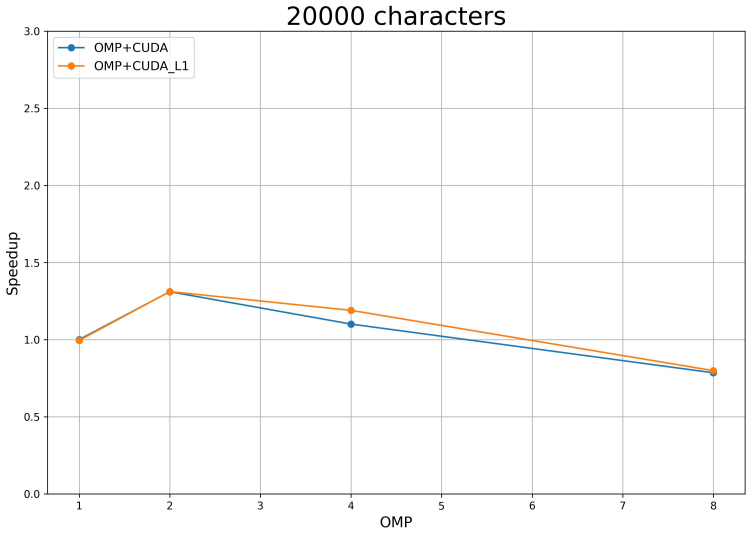
Timing table for 10000 characters.



Speedup plot for 10000 characters.

Modality	OMP	BlockSize	String_generation_time	Kernel_edit_distance_time	Program_execution_time	speedup	efficiency
Approximate	1	1024	0.000447	0.70133629	0.70180843	1	100
Approximate	2	1024	0.00047857	0.34700086	0.347501	1	100
Approximate	4	1024	0.000446	0.17598614	0.17645457	1	100
Approximate	8	1024	0.000443	0.09622186	0.096688	1	100
OMP+CUDA	1	1024	0	0.00844886	0.70128571	1.0007454	100.0745
OMP+CUDA	2	1024	0	0.00872486	0.26514286	1.310618	65.5309
OMP+CUDA	4	1024	0	0.00885343	0.16028571	1.1008752	27.5219
OMP+CUDA	8	1024	0	0.00840529	0.12314286	0.7851694	9.8146
OMP+CUDA_L1	1	1024	0	0.008564	0.70514286	0.9952713	99.5271
OMP+CUDA_L1	2	1024	0	0.00851043	0.265	1.3113245	65.5662
OMP+CUDA_L1	4	1024	0	0.00879729	0.14828571	1.1899634	29.7491
OMP+CUDA_L1	8	1024	0	0.00855343	0.121	0.7990744	9.9884

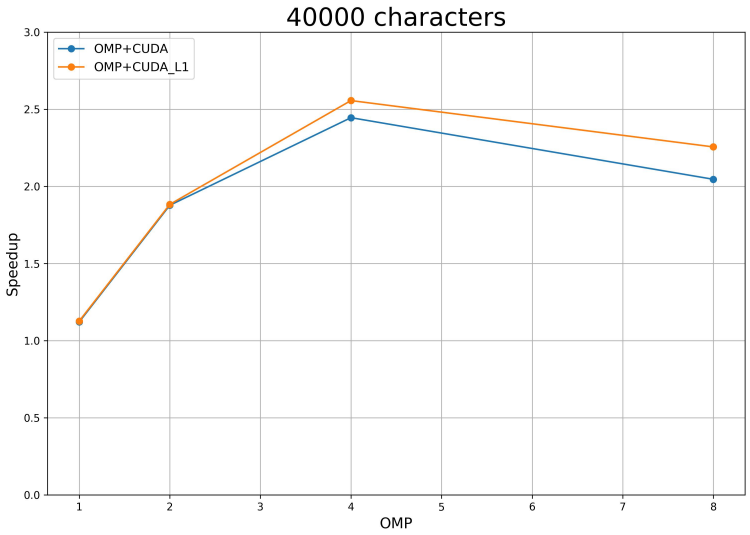
Timing table for 20000 characters.



Speedup plot for 20000 characters.

Modality	OMP	BlockSize	String_generation_time	Kernel_edit_distance_time	Program_execution_time	speedup	efficiency
Approximate	1	1024	0.00091257	2.78017729	2.78113186	1	100
Approximate	2	1024	0.00087071	1.38573757	1.38664843	1	100
Approximate	4	1024	0.000913	0.69946343	0.70042586	1	100
Approximate	8	1024	0.00086429	0.36036014	0.36126457	1	100
OMP+CUDA	1	1024	0.001	0.03949171	2.48085714	1.1210367	112.1037
OMP+CUDA	2	1024	0.001	0.03879929	0.73857143	1.8774737	93.8737
OMP+CUDA	4	1024	0.001	0.03815014	0.28642857	2.4453771	61.1344
OMP+CUDA	8	1024	0.001	0.03884986	0.17657143	2.0459968	25.575
OMP+CUDA_L1	1	1024	0.001	0.03879957	2.468	1.1268768	112.6877
OMP+CUDA_L1	2	1024	0.001	0.03874529	0.73628571	1.8833021	94.1651
OMP+CUDA_L1	4	1024	0.001	0.03877243	0.274	2.5562988	63.9075
OMP+CUDA_L1	8	1024	0.00114286	0.03993971	0.16014286	2.2558894	28.1986

Timing table for 40000 characters.

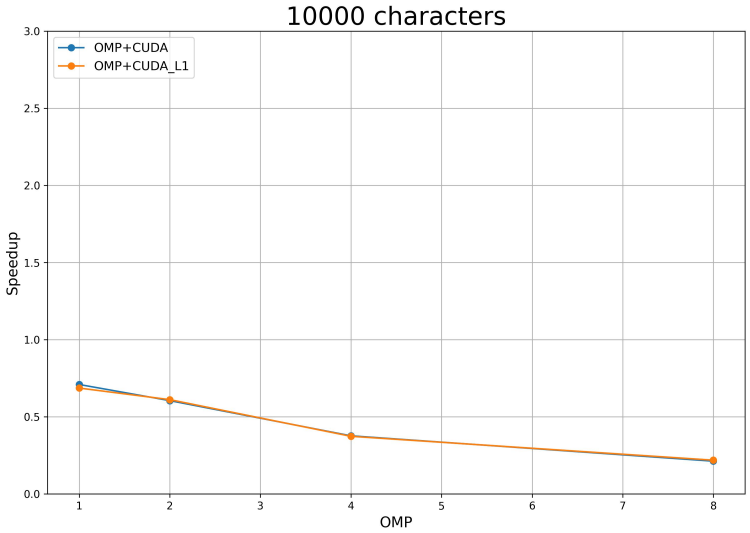


Speedup plot for 40000 characters.

1.7.4 Optimization 2

Modality	OMP	BlockSize	String_generation_time	Kernel_edit_distance_time	Program_execution_time	speedup	efficiency
Approximate	1	1024	0.00023357	0.17959714	0.17984386	1	100
Approximate	2	1024	0.00021129	0.08890086	0.089124	1	100
Approximate	4	1024	0.00025914	0.04339529	0.04367129	1	100
Approximate	8	1024	0.00023971	0.02459343	0.02484671	1	100
OMP+CUDA	1	1024	0	0.00454671	0.25357143	0.7092434	70.9243
OMP+CUDA	2	1024	0	0.00438943	0.14742857	0.6045233	30.2262
OMP+CUDA	4	1024	0	0.00440486	0.116	0.3764766	9.4119
OMP+CUDA	8	1024	0	0.00438771	0.11714286	0.2121061	2.6513
OMP+CUDA_L1	1	1024	0	0.00427086	0.26214286	0.6860529	68.6053
OMP+CUDA_L1	2	1024	0	0.00430371	0.14585714	0.6110362	30.5518
OMP+CUDA_L1	4	1024	0	0.004291	0.117	0.3732589	9.3315
OMP+CUDA_L1	8	1024	0	0.004322	0.11371429	0.2185013	2.7313

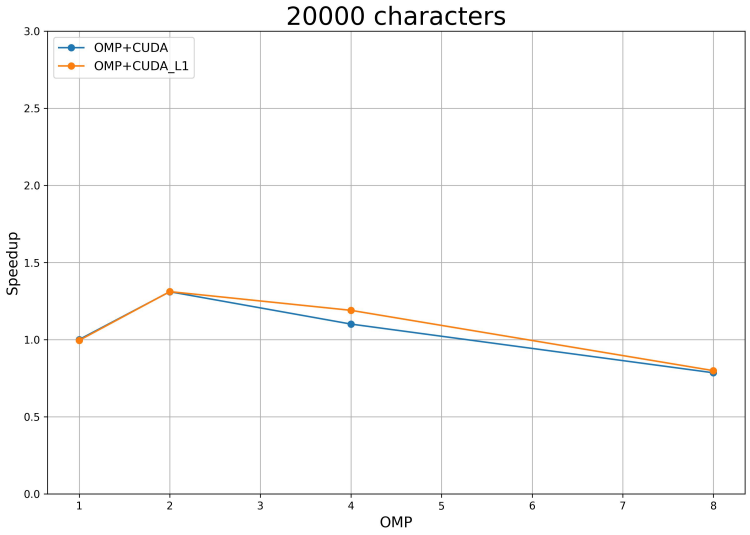
Timing table for 10000 characters.



Speedup plot for 10000 characters.

Modality	OMP	BlockSize	String_generation_time	Kernel_edit_distance_time	Program_execution_time	speedup	efficiency
Approximate	1	1024	0.000447	0.70133629	0.70180843	1	100
Approximate	2	1024	0.00047857	0.34700086	0.347501	1	100
Approximate	4	1024	0.000446	0.17598614	0.17645457	1	100
Approximate	8	1024	0.000443	0.09622186	0.096688	1	100
OMP+CUDA	1	1024	0	0.00844886	0.70128571	1.0007454	100.0745
OMP+CUDA	2	1024	0	0.00872486	0.26514286	1.310618	65.5309
OMP+CUDA	4	1024	0	0.00885343	0.16028571	1.1008752	27.5219
OMP+CUDA	8	1024	0	0.00840529	0.12314286	0.7851694	9.8146
OMP+CUDA_L1	1	1024	0	0.008564	0.70514286	0.9952713	99.5271
OMP+CUDA_L1	2	1024	0	0.00851043	0.265	1.3113245	65.5662
OMP+CUDA_L1	4	1024	0	0.00879729	0.14828571	1.1899634	29.7491
OMP+CUDA_L1	8	1024	0	0.00855343	0.121	0.7990744	9.9884

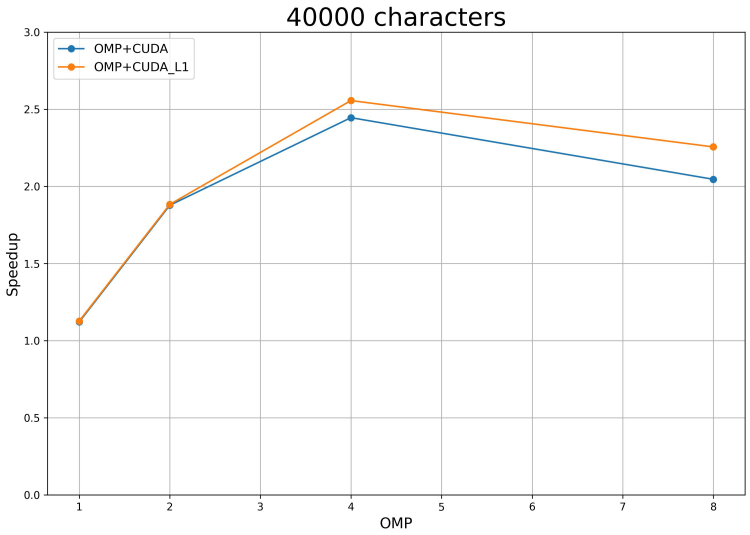
Timing table for 20000 characters.



Speedup plot for 20000 characters.

Modality	OMP	BlockSize	String_generation_time	Kernel_edit_distance_time	Program_execution_time	speedup	efficiency
Approximate	1	1024	0.00091257	2.78017729	2.78113186	1	100
Approximate	2	1024	0.00087071	1.38573757	1.38664843	1	100
Approximate	4	1024	0.000913	0.69946343	0.70042586	1	100
Approximate	8	1024	0.00086429	0.36036014	0.36126457	1	100
OMP+CUDA	1	1024	0.001	0.03949171	2.48085714	1.1210367	112.1037
OMP+CUDA	2	1024	0.001	0.03879929	0.73857143	1.8774737	93.8737
OMP+CUDA	4	1024	0.001	0.03815014	0.28642857	2.4453771	61.1344
OMP+CUDA	8	1024	0.001	0.03884986	0.17657143	2.0459968	25.575
OMP+CUDA_L1	1	1024	0.001	0.03879957	2.468	1.1268768	112.6877
OMP+CUDA_L1	2	1024	0.001	0.03874529	0.73628571	1.8833021	94.1651
OMP+CUDA_L1	4	1024	0.001	0.03877243	0.274	2.5562988	63.9075
OMP+CUDA_L1	8	1024	0.00114286	0.03993971	0.16014286	2.2558894	28.1986

Timing table for 40000 characters.

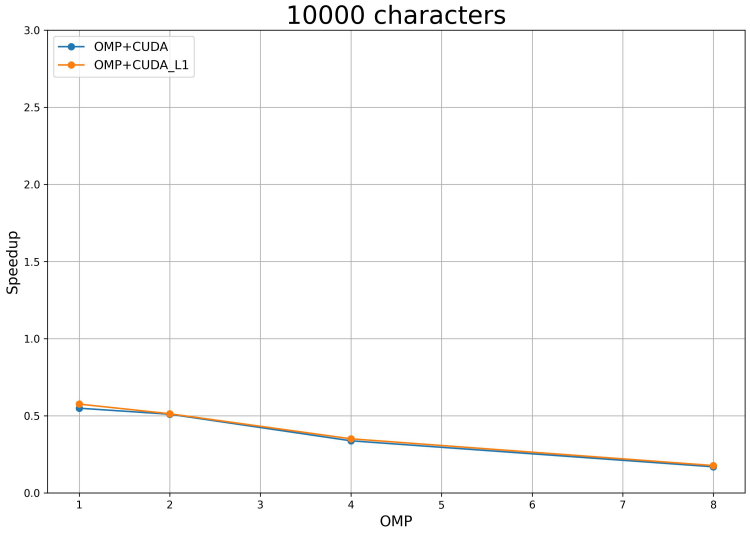


Speedup plot for 40000 characters.

1.7.5 Optimization 3

Modality	OMP	BlockSize	String_generation_time	Kernel_edit_distance_time	Program_execution_time	speedup	efficiency
Approximate	1	1024	0.00023414	0.14539657	0.14564786	1	100
Approximate	2	1024	0.00024086	0.07339686	0.07365343	1	100
Approximate	4	1024	0.00027471	0.04047143	0.040762	1	100
Approximate	8	1024	0.00024486	0.01936257	0.01962529	1	100
OMP+CUDA	1	1024	0	0.00425843	0.26528571	0.5490226	54.9023
OMP+CUDA	2	1024	0	0.004274	0.14442857	0.5099644	25.4982
OMP+CUDA	4	1024	0	0.00427643	0.12071429	0.3376734	8.4418
OMP+CUDA	8	1024	0	0.00435214	0.116	0.1691835	2.1148
OMP+CUDA_L1	1	1024	0	0.00434886	0.253	0.5756832	57.5683
OMP+CUDA_L1	2	1024	0	0.00444071	0.14357143	0.513009	25.6504
OMP+CUDA_L1	4	1024	0	0.00424957	0.11628571	0.3505332	8.7633
OMP+CUDA_L1	8	1024	0	0.004272	0.11071429	0.1772607	2.2158

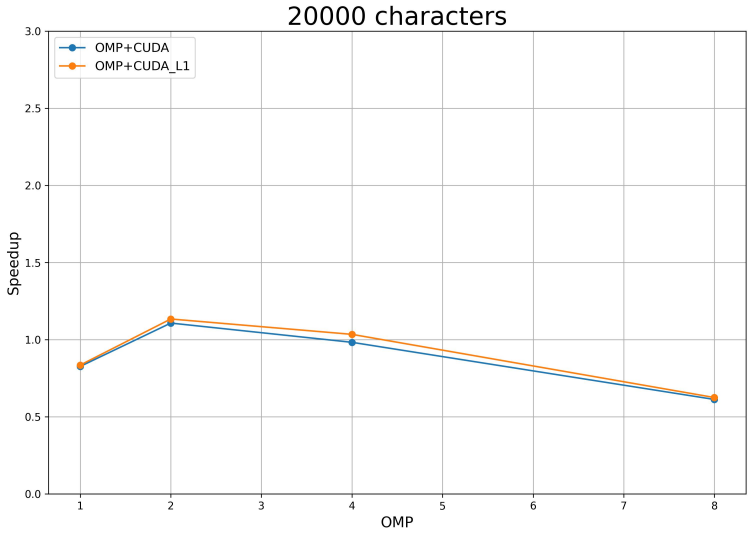
Timing table for 10000 characters.



Speedup plot for 10000 characters.

Modality	OMP	BlockSize	String_generation_time	Kernel_edit_distance_time	Program_execution_time	speedup	efficiency
Approximate	1	1024	0.00043771	0.58271371	0.58317829	1	100
Approximate	2	1024	0.00044514	0.29156757	0.29203586	1	100
Approximate	4	1024	0.00074529	0.151111	0.151883	1	100
Approximate	8	1024	0.00044643	0.07387286	0.074344	1	100
OMP+CUDA	1	1024	0	0.00849086	0.70571429	0.826366	82.6366
OMP+CUDA	2	1024	0	0.00873829	0.26371429	1.1073949	55.3697
OMP+CUDA	4	1024	0	0.00848143	0.15457143	0.9826072	24.5652
OMP+CUDA	8	1024	0	0.00852543	0.12142857	0.6122447	7.6531
OMP+CUDA_L1	1	1024	0	0.00854157	0.69828571	0.8351571	83.5157
OMP+CUDA_L1	2	1024	0	0.00860457	0.25757143	1.1338053	56.6903
OMP+CUDA_L1	4	1024	0	0.00857857	0.14685714	1.0342228	25.8556
OMP+CUDA_L1	8	1024	0	0.00842157	0.119	0.6247395	7.8092

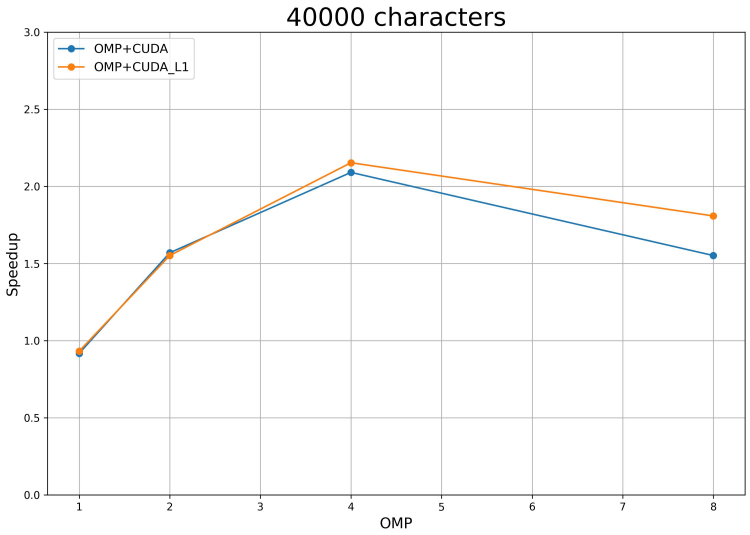
Timing table for 20000 characters.



Speedup plot for 20000 characters.

Modality	OMP	BlockSize	String_generation_time	Kernel_edit_distance_time	Program_execution_time	speedup	efficiency
Approximate	1	1024	0.00090414	2.304992	2.30594457	1	100
Approximate	2	1024	0.00089786	1.15569886	1.15664029	1	100
Approximate	4	1024	0.00088429	0.59489143	0.59583757	1	100
Approximate	8	1024	0.00090714	0.300535	0.30148571	1	100
OMP+CUDA	1	1024	0.001	0.039474	2.51242857	0.917815	91.7815
OMP+CUDA	2	1024	0.001	0.03934243	0.73685714	1.5696941	78.4847
OMP+CUDA	4	1024	0.00114286	0.03900114	0.285	2.0906582	52.2665
OMP+CUDA	8	1024	0.001	0.03999143	0.19428571	1.5517647	19.3971
OMP+CUDA_L1	1	1024	0.001	0.03883186	2.47442857	0.9319099	93.191
OMP+CUDA_L1	2	1024	0.001	0.03863857	0.74528571	1.5519421	77.5971
OMP+CUDA_L1	4	1024	0.001	0.038182	0.27671429	2.1532592	53.8315
OMP+CUDA_L1	8	1024	0.001	0.03935314	0.16671429	1.8083976	22.605

Timing table for 40000 characters.



Speedup plot for 40000 characters.

1.7.6 Final Considerations

We note that the trend of the case studies is essentially **equivalent to that of the OMP+MPI version**.

Naturally, with many more partitions, the execution with CUDA becomes much more **advantageous for much larger datasets**. For the first set, in fact, the speedup is always lower, because the **overhead times of CUDA** are longer than the algorithm itself, and therefore longer than the sequential version.

We observe that with the use of L1 cache, there is a **slight improvement**.

1.7.7 Documentation

Documentation here: `https://edit_distance_parallelized.surge.sh/`

1.7.8 How to run

Read the **README** inside the project folder.

Capitolo 2

Cluster parallelization (D'aniello)

In this second part of the report, we will discuss programming on **clusters**, both using the **Apache Hadoop** framework with the **Map-Reduce** programming paradigm, and using the **Apache Spark** framework.

2.1 Dataset analysis

RoundID	MatchID	Team Initials	Coach Name	Line-up	Shirt Number	Player Name	Position	Event
201	1096	FRA	CAUDRON Raoul (FRA)	S	0	Alex THEPOT	GK	
201	1096	MEX	LUQUE Juan (MEX)	S	0	Oscar BONFIGLIO	GK	
201	1096	FRA	CAUDRON Raoul (FRA)	S	0	Marcel LANGILLER		G40'
201	1096	MEX	LUQUE Juan (MEX)	S	0	Juan CARRENO		G70'
201	1096	FRA	CAUDRON Raoul (FRA)	S	0	Ernest LIBERATI		
201	1096	MEX	LUQUE Juan (MEX)	S	0	Rafael GARZA	C	
201	1096	FRA	CAUDRON Raoul (FRA)	S	0	Andre MASCHINOT		G43' G87'
201	1096	MEX	LUQUE Juan (MEX)	S	0	Hilario LOPEZ		
201	1096	FRA	CAUDRON Raoul (FRA)	S	0	Etienne MATTLER		
201	1096	MEX	LUQUE Juan (MEX)	S	0	Dionisio MEJIA		
201	1096	FRA	CAUDRON Raoul (FRA)	S	0	Marcel PINEL		
201	1096	MEX	LUQUE Juan (MEX)	S	0	Felipe ROSAS		
201	1096	FRA	CAUDRON Raoul (FRA)	S	0	Alex VILLAPLANE	C	
201	1096	MEX	LUQUE Juan (MEX)	S	0	Manuel ROSAS		
201	1096	FRA	CAUDRON Raoul (FRA)	S	0	Lucien LAURENT		G19'

Figura 2.1: First rows of the dataset.

The provided dataset contains information about football players who participated in the FIFA World Cup. In particular, in Figure 2.1, we can see the presence of **9 columns** indicating:

- **RoundiD:** Unique ID of the Round;
- **MatchiD:** Unique ID of the Match;
- **Team Initials:** Team country's three-letter initials;
- **Coach Name:** Name and country of the coach;
- **Line-up:** S = Line-up, N = Substitute;
- **Shirt Number:** number of the shirt;
- **Player Name:** Name of the player;
- **Position:** C = Captain, GK = Goalkeeper;

- **Event:** G = Goal, W = Own Goal, Y = Yellow Card, R = Red Card, SY = Red Card by second yellow, P = Penalty, MP = Missed Penalty, I = Substitution In, O = Substitute Out, OH = Substitution out in the half-time, IH = Substitution in in the half-time, U = Unknown event;

In the dataset, there are no particular issues, except for some player names with accented letters that are recognized in a strange way. This doesn't concern us, as we don't manually search for player names; the programs will simply read and compare them. Therefore, we won't bother fixing them. The only thing we will do is to ignore the first row, which is the header.

2.2 Hadoop Map-Reduce

Exercise 1 : Develop a program using Hadoop Map Reduce to analyze the available dataset differently from the second exercise (e.g., a statistic, a ranking, an index, etc.). .

For this task, the decision has been made to calculate the ranking of players who have scored the most goals overall. The number of players considered for the ranking is taken from the command line. Additionally, for each player, all the goals scored are listed, as an inverted index, indicating in which match they were made.

Now we will look at the solution code, then see how to run it, and finally evaluate its output.

2.2.1 Driver: Job chaining

Listing 2.1: DriverTopScorersWithInvertedIndex

```
/*
*   Driver to manage the two Jobs that make up this program: the first one is used
    to obtain all the players who scored
*   with pairings in all the matches they did; the second one to build the top.
*/

public class DriverTopScorersWithInvertedIndex {

    public static void main(String args[]) throws Exception {

//        Check if there are the right number of command line parameter.
        if (args.length < 4) {
            System.err.println("Usage: WorldCupAnalisysHadoop <inputFilePath>
                <firstJobOutputDir> <secondJobOutputDir> <NumberTopElement>");
            System.exit(-1);
        }

//        First command line parameter: dataset's path.
        Path inputDatasetPath = new Path(args[0]);
//        Second command line parameter: output directory path for the first job.
        Path outputDirInvInd = new Path(args[1]);
```

```
//       Third command line parameter: output directory path for the second job.
        Path outputDirTopK = new Path(args[2]);

        Configuration conf = new Configuration();

//       First job: InvertedIndex + Filter.
        Job job = Job.getInstance(conf);

        job.setJobName("World Cup Analisys - Step 1: Inverted Index");

//       Set path of the input dataset file for this job.
        FileInputFormat.addInputPath(job, inputDatasetPath);

//       Set path of the output folder for this job.
        FileOutputFormat.setOutputPath(job, outputDirInvInd);

//       Specify the class of the Driver for this job (this one).
        job.setJarByClass(DriverTopScorersWithInvertedIndex.class);

//       Set job input format: Text because in the dataset there are different
    column to split.
        job.setInputFormatClass(TextInputFormat.class);

//       Set job output format.
        job.setOutputFormatClass(TextOutputFormat.class);

//       Set map class.
        job.setMapperClass(MapperInvertedIndexAndFilter.class);

//       Set map output key and value classes: Text for the player name in the key;
    PlayerMatchGoal for the player match's goal.
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(PlayerMatchGoal.class);

//       Set reduce class.
        job.setReducerClass(ReducerInvertedIndexAndFilter.class);

//       Set reduce output key and value classes: Text for the player name in the
    key; Text for the concat of all the player matches goals in the value.
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Text.class);

//       Execute the job and wait for completion.
        if (!job.waitForCompletion(true)) {
            System.exit(1);
         }
```

```
        Configuration conf2 = new Configuration();
        conf2.set("k", args[3]);

//        Second job: TopK.
        Job job2 = Job.getInstance(conf2);

        job2.setJobName("World Cup Project - Step 2: TopK");

//        Set path of the input folder for this job (output folder of the previous
    job).
        FileInputFormat.addInputPath(job2, outputDirInvInd);

//        Set path of the output folder for this job.
        FileOutputFormat.setOutputPath(job2, outputDirTopK);

//        Specify the class of the Driver for this job (this one).
        job2.setJarByClass(DriverTopScorersWithInvertedIndex.class);

//        Set job input format: KeyValue because, from the previous job, we know
    that the first fields is the
//        player name and the second one is the concat of all the player matches
    goals.
        job2.setInputFormatClass(KeyValueTextInputFormat.class);

//        Set job output format
        job2.setOutputFormatClass(TextOutputFormat.class);

//        Set map class
        job2.setMapperClass(MapperTopKScorers.class);

//        Set map output key and value classes: NullWritable for the key as the TopK
    pattern includes;
//        PlayerTotalGoals fo the value because is the object of the Top.
        job2.setMapOutputKeyClass(NullWritable.class);
        job2.setMapOutputValueClass(PlayerTotalGoals.class);

//        Set reduce class
        job2.setReducerClass(ReducerTopKScorers.class);

//        Set reduce output key and value classes: Text for the player name in the
    key; Text for all the player matches goals in the value.
        job2.setOutputKeyClass(Text.class);
        job2.setOutputValueClass(Text.class);

//        Execute the job and wait for completion
        System.exit(job2.waitForCompletion(true) ? 0 : 1);
    }
}
```

2.2.2 Utility class 1: PlayerMatchGoal

Listing 2.2: PlayerMatchGoal

```
/*
*   Utility class to keep the match and all the goals of that match for a player.
*/

public class PlayerMatchGoal implements Writable {
    private String match;
    private List<String> goals;

    public PlayerMatchGoal() {
        this.match = "";
        this.goals = new ArrayList<>();
    }

    public PlayerMatchGoal(String match) {
        this.match = match;
        this.goals = new ArrayList<>();
    }

    public String getMatch() {
        return match;
    }

    public List<String> getGoals() {
        return goals;
    }

    public void addGoal(String goal) {
        goals.add(goal);
    }

    @Override
    public void write(DataOutput out) throws IOException {
        out.writeUTF(match);
        out.writeInt(goals.size());
        for (String goal : goals) {
            out.writeUTF(goal);
        }
    }
```

```
    @Override
    public void readFields(DataInput in) throws IOException {
        match = in.readUTF();
        int size = in.readInt();
        goals = new ArrayList<>(size);
        for (int i = 0; i < size; i++) {
            goals.add(in.readUTF());
        }
    }

//   Override the toString in order to structure it.
    @Override
    public String toString() {
        if(goals.isEmpty())
            return "";
        else {
            String matchGoal = match + ": ";
            for(String goal: goals) {
                matchGoal = matchGoal.concat(goal + ",");
            }
            return matchGoal.substring(0, matchGoal.length() - 1);
        }
    }
}
```

2.2.3 Mapper Job 1: Inverted index + Filter

Listing 2.3: MapperInvertedIndexAndFilter

```
/*
*   Mapper class that for each row, if the player has scored a goal, write in the
    context the player name,
*   paired to the match and goals of that row. We use LongWritable for the input
    key, just to ignore the
*   dataset header (offset = 0).
*/

public class MapperInvertedIndexAndFilter extends Mapper<LongWritable, Text, Text,
    PlayerMatchGoal> {
//   Instead of allocating a new variable each time within the function, we
    allocate them
//   within the class, only changing their values within the relevant method.
    private PlayerMatchGoal matchGoals;
    private Text playerName = new Text();
```

```
    @Override
    protected void map(LongWritable key, Text value, Mapper.Context context) throws
        IOException, InterruptedException {

//        Check to ignore the header.
        if(key.get() == 0)
            return;

//        Split all the dataser columns.
        String[] fields = value.toString().split(",");

//        Flag used to check if the player has scored at least one goal, in order to
    initialize the PlayerMatchGoal
//        object with the matchID and write it in the context.
        Boolean flag = true;

//        Check to ignore the raw without event (last column).
        if(fields.length == 9) {
//            Split all the events.
            for(String event: fields[8].split(" "))
//                Check if the event is a goal (or penalty).
                if(event.startsWith("G") || event.startsWith("P")) {
//                    Whe we find the first goal, initialize the object and change
    the flag value.
                    if(flag) {
                        matchGoals = new PlayerMatchGoal(fields[1]);
                        flag = false;
                    }
//                    Add the goal.
                    matchGoals.addGoal(event);
                }
        }

//        If the player scored at least one goal (filter) we write the pair (player
    name, match goals) in the context.
        if(!flag) {
            playerName.set(fields[6]);
            context.write(playerName, matchGoals);
        }
    }
}
```

2.2.4 Reducer Job 1: Inverted index + Filter

Listing 2.4: ReducerInvertedIndexAndFilter

```
/*
*   Reducer class that only concat all the PlayerMatchGoal in input for a player.
*/

class ReducerInvertedIndexAndFilter extends Reducer <Text, PlayerMatchGoal, Text,
    Text> {
//    Instead of allocating a new variable each time within the function, we
    allocate them
//    within the class, only changing their values within the relevant method.
    private Text invIndex = new Text();

    @Override
    protected void reduce(Text key, Iterable<PlayerMatchGoal> values, Context
        context) throws IOException, InterruptedException {
        invIndex.set("");

        for (PlayerMatchGoal value : values) {
            invIndex.set(invIndex.toString().concat("{" + value + "}; "));
        }

        context.write(key, invIndex);
    }
}
```

2.2.5 Utility Class 2: PlayerTotalGoals

Listing 2.5: PlayerTotalGoals

```
/*
*   Utility class to keep the playerName his goalNumber and the index of theese
    goals.
*/

public class PlayerTotalGoals implements WritableComparable<PlayerTotalGoals> {
    private String playerName;
    private Integer goalsNumber;
    private String goals;

    public PlayerTotalGoals() {
    }
```

```
    public PlayerTotalGoals(String playerName, Integer goalsNumber, String goals) {
        this.playerName = playerName;
        this.goalsNumber = goalsNumber;
        this.goals = goals;
    }

    public String getPlayerName() {
        return playerName;
    }

    public Integer getGoalsNumber() {
        return goalsNumber;
    }

    public String getGoals() {
        return goals;
    }

    @Override
    public void write(DataOutput out) throws IOException {
        out.writeUTF(playerName);
        out.writeInt(goalsNumber);
        out.writeUTF(goals);
    }

    @Override
    public void readFields(DataInput in) throws IOException {
        playerName = in.readUTF();
        goalsNumber = in.readInt();
        goals = in.readUTF();
    }

    @Override
    public int compareTo(PlayerTotalGoals other) {
//        We compare two PlayerTotalGoals based on their goals number.
        int result = other.goalsNumber.compareTo(this.getGoalsNumber());
//        In tie case we use the player Name alphabetical order.
        if (result == 0)
            result = this.playerName.compareTo(other.getPlayerName());
        return result;
    }

    @Override
    public String toString() {
        return playerName + "\t" + goals;
    }
}
```

2.2.6 Mapper Job 2: TopK

Listing 2.6: MapperTopKScorers

```
/*
*   Mapper class that for each row (player) count the number of goal and build the
    ranking based on this number.
*   In tie cases we use the player name alphabetical order.
*/
public class MapperTopKScorers extends Mapper<Text, Text, NullWritable,
    PlayerTotalGoals> {
//    Constant for the ranking size.
    private static Integer k;
    private TreeMap<PlayerTotalGoals, String> topK;

//    Inizialize the top before all the executions.
    @Override
    protected void setup(Context context) {
//        Get the ranking size from the context.
        k = Integer.valueOf(context.getConfiguration().get("k"));
        topK = new TreeMap();
    }

  @Override
    public void map(Text key, Text value, Context context) throws IOException,
        InterruptedException {
//        We take the goals number from the number of element "G" and "P" thanks to
    the this split series.
        Integer goalsNumber = 0;
        for(String matchGoal: value.toString().split("; "))
            goalsNumber += matchGoal.split(":")[1].split(",").length;
//        Create a new PlayerTotalGoals object with the current row data.
        PlayerTotalGoals playerTotalGoals = new PlayerTotalGoals(key.toString(),
            goalsNumber, value.toString());
        topK.put(playerTotalGoals, null);
//        Remove the last key from the ranking if the size is bigger than the
    selected k.
        if(topK.size() > k)
          topK.remove(topK.lastKey());
    }

  @Override
    protected void cleanup(Context context) throws IOException,
        InterruptedException {
//        Write all the PlayerTotalGoals from the top in the context.
        for(PlayerTotalGoals key: topK.keySet())
            context.write(NullWritable.get(), key);
    }
  }
```

2.2.7 Reducer Job 2: TopK

Listing 2.7: ReducerTopKScorers

```
/*
*   Reducer class that receives elements from the mappers
*   for their tops and builds a final "total" top in the same way.
*/

public class ReducerTopKScorers extends Reducer<NullWritable,
    PlayerTotalGoals,Text, Text> {
    private static Integer k;
    private Text playerName = new Text();
    private Text goals = new Text();

    @Override
    protected void setup(Context context) {
        k = Integer.valueOf(context.getConfiguration().get("k"));
    }

//   We can note that the initialization of the data structure and the final
    writing are within the function,
//   as there is only one group with a null key. Therefore, the reducer will be
    executed only once
//   with all the elements in the values.
    @Override
    public void reduce(NullWritable key, Iterable<PlayerTotalGoals> values, Context
        context) throws IOException, InterruptedException {
        TreeMap<PlayerTotalGoals, String> topK = new TreeMap();

        for(PlayerTotalGoals record: values) {
            PlayerTotalGoals newValue = new PlayerTotalGoals(record.getPlayerName(),
                record.getGoalsNumber(), record.getGoals());
            topK.put(newValue, null);
            if(topK.size() > k) {
                topK.remove(topK.lastKey());
            }
        }

        for(PlayerTotalGoals mapKey: topK.keySet()){
            playerName.set(mapKey.getPlayerName());
            goals.set(mapKey.getGoals());
            context.write(playerName, goals);
        }
    }
}
```

2.2.8 How to run

Inside the project archive, there are two folders. For now, we will focus on the contents of the one called ***Hadoop***. Inside it, we find two folders: the ***source*** folder contains the NetBeans project with the code; the ***cluster*** folder is the one that should be placed inside the Hadoop cluster volume folder.

After launching Docker, you need to open a terminal and navigate to the main folder of the Hadoop cluster (where the compose file is located). Here we can execute the command to build the cluster:

```
docker compose up -d
```

Now that the cluster is built, we need to connect to the master by launching a bash shell on it and connecting interactively with the command:

```
docker container exec -ti master bash
```

Firstly, we need to format this node as the namenode of the cluster with the command:

```
hdfs namenode -format
```

After that, now we need to start the distributed file system to work on and the process manager to launch the program with the commands:

```
$HADOOP_HOME/sbin/start-dfs.sh
$HADOOP_HOME/sbin/start-yarn.sh
```

Now we can navigate to the ***cluster*** folder that we copied into the volume using the command:

```
cd data/cluster
```

Here we can upload our input file *players1.csv* to HDFS using the dedicated protocol **hdfs://**, calling it */input* with the command:

```
hdfs dfs -put players1.csv hdfs:///input
```

Finally we can launch the program. We have to specify four parameters:

- **inputPath:** the input file Path (*/input* in this case);
- **firstJobOutputDir:** the output directory path for the first job;
- **secondJobOutputDir:** the output directory path for the second job;
- **NumberTopElement:** the number of element of the final Top;

So, to execute the jar, we have to use this command:

```
hadoop jar WorldCupAnalysisHadoop.jar /input /output1 /output2 10
```

After the program execution we can take it from HDFS with the command:

```
hdfs dfs -get /output2/part-r-00000 outputProject.txt
```

Now, in the project's ***cluster*** folder, we can find the document *outputProject.txt*, which contains the output of the program.

2.2.9 Output and final considerations

The program output is as follows and represents, in order, the top 10 players who have scored the most, along with the list of goals scored divided per match:

```
Sandor KOCSIS {1277: G3',G21',G69',G78'}; {1295: G109',G116'}; {1294: G24',G36',G50'}; {1248: G7',G88'};
ADEMIR {1191: G4'}; {1189: G17',G36',G52',G58'}; {1186: G57'}; {1187: G30',G79'};
Guillermo STABILE {1086: G8',G17',G80'}; {1087: G37'}; {1088: G69',G87'}; {1084: G12',G13'};
Helmut RAHN {1285: G85'}; {1389: G20'}; {1277: G77'}; {1391: G70'}; {1278: G18',G84'}; {1323: G32',G79'};
LEONIDAS {1151: G63',G74'}; {1150: G18',G93',G104'}; {1153: G57'}; {1152: G30'}; {1111: G55'};
Oscar MIGUEZ {1313: G30',G83'}; {1315: G71'}; {1231: G77',G85'}; {1185: G14',G40',G51'};
Oldrich NEJEDLY {1141: G67'}; {1143: G82'}; {1130: G21',G69',G80'}; {1152: P65'}; {1172: G118'};
Erich PROBST {1233: G51'}; {1238: G4',G21',G24'}; {1236: G33'}; {1237: G76'};
Gyorgy SAROSI {1106: P60'}; {1173: G28',G89'}; {1175: G40'}; {1174: G70'}; {1176: G65'};
Just FONTAINE {1388: G4',G85'}; {1386: G24',G30',G67'}; {1387: G44'};
```

This is the only output we will see for this program, because the only combination used is without a combiner and with a single reducer. For neither of the two jobs used, in fact, it makes sense to use a combiner or more than one reducer.

The first job performs an inverted index+filter, and for both, the combiner is not needed. For the inverted-index pattern, the reducer does nothing, so does the combiner; including it or using more reducers would only weigh down the execution. Similarly, for the filtering pattern, the reducer, like the combiner, should not be used at all.

The second job performs a topK pattern, which involves the use of internal state in the mapper that makes its output unique, rendering the combiner unnecessary. Additionally, the reducer must be unique since the produced topK result should be singular.

2.3 Spark

Exercise 2: Find the team (or teams in case of a tie) that has won the most matches with the fewest number of players on the field at the end of the game.

Now we will look at the solution code, then see how to run it, and finally evaluate its output.

2.3.1 Driver

Listing 2.8: SparkDriver

```
public class SparkDriver {

    public static void main(String[] args) {

        String inputPath = args[0];
        String outputPath = args[1];

        SparkConf conf = new SparkConf().setAppName("World Cup Analysis");

        JavaSparkContext sc = new JavaSparkContext(conf);

//        Build an RDD of Strings from the input textual file.
//        Each string contains the data of a player in the World Cup.
        JavaRDD<String> dataset = sc.textFile(inputPath);

//        Remove the header line.
        String header = dataset.first();
        JavaRDD<String> data = dataset.filter(row -> !row.equals(header));

//        First Step: Split the players based on the game played and group them.
        JavaPairRDD<String, Iterable<String>> matchesData = data.mapToPair(row -> {
//                    Get the matchId.
                    String matchId = row.split(",")[1];
                    return new Tuple2(matchId, row);
                }
        ).groupByKey();

//        Second Step: Determine the winning team for each match and count
//        the number of players on the field at the end of the game.
        JavaPairRDD<String, Tuple2<String, Integer>> winningTeamsByMatch =
            matchesData.flatMapValues(playerData -> {
//            Initialize two Map: the first one for the goal scored for each team;
//            the second one for the number of players on the field at the end of
    the game for each team;
            Map<String, Integer> goalsByTeam = new HashMap();
            Map<String, Integer> teamPlayers = new HashMap();
```

```
//          For each player we count the number of goal and/or red card.
          for (String player : playerData) {
              String[] fields = player.split(",");
//              Get the three-letters team initial.
              String team = fields[2];

//              If the team is not already present in one of the maps (and
    therefore also in the other),
//              we insert it in the goals Map with "0" and in the players count
    Map with "11".
              if(!goalsByTeam.containsKey(team)) {
                  goalsByTeam.put(team, 0);
                  teamPlayers.put(team, 11);
              }

//              If the current player doesn't have events we can skip him.
              if(fields.length == 9) {
//                  Take the single events.
                  String[] eventsByPlayer = fields[8].split(" ");

                  for(String events: eventsByPlayer) {
//                      If the event is a Goal or a Penalty we add a goal to the
    player's team.
                      if(events.startsWith("G") || events.startsWith("P"))
                          goalsByTeam.put(team, goalsByTeam.get(team) + 1);

//                      If the event is an Auto-Goal we subtract a goal to the
    player's team.
                      if(events.startsWith("W"))
                          goalsByTeam.put(team, goalsByTeam.get(team) - 1);

//                      If the event is a Red Card we subtract a player to the
    player's team
                      if(events.startsWith("R"))
                          teamPlayers.put(team, teamPlayers.get(team) - 1);
                  }
              }
          }

//          Get the two match's team;
          String team1 = (String) goalsByTeam.keySet().toArray()[0];
          String team2 = (String) goalsByTeam.keySet().toArray()[1];

//          We use a List of Tuple to get, for each match, the winning team and
    his number of players.
          List<Tuple2<String, Integer>> winningTeam = new ArrayList();
```

```
//          We get the winning team based on the goal.
//          In tie case we don't get anythng.
            if(goalsByTeam.get(team1) > goalsByTeam.get(team2))
                winningTeam.add(new Tuple2(team1, teamPlayers.get(team1)));
            else
                if(goalsByTeam.get(team1) < goalsByTeam.get(team2))
                    winningTeam.add(new Tuple2(team2, teamPlayers.get(team2)));

            return winningTeam.iterator();
        });

//      Third step: We need to filter the winning teams based on the remaining
//      number of players, keeping only those with the minimum number of players
    among all.

//      First of all we get all the number of players and find the min.
        Integer minPlayers = winningTeamsByMatch.map(entry ->
            entry._2()._2()).reduce((x, y) -> Math.min(x, y));

//      After doing this, we need to filter and keep only the matches that were
    won by
//      teams with the minimum number of players.
        JavaPairRDD<String, Tuple2<String, Integer>> winningTeamsWithMinPlayers =
            winningTeamsByMatch.filter(entry -> entry._2()._2().equals(minPlayers));

//      Finally, we can extract only the team name (since we kept them separated
    by match)
//      and count their occurrences to understand how many matches that team has
    won with the minimum number of players
        Map<String, Long> winningTeamsWithMinPlayersCountWins =
            winningTeamsWithMinPlayers.map(entry -> entry._2()._1()).countByValue();

//      The count return a map so we have to build a PairRDD with the pair (team,
    #winsWithMinPlayers).
        List<Tuple2<String, Integer>> temp = new ArrayList();
        for(String team: winningTeamsWithMinPlayersCountWins.keySet()) {
            temp.add(new Tuple2(team,
                winningTeamsWithMinPlayersCountWins.get(team).intValue()));
        }
        JavaPairRDD<String, Integer> winningTeamsWithMinPlayersCountWinsRDD =
            sc.parallelizePairs(temp);
```

```
//      Final step: For the last time, we need to filter the remaining teams to
    find the one that has won the most.

//      First of all we get all the number of wins and find the max.
        Integer maxWins = winningTeamsWithMinPlayersCountWinsRDD.map(entry ->
            entry._2()).reduce((x, y) -> Math.max(x, y));

//      Finally we filter only the team that has the max number of wins.
//      (We do a final map just to make the printout more readable.)
        JavaRDD<String> teamsWithMaxWinsWithMinPlayers =
            winningTeamsWithMinPlayersCountWinsRDD.filter(entry ->
            entry._2().equals(maxWins)).map(entry -> entry._1() + " wins " + maxWins
            + " match/es with " + minPlayers + " players");

        teamsWithMaxWinsWithMinPlayers.saveAsTextFile(outputPath);

        sc.close();
    }
}
```

2.3.2 How to run

Now we have to consider the ***Spark*** folder. Inside it, as the Hadoop one, we find two folders: the ***source*** folder contains the NetBeans project with the code; the ***cluster*** folder is the one that should be placed inside the Spark cluster volume folder.

After launching Docker, you need to open a terminal and navigate to the main folder of the Spark cluster (where the compose file is located). Here we can execute the command to build the cluster:

```
docker compose up -d
```

Now that the cluster is built, we need to connect to the master by launching a bash shell on it and connecting interactively with the command:

```
docker container exec -ti sp_master bash
```

The Spark cluster will place us in an internal folder, so the first thing we need to do is move to the project folder within the volume using the command:

```
cd /testfiles/cluster
```

Before running the program, make sure to have deleted the output folders ***outputSingle*** and ***outputCluster*** already present in the project folder within the volume. Subsequently, we can execute the two versions of the launch script:

```
./launch_single.sh
./launch_cluster.sh
```

In the appropriate folders generated in the project folder within the cluster volume, we will find the output.

2.3.3 Output and final considerations

The program output is as follows and represents the teams that won with the fewest number of players on the field at the end of the match. In particular, we also see the number of matches and the number of players:

```
HUN wins 1 match/es with 10 players
```

In particular, we can note that in this dataset, Hungary is the only team to have won one match with 10 players on the field at the end of the match.

The cluster output indeed contains the **same output in the second file**: this was predictable once the output of the "single version" was seen, as the data for this Hungary match is located in the second half of the dataset.