Genome index: MPI assignment

Due date: 7th June

1 Introduction

Human genome contains about 3.2 billion of base pairs. However, this is not the longest one. A Japanese flower called *Paris japonica* has genome of 149 billion pairs. Axolots have the longest genome in the animal kingdom - about 43 billions pairs.

To efficiently work on such datasets, biologists treat those genomes as strings and use structures and algorithms that were previously used while working with text data. One of such concepts is a *genome index*. This is a structure derived from the genome itself that allows fast querying of it.

Your goal is to build a genome index in the form of Suffix Array. Then, using indices of multiple genomes you will check how often some sequences appear in those genomes.

2 Suffix Arrays

2.1 Definition

Let $S = s_0 s_1 s_2 ... s_{n-1}$ be a string of length n. Each character s_i of the string is an element of a common alphabet $s_i \in \Sigma$. Further, we designate $S_i = s_i s_{i+1} ... s_{n-1}$ as the suffix of S starting at position i.

The Suffix Array of S, denoted by SA, is a length n array which contains the lexicographically sorted order of the suffixes. We set $SA[j] = i$ if and only if the suffix S_i is the j-th among the lexicographically sorted suffixes.

2.2 Querying

Given a string T and the SA of a string S, the positions on which T occurs in S can be found in $O(|T| \cdot log\ |N|)$ time.

The algorithm is based on binary search. Running binary search twice we find a, b such that ~~$S_{SA[a]} \leq T \leq S_{SA[b]}$~~ T is a prefix of both $S_{SA[a]}$ and $S_{SA[b]}$ (EDIT: 31.05.2022) and the $[a, b]$ range is of maximal length. If such range exists, there are $b - a + 1$ occurences of T in S, starting on positions $SA[a], SA[a+1], ..., SA[b]$. Suffix Array is used here in the following way: given j as the next binary search step, compare T with $S_{SA[j]}$. If T is a prefix of $S_{SA[j]}$, it's an exact match.

2.3 Interpretation

In our case S is the whole genome and T is some sequence of nucleobases (a single gene, for example). Both S and T are strings over the $\Sigma = \{A, C, G, T\}$ alphabet. The order of nucleobases is given by their letters values, i.e. $A < C < G < T$.

3 Task

Your task is to:

- Implement distributed algorithm to build Suffix Array;
- Build indices (Suffix Arrays) of multiple genomes;
- Respond to queries of type "How often does the sequence T occur in provided genomes?"

Note that your task is to build an index and *then* test its behavior on queries. We do not force this order by design to give you more space for parallelization during the query phase. However, solutions that analyze queries in order to build an index won't be accepted. The reason for this is that a *genome index* should be a general purpose structure that - once build - should work with any given set of queries.

The problem of creating the distributed Suffix Array was already studied. You may build the Suffix Array in any distributed manner you want, but we suggest using the following article as the basis for your algorithm: `https://dl.acm.org/doi/pdf/10.1145/2807591.2807609`.

In the article you should focus on:

- Section *3*, where definitions are introduced;
- Section *4*, where authors present their ideas on Suffix Arrays:
  - *4.1, 4.2* describe sequential algorithm that you can implement in order to verify your solution;
  - *4.2.1* introduces ideas on how to parallelize this approach;
  - Next, in the *4.3* section there is an improved algorithm. Using the ideas included in this section may help your program to score more points for performance.

4 Input and output

Programs will be tested automatically. Please stick to the format below.

4.1 Input

Your program will be compiled using the following instructions:

```
cd xx123456; make clean; make
```

Your program will be then launched inside a Slurm job as follows:

```
mpiexec ./genome_index :n :m :genome_in :queries_in :queries_out
```

where:

- `:n` is the number of genomes;
- `:m` is the number of queries;
- `:genome_in` - there are `:n` genomes, each stored in `{genome_in}_{0, ..., :n - 1}` file. Each file contains single string over the $\{A, C, G, T\}$ alphabet. You should not access those files directly;
- `:queries_in` is the file with queries. It contains `:m` strings over the $\{A, C, G, T\}$ alphabet. Each string is in a separate line;
- `:queries_out` is the file where you should put the results. Format is described in the *Output* section.

You should not access `:genome_in` files directly. Each process is supposed to read its own part of the input data via MPI I/O. We provide a library (`data_source.h`) that handles this task:

```
class DataSource {
public:
    DataSource(char *genome_in);        // Simply pass :genome_in from the program arguments

    uint64_t getTotalGenomeSize(int i);  // Returns total length of :i-th genome
    uint64_t getNodeGenomeSize(int i);   // Returns the length of :i-th genome part
                                         // assigned to the current node
    uint64_t getNodeGenomeOffset(int i); // Returns the first index of :i-th genome that is
                                         // assigned to the current node
    void getNodeGenomeValues(int i, char *buffer); // Reads data of length :getNodeGenomeSize(i)
                                                   // at offset :getNodeGenomeOffset(i)
                                                   // of the :i-th genome file using MPI I/O,
                                                   // into the buffer.
}
```

Each genome is distributed equally among all nodes, i.e. first processes receive $\lceil \frac{n}{p} \rceil$ characters each and the rest gets $\lfloor \frac{n}{p} \rfloor$ characters each.

4.1.1 Assumptions

- You should assume that a single node has not enough memory to store the entire Suffix Array or the input data of a whole genome.
- You may assume that genomes and queries are strings over the $\{A, C, G, T\}$ alphabet.
- You may also assume that the queries are much shorter than genomes, i.e. one node can store all queries in its memory.

4.2 Output

Your program should put results in the `:queries_out` file. There should be `:m` lines in this file, each line consists of `:n` integers. `:i-th` value in the `:j-th` line should be equal to the number of occurences of `:j-th` query in the `:i-th` genome.

4.3 Example

When your program is called as follows:
`mpiexec ./genome_index 3 2 example queries result`
And the files are:

```
example_0:
ACGTACACACCGCTACCGACCGTC

example_1:
ACGTCGACCTACCGTCATC

example_2:
ACCTGATCAGACTACGCTA

queries:
ACC
CGTC
```

Then your program should put the following in the `result` file:

```
3 2 1
1 2 0
```

In the provided archive there is a script `run_test.py`, which runs this test on your solution in conditions similar to the final ones.

5 Solution content

Please send us a single `.zip` file containing a single directory with your login (`ab123456`); the directory has at least the following files:

- `data_source.h`: This file cannot be modified.
- `data_source.cpp`: This file cannot be modified. For tests, we might use a different implementation of the library.
- `report.pdf`: a report describing your implementation. Describe the optimizations you implemented. Show weak and strong scaling results. For scaling, find instances and input parameters so that the measurements are realistic, but the (wall clock) run time does not exceed 3 minutes when more than 4 nodes are used.

6 Scoring

- correct MPI implementation of the parallel genome index (tested with queries): 12 points;
- report: 4 points (incorrect implementations do not get these points);
- performance: 9 points (incorrect implementations do not get these points).

Your solution must score at least 8 points to complete the course.

We will score correctness on our test data. If your solution passes most of the tests, but fails on some, we will contact you and you will be able to submit a patched version.

We will use Okeanos for performance testing.

To optimize performance, consider using advanced MPI operations, like asynchronous messages, collectives, custom datatypes, custom communicators.

Our performance tests will use `--tasks-per-node 24` unless you write in your report that your solution is more efficient with other value (e.g., you use `MPI+OpenMP` and you just need `--tasks-per-node 1` or `--tasks-per-node 2`).

Please do submit your assignment by the due date. If you're late, your score will be reduced by 1 point for every 12 hours (i.e.: if you're late by 2h, we subtract 1 point from your score; if you're late by 25h, we subtract 3 points).

There is a second due date - 14th June. Submitting by this due date is very risky. First, very good solutions submitted by this due date receive at most 10 points. Second, there will be less time for patching. Please do submit your assignment by the normal, first due date.