C++ Programming

Final Project
Implementing a Variant of the Smith-Waterman Algorithm
Secure Software Engineering Group
Philipp Schubert
Version 1.4

July 19, 2021

Solutions to this sheet are due on 30.08.2021 at 16:00. Please hand-in a digital version of your answers via PANDA at `https://panda.uni-paderborn.de/course/view.php?id=22691`. **Note:** If you copy text or code elements from other sources, clearly mark those elements and state the source. Copying solutions from other students is prohibited.

Solving this project is mandatory in order to pass the course and obtain the certificate. Copying solutions from other students is prohibited. **Caution:** If you are working on Linux/Unix/Mac you may need to use the additional compiler flag `-pthread` at the end of your compile command to specify the POSIX thread model to be used. The idea for this project originates from a similar project developed by Prof. Dr. Ralf Möller, Technische Informatik, Bielefeld University.

Introduction

The Smith-Waterman algorithm (SMA) is a well-known algorithm from Bioinformatics that is used to evaluate the similarities of (text) sequences. In this project, you will develop a small command-line application that implements a variant of this algorithm. Threads shall be used to make use of modern multi-core architectures. In this work, you will learn how C++ can be used to implement a more advanced algorithm efficiently. The resulting program's runtime may vary between several seconds up to one hour depending on the quality of your implementation.

Requirements of the Program

Your application has to meet the requirements as described in the remainder of this document. Furthermore, your program must be parameterizable using the following parameters: `<program>` `<input file` F_{in_1}`>` `<input file` F_{in_2}`>` `<number of threads` $\#T$`>` `<output file` F_{out}`>`. You are allowed to use further *optional* arguments. Your program should print an error/help message if it is run with no parameters or an incorrect parametrization.

Inputting Genome Sequences

As mentioned in the introduction, SMA is used to assess the similarities of character sequences. The origin of these character sequences does not matter to the algorithm. However, in the context of Bioinformatics, DNA-(desoxyribose nucleic acid)sequences or AA-(amino acid)sequences are usually compared to each other. This is why we restrict ourselves to analyze DNA-sequences in this project. To process and compare two sequences, they have to be loaded from disk into program variables to main memory first. You have to provide some functionalities to read the DNA sequences SOX3 and SRY. The DNA-sequences are stored in a file format called `.fas` (`fasta`). `fasta` files are plain text files that contain one or more sequences each of which is associated with an additional header line that is introduced by the '>' character. Here is an example:

```
> This is the header line
This is a DNA sequence ATGTCGACAAT...
Note that line breaks are allowed
as well as lower AND upper case letters.
```

DNA sequences may contain the characters 'A'/'a' (adenine), 'C'/'c' (cytosine), 'G'/'g' (guanine), 'T'/'t' (thymine) representing the four DNA bases that usually make up our genome. Sequence files can be relatively large, which is why you should read those files as single blocks. In order to do that, determine the size of the file by creating a variable of type **std::ifstream**. Use the member functions **seekg()** and **tellg()**. Once you have determined the file size, do not forget to reposition the input position indicator back to the beginning of the file. Next, you can allocate a buffer (use a suitable type here, **std::string**, for instance, and try to avoid using a variable of a low-level type such as **char***) that is large enough to hold the entire file contents. Once your buffer is set up correctly, you can read a file's contents directly to the buffer without the need for potentially expensive reallocation(s). Use the member function **read()** of your **std::ifstream** variable that expects a pointer to a buffer and the number of bytes it should read. If you decide on using **std::string**, you can get a pointer to **std::string**'s underlying buffer using its member function **data()**, which you can then pass to the **read()** function. By doing so, you achieve fast IO while being able to use the comfort of a high-level typed variable. After having read the files, remove the `fasta`-header line(s) as well as all '\n' (new line) and other whitespace characters and convert all lower case letters to upper case letters.

Comparison of the Sequences

The Smith-Waterman algorithm computes a score between two sequences m and n with length $|m|$ and $|n|$, accordingly. The score describes the similarity of these two sequences (higher is better) and can be computed with help of a matrix H with dimensions $(|m|+1)\times(|n|+1)$. The elements of the first row and first column of H are initialized as follows:

$$H(i,0) = 0,\ 0 \le i \le m \tag{1}$$
$$H(0,j) = 0,\ 0 \le j \le n \tag{2}$$

The remaining entries of the matrix are computed iteratively using the following three weights:

- ω_{match}
- $\omega_{mismatch}$
- ω_{gap}

The weight for a comparison between two characters a and b is defined as

$$\omega(a,b) = \begin{cases} \omega_{match}, & a = b \\ \omega_{mismatch}, & a \neq b \end{cases} \tag{3}$$

The elements of H with indices $1 \le i \le m$ and $1 \le j \le n$ are computed according to the following function:

$$H(i,j) = \max \begin{cases} 0 & \\ H(i-1,j-1) + \omega(a_i,b_j) & \text{match/mismatch} \\ H(i-1,j) + \omega_{gap} & \text{deletion} \\ H(i,j-1) + \omega_{gap} & \text{insertion} \end{cases} \tag{4}$$

After having computed the complete matrix H, the Smith-Waterman score (SMS) is the maximum of all of H's entries. Implement a function that computes the SMS for two strings. Check the correctness of your implementation by using the following example: $m = ACACACTA$ and $n = AGCACACA$ and use the weights $\omega_{match} = 2$, $\omega_{mismatch} = \omega_{gap} = -1$. Use these weights throughout the whole project. This example should produce the matrix shown in Table 1.

Table 1: Example matrix

	-	A	C	A	C	A	C	T	A
-	0	0	0	0	0	0	0	0	0
A	0	2	1	2	1	2	1	0	2
G	0	1	1	1	1	1	1	0	1
C	0	0	3	2	3	2	3	2	1
A	0	2	2	5	4	5	4	3	4
C	0	1	4	4	7	6	7	6	5
A	0	2	3	6	6	9	8	7	8
C	0	1	4	5	8	8	11	10	9
A	0	2	3	6	7	10	10	10	**12**

In this example, the SMS is 12 since 12 is the largest value stored in H. You can also check the correctness of your implementation using the web application provided at `http://rna.informatik.uni-freiburg.de/Teaching/index.jsp?toolName=Smith-Waterman`.

Do you really need to build the entire matrix to compute the SMS for two sequences? Adjust your implementation such that is uses only two rows for the computation of the score (you have to keep track of the largest value computed so far; when having computed the very last row, that will be your SMS).

Parallelization of the Algorithm

The similarity of the gene sequences SOX3 and SRY should now be calculated in parallel using your SMS implementation. Therefore, each segment of 50 consecutive characters in the first sequence s_{m_i} must be compared to each segment of 50 consecutive characters of the second sequence s_{n_j} ($|s_{m_i}| = |s_{n_j}| = 50$). If an SMS comparison of two 50 character segments leads to a score of at least 70 (you can try a few different thresholds), the start positions of the two segments i and j in their respective sequence as well as the score should be stored in a **std::vector**. It makes sense to create a small data structure **triple** (or use **std::tuple**) that is capable of storing these three values and to create a global variable of type **std::vector<triple>** to store the comparisons that have a score larger than the threshold; these are your raw results.

The parallelization using multiple threads can be done by splitting one of the sequences (for instance the first one m) in $\#T$ different regions r_{m_i}. Each 50 character segment within the smaller regions r_{m_i} must be compared to each 50 character segment in the second sequence. Do not forget any comparisons at the borders of the regions. Also, do not perform computations twice. You may wish to make use of an STL container's member function **at()** when accessing the character sequences which performs range checking and throws an exception whenever you accidentally try to perform an access out of bounds. Once your are confident that your implementation is correct, you can then switch over to **operator[]** to potentially increase performance.

Design your program in such a way that it receives the number of threads $\#T$ to be used as a command-line parameter. To implement the design discussed in the above, it makes sense to design a callable **class** (that is a class implementing **operator()**—the call operator) that receives all the data it needs to know in order to do the processing at its construction. Implement the call operator to start the desired computation(s). Create instances of this class in a suitable way and hand them over to **std::thread** variables that start the actual computations described in your callable class by calling the call operator.

If a thread detects that the result of a segment-comparison produces an SMS ≥ 70, save the result in the global variable of type **std::vector**<**triple**>. (Hint: use the member function **push_back()** of **std::vector** that allows you to add an entry to the end of the **std::vector**-typed variable. If required, **push_back()** automatically adjusts the size of the **std::vector** variable). Avoid race conditions by using a **std::mutex** lock, for instance!

Since all of the threads are *only* reading from the two gene sequences, it makes sense to store them in two global variables (synchronization is not required in this case). All threads can share the gene sequences and thus unnecessary copying of subsequences can be avoided which would otherwise highly degrade performance. (You may have to adjust the parameter list of your function that implements the SMS computation to match such a design.) Do not forget to join your threads once you have created them.

Post Processing and Outputting the Results

After having computed and stored the raw results in a **std::vector** variable, one post processing step must be performed. For each start position in a sequence find the corresponding start position in the other sequence with the largest score. Store those findings—the post-processed findings—in a separate **std::vector** variable. For instance, the following (fictional) results:

start in SOX3	start in SRY	score
100	2010	81
100	2011	83
100	2012	86
142	541	99
142	542	81
142	543	94

will be post-processed to:

start in SOX3	start in SRY	score
100	2012	86
142	541	99

Finally, write each entry of the post-processed results to a file in `.csv` (comma-separated values) format as shown below:

```
start in SOX3,start in SRY,score
100,2012,86
142,541,99
```

You can use the Python script `plotSMSresults.py` to plot the results in a scatter plot. Your plots should look similar to the ones shown in Figure 1 and Figure 2.

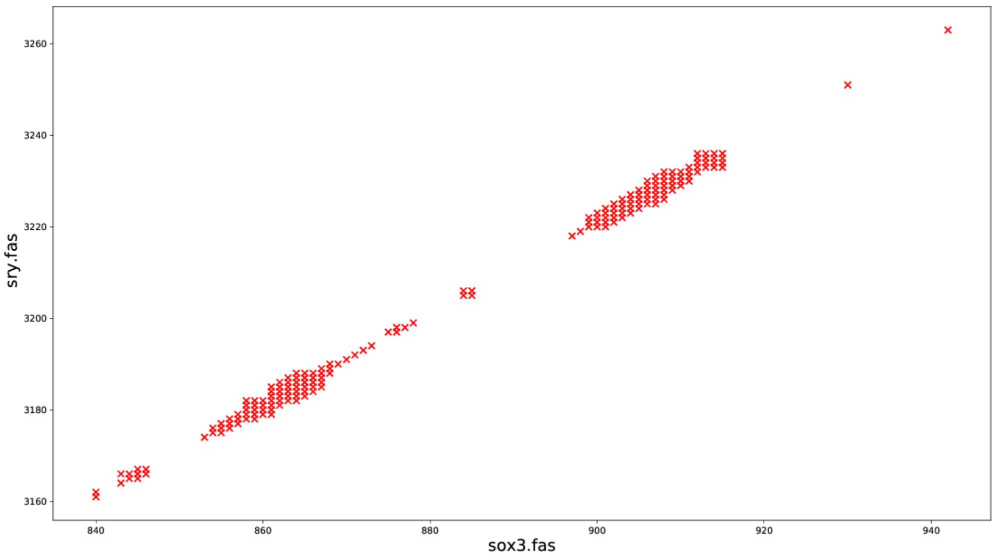

Figure 1: Raw results of the comparison between SOX3 and SRY.

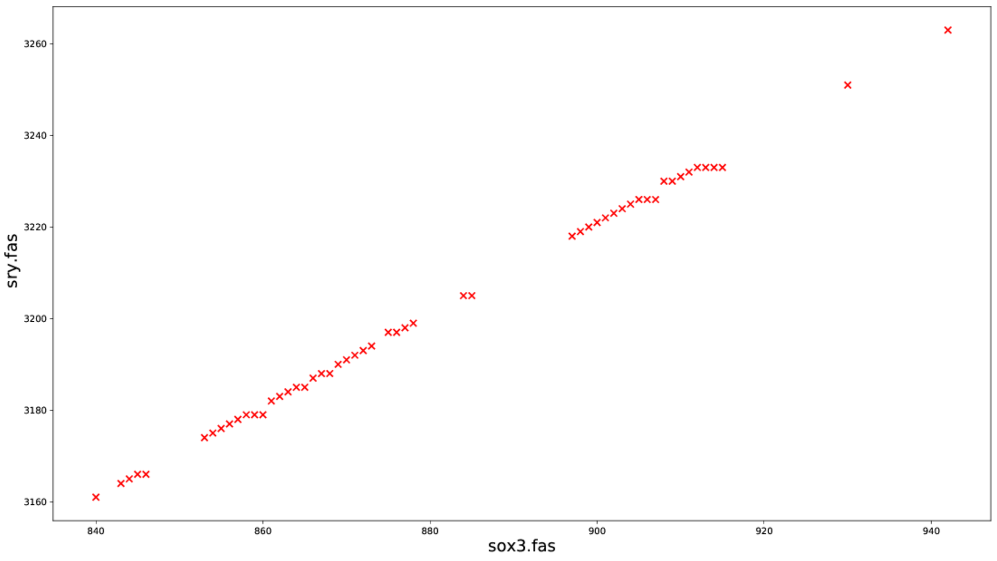

Figure 2: Post-processed results of the comparison between SOX3 and SRY.

Help

If you need help, do not understand the exercise, or get stuck while trying to solve the exercise, please feel free to send me an email. If necessary, we can also schedule a (remote) meeting to discuss problems.