Parallel Edit Distance

Parallel Programming for Machine Learning

Niccolò Arati

28/03/2024

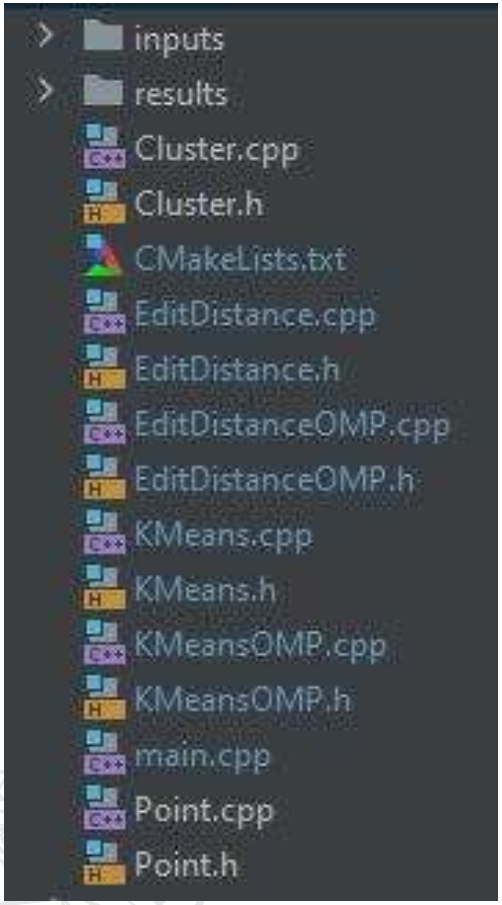
Introduzione

Edit Distance, chiamata anche Distanza di Levenshtein, e una misura di similarità tra due stringhe A e B. Considera numero minimo di **operazioni base** tra caratteri (inserimento, sostituzione, rimozione) da effettuare per trasformare la stringa A nella stringa B.
Si e scelto di valutare ogni operazione di base con lo stesso impatto, infatti ciascuna di esse aumenta di 1 il valore di Edit Distance.

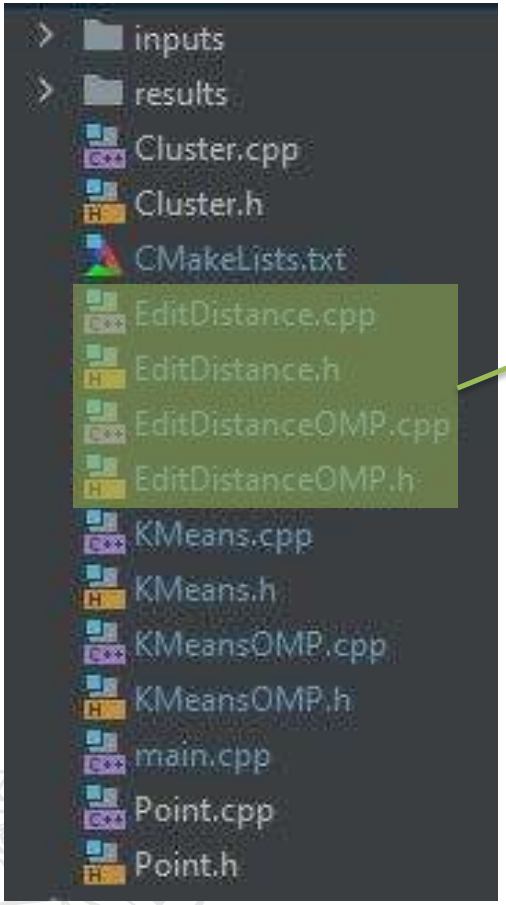
Sono presenti tre implementazioni di Edit Distance **sequenziali** e due implementazioni **parallele**. Lo scopo di questo elaborato è quello di osservare lo **speedup** ottenuto con le versioni parallele rispetto a quelle sequenziali.

Il linguaggio di programmazione utilizzato è il C++ (compilatore MinGW, IDE CLion) e la parallelizzazione è stata svolta con **OpenMP**.
Tutti gli esperimenti sono stati svolti sul server ssh ”papavero.dinfo.unifi.it”.

Organizzazione del codice

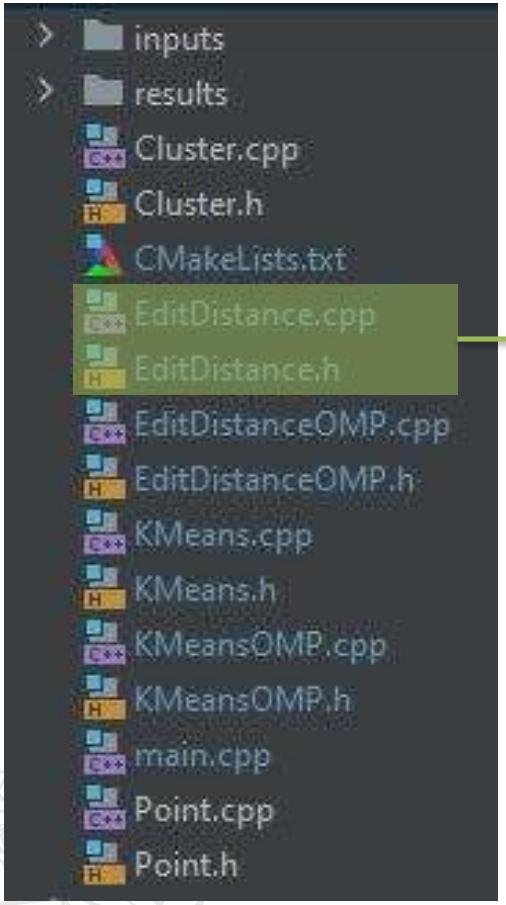


Organizzazione del codice



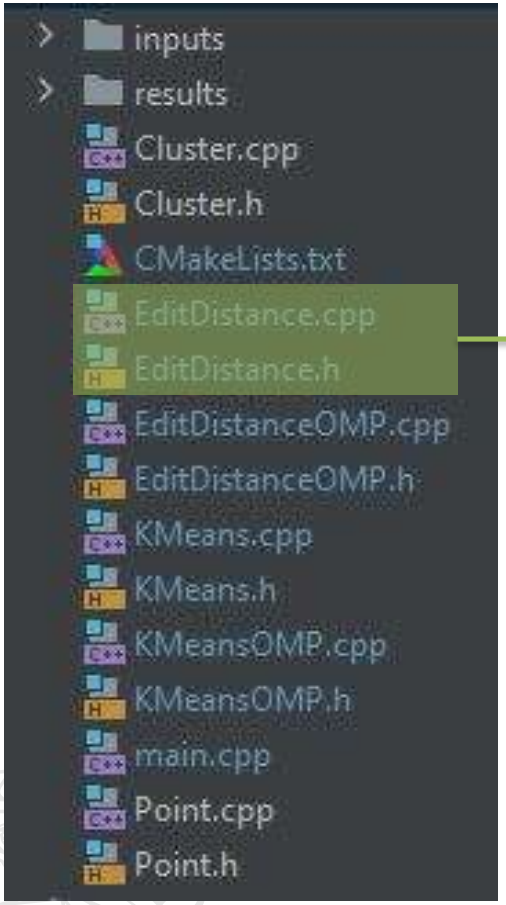
File compresi in questo progetto. I restanti sono relativi ad un altro progetto con directory comune

Organizzazione del codice



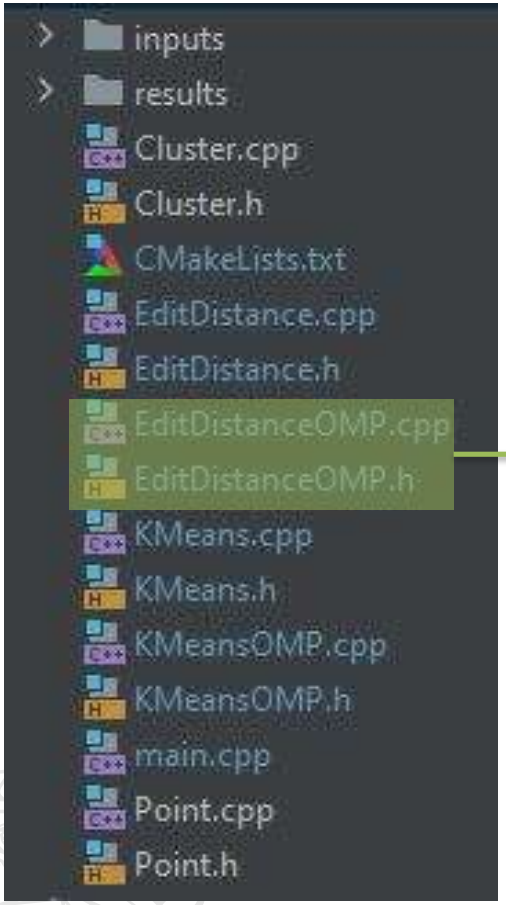
File contenenti le implementazioni sequenziali di Edit Distance: Full Matrix, Skew Diagonal e Matrix Row

Organizzazione del codice



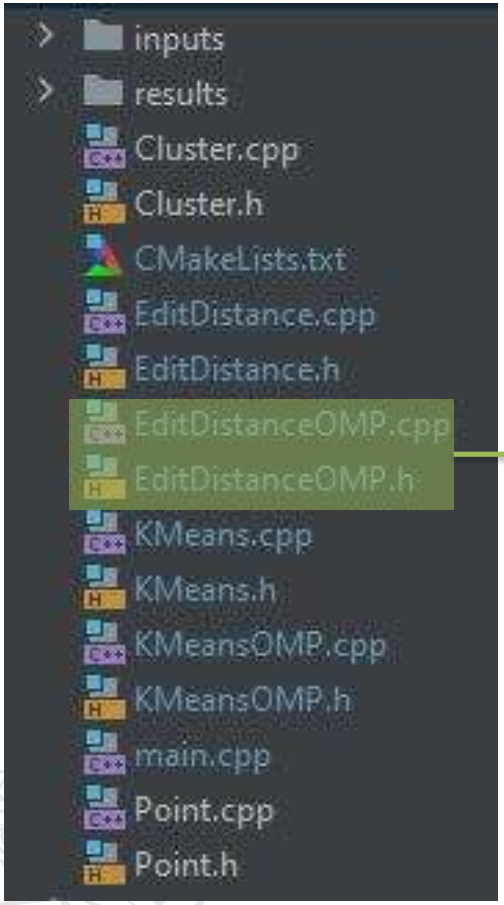

```
int levenshteinDistFM(const std::string& word1, const std::string& word2, int length1, int length2);

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);
```

Organizzazione del codice



File contenenti le due versioni parallelizzate di Edit Distance con approccio Skew Diagonal

Organizzazione del codice




```
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);
```

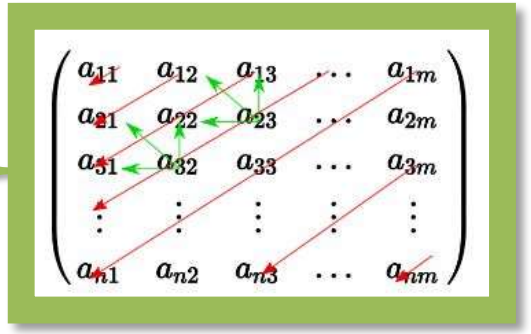
Algoritmo Sequenziale: Full Matrix

```
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;
}
```

Calcolo matrice di Edit Distance

Algoritmo Sequenziale: 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;
```



Algoritmo Sequenziale: 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];
}
```

Calcolo della i-esima riga della matrice basandosi sulla riga (i-1)esima

Skew Diagonal Parallelizzato

Una singola parallel overhead

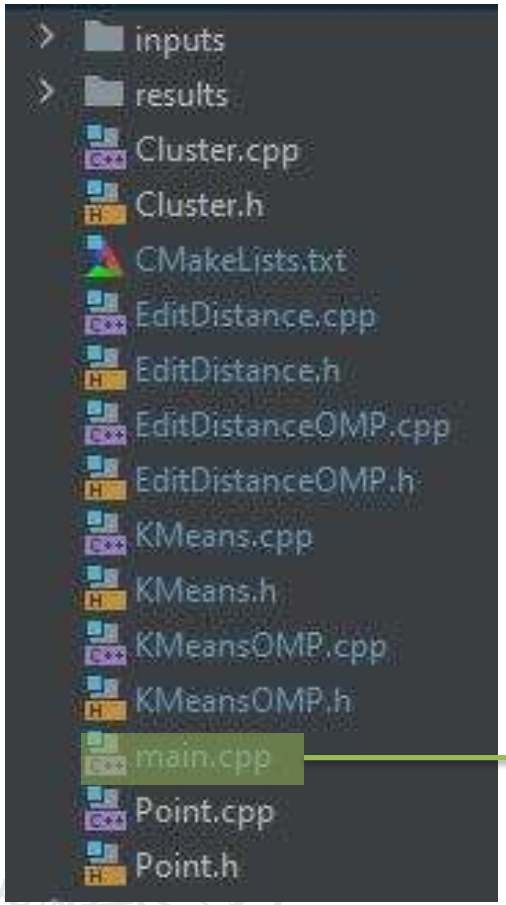
```
#pragma omp parallel default(none) firstprivate(dMIN, dMAX, M, N) \
shared(distMatrix, word1, word2) num_threads(threads)
{
    for (int d = dMIN; d <= dMAX; d++) {
        int iMIN = std::max(d, 1);
        int iMAX = std::min(M + d, N - 1);
        #pragma omp for
        for (int i = iMIN; i 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

Skew Diagonal Parallelizzato

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

Parallelizzazione con singola 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

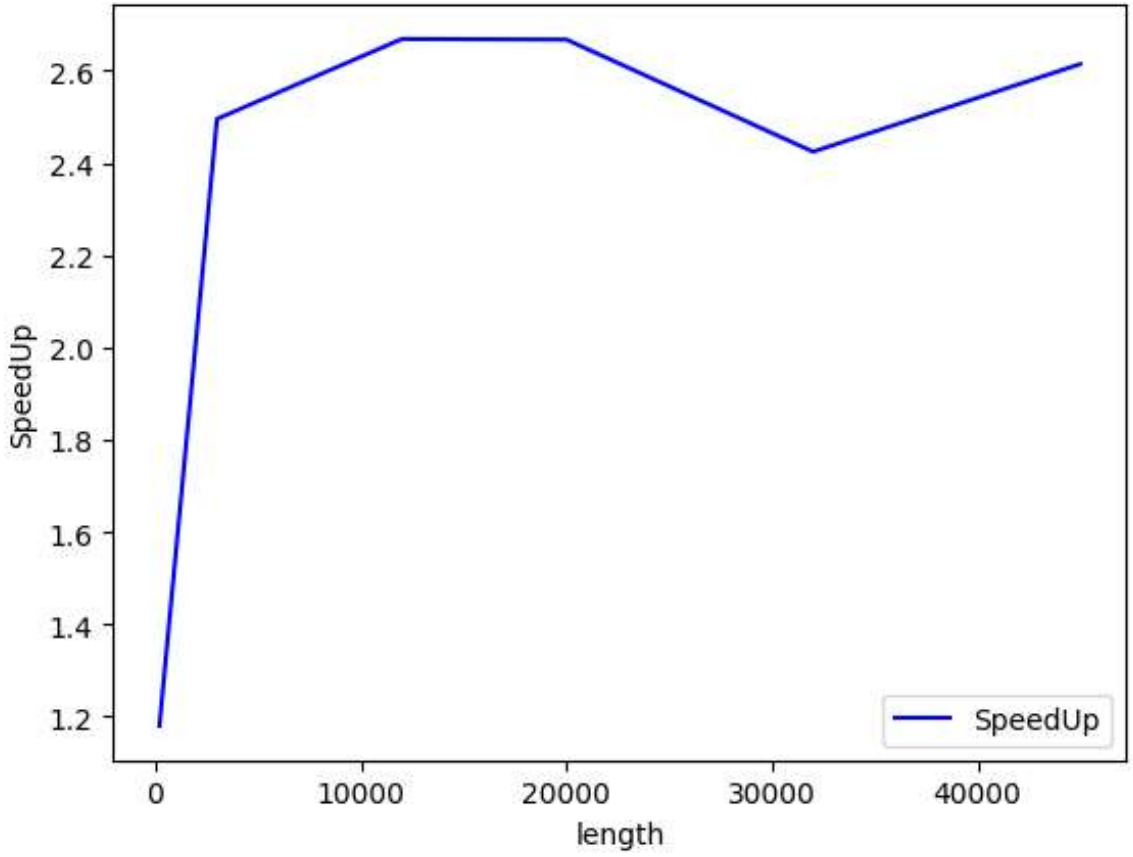
Test 1

Parallelizzazione con 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

Variare lunghezza delle stringhe, con 8 threads e speedup calcolato tra Full Matrix e Skew Diagonal con singola parallel overhead



Test 3

Variare numero di threads, con lunghezza stinghe di 20000