Assignment 1
Parallel Computation of Genome Sequence Similarity

CS3210 – 2020/21 Semester 1

29 Aug 2020

Learning Outcomes

This assignment provides you the opportunity to apply your understanding of parallel programming with shared memory-paradigms (processes, POSIX threads and OpenMP) to solve real-world problems, especially relevant in the context of the current COVID-19 pandemic.

1 Problem Scenario

In computational biology, comparing the similarity of two (DNA/RNA) genomes possesses multiple applications such as sequence alignment, identification of mutation-prone genome regions, and re-construction of evolutionary patterns of virus strains. A common measure of the similarity of two genomes used in practice is that of the **Longest Common Subsequence (LCS)**.

In this assignment, you are required to parallelize the computation of the LCS of two genomes with (i) processes, (ii) POSIX threads (`pthreads`) and (iii) OpenMP. For simplicity, we will only consider DNA sequences.

This assignment is divided into two parts:

- Part 1: Process and pthread implementations, **due on Mon, 14 Sep, 11am**
- Part 2: OpenMP implementation and analysis, **due on Mon, 28 Sep, 11am**

1.1 Definitions

A DNA **sequence** is a string of symbols from the set of characters $\{A, T, G, C\}$, representing in order the four consitutent nucleotides: adenine (A), thymine (T), guanine (G) and cytosine (C).

In general for strings, a **subsequence** of a string S, is a sequence of characters that appear from left-to-right order in S, but not necessarily consecutively. In other words, a subsequence is obtained from S by removing zero or more characters. Suppose we have a DNA sequence $AGCTACT$, then G, CAT, $ACTAT$ and $AGCTACT$ are all subsequences. However, CCA is not a subsequence, since there is no T following the last C in the sequence above.

We now define the **Longest Common Subsequence (LCS)** in the context of DNA sequences. Given two DNA sequences X, Y, the longest common subsequence $LCS(X, Y)$ is the longest DNA sequence Z that is a subsequence of **both** X and Y. This sequence Z may not be unique, there may be multiple such subsequences $Z_1, Z_2, ...$ of equal (maximal) length.

Since we are interested in the degree of similarity between two DNA sequences, we are only concerned with the **length of this LCS**, and not the subsequence itself.

1.2 Sequential Algorithm

(You may skip this section if you have encountered the LCS problem in your advanced algorithm classes.)

One approach to computing the LCS of two DNA strings X and Y (here we assume X is shorter than Y) involves enumerating over all possible subsequences of X, and then checking if they are subsequences of Y. However, this is a bad approach in general, as there are $2^{|X|}$ such subsequences ($|X|$ is the length of X). Given the typical length of DNA strings, this is clearly unfeasible!

Instead, we can apply a technique known as **dynamic programming** to compute the LCS, due to the recursive nature of the problem. This technique allows us to reduce the sequential execution time of the algorithm by eliminating repeated computation of smaller instances of the problem, i.e. those that occur with prefixes of the DNA sequences.

Wikipedia contains a good explanation of the standard dynamic programming solution for computing the LCS. You can read more about it here.

This approach relies on expressing the length of the LCS recursively. For two sequences $X = x_1x_2...x_{|X|}$ and $Y = y_1y_2...y_{|Y|}$, we construct a score matrix LCS of size $(|X| + 1) \times (|Y| + 1)$, in which $LCS[i, j]$ (with $1 \leq i \leq |X|, 1 \leq j \leq |Y|$) records the length of the longest common subsequence for substrings $x_1x_2...x_i$ and $y_1y_2...y_j$. We fill the entries by the following recurrence formula:

$$LCS[i,j] = \begin{cases} 0, & i \text{ or } j \text{ is } 0 \\ LCS[i-1, j-1] + 1, & x_i = y_j \\ \max(LCS[i-1, j], LCS[i, j-1]), & \text{otherwise} \end{cases} \tag{1}$$

For example, consider two DNA strings $X = ATCGAG$ and $Y = ACGTAC$. The matrix LCS that we construct is:

$$LCS[i,j] = \begin{bmatrix} & 0 & A & T & C & G & A & G \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ A & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ C & 0 & 1 & 1 & 2 & 2 & 2 & 2 \\ G & 0 & 1 & 1 & 2 & 3 & 3 & 3 \\ T & 0 & 1 & 2 & 2 & 3 & 3 & 3 \\ A & 0 & 1 & 2 & 2 & 3 & 4 & 4 \\ C & 0 & 1 & 2 & 3 & 3 & 4 & 4 \end{bmatrix} \tag{2}$$

Observe the dependencies among rows, columns and diagonals within this matrix to find methods to parallelize this computation. Our provided implementation, `LCS-seq.c`, uses the observation that the matrix can be reduced to a $2 \times \min(|X| + 1, |Y| + 1)$ matrix as the dynamic programming approach only needs the current and previous row of the matrix.

1.3 Inputs and Outputs

All your programs should accept two command-line arguments, each a path to an input file containing one DNA sequence. Your program should compute the length of the LCS of the two supplied DNA sequences (found in the two files), and output a single integer denoting the length of the LCS.

Sample Program Execution

```
$ ./LCS-seq DNA-35000-1.in DNA-35000-2.in
```

Input files strictly follow the structure shown below:

- N – Length of this DNA sequence
- S – The DNA sequence itself

Sample Input

```
8
AGCATTAC
```

Your output should strictly follow the structure shown below (output to standard output, not to a file):

- l - the length of the LCS of the two supplied DNA strings

Do not output any other text, as this may cause your submission to be graded incorrectly. Disable or remove any logging prints for debug purposes in your submission.

1.4 Starter Code

We provide a sequential implementation of the bottom-up dynamic programming solution in `LCS-seq.c`. Here, we compute in row-wise fashion the LCS for all pairs of prefixes of both DNA strings in a matrix, using solutions to smaller instances of the problem to construct solutions to larger instances. To reduce the memory footprint of the program, we only retain one row of results from the matrix at all times, since that is sufficient to compute the next row of the matrix.

Additionally, we provide two sets of input files that you may use for your testing. You are advised to produce new test cases for you program.

Start by understanding how the provided implementation works. You may parallelize our implementation by observing the independence of the diagonals in the matrix, or you can come up with new algorithms that are easier or faster to parallelize. You might find multiple research papers on parallelizing the computation of the LCS if you run a Google search. Feel free to apply any algorithm, **but make sure that you cite the resources in your report.**

Your parallel implementations should give the same result (output) as the provided sequential implementation, and execute faster on the lab machines.

1.5 Optimizing your Solution

The provided implementation is relatively simple, and thus it may seem difficult at first to identify parallelism that can be extracted by a multi-process/multi-threaded model. As such, you might need to try several approaches to parallelize the algorithm, or even consider alternative algorithms. In your submission, you are advised to retain your alternative implementations (even if they are slower or not correct), and explain the changes you have made in each.

Distinguish any alternative implementations you include in your submission clearly from the final parallel implementations to be graded.

You may obtain **bonus marks** by using Advanced Vector Extensions (AVX) instructions with any of the implementations to further improve the execution time and scalability (speedup) for your programs.

You should demonstrate your parallel implementation scales with increasing input size (DNA sequence lengths), number of threads/processes and hardware capabilities. To analyze the improvements in performance, you should measure the execution time for carefully chosen input DNA lengths with an increasing number of threads/processes.

For the Part 2 submission, you are advised to focus more on optimizing your implementation, and comparing between OpenMP and your multi-process/multi-threaded implementations. You are allowed to change/improve your implementations in Part 1 for your Part 2 submission. **Explain the changes briefly in your report, if any**.

2 Admin Issues

2.1 Running your Programs

For all three implementations, run your program with varied input sizes (DNA sequence lengths) and number of processes/threads used. You should select DNA sequence lengths that have meaningful execution times when solved by the provided sequential implementation.

For performance measurements, run your program at least 3 times and take the shortest execution time. You should use the machines in the lab for your measurements:

- `soctf-pdc-001` - `soctf-pdc-008`: (Xeon Silver 4114)
- `soctf-pdc-009` - `soctf-pdc-016`: (Intel Core i7-7700K)
- `soctf-pdc-018` - `soctf-pdc-019`: (Dual-socket Xeon Silver 4114)
- `soctf-pdc-020` - `soctf-pdc-021`: (Intel Core i7-9700)

If you are computing the speedup of your parallel implementation, compute it using the provided sequential implementation as the baseline.

2.2 Bonus

You may obtain up to 3 bonus marks for the following:

- **up to 2 bonus marks for Part 1**: for using AVX on any of your implementations to improve the execution time and scalability. Measurements have to be shown in your report to demonstrate the improvements.
- **up to 1 bonus mark for Part 2**: for achieving the best speedup among all OpenMP submissions. We will assign in total 4 bonus marks to the class, one mark for obtaining the best speedup on each type of lab machine (listed above). If an implementation tops on multiple machines only one bonus mark will be allocated to the student, and we will consider the next best speedup. Partial marks can be obtained for the second and third best on each machine.

2.3 FAQ

Frequently asked questions (FAQ) received from students for this assignment will be answered here. The most recent questions will be added at the beginning of the file, preceded by the date label. **Check this file before asking your questions.**

If there are any questions regarding the assignment, please post on the LumiNUS forum or email Keven (keven@comp.nus.edu.sg) or Richard (e0191783@u.nus.edu).

2.4 Submission Instructions

You are allowed to work in groups of maximum two students for this assignment. You may discuss the assignment with others but in the case of plagiarism, both parties will be severely penalized.

Assignment submission will be done in in two rounds:

1. **Mon, 14 Sep, 11am – Assignment 1 – part 1 [8 marks]**: **Both** process and `pthread` implementations (`LCS-threads.c` and `LCS-processes.c`), accompanied with test cases and a report that includes performance measurements for the sequential, multi-threaded and multi-process implementations. About 7 marks will be allocated for your implementation and 1 mark for your report with explanations and measurements (subject to change).
2. **Mon, 28 Sep, 11am – Assignment 1 – part 2 [8 marks]**: **All three** implementations (`LCS-threads.c`, `LCS-processes.c`, and `LCS-omp.c`), accompanied with test cases and a report that includes a performance comparison between all three parallel implementations. About 5 marks will be allocated for the OpenMP implementation, and 3 marks for your performance comparison (subject to change).

Your report should include:

- A brief description of your program's design (how they work) and implementation assumptions.
- A brief explanation of the parallel strategy you used in your multi-process, multi-threaded and OpenMP implementations, e.g. synchronisation, work distribution, etc.
- Any special consideration or implementation detail that you consider non-trivial.

- Details on how to reproduce your results, e.g. inputs, execution time measurement, etc.
- Execution time and speedup measurements of your sequential and parallel implementations. Run your implementations on at least two types of nodes (machines) in our lab. Vary input size, number of cores used, number of processes and threads, etc.
- Analyze your measurement results and explain your observations. Speedup computation and performance comparison between the provided sequential and your parallel implementations should be included in your final report (part 2). State your assumptions.
- In your report for Part 2, present at least the following: graphs showing the execution time (y-axis) variation with input size (DNA sequence lengths), number of processes/threads (x-axis) (fixed input size of 45,000), number of cores used (fixed input size and fixed number of threads). You can vary the number of threads/processes from 1 to 256. Feel free to add any other graphs or insights that you find.

There is no minimum or maximum page length for the report. Be **comprehensive**, yet **concise**.

Submit your assignment before the deadline under LumiNUS Files. Each student must submit one zip archive named with your student number(s) (A0123456Z.zip - if you worked by yourself, or A0123456Z_A0173456T.zip - if you worked with another student) containing the following files and folders:

1. Your C/C++ code for `LCS-threads.c`, `LCS-processes.c`, and `LCS-omp.c`
2. `README` text file, minimally with instructions on how to compile and execute the code. Mention the lab machines that you used in your testing.
3. Report in PDF format (part1_report.pdf and part2_report.pdf)
4. A folder, named `testcases`, containing any additional test cases (input and output) that you might have used
5. A folder, named `scripts`, containing any additional scripts you used to measure the execution time and extract data for your report.

Note that for submissions made as a group, only the most recent submission (from any of the students) will be graded, and both students receive that grade. A penalty of 10% per day (out of your grade) will be applied for late submissions.