People's Democratic Republic of Algeria
University of Science and Technology HOUARI BOUMEDIENE
Faculty of Computer Science

Parallelization of the Smith-Waterman Local Alignment Algorithm

Written by

- MOUHOUB Massinissa
- ZIGADI Sadji Amine

Contents

1 Presentation of the Smith-Waterman Algorithm **3**
- 1.1 Definition . . . 3
- 1.2 Main Steps of the Algorithm . . . 3
  - 1.2.1 Score Matrix Initialization . . . 3
  - 1.2.2 Matrix Filling . . . 4
  - 1.2.3 Alignment Search . . . 4
  - 1.2.4 Pseudo-code . . . 6
- 1.3 Walkthrough of an Example . . . 6
  - 1.3.1 The Initial Matrix . . . 6
  - 1.3.2 Matrix Filling . . . 6
  - 1.3.3 Interpretation of Results . . . 7
- 1.4 Time and Space Complexity of the Algorithm . . . 8
  - 1.4.1 Time Complexity . . . 8
  - 1.4.2 Space Complexity . . . 8

2 Parallelization of the Smith-Waterman Algorithm **9**
- 2.1 Analysis of the Sequential Script . . . 10
- 2.2 Parallel Script . . . 11

3 Performance Analysis **13**

Introduction

Bioinformatics is a key field that bridges biological sciences with computer science, enabling the resolution of complex problems in biological data processing. DNA sequence alignment is an essential task in this domain, and the Smith-Waterman algorithm is one of the most widely used methods for local sequence alignment. This algorithm is particularly important for identifying similarities between different genetic sequences and is based on dynamic programming to find the optimal solution.

This project focuses on the parallelization of the Smith-Waterman algorithm using parallel programming techniques in Python. The objective is to improve the algorithm's efficiency by reducing execution time, particularly for large biological databases. To achieve this, we will analyze the algorithm's time and space complexity, identify bottlenecks using profiling tools, and propose a parallelization model tailored for multicore architectures.

The report will detail the various implementation stages, including a presentation of the Smith-Waterman algorithm, its complexity analysis, the proposed parallelization model, and a performance evaluation comparing the sequential and parallel versions.

Chapter 1

Presentation of the Smith-Waterman Algorithm

The Smith-Waterman algorithm is a local alignment method used in bioinformatics to find similar regions between two DNA, RNA, or protein sequences. Unlike the Needleman-Wunsch algorithm, which performs a global alignment, Smith-Waterman focuses on identifying optimal subsequences while allowing gaps and substitutions.

1.1 Definition

The Smith-Waterman algorithm identifies optimal local alignment regions between two sequences by maximizing an alignment score based on a substitution matrix and penalties for gaps. It is particularly useful when only certain parts of the sequences exhibit significant similarity.

1.2 Main Steps of the Algorithm

The steps of the algorithm are summarized below. See 1.1 for better visualization.

1.2.1 Score Matrix Initialization

A matrix $\mathbf{H}$ of dimensions $(m+1) \times (n+1)$ is created, where m and n are the lengths of the two sequences respectively. The initial values of the first row and first column are all set to zero.

1.2.2 Matrix Filling

Each cell $H(i, j)$ is computed based on the following scores:

- $H(i-1, j-1) + score(x_i, y_j)$; knowing that the function $score(x_i, y_j)$ returns a value depending on whether x_i and y_j are similar or not (match or mismatch).
- $H(i-1, j) - gap\_penalty$; where $gap\_penalty$ is a defined value representing the cost associated with the introduction or extension of a gap in an alignment.
- $H(i, j-1) - gap\_penalty$.

$H(i, j)$ takes the maximum value among these options.

1.2.3 Alignment Search

The maximum score in the matrix indicates the end of the optimal local alignment region. Starting from the cell with the maximum score, we trace back through the matrix following the cells that contributed to the score until reaching a cell with a score of 0.

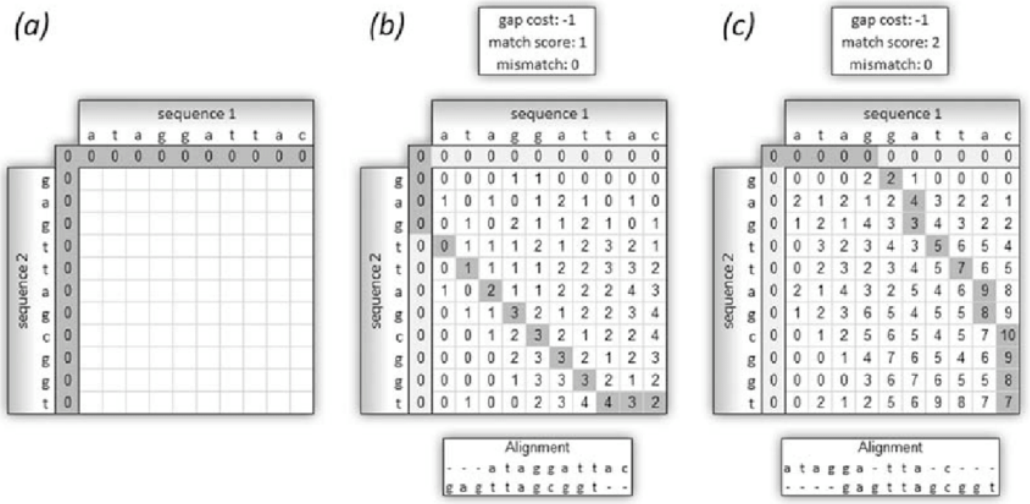

Figure 1.1: Steps of the Smith-Waterman algorithm execution.

Algorithm 1 Local Alignment with Smith-Waterman

Require: Two sequences `seq1` (length m) and `seq2` (length n)
Ensure: Local alignment and alignment score

Initialize a matrix H of size $(m+1) \times (n+1)$ with zeros.
Define the penalties: `match`, `mismatch`, and `gap`.
for $i \leftarrow 1$ to m **do**
    for $j \leftarrow 1$ to n **do**
        $match\_score \leftarrow H[i-1][j-1] + score(seq1[i], seq2[j])$
        $delete\_score \leftarrow H[i-1][j] - gap\_penalty$
        $insert\_score \leftarrow H[i][j-1] - gap\_penalty$
        $H[i][j] \leftarrow \max(0, match\_score, delete\_score, insert\_score)$
    end for
end for
Find the maximum value in H and its position $(i_{\max}, j_{\max})$.
Perform a *traceback* from $(i_{\max}, j_{\max})$ to reconstruct the alignment:
while $H[i][j] > 0$ **do**
    if $H[i][j] == H[i-1][j-1] + score(seq1[i], seq2[j])$ **then**
        Add $seq1[i]$ and $seq2[j]$ to the alignment.
        $i \leftarrow i-1$, $j \leftarrow j-1$
    else if $H[i][j] == H[i-1][j] - gap\_penalty$ **then**
        Add a gap to `seq2`.
        $i \leftarrow i-1$
    else
        Add a gap to `seq1`.
        $j \leftarrow j-1$
    end if
end while
return The alignment and the maximum score.

1.2.4 Pseudo-code

1.3 Walkthrough of an Example

Sequences

- Sequence 1: `GATTACA`.
- Sequence 2: `GCATGCU`.

Parameters

- Match: +2.
- Mismatch: -1.
- Gap: -2.

1.3.1 The Initial Matrix

We start with a matrix filled with zeros:

		G	C	A	T	G	C	U
	0	0	0	0	0	0	0	0
G	0							
A	0							
T	0							
T	0							
A	0							
C	0							
A	0							

Table 1.1: Step 1: Matrix Initialization

1.3.2 Matrix Filling

We fill the rest of the matrix using the method mentioned previously, see subsection 1.2.2. We obtain the following:

		G	C	A	T	G	C	U
	0	0	0	0	0	0	0	0
G	0	2	0	0	0	2	0	0
A	0	0	0	2	0	0	1	0
T	0	0	0	0	4	2	0	0
T	0	0	0	0	2	3	1	0
A	0	0	0	2	0	1	2	0
C	0	0	2	0	1	0	3	1
A	0	0	0	4	2	0	1	2

Table 1.2: Step 2: Matrix Filling

		G	C	A	T	G	C	U
	0	0	0	0	0	0	0	0
G	0	2	0	0	0	2	0	0
A	0	0	0	2	0	0	1	0
T	0	0	0	0	4	2	0	0
T	0	0	0	0	2	3	1	0
A	0	0	0	2	0	1	2	0
C	0	0	2	0	1	0	3	1
A	0	0	0	4	2	0	1	2

Figure 1.2: Result of the local alignments

1.3.3 Interpretation of Results

After filling, we find the region with the maximum score and perform a traceback to determine the optimal local alignment, tracing back until a 0 is found. Multiple local alignments can be found, as is the case in our example.

In the treated example, we have the following two alignments:

- Local alignment 1: `AT`.
- Local alignment 2: `CA`.

1.4 Time and Space Complexity of the Algorithm

1.4.1 Time Complexity

In order to find the time complexity of the Smith-Waterman algorithm, we will compute the complexity of all individual steps, then deduce the total complexity.

- **Matrix Initialization** Nested loop, the first has a complexity of n (n the size of the first sequence) and m (m the size of the second sequence), so we will have $O(n \times m)$.
- **Matrix Filling** Same as the matrix initialization, and we have the maximum value step in addition which will cost a linear complexity ($O(4) = O(1)$, 4 because we compare between 4 values; diagonal, horizontal, vertical and 0), so we will have $O(n \times m)$.
- **Interpretation of Results** We have a while loop that will execute at worst the minimum size between the two sequences, so we will have $O(min(n, m))$.

Therefore the time complexity is $O(n \times m)$.

1.4.2 Space Complexity

For the algorithm, we use a single matrix of size $n \times m$, so the space complexity is $O(n \times m)$.

Chapter 2

Parallelization of the Smith-Waterman Algorithm

First, we present the sequential *Python* script of the Smith-Waterman algorithm.

```
1  def smith_waterman(seq1, seq2, match_score, mismatch_score,
       gap_penalty):
2      rows = len(seq1) + 1
3      cols = len(seq2) + 1
4      H = [[0 for _ in range(cols)] for _ in range(rows)]
5
6      max_score = 0
7      max_positions = []
8
9      for i in range(1, rows):
10         for j in range(1, cols):
11             value_1 = H[i-1][j-1] + (match_score if seq1[i-1] ==
                   seq2[j-1] else mismatch_score)
12             value_2 = H[i-1][j] + gap_penalty
13             value_3 = H[i][j-1] + gap_penalty
14
15             H[i][j] = max(0, value_1, value_2, value_3)
16
17             if H[i][j] > max_score:
18                 max_score = H[i][j]
19                 max_positions = [(i, j)]
20             elif H[i][j] == max_score:
21                 max_positions.append((i, j))
22
23     print("Scoring Matrix:")
```

```
24      for row in H:
25          print(" ".join(map(str, row)))
26
27      def traceback(i, j, align1, align2, alignments):
28          if i == 0 or j == 0 or H[i][j] == 0:
29              alignments.append((align1[::-1], align2[::-1]))
30              return
31
32          current_score = H[i][j]
33          diagonal = H[i-1][j-1]
34          up = H[i-1][j]
35          left = H[i][j-1]
36
37          if current_score == diagonal + (match_score if seq1[i-1] ==
                seq2[j-1] else mismatch_score):
38              traceback(i-1, j-1, seq1[i-1] + align1, seq2[j-1] +
                    align2, alignments)
39
40          if current_score == up + gap_penalty:
41              traceback(i-1, j, seq1[i-1] + align1, "-" + align2,
                    alignments)
42
43          if current_score == left + gap_penalty:
44              traceback(i, j-1, "-" + align1, seq2[j-1] + align2,
                    alignments)
45
46      all_alignments = []
47      for pos in max_positions:
48          traceback(pos[0], pos[1], "", "", all_alignments)
49
50      unique_alignments = list(set(all_alignments))
51
52      return unique_alignments
```

Listing 2.1: Script de Smith-Waterman séquentiel

2.1 Analysis of the Sequential Script

In order to identify which tasks to parallelize in the sequential code, we ran several tests using the *cProfile* module.

With two sequences of size 108, we obtain the results shown in fig 2.1.

From fig 2.1, we notice that the step that takes the most time in the

```
Profiling Results:
         22381521 function calls (3157581 primitive calls) in 13.992 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.014    0.014   13.992   13.992 profiling.py:13(main)
        1    1.495    1.495   13.978   13.978 main.py:26(smith_waterman)
19223941/1   12.399    0.000   12.476   12.476 main.py:31(traceback)
  3145799    0.078    0.000    0.078    0.000 {method 'append' of 'list' objects}
        1    0.005    0.005    0.006    0.006 main.py:1(initialize_and_fill_matrix)
    11664    0.001    0.000    0.001    0.000 {built-in method builtins.max}
      109    0.000    0.000    0.000    0.000 main.py:4(<listcomp>)
        1    0.000    0.000    0.000    0.000 {built-in method builtins.print}
        3    0.000    0.000    0.000    0.000 {built-in method builtins.len}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Figure 2.1: Profiling result of two sequences of 108 characters.

algorithm is the `trackback` function, which finds all local alignments between the two sequences. Therefore we will create a parallel script that parallelizes this functionality.

2.2 Parallel Script

Here is the parallel version:

```
1     def traceback_worker(args):
2         i, j, seq1, seq2, H, match_score, mismatch_score,
              gap_penalty = args
3         alignments = []
4
5         def traceback(i, j, align1, align2):
6             if i == 0 or j == 0 or H[i][j] == 0:
7                 alignments.append((align1[::-1], align2[::-1]))
8                 return
9
10            current_score = H[i][j]
11            diagonal = H[i-1][j-1]
12            up = H[i-1][j]
13            left = H[i][j-1]
14
15            if current_score == diagonal + (match_score if seq1[i-1]
                  == seq2[j-1] else mismatch_score):
16                traceback(i-1, j-1, seq1[i-1] + align1, seq2[j-1] +
                      align2)
17
18            if current_score == up + gap_penalty:
```

```
19                traceback(i-1, j, seq1[i-1] + align1, "-" + align2)
20
21            if current_score == left + gap_penalty:
22                traceback(i, j-1, "-" + align1, seq2[j-1] + align2)
23
24        traceback(i, j, "", "")
25        return alignments
26
27
28    def smith_waterman(seq1, seq2, match_score, mismatch_score,
          gap_penalty):
29        H, max_score, max_positions =
              initialize_and_fill_matrix(seq1, seq2, match_score,
              mismatch_score, gap_penalty)
30
31        args = [(pos[0], pos[1], seq1, seq2, H, match_score,
              mismatch_score, gap_penalty) for pos in max_positions]
32
33        with multiprocessing.Pool(processes=2) as pool:
34            results = pool.map(traceback_worker, args)
35
36        all_alignments = [alignment for result in results for
              alignment in result]
37        unique_alignments = list(set(all_alignments))
38
39        return unique_alignments, max_score
```

Here the `traceback_worker` function is the main function that allows the program to be parallelized. Each process will execute this function in parallel and will generate the alignment.

Chapter 3

Performance Analysis

After several tests, we found the following results: We find that the larger

	seq	par			
		2	4	6	8
10	0.000384	0.061976	0.057896	0.073779	0.07761
20	0.000354	0.054117	0.071049	0.077192	0.099325
50	0.001179	0.05806	0.072039	0.078288	0.093207
100	0.006348	0.065681	0.074773	0.063099	0.09565
150	0.017743	0.065865	0.087802	0.089017	0.100491

Table 3.1: Results across several sequence sizes.

the sequence size becomes, the more interesting parallelism becomes. We compute the speedup and efficiency for the sequence size of 150 characters: We notice that parallelization is not very interesting in the case of Smith-

	Time	Speedup	Efficiency
Seq	0.017743	/	/
2	0.065865	0.27	0.135
4	0.087802	0.2	0.05
6	0.089017	0.19	0.03
8	0.100491	0.17	0.02

Table 3.2: Speedup and efficiency

Waterman.

Conclusion

The parallelization of the Smith-Waterman algorithm becomes particularly interesting when it comes to processing multiple local alignments simultaneously, which is common in bioinformatics applications. By using a multi-core architecture, we managed to deduce that the execution time of the algorithm becomes more favorable in the case where there are multiple local alignments, by exploiting the possibilities offered by parallel programming. This improvement allows for faster processing of large amounts of biological data, making the algorithm more suited to the requirements of modern genetic databases. In summary, parallelization brings significant performance gains, particularly in scenarios with multiple local alignments to perform.