

Approximate String Matching using Edit distance with application to genomic

Michele Inchingolo
Institutional ID: 482748
Email:

michele.inchingolo01@universitadipavia.it

1. INTRODUCTION

The goal of this project is to analyze the differences in performance between serial and parallel implementation for a certain problem, that in this case corresponds to approximate string matching with application to genomic or proteomics.

There are two variants of approximate string matching problem, *k-mismatch* problem and *k-difference* problem. I decided to study the *k-difference* problem. Given a pattern $P = p_1 p_2 \dots p_m$ with length m , a text $T = t_1 t_2 \dots t_n$ with length n , and a positive integer threshold k , the *k-difference* problem concerns finding all substrings of T , whose edit distance to P is at most k .

The edit distance (Levenshtein distance) is defined as the minimum number of edits (insertion, deletion, and substitution of a single character) needed to make the two strings equal. This metric has been used in order to quantify how much a substring of T is different from pattern P . Taking into account this definition it is possible to notice that if k has been set to zero, it will be possible to search for substrings that exactly match with the pattern.

The implementation for this problem will be able to find both the number of substrings and the sequence of edit operations.

2. EDIT DISTANCE

As said before the edit distance measure the differences between two strings in terms of edit operations, this statement can be summarized with this recursive expression:

$$\text{lev}_{a,b}(i, j) = \begin{cases} \max(i, j) & \text{if } \min(i, j) = 0, \\ \min \begin{cases} \text{lev}_{a,b}(i-1, j) + 1 \\ \text{lev}_{a,b}(i, j-1) + 1 \\ \text{lev}_{a,b}(i-1, j-1) + 1_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases}$$

(1). Recursive definition for edit distance

where $\text{lev}_{a,b}(i, j)$ as to be intended as the minimum number of differences between $p_1 p_2 \dots p_i$ and any substring of T at the j -th index, with a and b respectively equal to P e T .

For each case in which i is equal to zero we are comparing an empty pattern with each substring of T that ends in j , so for each comparison the edit distance will be equal to the cost of inserting j characters to the pattern. The same happens when j is equal to zero, with the difference that in this case the cost has to be referred to deletion of i characters from the pattern.

In order to perform an approximate string searching it was necessary to modify this statement, so when i is equal to zero I will always consider edit distance zero too, since I want an edit distance search where the cost of starting the matching is independent of the position in the text, so that a match in the middle is not prejudiced against. This is due to the fact that the edit distance in the original statement has to be applied for comparing the two whole strings, instead it's needed to perform a comparison between the entire pattern and each substring in the text, without paying each time a cost equal to j , correspondent to the number of characters that it will be necessary to add at the pattern in order to start the matching at that index.

In all other cases in which both i and j are different from zero, it's necessary to select the minimum between three possibilities to extend smaller strings, that are $p_1 p_2 \dots p_i$ and $t_1 t_2 \dots t_j$:

- $\text{lev}_{a,b}(i-1, j) + 1$, means that there is an extra character in the pattern to account for, so it's not necessary to advance the text pointer and pay the cost of an insertion.
- $\text{lev}_{a,b}(i, j-1) + 1$, now it's present an extra character in the text to remove, so it's not necessary to advance the pattern and pay the cost for a deletion.
- $\text{lev}_{a,b}(i-1, j-1) + 1_{(a_i \neq b_j)}$, means we either match or substitute the i -th and j -th characters, depending upon whether the previous characters are the same or not.

Each edit operation costs 1 except for the match that does not require any operation.

It's possible to select different costs for each operation in order to perform more flexible search strategies, but I preferred this one since it allows to find the minimum edit distance that is perfect for the purpose of the project.

The main problem with this algorithm is that it's very time-consuming, since it should recalculate the previous values recursively. A simple observation is that most of these recursive calls are computing things that have been previously computed, so it's only necessary to store the values for each (i, j) pair in a matrix. This method takes advantage from dynamic programming implementation of the algorithm^[6], and it will lead to the creation of a matrix of dimension $[m+1, n+1]$.

j	0	1	2	3	4	5	6	7	8
i		C	A	T	G	A	C	T	G
0	0	0	0	0	0	0	0	0	0
1	T	1	1	1	0	1	1	1	0
2	A	2	2	1	1	1	1	2	1
3	C	3	2	2	2	2	1	2	2
4	T	4	3	3	2	3	2	1	2
5	G	5	4	4	3	2	3	2	1

Figure 1. Dynamic programming matrix for edit distance with all zero in the first row for approximate string matching.

The figure represents an example of what has been discussed so far and it is easy to notice that the solution for the k -difference approximate string matching problem is to consider all the j -th indexes for which the values on the last row of the matrix are lower than, or equal, to threshold k .

So counting all the j -th indexes that respect this condition will return the total number of substrings found, but in addition to this information it's also possible to return the sequence of edit operations. This is possible because during the computation of the matrix it will also stored the type of edit operation, since each of them has been identified with a number: 0 for match/substitution, 1 for insert and 2 for delete.

An example, taking the pattern in figure above $P = TACTG$ and the substring of T at index 5, $CATGA$, between which there is an edit distance of 3, the image below shows the sequence of edit operations that differentiate the two strings.

j	0	1	2	3	4	5	6	7	8
i		C	A	T	G	A	C	T	G
0		-1	-1	-1	-1	-1	-1	-1	-1
1	T	2	0	0	0	0	0	0	0
2	A	2	0	0	2	0	0	2	0
3	C	2	0	2	0	0	0	1	0
4	T	2	2	0	0	0	2	0	1
5	G	2	2	0	2	0	2	2	0

Figure 2. Example of reconstructing edit path between pattern and substring CATGA.

The green arrows represent the insert operation, the red ones the delete operations and the blue ones match/substitution operations.

T	A	C	T	G	
↑	↑	↓	↑	↑	↑
S	M	D	M	M	I
C	A	T	G	A	
C	A	T	G	A	
↑	↑	↑	↑	↑	↑
S	M	M	M	I	
C	A	T	G	A	

Image on the left represents how to apply the edit path in order to transform the pattern in CATGA.

Three edit operations are necessary:

- 1) Deletion of C.
- 2) Insertion of A.
- 3) Change T in C.

3. DATASET

3.1. INPUT DATA

Since the project focuses on genomic applications I decided to use FASTA formatted data both for pattern and text, since is commonly used in genomic or proteomic applications.

This format is a text-based format for representing either nucleotide sequences or peptide sequences, in which base pairs or amino acids are represented using single-letter codes. A sequence in FASTA format begins with a single-line description, followed by lines of sequence data. The description line is distinguished from the sequence data by a greater-than (" $>$ ") symbol in the first column.^[1]

In order to read properly this kind of formatted files I used a FASTA parser^[2], that also works with *gzip* compression. That's useful since DNA sequences are very huge files and so it is frequent to download that from genomic banks as compressed archives.

3.2. OUTPUT DATA

The results have been written out in *gzip* compressed file and for each substring found has been shown two different information:

1. Sequence of edit operations.
2. Substring of genomic sequence

4. ANALYSIS OF THE ALGORITHM

4.1. ASSUMPTION

As I said the project focuses on genomic application so it frequently happens that DNA sequences to scan are very huge files, therefore dynamic programming implementation of the algorithm will create a matrix that could occupy too much space in memory. So it's not feasible to instantiate a matrix with dimension $[m+1, n+1]$, with n that could be in the order of millions of characters and m in the order of thousands. Hence I decided to subdivide the whole DNA in subsequences, and compare the pattern with each of them. This will create sub matrices of lower dimension whose sizes will be bounded by a fixed value. The criterion for this operation is to divide the entire DNA length in two until the dimension of the sub matrix is greater than a fixed size of memory, expressed in Byte and set as a constant for the whole execution of the program. This solution led to a better control of the size occupied in memory being able to manage also very large DNA sequences.

This solution implies another problem: the submatrices created will represent the comparison between the pattern and all of the substrings in a certain region of the whole text, as if each region is a new DNA sequence that has no correlation with the rest of the text.

This problem is simply solved in this way: each new sub matrix will be initialized taking into account the values of the previous sub matrix, so each new sub matrix will be created as the prosecution of the previouses. In this way it's preserved each value of the original huge matrix of dimension $[m+1, n+1]$, and so it's still possible to search for the pattern in the entire DNA sequence.


	j	0	1	2	3		0	1	2	3	4
i			C	A	T		G	A	C	T	G
0		0	0	0	0		0	0	0	0	0
1	T	1	1	1	0		1	1	1	0	1
2	A	2	2	1	1		1	1	2	1	1
3	C	3	2	2	2		2	2	1	2	2
4	T	4	3	3	2		3	3	2	1	2
5	G	5	4	4	3		2	3	3	2	1

Figure 3. Subdivision of the original matrix in sub matrices. Notice that the first row is always initialized with zero and that each new text region starts from index 0 instead of 1.

So for each new region of the text that will be compared with the pattern, it's only necessary to preserve in memory two sub matrices: the one that represents the current comparison and the one of the previous search. The same thing has been done for the matrix that memorizes the type of edit operations allowing to reconstruct the edit path, and retrieve the related substring, also for the matching at the boundary between the two sub matrices.

4.2. ANALYSIS OF SERIAL IMPLEMENTATION

After these assumptions the implementation of the algorithm mainly consists in two parts:

1. Compute the dynamic programming matrix
2. Searching for the substrings along with their edit operations and saving the results.

Starting from the recursive definition for the problem stated at the beginning⁽¹⁾, the dynamic programming matrix could be computed both by rows or by columns. Since for each (i,j) pair of the matrix I needed to store two values, the type of edit operation and the minimum edit cost, each element of the matrix will be stored in a struct data type defined in this way.

```
typedef struct
{
    int cost;
    int parent;
} cell;
```

Where parent stands for the type of edit operation that involves the minimum cost to reach that cell.

The matrix needs to be initialized, the first row is always set to zero for the cost member of the struct instead the parent is always set to -1, this because it serves as the ending condition for the recursive function that reconstructs the edit path. After that the first column is initialized: the cost member is set as $\max(i,j)$ given j equal to zero, instead the parent member is always set to 2, that is the number that defines the delete operation. As said before the total computation has been subdivided in many sub matrices each one seen as the prosecution of the previous ones.

The column initialization just described as to be intended only for the first sub matrix, instead for the subsequents sub matrices the first column has to be initialized taking into account the cost and parent values of the last column from the previous sub matrix, see figure 3. So, each cell of the first column for the new sub matrix will be calculated as if they belonged to the previous sub matrix. Instead the first row it's always initialized in the same way, in both cases.

After that it's possible to start the computation of the matrix, here the code for the implementation.

```
int offset_text_index;
(mat_p == NULL) ? (offset_text_index = -1) : (offset_text_index = 0);
for (i = 1; i < nr; i++)
{
    for (j = 1; j < nc; j++)
    {
        int mat_index = nc*i + j;
        int match_cost = (s[i - 1] == t[j + offset_text_index]) ? 0 : 1;
        opt[MATCH] = mat[mat_index - nc - 1].cost + match_cost;
        opt[INSERT] = mat[mat_index - 1].cost + 1;
        opt[DELETE] = mat[mat_index - nc].cost + 1;
        mat[mat_index].cost = opt[MATCH];
        mat[mat_index].parent = MATCH;
        for (k = INSERT; k <= DELETE; k++){
            if (opt[k] < mat[mat_index].cost)
            {
                mat[mat_index].cost = opt[k];
                mat[mat_index].parent = k;
            }
        }
    }
}
```

(2). Dynamic programming implementation of edit distance

The algorithm above represents the dynamic programming implementation for the recursive problem definition, with the addition of storing also the type of operation and not only the minimum cost. The variable `offset_text_index` serves to compare the correct characters from pattern and the current region of text, depending on whether I'm calculating the first sub matrix or not.

The second part of the algorithm consists in scanning the last row of the sub matrix just calculated and memorizing all of the j -th indexes at which the cost value of the cell is lesser than, or equal, k . In my implementation this k parameter has been passed through the command line and it has to be intended as a percentage of the total pattern length.

Then starting from the set of j -th indices found it's now possible to scan the matrix in order to reconstruct the sequence of edit operations and also to retrieve the corresponding substring from the text.

```
if (mat[mat_index].parent == -1)
    return;
if (mat[mat_index].parent == MATCH)
{
    reconstruct_path(s, t, t_prev, i - 1, j - 1, edit_path, nr, nc, matp, ncol_prev, mat);
    match_out(s, t, t_prev, i, j, matp, edit_path);
    return;
}
if (mat[mat_index].parent == INSERT)
{
    reconstruct_path(s, t, t_prev, i, j - 1, edit_path, nr, nc, matp, ncol_prev, mat);
    insert_out(t, j, edit_path);
    return;
}
if (mat[mat_index].parent == DELETE)
{
    reconstruct_path(s, t, t_prev, i - 1, j, edit_path, nr, nc, matp, ncol_prev, mat);
    delete_out(s, i, edit_path);
    return;
}
```

(3). Recursive edit path function

The previous snippet of code is the recursive function that allows the reconstruction of the sequence of edit operations. It works starting from a selected cell and then evaluates its parent value, hence according to the results of the evaluation the function recalls itself changing the cell that will be checked. The recursion ends when the function arrives at the first row of the matrix where the parent value has been set to -1. The functions `match_out`, `delete_out`, `insert_out` only serve to create the string of edit operations that will be saved as output.

The following snippet of code shows how it's possible to handle the edit path reconstruction at the boundary between two subsequent sub matrices. Since the recursive function will evaluate parent values as if they belong to the whole original matrix, it could happen that at the boundary the j -th index becomes negative. In this case it's necessary to change reference for the matrix, pointing at the previous one, and accordingly change the indexes in order to continue with the edit path reconstruction.

```
int mat_index;
if (j < 0){
    mat = matp;
    mat_index = ncol_prev*i + (ncol_prev + j);
}else{
    mat_index = nc*i + j;
}
```

This recursive function is named `reconstruct_path` and it is part of a more generic function `substring_matched` that also provides a method to retrieve related substring.

```
for (int j = 0; j < n_submatch; j++){
{
    int index_last_char_sub = indexes[j] + offset;
    reconstruct_path(p, t, text_prev, last_row, index_last_char_sub,
        edit_path, nr, nc, matp, ncol_prev, mat);
    substring_from_text(substring, t, text_prev, matp,
        edit_path, index_last_char_sub - offset);

    int index_sub_mat;
    for (int c = 0; c < max_col_sub; c++){
        index_sub_mat = max_col_sub*(j*2) + c;
        //Row for edit path
        subtrings[index_sub_mat] = edit_path[c];
        //Row for DNA substring
        index_sub_mat = index_sub_mat + max_col_sub;
        subtrings[index_sub_mat] = substring[c];
    }

    memset(edit_path, 0, max_col_sub);
    memset(substring, 0, max_col_sub);
}
```

(4). Substrings matched function

Then the edit paths with the substrings will be stored in a matrix whose content will be then saved on disk in a compressed text file. All of these functions are contained in another function called `sequence_reader`, that could be seen as the main method in which the DNA sequence has been subdivided and then for each of this subdivision all the previous actions have been performed. For simplicity the snippet has not been shown since it's a very long part of code, and a snapshot of it won't add more information of what that has already said.

5. A-PRIORI STUDY OF AVAILABLE PARALLELISM

Looking at the whole algorithm I found that the two parts that could be parallelized are the two most important sections, i.e the computation of the matrix and the searching and saving of the edit path along with the substrings. The scan on the last row of the matrix, that returns the j -th indices along with the number of occurrences it's not so time consuming, so I preferred to not parallelize. The iteration on the whole DNA sequence, instead could not be parallelized at all since there is a loop-carried dependency because, given a subdivision of the text the related sub matrix will be dependent on the sub matrix calculated in the previous iteration.

Regarding the matrix, each time an evaluation about the minimum cost has to be performed, this operation requires elements of the matrix that reside or on the previous diagonal, or on the same row but previous column, or on the same column but previous row. Hence, looking at the code snippet of the implementation⁽²⁾, for each computation there is a loop-carried dependency both in the inner for loop and in the outer for loop, and this happens independently from the row-order or column-order instantiation of the matrix.

The solution that allows to remove at least the inner loop-carried dependency is to compute the matrix in diagonal order.



Figure 4. Edit distance matrix computed in diagonal order

From the image it's simple to notice that now each element on the diagonal could be calculated independently from the others, since each computation only needs to read values that have been calculated in the previous diagonals, without considering the values that will be computed on the same one.

The parallelization of the second part of the algorithm concerns dividing in equal parts the set of j -th indices among the threads, so that each of them will be responsible to find the corresponding sequence of edit operations along with the substrings and then save these results in different files.

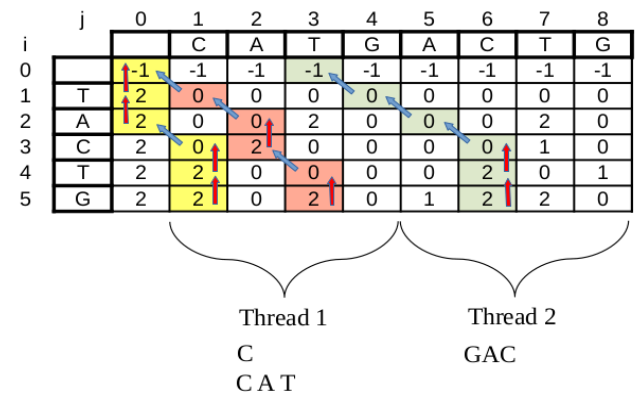


Figure 5. Example of how edit path and substrings search has been divided among two threads.

Before implementing the parallelization of the serial code the theoretical speedup that could be obtained with this parallelization has been studied using the **Amdahl's law**:

$$\frac{1}{(1-f) + f/n}$$

which states that the theoretical speedup is always limited by the part of the task that cannot benefit from the

improvement^[3]. In the formula f represent the ratio between the execution time of the code part that will be parallelized over the total time of execution for the task, always considering the serial implementation. Instead n represents the number of physical cores.

Taking measurements with different values of k , pattern and DNA length, it has obtained a percentage of 98% of parallelizable code in all of these cases. So, the theoretical speedups have been calculated varying the number of cores:

- 2 cores, speedup 1.96
- 4 cores, speedup 3.77
- 8 cores, speedup 7.017
- 16 core, speedup 12.03
- 24 cores, speedup 16.43

Higher percentage of parallelizable code has been obtained, since the two parts that will be parallelized consist almost the totality of the searching pattern problem faced.

6. PARALLEL IMPLEMENTATION

The parallel implementation for the dynamic programming table is shown below.

```
for (int c = 2; c < num_diagonals; c++) {
    length_curr_diag = MIN2(plen + 1, c + 1);
    first_col = MIN2(c - 1, nc - 1);
    first_row = MAX2(1, c - nc + 1);

    int n_iterations = length_curr_diag - first_row;

    if (first_col - n_iterations < 0) {
        n_iterations--;
    }

    int offset_text_index;
    (mat_p == NULL) ? (offset_text_index = -1) : (offset_text_index = 0);

    int num_max_threads = omp_get_max_threads();
    if (num_threads == num_max_threads) {
        num_threads = num_threads - 2;
    } else if (num_threads > num_max_threads) {
        num_threads = num_max_threads - 2;
    }

    #pragma omp parallel for num_threads(num_threads) \
    firstprivate(nc, first_col, first_row, n_iterations, offset_text_index) private(opt)
    for (int col = first_col; col > first_col - n_iterations; col--) {
        int row = first_row + (first_col - col);
        int mat_index = nc * row + col;
        int match_cost = (p[row-1] == t[col + offset_text_index]) ? 0 : 1;

        opt[MATCH] = mat[mat_index - nc - 1].cost + match_cost;
        opt[INSERT] = mat[mat_index - 1].cost + 1;
        opt[DELETE] = mat[mat_index - nc].cost + 1;
        mat[mat_index].cost = opt[MATCH];
        mat[mat_index].parent = MATCH;
        for (int k = INSERT; k <= DELETE; k++) {
            if (opt[k] < mat[mat_index].cost) {
                mat[mat_index].cost = opt[k];
                mat[mat_index].parent = k;
            }
        }
    }
}
```

(5) . Parallel implementation edit distance matrix

The initialization for the first row and column is the same as the serial implementation along with the evaluation for the minimum cost and edit type operation. The main change is to adjust, according to the current diagonal, the indices of the matrix in order to correctly compute each element.

The matrix (mat), the current subsequence of DNA (t) and the pattern (p) have been left as shared variables, since each thread computing an element on the diagonal will always work in a different position in the matrix, also when match operation has to be evaluated, the comparison between pattern and text characters always happen at different positions, so won't be conflict both in reading and writing

operations, see figure 4. The variables included in the firstprivate (first_col, first_row, nc, n_iterations) clause are needed to be initialized with their original values in order to correctly compute diagonals, and also private in order to avoid conflicts. offset_set_index also needed to be set with the original value and private for avoiding conflicts, despite other variables this one it's useful to correctly compare DNA subsequence and pattern characters.

The computation previously has to calculate the length of the current diagonal and then, starting from the first column and first row, each element has been computed. Going backwards the next column has set and the next row has been set taking into account how much we have been moved from the first column. Private *opt* it's only used as a local static array that memorizes the cost of edit operations.

The second part of the algorithm that has been parallelized is this :

```
#pragma omp parallel firstprivate(k_edit_dist, length_sequence, ncol_curr, ncol_prev, nrows) num_threads(num_threads)
{
    int *sub_indexes_k_thread;
    int num_ind_k_thread = 0;

    #pragma omp for schedule(static)
    for (int k = 0; k < num_submatch; k++) {
        if (num_ind_k_thread == 0) {
            num_ind_k_thread = num_ind_k_thread + 1;
            sub_indexes_k_thread = malloc(sizeof(int));
            sub_indexes_k_thread[num_ind_k_thread - 1] = submatch_indexes[k];
        } else {
            num_ind_k_thread = num_ind_k_thread + 1;
            sub_indexes_k_thread = realloc(sub_indexes_k_thread, num_ind_k_thread * sizeof(int));
            sub_indexes_k_thread[num_ind_k_thread - 1] = submatch_indexes[k];
        }
    }

    int max_lenght_substring_matched_t = strlen(pattern) + k_edit_dist + 1;
    int max_rows_thread = 2 * num_ind_k_thread;
    char *substrings_matched_thread;
    substrings_matched_thread = calloc(max_rows_thread * max_lenght_substring_matched_t, sizeof(char));

    substrings_matched(pattern, text_curr, text_prev,
        sub_indexes_k_thread, num_ind_k_thread, nrows,
        ncol_curr, Mcurr, max_rows_thread,
        max_lenght_substring_matched_t, Mprev, ncol_prev,
        substrings_matched_thread);

    save_substrings_matched(pattern, k_edit_dist, length_sequence, max_rows_thread,
        max_lenght_substring_matched_t, substrings_matched_thread, Mprev);
    free(substrings_matched_thread);
}
```

(6) . Parallel implementation of search and save phase

In this parallel region there is a parallelized cycle in which the set of j -th indices found has been divided among the thread in equal parts, so each of them will be charged with an equal amount of work, avoiding having thread slower than other.

So, each subset of indices, (*sub_indexes_k_thread) , is saved as a private variable for each thread, along with the length of this subset, (num_ind_k_thread). Also (*substrings_matched_thread) is a private variable that stores for each thread the set of edit paths along with the related substrings.

After that the function substrings_matched⁽⁴⁾ has been called. It will find the edit path and the related substrings, using this time the private set of indices previously set, storing these results in *substrings_matched_thread.

For the same reasons expressed before, the two matrices, Mcurr and Mprev, the pattern and the two parts of the whole text, text_curr and text_prev, have been left shared since the threads will work on different positions both in reconstructing edit path and both in retrieving substrings, see figure 5.

The variables in the first private clause are integer variables that have been copied and initialized with original values for each thread, in order to avoid conflicts. `ncol_curr`, `ncol_prev` and `nrows` are used for index correctly `Mcurr` and `Mprev`, instead `k_edit_dist` and `length_sequence` are used to access the correspondent directory in which the results will be saved. `k_edit_dist` is also used to set the maximum length that the substring could reach.

At the end each thread will save its results in a proper file in order to take advantage from the parallel writing.

7. TESTING AND DEBUGGING

In order to test the correctness of my intuitions different tests have been performed. The parameters that could influence the execution time are: the length of DNA sequence, the length of the pattern, the k -threshold, expressed in percentage over the pattern length and the size of the sub matrix.

In my tests I mainly focused on the variation of the k -threshold and pattern length. The k it's linked to the probability to find more or less substrings, because this parameter manages the flexibility of the search: an higher k allows to find substrings more different from the pattern, instead a lower k will conduct to substrings that are more similar to the pattern.

So, the threads involved in the parallelization of search and save phase⁽⁶⁾, will have to do with an heavier or lighter workload in relation the value of k . With a good proportion between workload and number of threads will lead to an improvement in performance. What about the pattern length, this measure will define the amount of diagonal elements, see figure 4, and so the workload for the related parallelized task⁽⁵⁾. Hence an appropriate proportion between pattern length and number of threads, will increase the performance.

The variations of these selected parameters have been tested both for serial and parallel implementations, in order to compare the performance. The execution times have been monitored using the `omp_get_wtime()` function properly affixed on the two parallelizable parts of the program. During each iteration on DNA sequence, both the execution time for computing the matrix and for the search and save phase have been displayed on console in order to monitor the program execution. Tests on my device have been performed on Ubuntu 20.04 LTS as operating system with an intel core i7-4720HQ with 4 core, 8 threads and 8 GB of RAM.

8. PERFORMANCE ANALYSIS

First cases have been performed using the DNA sequence of human chromosome 22 downloaded from the NCBI^[4], properly cleaned by the 'N' characters that could represent both problems in DNA sequencing or poor quality of DNA sample. So, taking fixed this parameter along with the maximum size of the sub matrix (512 MByte), different tests have been performed varying the pattern length and k threshold.

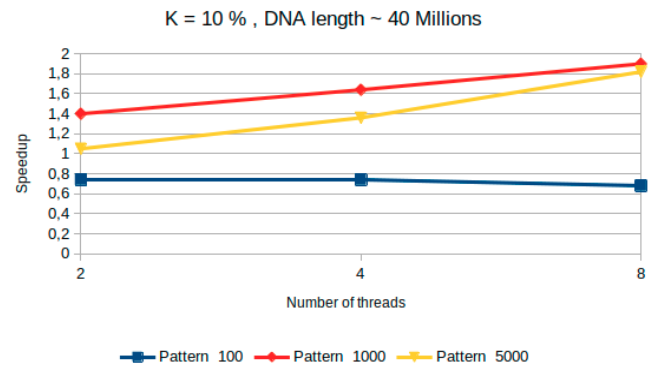


FIGURE 5. Searching with different pattern length on my PC with low k .

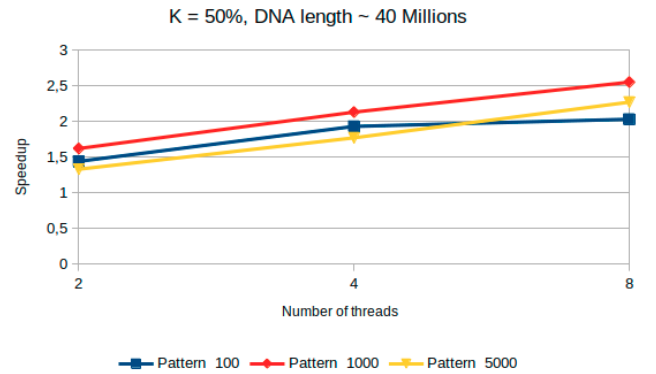


FIGURE 6. Searching with different pattern length on my PC with high k

In figure 5 has been shown the case with a low value of k , this implies that a few number of substrings will be found and so the execution time for search and save phase could be neglected. Therefore, the execution time for computation of the matrix will be the most important and, as explained in the test section, the length of the pattern will manage the threads workloads and so the performance. In the case of a pattern length of 100 the workload it's not well balanced, so the overhead introduced by the parallelization will lead to a decrease in performance. This effect will be more consistent as the number of threads increases. Instead with a longer pattern, the overhead it's no longer a problem and so an increase in performance could be appreciated.

In figure 6, instead represent the case for a higher value of k . This time also the execution time for the search and save phase will be relevant, along with the workload for the related parallelized part. In this case it's possible to notice an increase in performance for all the pattern length tested, also for the lowest one. Since the worsening of performance for the matrix computation has been counterbalance from an improvement in the search and save phase, leading to a better performance for the whole searching task. For higher pattern length with higher k , the performance increases further respect to a lower value of k .

Then the same tests were performed on a virtual machine instantiated on Google Cloud Platform with 24 cpus and 64 GB of ram, using Ubuntu 20.04 LTS as operating system.

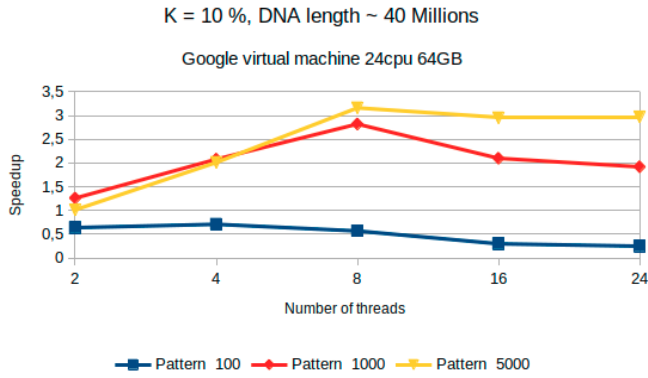


FIGURE 7. Searching with different pattern length on a virtual machine with low k.

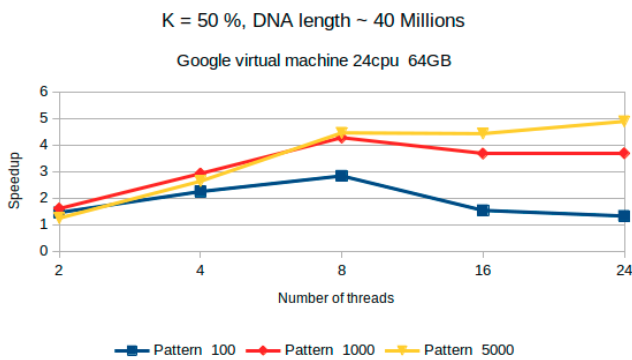


FIGURE 8. Searching with different pattern length on a virtual machine with high k.

In figure 7, it's possible to notice that for the lowest pattern length the performance trend is the same as the previous case tested on my device, for the same reasons. Instead this time the pattern length of 5000 seems to be the one that benefits most from the parallelization, despite the pattern length 1000 that on my device was the one with best performance, see both figure 5 and 6. In fact on google virtual machine higher number of threads will lead to a worsening in performance for length 1000, due to a bad workload balance.

In figure 8, a higher value of k has been chosen. The pattern length 100 could also benefit from the parallelization, and the performance for the other two lengths have been further increased with respect to the case shown in figure 7.

In the second case the change of DNA sequence has been increased and the related performance studied. This time the DNA sequence of the human 13 chromosome has been taken into account, always downloaded from NCBI^[4]. So different tests have been done varying the pattern length and k threshold, both on my personal device and both on Google cloud virtual machine, with the same properties of the previous tests. In this case has been investigated the effect of increasing the DNA sequence over the performance.

Observing all figures from 9 to 12, it's possible to notice that the speedup trends are very similar to the previous case,

in which a lower DNA sequence was taken into account. This happens because with my implementation increasing the length of DNA will lead to more iterations that have to be done, but inside each iteration the parameters that will affect the performance will be always the pattern length and the k-threshold. Therefore searching in a longer DNA sequence will only increase the total amount of time needed to execute the searching but performance won't be affected by this increase. Then all the considerations done for the previous cases with a lower DNA are still valid.

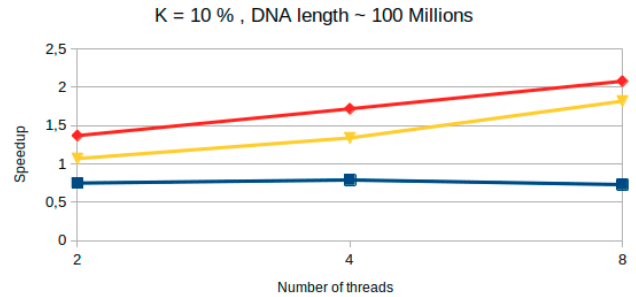


FIGURE 9. Searching with different pattern length on my PC with low k.

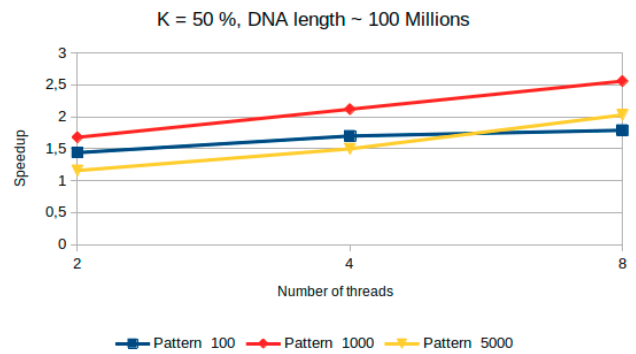


FIGURE 10. Searching with different pattern length on my PC with high k.

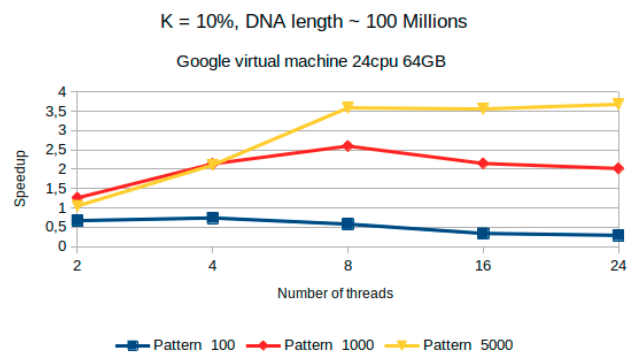


FIGURE 11. Searching with different pattern length on a virtual machine with low k.

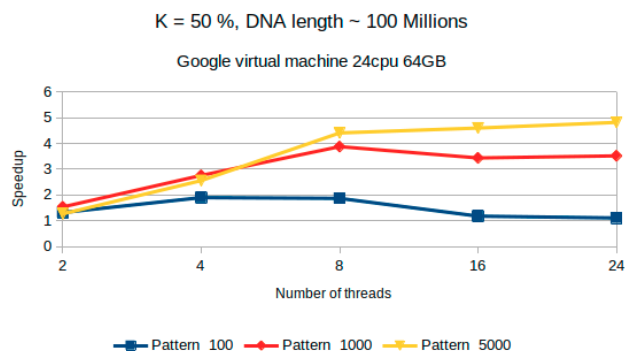


FIGURE 12. Searching with different pattern length on a virtual machine with low k.

All the pattern sequences used for the tests have been randomly generated using this web tool ^[5] and formatted using FASTA format^[1].

A note must be made, when the test has been performed using all the threads available for the current machine, 8 threads for my device and 24 for the virtual machine, a scaling for the parallelization of the matrix⁽⁵⁾ has to be done. Since the parallelized search and save phase made in previous iterations will affect the overhead for the parallelization of the matrix, because all the thread have been used for saving in different files. So using the system resource monitor has been found that even if the parallelized save has been done, still some threads are busy. This implies that, if all possible threads are requested for matrix parallelization, there will be an increase in overhead leading in worsening the performance. Therefore two options were possible for scaling: using the `omp_set_dynimac(true)` directive or decreasing the number of threads used for matrix parallelization by two threads. Hence, both these options have been tried analyzing the performance obtained, and in the end the manual scaling has been chosen.

CONCLUSION

Despite Amdahl's law suggesting that a high percentage of the code could be parallelized and so high values for speedup could be reached, the measurements done show that in practice lower improvements have been reached. This could be linked to the nature of the problem to parallelize along with the parallel implementation used for this project. For example, taking into account the edit distance matrix⁽⁵⁾ could be highlighted that also the parallel implementation has been bounded by the computation of previous diagonals preventing the parallelization of the whole matrix. Instead the parallel implementation suggested for the searching and saving of edit operations sequence⁽⁶⁾ and substrings, seems to be quite good since when the time for this task becomes considerable, the performance improves.

In the end, this project suggests parallel implementation that in most cases guarantees an improvement in performance respect to the serial implementation, keeping in mind to properly balance the workload among threads.

REFERENCES

- [1] [FASTA Format](#)
- [2] [FASTA/FASTQ Parser](#)
- [3] [Amdahl's law](#)
- [4] [NCBI Genome Bank](#)
- [5] [Random DNA sequences](#)
- [6] [Implementation of approximate string matching using edit distance](#)