Smith Waterman with Backtracking

Presented by
Prerak Srivastava
Abhijnan Vegi

International Institute of Information and Technology, Hyderabad

prerak.s@research.iiit.ac.in
abhijnan.vegi@students.iiit.ac.in

May 7, 2022

Outline

Problem Statement

- The Smith-Waterman algorithm is an algorithm that performs **local sequence alignment**, which means that the it finds the region with **highest similarity** between two sequences. These sequences can be nucleic acid sequences or protein sequences.
- The local alignment problem states that given two strings S_1 and S_2 , we are required to find substrings α from S_1 and β from S_2 whose optimal global alignment value is maximum over all pairs of substrings from S_1 and S_2.

Prerequisites

- **Substitution Matrix:** A substitution matrix is a collection of scores for aligning characters of a sequence with one another. These scores generally represent the level of similarity between the characters and the context depends on the sequences that are being compared.
- **Gap Penalty:** When aligning sequences, sometimes introducing gaps in the sequences can allow an alignment algorithm to match more terms than a gap-less alignment. However, it should still be a priority for the algorithm to minimize the number of gaps in an alignment as too many gaps can cause an alignment to be meaningless.

Nomenclature

- Let the two sequences be α and β of sizes m and n respectively. The DP matrix that will be used will be of size $(m+1) * (n+1)$.
- Let $F(i, j)$ represent the value of the cell in the i^{th} row and j^{th} column in the DP matrix.
- Let $s(i, j)$ represent the value of the cell in the i^{th} row and j^{th} column in the substitution matrix, which represents the similarity score of the i^{th} character from α and the j^{th} character from β.
- The scoring system and the gap penalty for our implementation is as follows:
  - **Scoring System**: +3 if match, -3 if mismatch
  - **Gap Penalty:** 2

Algorithm

- The filling up of the DP matrix is done by using the following equation

$$F(i,j) = \max \begin{cases} 0 \\ F(i-1,j-1) + s(i,j) \\ F(i-1,j) - d \\ F(i,j-1) - d \end{cases}$$

- In the following function,
  - 0 represents that there is no similarity between α_i and β_j
  - $F(i-1,j-1)$ represents the similarity score of aligning α_i and β_j
  - $F(i-1,j) - d$ represents that α_i is aligned with a gap
  - $F(i,j-1) - d$ represents that β_i is aligned with a gap
- For ever cell of the DP matrix, we add a pointer which points to the cell which was the source for this cell. This pointer is later used for backtracking.
- **Time Complexity:** $O(mn)$
- **Space Complexity:** $O(mn)$

Compute Configuration

- Prerak's PC:
  - RAM: 16GB
  - CPU: intel i5-9300
  - Number of available cores: 8
- Abhijnan's PC:
  - RAM: 32GB
  - CPU: Ryzen 7 4800-H
  - Number of available cores: 16

Performance Evaluation Framework

- The performance evaluation is done by calculating the **Execution Time** of the program
- The function *tick* and *tock* in helpher.h are used to calculate the execution time of any function in milliseconds.

Datasets

- Small

Name	Size of sequence 1	Size of sequence 2
$test_1$	12KB	12KB
$test_2$	60KB	60KB
$test_3$	164KB	172KB
$test_4$	532KB	540KB

- Medium

Name	Size of sequence 1	Size of sequence 2
$test_1$	3MB	3.2MB
$test_2$	22MB	23MB

Approaches

- Baseline
- Linear
- Diagonal Computation
- Parallelized Diagonal Computation

Baseline Method

- Method:
  - The DP matrix is filled in a row major fashion and stored (each row is calculated completely before moving on to the next row)
  - *maxScore* keeps a track of the position of the highest score in the matrix and is checked in every cell's calculation.
  - The traceback starts from the cell with the highest score and traces back until a 0 is encountered.

Baseline Benchmark

- No Flags

Name	Size of sequence 1
$test_1$	2084.279
$test_2$	Segfault
$test_3$	Segfault
$test_4$	Segfault

- Flags: `-Wall -g -O2 -std=c99 -march=native -mtune=native`

Name	Execution Time
$test_1$	1312.946
$test_2$	Segfault
$test_3$	Segfault
$test_4$	Segfault

Shortcomings of Baseline and Scope for Optimization

- **No parallelization**: Not using multiple cores of your CPU or any SIMD instructions to optimize the execution of the code.
- **Tracebacking Problem**: Consider the case where the sequences are of size 1MB each (10^6 bytes), this implies that the DP matrix we need to calculate the traceback would be of size 1TB, which is an absurd amount of data that is required to be stored. Since each character from the final alignment is dependent on the current row and the previous row, the scope for optimizing the backtracking becomes extremely small as we are required to store all the rows of the matrix in order to not lose any characters from the final alignment.

Linear Method

- Method:
  - The DP matrix is filled in a row major fashion and not stored (each row is calculated completely before moving on to the next row). Each row is stored in the *top* and *curr* variable, which is replaced while computing the next row.
  - *maxScore* keeps a track of the position of the highest score in the matrix and is checked in every cell's calculation.

Linear Benchmark

- No Flags

Name	Size of sequence 1
$test_1$	892.903
$test_2$	28893.43
$test_3$	239601.98
$test_4$	$\geq$ 300000

- Flags: `-Wall -g -O2 -std=c99 -march=native -mtune=native`

Name	Execution Time
$test_1$	251.541
$test_2$	7497.219
$test_3$	63481.341
$test_4$	$\geq$ 100000

- **Speedup:** 8x

Diagonal Computation Method

- Method:
  - Since the DP function requires the value of every (i, j) cell to be dependent on the $(i-1, j), (i, j-1), (i-1, j-1)$ cells, we can see that there is scope for computation along the **diagonal** as shown below:

0	0	0	0	0
0	1	3	6	10
0	2	5	9	13
0	4	8	12	15
0	7	11	14	16

  Here, the number in each cell (other than the 0s) represent the order in which their value will be computed. The colors show the diagonals along which the computation is done.
  - *maxScore* keeps a track of the position of the highest score in the matrix and is checked in every cell's calculation.

Diagonal Computation Benchmark

- No Flags

Name	Size of sequence 1
$test_1$	932.299
$test_2$	29521.036
$test_3$	245205.252
$test_4$	$\geq$ 300000

- Flags: `-Wall -g -O2 -std=c99 -march=native -mtune=native`

Name	Execution Time
$test_1$	142.881
$test_2$	4623.872
$test_3$	39466.589
$test_4$	$\geq$ 100000

- **Speedup:** 14.67x

Parallelized Diagonal Computation Method

- Method:
  - The method is the same as the diagonal computation approach, but every cell in a diagonal is computed parallelly.
  - A thread is spawned for each cell in the diagonal which performs the computation
  - In the initial diagonals, the work done by each threads is less. As the length of the diagonals get bigger and bigger, the amount of work by each thread increases.

Parallelized Diagonal Computation Benchmark

Flags:

```
-Wall -g -O2 -std=c99 -march=native -mtune=native -fopenmp
```

Name	Execution Time
$test_1$	185.368
$test_2$	3210.834
$test_3$	27087.926
$test_4$	$\geq$ 100000

Speedup: 21.37x

Performance Comparison