Parallel Edit Distance
Parallel Programming for Machine Learning

Niccolò Arati
niccolo.arati@edu.unifi.it

Abstract

*This project aims to analyze the advantages of a parallel implementation of the Edit Distance algorithm. Specifically, it aims to observe the **speedup**, calculated as the ratio between the sequential execution time and the parallel execution time, while varying different configurations of the algorithm's main parameters, as well as different implementations of the sequential algorithm.*
Additionally, the results will be analyzed by changing the number of threads related to the parallel configuration.

1. Introduction

Edit Distance, also known as Levenshtein Distance, is a measure of similarity between two strings, A and B. Specifically, it indicates the minimum number of basic operations required to transform string A into string B.
The basic operations considered in this work are:

- **Insertion** of a character;
- **Substitution** of a character;
- **Deletion** of a character.

The algorithm can be summarized with the following recursive formula:

$$lev_{A,B}(i,j) = \begin{cases} max(i,j), \text{if } min(i,j) = 0 \\ min \begin{cases} lev_{A,B}(i-1,j) + 1 \\ lev_{A,B}(i,j-1) + 1 \\ lev_{A,B}(i-1,j-1) + id \end{cases} \end{cases} \tag{1}$$

In Eq. (1), $id = \mathbf{1}(A_i \neq B_i)$, where $\mathbf{1}$ corresponds to the indicator function.
Observing the formula, for each element (i, j) of the matrix, corresponding to the comparison between the i-th character of A and the j-th character of B, the value is calculated as follows:

- If the characters corresponding to indices i and j are equal, the Edit Distance value remains the same as the one assigned to $(i - 1, j - 1)$, as no basic operations are needed.
- Otherwise, removal (first row), insertion (second row), and substitution (third row) operations are considered, and the operation that minimizes the increase in the Edit Distance value is chosen.

Each basic operation is evaluated with the same impact, as each of the three increments the Edit Distance value by 1.
The purpose of the project is to measure the speedup obtained by parallelizing the algorithm and comparing it with its sequential version. OpenMP was used for parallelization, and the programming language used is C++.
All experiments were conducted on the SSH server "papavero.dinfo.unifi.it."

1.1. Sequential versions of the algorithm

Three sequential versions of the algorithm have been implemented:

- **Full Matrix**: utilizes a 2D matrix to store the intermediate results of distance calculation. The dimensions of the matrix correspond to the lengths of the two compared strings. The algorithm starts with an empty string and iteratively computes the various elements of the matrix row by row.
- **Matrix Row**: stores two rows of the matrix at a time, thereby reducing the complexity in terms of occupied memory. The algorithm iterates between the strings row by row, storing the current and previous iteration's calculations in two vectors corresponding to the i-th and (i - 1)th rows of the complete matrix.
- **Skew Diagonal**: constructs the matrix by traversing it diagonally from right to left. Fig. 1 intuitively illustrates the general idea of the algorithm.

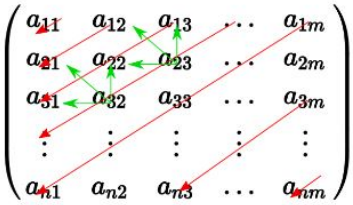

Figure 1. The red arrows represent the skew diagonals of the matrix. Each element of the diagonals requires 3 elements to be computed, these 3 elements are indicated by the green arrows and they belong to the previous diagonal, so they are already known.

1.2. Parallel version of the algorithm

The most common approach to implement Edit Distance is the Full Matrix, but it is not parallelizable because it presents a **race condition**. This occurs because two nested for loops are used to traverse the matrix, and the inner loop depends on the index of the outer loop.
For this reason, the **Skew Diagonal** version is parallelized, as each element of the considered diagonal depends on 3 elements belonging to the previous diagonal, which has already been calculated in the previous iteration. With this approach, the race condition is avoided, allowing parallelization.

2. Code Structure

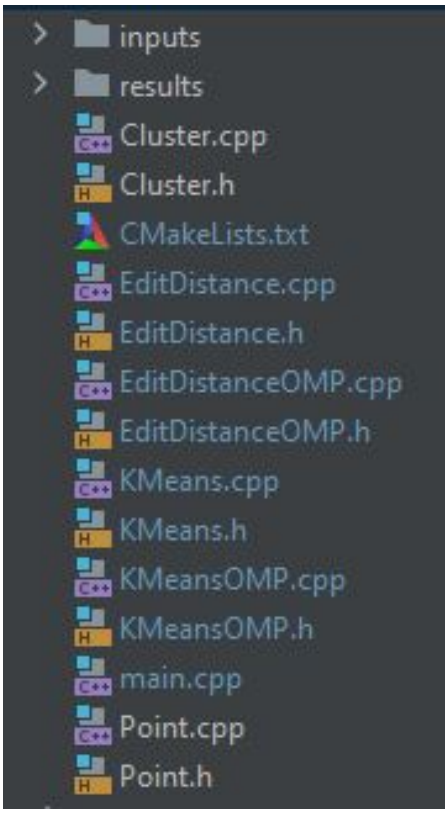

Figure 2. Code structure.

CLion was used as the IDE and MinGW as the compiler.
The code, whose files are shown in Fig. 2, contains files belonging to another project with a common directory. The files relevant to this project are:

- **EditDistance.h/cpp:** contain sequential implementations of the algorithms;
- **EditDistanceOMP.h/cpp:** contain the parallel implementation of the algorithm;
- **main.cpp:** contains functions related to the tests and other auxiliary functions.

In each of the following implementations, the matrix containing the Edit Distance results (or

the vectors corresponding to the rows of the matrix for Matrix Row) has been stored using a **std::vector** item, in order to automatically handle any possible memory leaks.

2.1. Sequential Implementations

In the files mentioned earlier, there are declarations and definitions of 3 functions, one for each sequential version of Edit Distance that we decided to implement.
The declarations are shown in Fig. 3.
levenshteinDistFM implements the Full Matrix approach of Edit Distance, requiring as input the two strings to compare and their lengths. As output, it returns an integer corresponding to the Edit Distance value between the two considered strings.
levenshteinDistSD implements the Skew Diagonal approach, similarly requiring the two strings to compare with their lengths as input, and returning an integer with the Edit Distance value.
levenshteinDistMR implements the Matrix Row approach, with the function requiring the same inputs as the previous two functions, and the output corresponding to the one of the previous functions as well.

```
int levenshteinDistFM(const std::string& word1, const std::string& word2, int length1, int length2);

int levenshteinDistSD(const std::string& word1, const std::string& word2, int length1, int length2);

int levenshteinDistMR(const std::string& word1, const std::string& word2, int length1, int length2);
```

Figure 3. Declarations of the 3 sequential versions implemented of Edit Distance

2.2. Parallel implementation

In the file mentioned earlier, there are declarations and definitions of 2 functions, which implement the Skew Diagonal approach by applying parallelization with two different ways. The declarations require the same inputs as the sequential functions, with the addition of an integer indicating the number of threads to use for parallelization.
The difference between the two functions will be further analyzed in Sec. 3.

2.3. main.cpp

In the file, there are tests related to the parallelization of the Edit Distance algorithm (presented in Sec. 4), along with the **main** function that executes those tests and the **random_string** function.
The latter is the function responsible of randomly generating the strings to be compared with Edit Distance. It requires as input the desired length of the string and the seed to potentially replicate the string generation. For generating the string, all alphanumeric characters have been considered, including both uppercase and lowercase letters, as well as digits.

3. Parallelization with OpenMP

The code containing the implementation of the Skew Diagonal approach has been parallelized in the part related to the calculation of the Edit Distance matrix. There are two for loops: the first one iterates over the diagonals, while the second one computes the value of the elements belonging to the current diagonal.
Two possible approaches to parallelization have been considered.

3.1. Single parallel overhead

```
#pragma omp parallel default(none) firstprivate(dMIN, dMAX, M, N) \
shared(distMatrix, word1, word2) num_threads(threads)
{
    for (int d = dMIN; d <= dMAX; d++) {
        int iMIN = std::max(d, 1);
        int iMAX = std::min(M + d, N - 1);
        #pragma omp for
        for (int i = iMIN; i <= iMAX; i++) {
            int k = d < 0 ? 1 : -1;
            int j = M + d - i + k;
            if (word1[i - 1] != word2[j - 1]) {
                distMatrix[i * N + j] = std::min(std::min(distMatrix[(i - 1) * M + j],
                                                          distMatrix[i * M + j - 1]),
                                                 distMatrix[(i - 1) * M + j - 1]) + 1;
            } else {
                distMatrix[i * N + j] = distMatrix[(i - 1) * N + j - 1];
            }
        }
        if (d == -1) {
            d += 2;
        }
    }
}
```

Figure 4. Parallelization using a parallel block at the outer for loop.

As observable in Fig. 4, for the first approach, a **omp parallel** section is declared in correspondence of the outermost for loop, while an **omp for**

section is specified for the innermost for loop.
Threads are spawned at the first for loop and then they are distributed among the various iterations of the second for loop. Consequently, threads are generated only once for the entire execution of the algorithm, theoretically improving the execution time. This aspect will be examined in Sec. 4.1.
Regarding the clauses of the parallel section, the indices related to the diagonals of the matrix and the lengths of the strings are specified as firstprivate, allowing each thread to compute the indices for the inner for loop in a local variable. Due to this dependency, it was not possible to specify a collapse clause.
The matrix containing the Edit Distance values and the two strings are set as shared.

3.2. Multiple parallel overhead

```
for (int d = dMIN; d <= dMAX; d++) {
    int iMIN = std::max(d, 1);
    int iMAX = std::min(M + d, N - 1);
    #pragma omp parallel for default(none) firstprivate(iMIN, iMAX, d, M, N) \
    shared(distMatrix, word1, word2) num_threads(threads)
    for (int i = iMIN; i <= iMAX; i++) {
        int k = d < 0 ? 1 : -1;
        int j = M + d - i + k;
        if (word1[i - 1] != word2[j - 1]) {
            distMatrix[i * N + j] = std::min(std::min(distMatrix[(i - 1) * M + j],
                                                      distMatrix[i * M + j - 1]),
                                             distMatrix[(i - 1) * M + j - 1]) + 1;
        } else {
            distMatrix[i * N + j] = distMatrix[(i - 1) * N + j - 1];
        }
    }
    if (d == -1) {
        d += 2;
    }
}
```

Figure 5. Parallelization using a parallel block at the inner for loop.

In Fig. 5, we observe how both the omp parallel section and the omp for section are specified simultaneously at the innermost for loop. By doing so, threads will be generated and assigned for each iteration of the outermost for loop, which should result in higher computational times compared to the previous approach.
Observing the clauses of the parallel for section, compared to the previous approach the indices of the inner for loop are indicated as firstprivate, while the shared variables remain the same.

4. Esperiments conducted

Returning to the main.cpp file, the auxiliary functions written for the tests are as follows:

- **testStringSearchFMTime**: takes as input the two strings to be compared and it measures the time taken by the Full Matrix algorithm to compute the Edit Distance. Additionally, a boolean parameter, repeat, can be specified as true to repeat the test 10 times and average the obtained temporal results.
- **testStringSearchSDTime**: measures the computational time of the Skew Diagonal algorithm, with the same inputs as the previous function.
- **testStringSearchMRTime**: measures the computational time of the Matrix Row approach, with the same inputs as before.
- **testStringSearchSD_OMPTime**: measures the time needed for the execution of the Skew Diagonal algorithm parallelized using the first approach. In addition to the inputs required by the previous functions, it also requires an integer indicating the number of threads used for parallelization.
- **testStringSearchSD_OMP2Time**: similar to the previous function, but measures the time of the second approach for parallelizing Skew Diagonal.
- **compareTimeStringSearch**: executes the 5 preceding functions and compares the temporal results of the sequential algorithms with the parallel ones, saving the corresponding speedup values in a vector. This vector is then returned as output.
- **compareTimeStringSearchSD**: compares the computational times of Full Matrix and Skew Diagonal parallelized using the first approach, saving the speedup values in a vector returned as output. The reasons for specifically comparing these two algorithms are indicated in Sec. 4.1.

- **testEditDistance**: function where the tests are defined, it is executed by the main program.

In general, the parameters considered for the experiments are the number of threads implied in the parallelization and the length of the strings to compare, in addition to the different approaches presented for Edit Distance.
Results are reproduced 10 times and then averaged, and they are saved in specific CSV files located in the "results" folder. Graphs are generated using Python some code (not provided).
The results of the tests are saved in separate CSV files in the "results" folder.
Below are the tests conducted on the Edit Distance algorithms.

4.1. Test1: varying length with various sequential and parallel approaches

With this first test, we aim to compare the 3 sequential approaches with the two parallel versions. For comparison, various string lengths are considered (as indicated in Tab. 1), while the number of threads is kept constant at 16.
There are no specific expectations for this test; the only thing we could hypothesize is that the sequential Matrix Row approach might be the most challenging to improve (in terms of computational time) since it utilizes less memory.
For the two parallel versions we expect instead to observe what was assumed in Sec. 3 when presenting the two parallelizations.

	FM	SD	MR
P1, len=5k	3.2071	3.70324	2.31112
P2, len=5k	2.85164	3.29279	2.05497
P1, len=15k	3.3498	5.15474	2.43479
P2, len=15k	3.18759	4.90512	2.31688
P1, len=28k	3.63548	6.48565	2.58662
P2, len=28k	3.52203	6.28326	2.5059

Table 1. Table with the values of speedup varying by sequential algorithm (FM indicates Full Matrix, SD Skew Diagonal, and MR Matrix Row), parallel algorithm (P1 indicates the first approach, P2 the second), and word length (indicated by len).

Starting from the comparison between the three sequential algorithms, we notice that for each considered length there is a positive speedup, indicating the actual utility of parallelization with lengths comparable to those considered in the table. The sequential Skew Diagonal approach appears to be the technique that takes the longest execution time while, as anticipated, Matrix Row turns out to be the fastest of the three, although still not comparable to parallelization in terms of performance.
We also observe how increasing the length of the strings leads to a corresponding grow up in speedup, indicating that parallelization can better handle data structures containing intermediate calculations for Edit Distance by assigning work to threads (this aspect will be further analyzed in Sec. 4.2).
Finally, observing the comparison between the two parallel approaches, it is noted that the approach with a single parallel overhead is faster, although the improvements are rather modest, especially as the length of the compared strings increases (with a length of 28000, there is a difference of about 0.1 between the two speedups).
This suggests that with a lower number of threads, the approach with more parallel overhead could have potentially better performance because with more overhead, more time will be spent on thread management (generation and assignment of work), but by reducing the number of threads we limit this impact.
Therefore, the code was reran with a number of threads equal to 4.

	FM	SD	MR
P1, len=5k	1.94831	2.22286	1.3185
P2, len=5k	2.02005	2.3047	1.36705
P1, len=15k	1.85594	2.8919	1.34012
P2, len=15k	1.8461	2.87656	1.33301
P1, len=28k	1.84884	3.40718	1.33429
P2, len=28k	1.86523	3.43739	1.34611

Table 2. Table with speedup values varying by sequential algorithm, parallel algorithm, and word length (as in Tab. 1) using 4 threads for parallelization.

Observing the results presented in Tab. 2, we notice that this time the higher speedup is achieved, albeit slightly, with the second parallel

approach.
Between the two parallel approaches, there does not seem to be a significant difference; therefore, for the continuation of the tests, the implementation with a single parallel overhead was chosen, as it has slightly better performance using more threads. Among the objectives of the tests there is also to observe the performance by leveraging the hardware as much as possible, and choosing this approach ensures us lesser computational time.
Regarding the choice of sequential algorithms, it was decided to use the Full Matrix approach, not only because it is the most commonly used in practice, but also because from the results it offers performance on average compared to the other two sequential versions.

4.2. Test2: varying length

For the second test, we vary the length of the strings while keeping the number of threads constant at 8 to focus as much as possible on the actual impact of the string length parameter.
What we expect is that as the length of the strings increases, the value of speedup also increases, as we are affecting the complexity of the sequential Edit Distance algorithm, which generally is:

$$\mathcal{O}(n * m) \tag{2}$$

In Eq. (2), n and m represent the lengths of the two strings to be compared.

Observing the results in Fig. 6, what we expected partially occurred. Up to a length of 12000, the growth of the speedup is consistent, while from 20000 onwards, it stabilizes at around 2.6 (with a slight decrease to 2.4 at a length of 32000).
We can hypothesize that this occurs due to heap saturation, which may poorly handle the allocation of a 32000x32000 matrix. Even with workload distribution among the various threads the situation does not improve, although from Tab. 1, we can observe that with a length of 28000 and 16 threads, we achieve a speedup of approximately 3.5.
One possible solution is therefore to increase the number of threads working on the calculation of the Edit Distance matrix, at least until reaching a certain value beyond which we will also saturate the CPU (at that point, further increasing the number of threads will become ineffective).

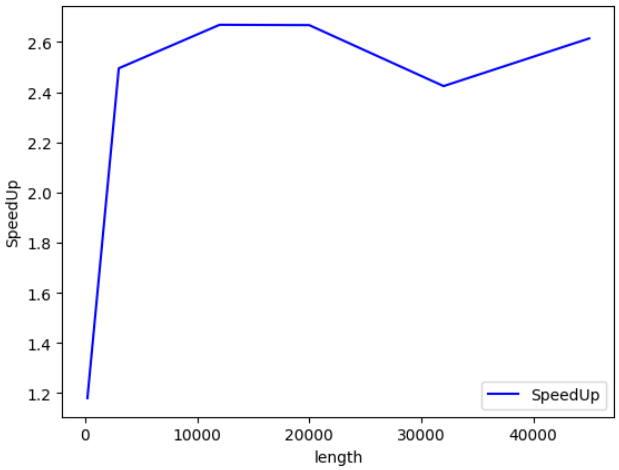

Figure 6. Values of speedup as the length of the compared strings increases. The lengths considered are 200, 3000, 12000, 20000, 32000, and 45000.

4.3. Test3: changing nThreads

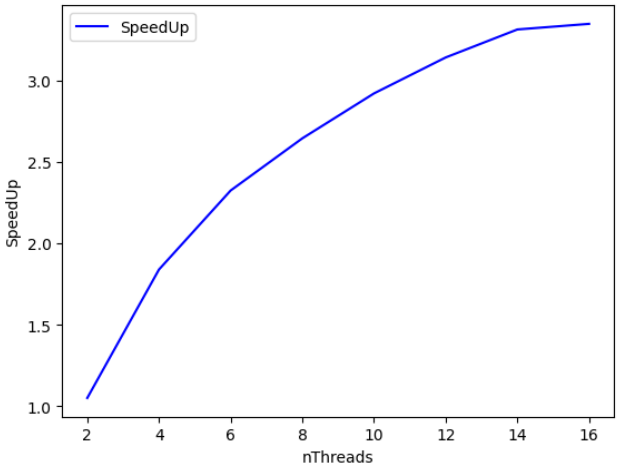

Figure 7. Values of speedup obtained by varying the number of threads used for the parallelized version of Edit Distance.

Finally, for the last test, we evaluate the impact of the parameter related to the number of threads, keeping the length of the strings constant at 20000. The number of threads is varied from 2 to 16, considering multiples of 2.
We expect that as the number of threads increases,

the performance of the parallelized algorithm will also increase, leading to higher values of speedup. As observed from the results in Fig. 7, with the increase in the number of threads the speedup between the sequential and parallel algorithms also increases. Naturally, the increase in speedup is not infinite; there will be a threshold value for the number of threads beyond which the CPU will become saturated.