Chair for Clinical Bioinformatics
Pascal Hirsch
Annika Engel
Saurabh Pandey
Ilia Olkhovskii

Programming Course
5th Assignment

Hand in until 31.01.24 at 11:59 pm

General remarks

Please make sure that you follow all rules stated in the general remarks PDF in the moodle. The number of barbells ⤡ for each task describes the expected difficulty. The maximum of number of barbells per task will be 4.

Task 1 (10 P.): ⤡

1. A common variable `counter` is incremented from 5 threads 5000 times. The result is 24756. How is that possible? (1 P.)
2. How could you correct the problem? Provide a short code example. (2 P.)
3. Why should `lock_guard` be preferred to explicit locking? (1 P.)
4. What is an alternative to locking variables and why? (1 P.)
5. Can multiple threads access a variable that is read-only safely without locking? (1 P.)
6. Which type of mutex would you use if you have multiple threads reading from the same value and only one writing to the value? Provide a short code example. (2 P.)
7. How would you parallelize a for loop with the Intel TBB? How with OpenMP? (2 P.)

Task 2 (12 (4+8) P.): (⤡ + ⤡⤡⤡)

For this task we provide code for a program that expects a FASTA file as input containing small NGS reads (first argument). The second argument is the name of the output FASTA file. Since many reads occur multiple times, this program "collapses" the reads, such that downstream tools can work with a much smaller FASTA file. Collapsing means: if the exact same sequence occurs Z times in the

FASTA file, we generate a new FASTA header for this sequence by adding the number of occurrences like this: `>seq_12345_x`Z

The new unique sequences should be given incremental numbers in the header according to the following scheme: `seq_1, seq_2, ...` to be uniquely identifiable.

The output is the new collapsed FASTA file with reads sorted according to their frequency (highest frequencies first), length (descending order), and last alphabetically (descending order).

The collapser should also be able to work with already (partially) collapsed input files, meaning we have to check if the FASTA header ends with `_x`Z, and then handle this accordingly.

Your task is to first write down a (non-exhaustive) list of five mistakes or inefficient approaches of the provided code.

Then, to get all points, you should fix the code and make it as fast as possible. If necessary you can rewrite the whole program. It should run for the provided example FASTA file in under 0.5 seconds on the course virtual machine `progbfx.cs.uni-saarland.de`.

Task 3 (12 (4+8) P.): (➘ + ➘➘➘)

For this task we provide code for a program that expects a FASTQ file as input and outputs a collapsed FASTQ file as output. Collapsing is done similar to the collapsing for FASTA files:

1. find equal sequences
2. count them
3. the header for the collapsed sequences is in the following format: @seq_number_xZ for Z occurrences of the sequence
4. then to obtain the quality scores you should first compute the mean of the probability that a base call is wrong for each position of the sequences that are collapsed into one, and then transform the probability back to a quality score.

The base quality is in Sanger format. It can encode a Phred quality score from 0 to 93 using ASCII 33 to 126. The probability p that a base call is wrong can be derived from the base quality Q: $Q = -10 \cdot log10(p)$. The output should be sorted by decreasing frequency, then by the sum of the probabilities of a sequence (descending order) and finally alphabetically (descending order).

Your task is to first write down a (non-exhaustive) list of five mistakes or inefficient approaches of the provided coded.

Then, to get all points, you should fix the code and make it as fast as possible. If necessary you can rewrite the whole program. It should run for the provided example FASTQ file in under 0.5 seconds on the course virtual machine `progbfx.cs.uni-saarland.de`.

Task 4 (13 P.):

In this task your goal is to implement a parallel version of the Burrows-Wheeler transformation (BWT). First, your program should read all lines from the standard input, remove duplicates, and lexicographically sort the remaining lines in ascending order using a concurrent merge-sort. Make sure that your implementation properly removes trailing new-line separators in the input strings. Second, compute the BWT of every unique line and output them line by line in the corresponding sorted order computed on the input lines as described above. For calculating the BWT, compare characters according to the ASCII table. Please note that your implementation should append a dollar-sign ($) to the end of each input line before computing the BWT. In addition, the implementation should raise an exception in case any of the input lines contains a dollar-sign already. The first and only program argument given defines the maximum number of threads the implementation may use at any time. Optimize your code such as to use as many threads as possible while satisfying the upper bound parameter.

Example:

```
./task4 4
Mississipi
The temperature was freezing last night
55AC-238947201
Mississipi
```

Output:

```
1C207-29$54385A
i$pssMissii
sgteet$wlrhrrtpe niTgnz ei meufaash at e
```

Task 5 (13 P. (6+7)):

a) Write a function that reads a sorted GFF3 file and returns a vector of GFF entries. Store the GFF entries as follows:

```
struct GFFEntry {
        std::string seqid;
        std::string source;
        std::string type;
        uint64_t start;
        uint64_t end;
        float score;
        char strand;
        uint8_t phase;
        std::map<std::string, std::string> tags;
};
```

Your function should have the following signature:

```
std::vector<GFFEntry> readGFF(std::string filename);
```

Further write a function with the following signature that writes a GFF3 file:

```
write_mean_coverage(const std::vector<GFFEntry>& entries,
std::ostream& os, const  std::string seqid,
const unsigned long start, const unsigned long end)
```

Then, develop a program that reads a sorted GFF3 file and outputs the mean coverage of GFF3 features on the standard output. The input file is passed as first argument. The second argument the is sequence identifier that should be filtered and the third and fourth argument specify the range of the allowed starting coordinates (closed interval, one-based). We define the mean coverage of features the number of bases overlapping between all GFF3 entries in the queried interval divided by the overall range of this interval (see example). Note that boundaries are always included in an interval. For your testing purposes, given an unsorted GFF3 file you can create a sorted version by applying the bedtools sort command, where necessary.

You should call your program as follows: `./task5a input_file seqid start end`
e.g. `./task5a task5.gff3 chr1 50000 200000`

b) To get all the points implement the same functionality but using two concurrent threads. One that reads your file and one that **concurrently** both processes and outputs the requested range to the standard output. Implement the functionality in the following function:

```
void readwritemean_coverage(std::string input_filename,
std::string seq_id,
uint64_t start, uint64_t end);
```

Example.gff3:

```
#<Many gff3 header lines that we skip here>
chr8    .       miRNA   1       1       .       +       .       ID=1
chr8    .       miRNA   1       3       .       -       .       ID=2
```

Example call: `./task5b Example.gff3 chr8 1 1`
Output:
2