PC-2019/20

Niccolò Corsani
E-mail address
niccolo.corsani@stud.unifi.it

Abstract

The purpose of "Edit Distance" is to calculate the difference between two strings. The difference is defined in terms of the elementary operations necessary to convert one string into another. In particular, the elementary operations are: insertion, elimination, replacement. The sum of the number of the total of these transactions is the value that represents the difference. For example, two changes are needed to transform "bar" into "biro":

1. "bar" - "bir" (replacement of 'a' with 'i')

2. "bir" - "biro" (insertion of 'o')

The algorithm in question is used in many types of applications, such as: The automatic correction of spelling errors, the analysis of "Natural language processing" and finally in the biological field in the analysis of nucleotide DNA sequences. This area in particular is having an ever greater development given the size of the data to be computed.

1. Introduction

The project was made in java and the technologies used to achieve parallelism were those of:

-ThreadPool.

-Executor

-Future

At the basis of the algorithm there is the computation of a matrix. This matrix will have a size equal to length-string-1 * length-string-2. The algorithm has been implemented with three different techniques: Sequential case:

-Parallel case on columns

-Parallel case on diagonals

1.1. ManageThread class

The manageThread class is the middle class of the software. This deals with the complete management of threads in every aspect. In particular, it is composed of 2 methods:

-"ManageThreadsColumn" and "ManageThreadDiagonal".

The two methods deal with instantiating the various threads, which will process the respective data. The static class Executors was used to manage the threads. This class allows you to create different types of "Thread-Pool" instances in "Factory" mode depending on the purpose. For example:

-exServFixed=Executors.newFixedThreadPool(4)

-exServFixed=Executors.newScheduledThreadPool(4)

Once the Threadpool has been created, a while cycle begins in which through the "submit(new class)" method it is possible to create an instance of the class and execute the "Call" method. This practice is similar to what is done with the "Thread" class and the "start" method, however with "Executor" there is the possibility of obtaining a return value from the call method. The value that returns the call method, in addition to providing useful information, can be used to manage the synchronization of the various threads. To achieve this, the "Future" class is used in combi-

nation with the ”ExecutorService” class.

An instance of Future is first created by specifying the type of the return value of the ”call” function. When the ”submit” method is then called, proceed as follows:

```
Future<Integer> future;
     while (counterColumns <=
         stringaColonne.length()) {
       /* non bloccante */
       future = exServFixed.submit(new
           EditDistanceParallel(stringaRighe,
           stringaColonne, counterColumns,
           table));
       counterColumns++;

     }

     while (!future.isDone());
```

In this way, the mainThread will wait for the conclusion of the execution of the various threads. To observe the obtained value, you can use the ”future.get ()” function. Note how ”future = exServFixed.submit (...)” is not blocking and therefore the parallelism of the Threads is maintained.

What has been said so far represents the non-functional behavior of the system, the functional implementation of the system and therefore the ”Edit distance” calculation algorithm will now be described.

1.2. Edit Distance parallel implementation

The algorithm starts from the while loop described previously. Each instance launched by the ”submit” method receives as a parameter the string of rows, the string of columns, a counter, and the matrix of values. In the case of the ”ManageThreadsDiagonal” method, the counter represents the value of the column or row from which to start the computation. In particular, the counter

	""	P	A	R	T
""	0	1	2	3	4
S	1	1	2	3	4
P	2	1	2	3	4
A	3	2	1	2	3
R	4	3	2	1	2
T	5	4	3	2	1
A	6	5	4	3	2
N	7	6	5	4	3

The orange thread needs the blue one to be ahead of it, as well as the blue one with the pink one.

goes from 1 to stringCollonna.length() and then starts again from 1 up to stringRighe.length(), then it will be the class ”EditDistanceDiagonal” thath will manage the computations of the diagonal through a for loop. This class has the purpose of implementing the computation on the diagonals, ensuring synchronization between the various threads.

Since the computations starts from the first row and first column in the matrix and then go to larger values, the condition for which a Thread can continue its computations is that every other Thread computing at his left is ahead of it.

```
int tableDiagonal[][] = new
    int[stringaRighe.length() +
    1][stringaColonne.length() + 1];

int counterColumns = 1;

exServFixed =
    Executors.newFixedThreadPool(4);

long inizio =
    System.currentTimeMillis();

while (counterColumns <
    stringaColonne.length()) {
  future = exServFixed
      .submit(new
          editDistanceDiagonal(stringaRighe,
          stringaColonne,
```

```
            counterColumns,
            tableDiagonal));
    counterColumns++;
}
while (!future.isDone());
int counterRighe = 1;
while (counterRighe <=
    stringaRighe.length()) {
    future = exServFixed
        .submit(new
            editDistanceDiagonal(stringaRighe,
            stringaColonne,
            counterRighe,
            tableDiagonal));
    counterRighe++;
}
while (!future.isDone());
```

In the case of the ”ManageThreadsColumn” method, the procedure is similar to that performed with the diagonals. The ”ManageThreads” class manages the call of threads with the same arguments mentioned above. In this case, however, the counter does not represent rows and columns, but only the columns. Once each Thread gets the value of the column from which to start the computation, each Thread works managing the synchronization as in the case of diagonals.

Since this synchronization between threads, in addition to the management of the instances of the classes that implement ”Callable” generates a lot of overhead and finally due to the low computation time of diagonals and columns, the sequential execution time is less than the parallel one. As a solution to this a future implementation is possible, combining the sequential and parallel approach. Before proceeding with the performance analysis, a note useful for the continuation of the software is required. In fact, it is necessary to explicitly declare to allocate a memory in the Heap of a certain size. This is because the matrix that will be generated for computational purposes can take very large values. To do this it is need the ”-Xmx2048m” command when compiling the .java file or otherwise in eclipse it is possible to add this instruction to the execution properties. This will have the effect of increasing the size of the memory useful for program to 2048 Megabyte.

1.3. Performance

As a premise in this paragraph it should be noted that the computer on which the software was tested has a processor with a total of 4 cores. The performances were tested by varying the two parameters: String size, Thread number. Graphs were made by analyzing the times on strings of different lengths.

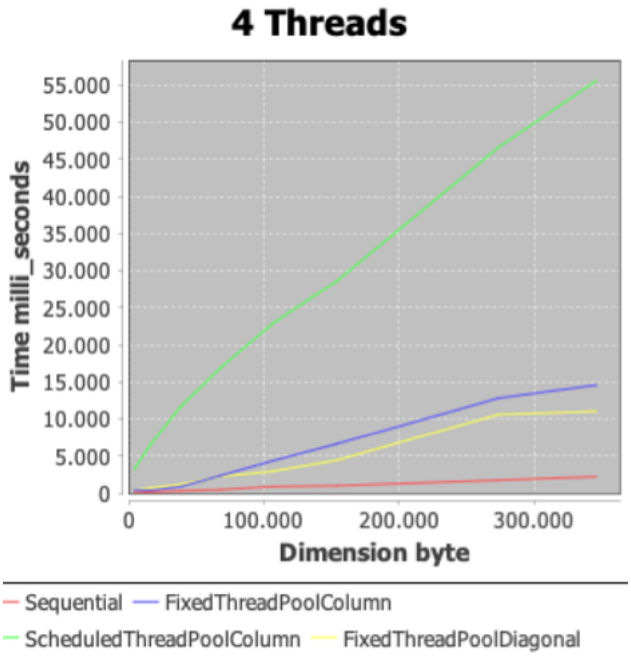
The first graph represents the 4 cases with 4 threads:

1) The sequential case.

2) The parallel case with FixedThreadPool (computation on columns).

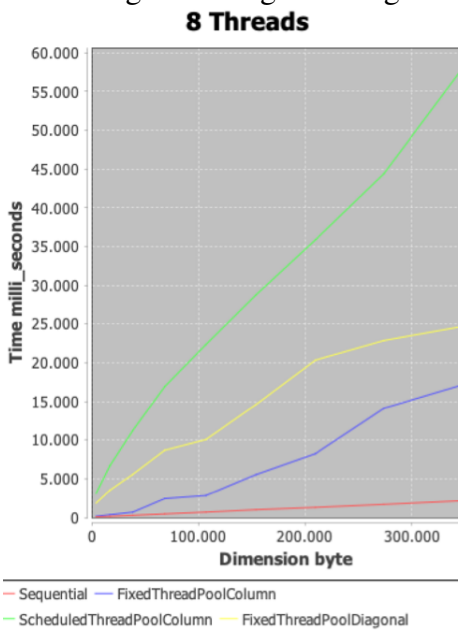
3) The parallel case with ScheduledThreadPool (computation on columns).

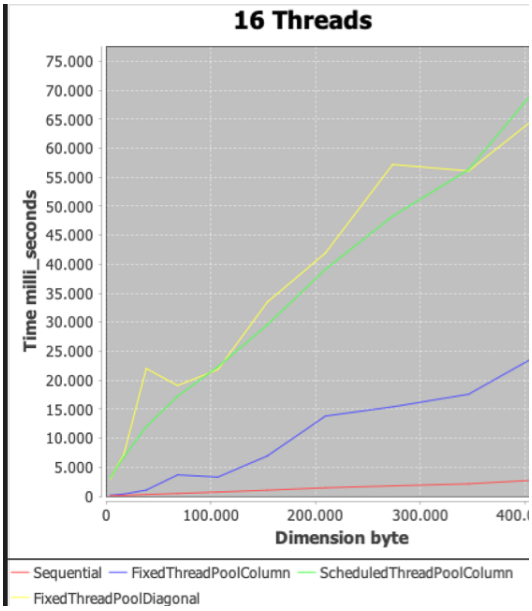
4) The parallel case with FixedThreadPool (computation on diagonals).



As already mentioned above, the sequential case is the most performing, this because the

computations on both columns and diagonals are not proportioned to the hoverhead they generate. In particular, the mixed ”parallel / diagonal” approach is the best solution. Furthermore, it can be easily deduced that, in the case of computations on columns, the execution times are greater than in a sequential version, the greater the ”stringRows.length/ StringString.length” is .





In the images above you can see the performances with 8 and 16 Threads.

Each graph was calculated for 10 different string dimensions.

It is clear that parallel execution time is subject to a large overhead. This is due to the while loop that manages instances of the classes that implement ”Callable”.

1.4. Solution less overhead

As can be seen in the code below, the ”new EditDistanceParallel (..)” instruction must be limited to the minimum.

```
while (counterColumns <=
    stringaColonne.length()) {
       /* non bloccante */
       future = exServFixed.submit(new
           EditDistanceParallel(stringaRighe,
           stringaColonne, counterColumns,
           table));

       counterColumns++;
}
```

To achieve this, an alternative solution has been created, where each Thread manages the iterations on multiple diagonals and not just one. This software has been developed for both columns and diagonals. The main idea is therefore to create only 4 instances, each of which will synchronize at best with the other threads. In particular, the synchronization techniques are the same as those implemented in the previous software:

Each thread checks that there is all the data to compute right and if not, release the CPU.

```
  for (int i = 1; i < 5; i++) {
       future = exServFixed.submit(new
           EditDistance(i, table,
           stringaRighe, stringaColonne));
    }
```

Once a Thread has finished computing a diagonal or column, increase its counter by 4 and start computing the next diagonal or column. As for the case on columns, each Thread will execute this code:

```
while (threadNumber <= s1.length()) {
       for (int colonna = 1; colonna <
           table[threadNumber].length;
           colonna++) {
         while (table[threadNumber -
             1][colonna] == -1 ||
```

```
            table[threadNumber][colonna -
            1] == -1
             || table[threadNumber -
                1][colonna - 1] == -1) {
          Thread.yield();
                 }
        int del = table[threadNumber -
            1][colonna] + 1;
        int ins =
            table[threadNumber][colonna -
            1] + 1;
        int rep = table[threadNumber -
            1][colonna - 1]
             + (s1.charAt(threadNumber -
                1) == s2.charAt(colonna -
                1) ? 0 : 1);
        table[threadNumber][colonna] =
            Math.min(del, Math.min(ins,
            rep));
      }
      threadNumber = threadNumber + 4;
    }
    return table[s1.length()][s2.length()];
```

A little more complex is the situation with regards to the diagonals, since the algorithm is divided into two parts: The first part starts its computation from the first row, scrolling through all the columns. The second part scrolls all the rows starting from the last column. To do this, two while loops have been implemented:

```
while (counterColumns < 5) {
  future = exServFixed
       .submit(new
           editDistanceDiagonal(stringaRighe,
           stringaColonne,
           counterColumns,
           tableDiagonal, 0));
  counterColumns++;

}

while (!future.isDone())
  ;
future.get();

System.out.println("Inizio
    righe......");

int counterRighe = 1;
exServFixed =
    Executors.newFixedThreadPool(4);
while (counterRighe < 5) {
  future = exServFixed
       .submit(new
           editDistanceDiagonal(stringaRighe,
           stringaColonne,
           counterRighe,
           tableDiagonal, 1));

  counterRighe++;
}

while (!future.isDone())
  ;
System.out.println("L'ultimo Thread ha
    completato il suo lavoro: " +
    future.get());
```

1.5. Performance less overhead

The change is very useful, the graph shows that for a given size of the analyzed matrix the time is much shorter. For example, for 200,000 byte matrices, from 10 seconds to less than one second for the columns, the difference is smaller for the diagonals. The graphs below show the curves for a 4 Thread FixedThreadPool. The dimensions analyzed are similar to those shown above.