Shared-Memory Programming: Genome Search

In this assignment, you will write a Posix threads ("pthreads") program that searches the human genome for a nucleotide sequence.

1 Read

Like all life on earth, our genome is encoded in long strands of *deoxyribonucleic acid* (DNA). In humans, DNA is partitioned across 24 *chromosomes*. Each chromosome comprises a *nucleotide sequence* ranging in size from 27,442,533 nucleotides in the Y chromosome to 248,517,897 nucleotides in chromosome 2. In total, the human genome comprises 3,149,694,148 nucleotides.

The standard encoding of nucleotide sequences uses four letters to represent the constituent molecules as detailed in Table 1.

Letter	Amino Acid
A	adenosine
C	cytidine
G	guanine
T	thymidine

Table 1: Nucleotide encoding

It is helpful to be able to search chromosomes for occurrences of particular sequences of nucleotides. We might wish to find, for example, the number of times the nucleotide sequence `GATTACA` appears in the Y chromosome (4,937 times, as it turns out).

1.1 File Format

Fasta is a standard file format for encoding nucleotide sequences. Each Fasta file that we will use encodes the nucleotides in a *single* chromosome.

For example, here are a few lines from the Fasta file that encodes the X chromosome.

```
CGCTTGCAGTGGAAAGCAGGGAGTTGTGAGCTTGTGTCATAGAATATACAATTTCCTTGTGGTGTCCCAC
AAAAAGCTATCTCCCATTCAGATTGTTCCTACGTAGTTGTGCCAGTCATAACAAGAACCAAGATC
>ref|NT_167197.2| Homo sapiens chromosome X genomic scaffold, GRCh38.p12 Primary Assembly HSCHRX_CTG7
TGTCTGTGTATGTATATATATATATATATATATATATATATATATATATATATATGCACACATAGCATGT
ACATCGCTCAAAGCCCAGGTTTTTGGACATCAGTGTAACTGACCTGCTGATTAGCGTATGGTGAGGACTG
```

The bulk of the lines are strings of the four characters listed in Table 1. You will also see the character `N`, meaning "any.' Interspersed throughout the file are lines containing descriptive text; these lines start with the greater-than symbol (`>`). We are only interested in the nucleotide sequence, not the descriptive text.

1.2 Reference Implementation

This assignment includes a sequential reference implementation of genome search. The reference implementation:

1. Loads genome sequences from one or more FASTA files.
2. Searches all sequence data for a pattern.
3. Reports the number of matches of the pattern in the sequence data.
4. Optionally reports the locations of the pattern in the sequence data.
5. Reports the duration of the matching operation.

Refer to Section 2 for information about obtaining, compiling, and running this program.

As mentioned, the reference implementation can load genome sequences from one or more FASTA files. This allows us to experiment with performance on different problem sizes (i.e., varying numbers of nucleotides). For example, we can run the program on data for a single chromosome, multiple chromosomes, or the entire genome (all 24 chromosomes). As a simplification, the program simply concatenates together the nucleotide sequences from the FASTA files read. This simplification means that the program may find spurious matches of the pattern at chromosome boundaries.

2 Set Up

Set up to do this assignment as follows:

1. Download the `handout.tar.gz` file from the course web site.
2. Create a working directory for the assignment and `cd` into it.
3. Unpack the tarball from a BASH prompt:

   ```
   tar zxvf handout.tar.gz
   ```

4. Download the genome data from the National Institutes of Health (NIH) by running the `get-data.sh` shell script. It should download all 24 chromosome files to your working directory.

   The genome files from NIH are zipped (`.gz`) to save space. *Leave them zipped!* The files total about 875 MB zipped, but expand to more than 3 GB unzipped. The reference implementation includes a function that reads a zip file directly into memory.

5. Compile the `sg.c` file, which contains a sequential implementation of the genome search code. The included `Makefile` makes this easy; at the BASH prompt, type:

   ```
   make sg
   ```

6. Try executing the `sg` program a few times to get a sense of what it does. For example:

   ```
   ./sg -m 300 -p CATGGGCAT hs_ref_GRCh38.p12_chr2.fa.gz
   ```

will search for the pattern `CATGGGCAT` in chromosome 2, allocating 300 MB of space to read in the genome data. You can get help from `sg` by running it with the `-h` flag.

Spend some time "bonding" with this code. You may reuse any of it in your parallel implementation. I've included lots of comments for your benefit—*read them* and understand the code.

3 Code

Write a shared-memory parallel program using Posix threads that duplicates the functionality of the reference program but can run on multiple threads. Your parallel implementation should:

1. Allow the number of threads to be specified using a command-line parameter called `n` that takes as an argument the number of threads to run. For example:

   ```
   myprog -n 4 ...
   ```

   would run the program on four threads.
2. Partition the genome search as efficiently as possible across the threads at run time. Each thread should maintain a local count of the number of matches.
3. Calculate the global number of matches.
4. Return the *same* number of matches for a given pattern and nucleotide sequence *regardless* of the number of threads.
5. Report
   (a) The global number of matches of the pattern in the nucleotide sequence
   (b) The parallel run time, t_p, of the search. Start timing immediately before the first thread is created, and stop timing immediately after the last thread is joined. Do *not* include the time it takes to load the genome data from disk.

It's *extremely* helpful to run the compiler with all warnings enabled to help catch problems. With `gcc`, the Gnu C compiler, use the `Wall` flag as follows:

```
gcc -Wall ...
```

This flag makes the compiler very picky, which is *exactly* what you want. Trust me.

4 Experiment

Gather experimental data on the behavior of your parallel implementation. Measure t_p for combinations of data size, pattern size, and number of threads, as described here.

4.1 Data Size

Run your code on data for a single chromosome, the entire genome (i.e., all chromosomes), and representative numbers of chromosomes in between.

Each chromosome has a different number of nucleotides. When reporting the data size for your experiments, report the *total number of nucleotides*, not the number of chromosomes.

4.2 Pattern Size

Run your code on patterns of various sizes: small, large, and somewhere in between. Try patterns that occur in the data and patterns that don't.

4.3 Number of Threads

Run your code on various numbers of threads, including at least the following:

1. *one* thread
2. *two* threads
3. as many threads as *cores* on your processor
4. as many threads as *hyperthreads* supported by your processor

You are welcome to experiment with other numbers of threads at your discretion.

5 Submit

Submit to the course web site a tarball (Section 5.1) containing the following deliverables.

1. All the source code for your parallel genome search program (i.e., source files, header files, Makefile, etc.). Your program must:
   (a) (10 points) Load genome sequences from one or more FASTA files named on the command line.
   (b) (10 points) Partition the search operation across multiple threads, the number of which is given on the command line at run time.
   (c) (20 points) Search for a pattern in all sequence data in parallel.
   (d) (10 points) Correctly report the total number of matches of the pattern across all threads.
   (e) (5 points) Correctly report the duration of the matching operation.
2. A written overview of your parallel implementation detailing your experiments, results, graphs, and analysis. Specifically, your report should include:
   (a) (5 points) Strategy for partitioning the search across multiple threads
   (b) (5 points) Strategy for reliably computing the global number of matches across all threads
   (c) (5 points) Strategy for choice of the experimental parameters detailed in Section 4 of this write-up
   (d) (5 points) For all experiments, the number of threads (p), measured parallel run-time (t_p), and calculated speedup (S) and efficiency (E)
   (e) (8 points) Graph(s) of S versus p for various data and pattern sizes
   (f) (7 points) Graph(s) of E versus p for various data and pattern sizes
   (g) (10 points) Written analysis of the performance of your implementation that interprets the empirical data you have gathered against your design and expectations

Refer to some of my papers (posted on the course web site) for examples of how to report the design, implementation, and performance of a parallel algorithm. In particular, your data tables (of p, t_p, S, and E) and your graphs of S and E should echo my approach in these papers.

To receive full credit:

1. Your *code* must be clear, complete, well-commented, and easy to understand. It should compile cleanly (i.e., no errors or warnings when compiled with `-Wall`).
2. Your *prose* must be clear, grammatical, spelled correctly, and punctuated properly. Write complete sentences containing actual *verbs*. Use the active voice. Strive to make your technical writing *not suck*.

5.1 Tarballs

To create a tarball called `foo.tar.gz` containing files `bar`, `baz`, and `quux`, run the following command at the Bash prompt:

```
tar zcvf foo.tar.gz bar baz quux
```

The flags to `tar` mean `z`ipped, `c`reate, `v`erbose, and `f`ile. Refer to the `tar` manual page for details (run '`man tar`' from the Bash prompt).

5.2 LaTeX

I encourage you to learn and use LaTeX for all your technical writing. Reports submitted in this format will receive more favorable grading than those submitted in another format.

A good starting place is The LaTeX Project, which includes details on installing and learning LaTeX. I (and many of the faculty) would be delighted to help you learn or hone your skill with LaTeX (and the underlying TeX system on which it's based).

Question	Points	Score
1	55	
2	45	
Total:	100	