Parallel Edit Distance

Parallel Programming for Machine Learning

Niccolò Arati

28/03/2024

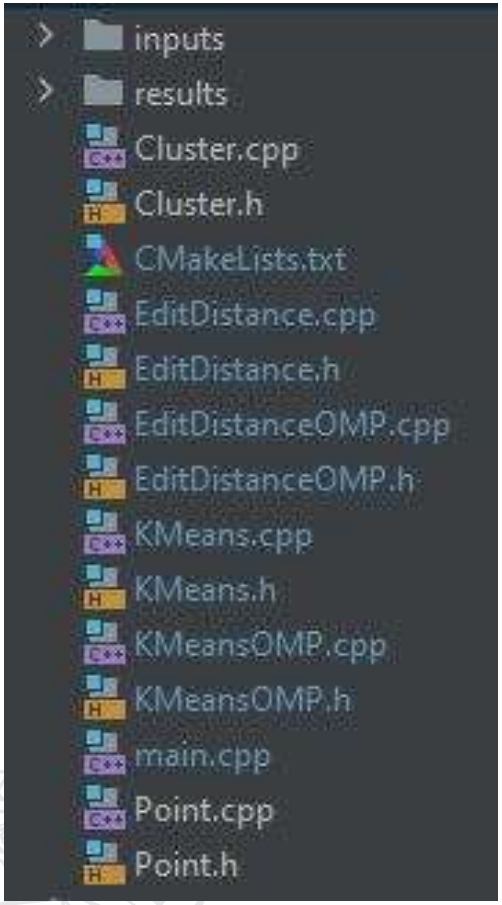
Introduction

Edit Distance, also known as Levenshtein Distance, is a measure of similarity between two strings A and B. It considers the minimum number of **basic operations** between characters (insertion, substitution, deletion) needed to transform string A into string B.
It was chosen to evaluate each basic operation with the same impact, so each of them increases the Edit Distance value by 1.

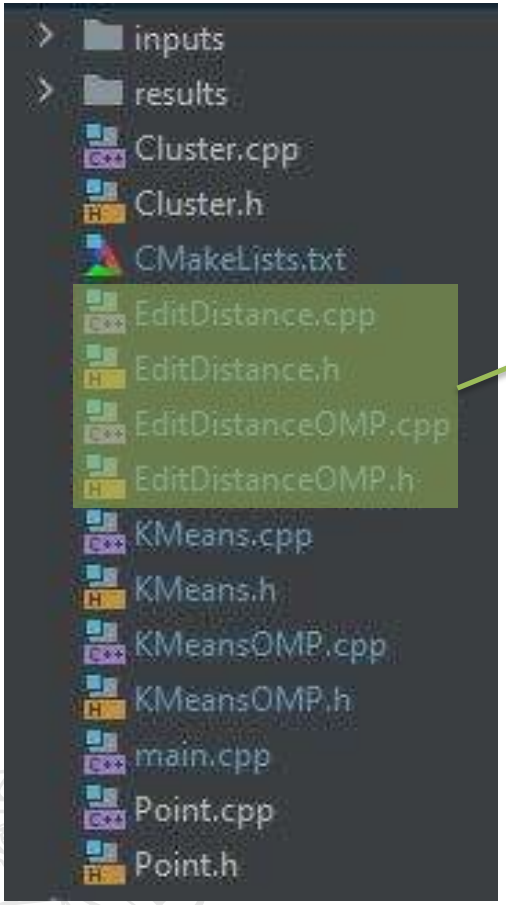
There are three **sequential** implementations of Edit Distance and two **parallel** implementations. The purpose of this work is to observe the **speedup** obtained with the parallel versions compared to the sequential ones.

The programming language is C++ (MinGW compiler, CLion IDE), and parallelization was done with **OpenMP**.
All experiments were conducted on the ssh server "papavero.dinfo.unifi.it".

Code Structure

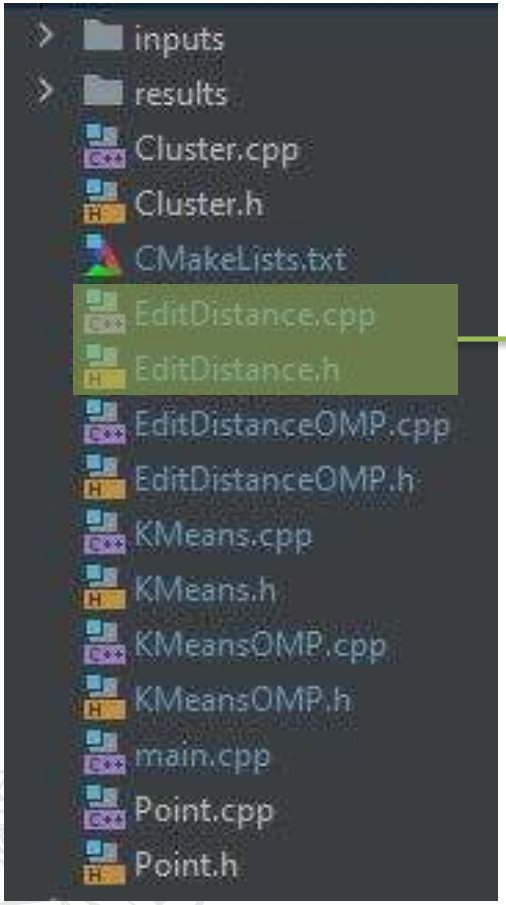


Code Structure



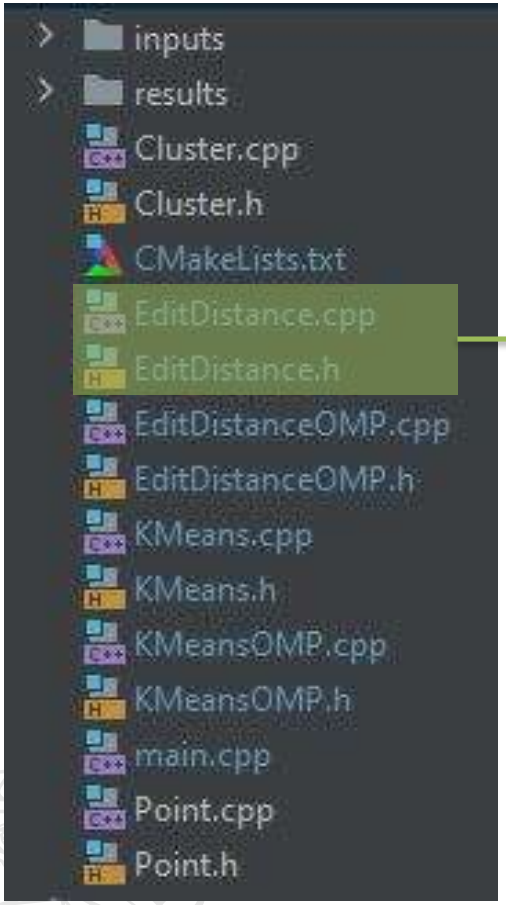
Files included in this project. The remaining ones are related to another project with a common directory.

Code Structure



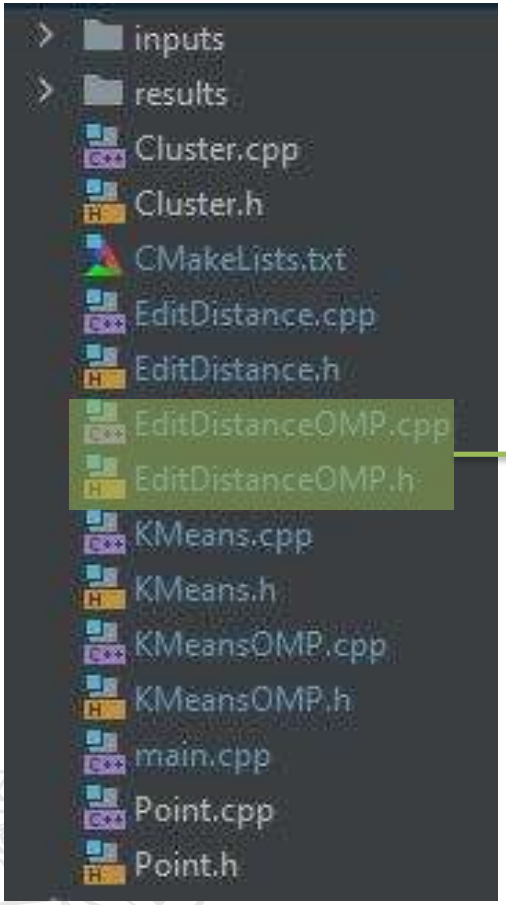
Files containing the sequential implementations of Edit Distance: Full Matrix, Skew Diagonal, and Matrix Row.

Code Structure



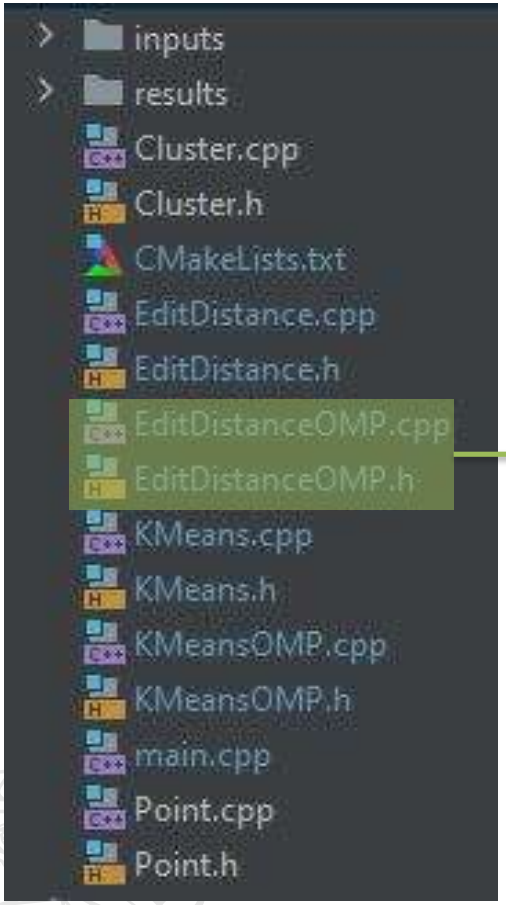

```
int levenshteinDistFM(const std::string& word1, const std::string& word2, int length1, int length2);

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);
```

Code Structure



Files containing the two parallelized versions of Edit Distance using as base the Skew Diagonal approach.

Code Structure




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

int levenshteinDistSD_OMP2(const std::string& word1, const std::string& word2, int length1,
                           int length2, int threads);
```

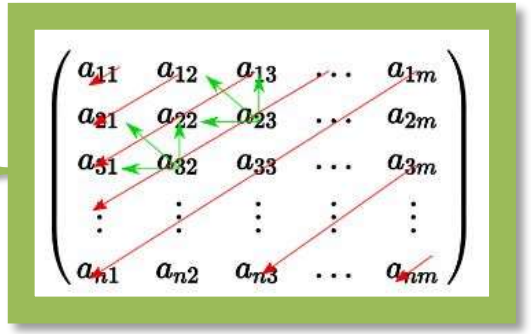
Sequential Algorithm: Full Matrix

```
int levenshteinDistFM(const std::string& word1, const std::string& word2, int length1, int length2) {
    int N = length1 + 1;
    int M = length2 + 1;

    std::vector<int> distMatrix( n: N * M);

    if (length1 == 0) {
        return length2;
    }
    if (length2 == 0) {
        return length1;
    }
    for (int i = 0; i <= length1; i ++) {
        distMatrix[i] = i;
    }
    for (int j = 0; j <= length2; j ++) {
        distMatrix[j * M] = j;
    }
    //matrix filling
    for (int i = 1; i <= length1; i ++) {
        for (int j = 1; j <= length2; j ++) {
            int substitutionCost = (word1[i - 1] == word2[j - 1]) ? 0 : 1;
            distMatrix[i * N +j] = std::min(std::min(distMatrix[(i - 1) * M + j] + 1, //deletion cost
                                                     distMatrix[i * M + j - 1] + 1), //insertion cost
                                            distMatrix[(i - 1) * M + j - 1] + substitutionCost); //substitution cost
        }
    }

    int result = distMatrix[length1 * N + length2];
    return result;
}
```

Edit Distance matrix computation.

Sequential Algorithm: Skew Diagonal

```
//begin algorithm
int dMIN = 1 - M;
int dMAX = N - 1;
for (int d = dMIN; d <= dMAX; d ++) {
    int iMIN = std::max(d, 1);
    int iMAX = std::min(M + d, N - 1);
    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;
    }
}
int result = distMatrix[length1 * N + length2];
return result;
```



Sequential Algorithm: Matrix Row

```
int levenshteinDistMR(const std::string& word1, const std::string& word2, int length1, int length2) {
    std::vector<int> prevRow( n: length2 + 1,  value: 0);
    std::vector<int> currRow( n: length2 + 1,  value: 0);

    if (length1 == 0) {
        return length2;
    }
    if (length2 == 0) {
        return length1;
    }
    for (int j = 0; j <= length2; j ++) {
        prevRow[j] = j;
    }
    for (int i = 0; i < length1; i ++) {
        currRow[0] = i + 1;
        for (int j = 0; j < length2; j ++) {
            int deletionCost = prevRow[j + 1] + 1;
            int insertionCost = currRow[j] + 1;
            int substitutionCost;
            if (word1[i] == word2[j]) {
                substitutionCost = prevRow[j];
            }
            else {
                substitutionCost = prevRow[j] + 1;
            }
            currRow[j + 1] = std::min(std::min(deletionCost, insertionCost), substitutionCost);
        }
        prevRow = currRow;
    }
    return currRow[length2];
}
```

Calculation of the i-th row of the matrix based on the (i-1)-th row.

Parallelized Skew Diagonal

Single parallel overhead

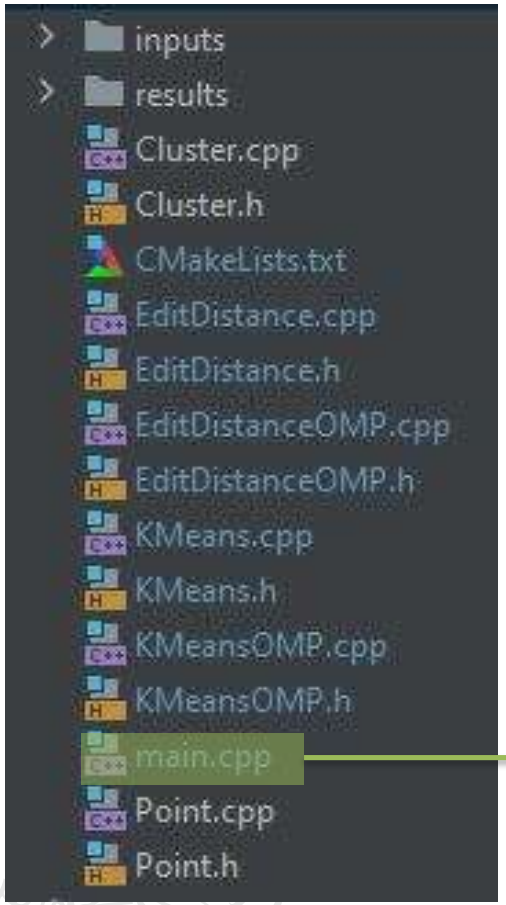
```
#pragma omp parallel default(none) firstprivate(dMIN, dMAX, M, N) \
shared(distMatrix, word1, word2) num_threads(threads)
{
    for (int d = dMIN; d <= dMAX; d++) {
        int iMIN = std::max(d, 1);
        int iMAX = std::min(M + d, N - 1);
        #pragma omp for
        for (int i = iMIN; i <= 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;
        }
    }
}
```

Parallelized Skew Diagonal

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;
    }
}
```

Test




```cpp
//test EditDistance
std::string random_string(std::size_t length, int seed) {...}

//Sequential tests
double testStringSearchFMTime(const std::string& word1, const std::string& word2, bool repeat) {...}

double testStringSearchSDTime(const std::string& word1, const std::string& word2, bool repeat) {...}

double testStringSearchMRTime(const std::string& word1, const std::string& word2, bool repeat) {...}


//Parallel tests
double testStringSearchSD_OMPTime(const std::string& word1, const std::string& word2, int nThreads, bool repeat) {...}

double testStringSearchSD_OMP2Time(const std::string& word1, const std::string& word2, int nThreads, bool repeat) {...}


//compare all algorithms
std::vector<double> compareTimeStringSearch(const std::string& word1, const std::string& word2, int nThreads,
                                            bool repeat = true) {...}

//compare Full Matrix and Skew Diagonal Parallel
double compareTimeStringSearchSD(const std::string& word1, const std::string& word2, int nThreads,
                                             bool repeat = true) {...}

void testEditDistance() {...}
```

Test 1

Parallelization with a single parallel overhead, 16 threads

	Full Matrix	Skew Diagonal	Matrix Row
P1, len = 5000	3. 2071	3. 70324	2. 31112
P2, len = 5000	2. 85164	3. 29279	2. 05497
P1, len = 15000	3. 3498	5. 15474	2. 43479
P2, len = 15000	3. 18759	4. 90512	2. 31688
P1, len = 28000	3. 63548	6. 48565	2. 58662
P2, len = 28000	3. 52203	6. 28326	2. 5059

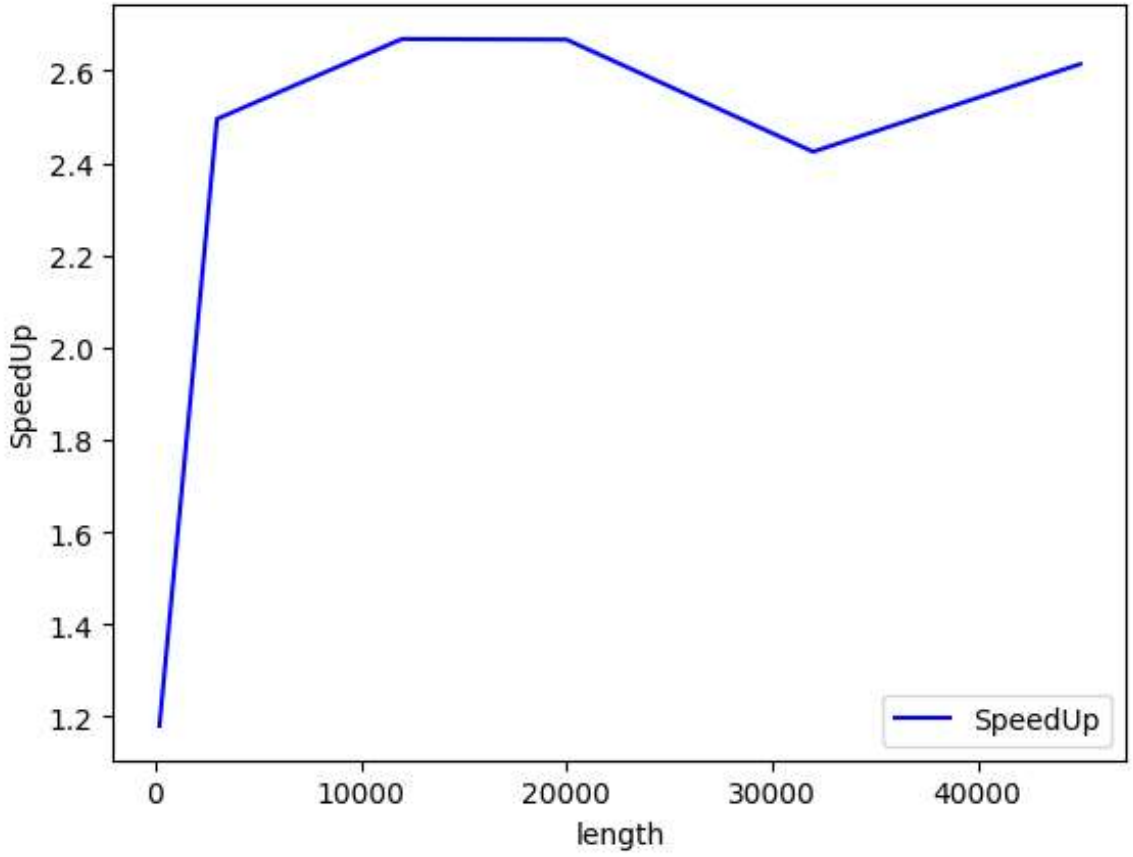
Test 1

Parallelization with multiple parallel overhead, 4 threads

	Full Matrix	Skew Diagonal	Matrix Row
P1, len = 5000	1. 94831	2. 22286	1. 3185
P2, len = 5000	2. 02005	2. 3047	1. 36705
P1, len = 15000	1. 85594	2. 8919	1. 34012
P2, len = 15000	1. 8461	2. 87656	1. 33301
P1, len = 28000	1. 84884	3. 40718	1. 33429
P2, len = 28000	1. 86523	3. 43739	1. 34611

Test 2

Varying string lengths, speedup calculated between Full Matrix and Skew Diagonal with single parallel overhead (8 threads)



Test 3

Varying the number of threads, with strings length of 20,000.