DNA Sequence Alignment

EduHPC'24 Peachy Assignment

1 Introduction

You are provided with a sequential code that uses a brute-force method to detect exact alignments (matches) of a series of nucleotide patterns in a longer genetic DNA sequence.

Example:

```
Sequence: CCGTACCTGTGACCGTTAAAACTTTC
Pattern:             ACCGT
```

The given pattern aligns at position 11, considering that the first position is indexed as 0

The genetic sequence is stored in a one-dimensional array of characters, each character representing a nucleotide type (ACGT). It is randomly generated taking into account the frequency of probability of each nucleotide type received as program arguments. The patterns can be either completely random (they may or may not be found in the main sequence), or randomly chosen samples from the original sequence (they will always be found). All patterns are stored in a vector of character strings.

The proposed DNA sequence alignment approach is based on looking for an exact match of each pattern starting at each position of the original sequence. Although there are more efficient algorithms, this brute force method is used because it is simple, very regular, and parallelizable. To reduce the execution time, the program stops the search at the first match of each pattern. After all searches have finished, the program calculates the total number of patterns found, and for each position of the main DNA sequence, the number of patterns that are aligned covering that position. Finally, a series of checksums are computed and displayed to easily check correctness of the program comparing them with those obtained with the sequential version.

2 Sequential Code Description

Program arguments: These arguments are used to describe the input scenario. They are designed to give the user a lot of control to generate scenarios of different types and with different computational workloads.

1. `<seq_length>`: Length of the main DNA sequence.
2. `<prob_G>`: Probability of occurrence of G-nucleotides.
3. `<prob_C>`: Probability of occurrence of C-nucleotides.
4. `<prob_A>`: Probability of occurrence of A-nucleotides.

   (The probability of occurrence of T-nucleotides is calculated internally by subtracting the sum of the above probabilities from 1)

5. `<pat_rng_num>`: Number of random patterns.
6. `<pat_rng_length_mean>`: Mean length of the patterns.
7. `<pat_rng_length_dev>`: Length deviation of those patterns.

8. `<pat_samples_num>`: Number of patterns that are samples of the original sequence.
9. `<pat_samp_length_mean>`: Mean length of the sample patterns.
10. `<pat_samp_length_dev>`: Length deviation of the sample patterns.
11. `<pat_samp_loc_mean>`: Mean location of the beginning of the samples.
12. `<pat_samp_loc_dev>`: Deviation from the start location.
13. `<pat_samp_mix:B[efore]|A[fter]|M[ixed]>`: This parameter means:
    - `B`: The sample patterns are sorted before the random ones in the patterns list.
    - `A`: The sample patterns are after the random ones in the list.
    - `M`: The sample and random patterns are interleaved.
14. `<long_seed>`: A random seed for the generation of different and reproducible cases for the same arguments.

Students are encouraged to generate their scenarios with significant computational workload to perform their time measurement tests. It is recommended to first check that the results are correct in different scenarios. Several program argument examples are provided along with the code.

Results: The program shows the execution time of the target computation part (without initialization times), and the checksums that are used to verify the correctness of the modified program.

Debug Mode: If the code is compiled with the flag -DDEBUG (included in the provided Makefile), a piece of code is activated to show the values of the input arguments, the generated sequence and patterns and the results of the ancillary arrays used for the checksums. It can help to detect possible errors or to better understand what the program does using very small examples. The DEBUG parts will never be used for time measurements. Students can modify or add similar parts with conditional compilation dependent on the DEBUG symbol that will be ignored in the final version.

3 Project Goal

The goal is to parallelize the code, using each parallel programming model proposed by the teacher, without changing the algorithm being applied or the program results. Optimize the code, check that the results are correct and equivalent to the sequential execution with the same input arguments, and try to obtain the best possible performance. It is important to distinguish the pieces of code that can be parallelized directly, those that need modifications, etc.

A note about random numbers and parallelism

The generation of a pseudo-random sequence of numbers with the classical C library functions is inherently sequential. Our code provides a linear congruently pseudo-random generator that provides the tools to generate parts of the sequences in parallel. It provides functions to: (1) compute the next random value with a uniform or a normal (mu, sigma) distribution, and (2) do a jump ahead in the sequence for a given number of steps.

Part of the program to parallelize and optimize

The sequential code indicates the functions and parts of the main code that students should parallelize and optimize. They are clearly identified with special comments. Students should refrain to alter or modify the code outside the marked functions and parts, especially those where time measurements, initialization, or results output are carried out.

In the parts to be modified, changes and optimizations can be made to the code and new data structures can be created, modified, or deleted, as long as they do not alter the idea of the brute-force algorithm.

Beware of modifications that can significantly alter the exploitation of the parallelism resources with the proposed parallel programming model, which is the purpose of the assignment.

The DEBUG code is not measured for the final performance result, so there is no need to worry about it. It is there only to help you to understand or debugthe code after changing it.

The program is compiled with the maximum compiler optimization level (`-O3` option), as the aim is to focus on program changes that do not interfere with compiler optimizations.