Project 3: Mutual exclusion and map reduce pattern in OpenMP (100 points)

Submission guidelines:

Please submit **2 files** in Canvas.

- a PDF report file (filename: **Lastname_Firstname_Project_3_Report.pdf**)
- a ZIP file (filename: **Lastname_Firstname_Project_3.zip**) containing:
  - `Problem_2/`
    - `compute_average_TF_Exp1_critical.c`
    - `compute_average_TF_Exp1_atomic.c`
    - `compute_average_TF_Exp1_locks.c`
    - `compute_average_TF_Exp2_schedule.c`
    - `Makefile`
  - `Problem_3/`
    - `compute_median_TF_Exp1_baseline.c`
    - `compute_median_TF_Exp2_mapreduce.c`
    - `Makefile`
  - `Problem_4/`
    - `kmeans_clustering.c`
    - `Makefile`

All test-cases will be run on Schooner when graded. So, please make sure your programs run in Schooner before you turn in your project.

Test data:

The test data are provided on Schooner at
`"/home/oucspdnta/current/test_data/Project_3_Tests/"`.

Autograder Instructions

The autograder for Project 3 has been changed to be more modular and to allow individual problems to be run in each project. The autograder now consists of multiple scripts that can be run individually to test individual problems.

Where to get the Autograder:

- The autograder and the test data is located on Schooner at:
  `/home/oucspdnta/current/test_data/Project_3_Tests/`
- To copy the autograder and the test data to your own home directory, run the command:
  `cp -r /home/oucspdnta/current/test_data/Project_3_Tests/ .`

Required File Structure:

- As noted within the python script, the file structure of the project is necessary:
  - `test_data/`
    - `DNA_Files/`
    - `Problem_2/`
    - `Problem_3/`
    - `Problem_4/`
  - `LastName_FirstName_Project_3/`
    - `Problem_2/`
      - `compute_average_TF_Exp1_critical.c`
      - `compute_average_TF_Exp1_atomic.c`
      - `compute_average_TF_Exp1_locks.c`
      - `compute_average_TF_Exp2_schedule.c`
      - `Makefile`
    - `Problem_3/`
      - `compute_median_TF_Exp1_baseline.c`
      - `compute_median_TF_Exp2_mapreduce.c`
      - `Makefile`
    - `Problem_4/`
      - `kmeans_clustering.c`
      - `Makefile`
    - `autograder_project_3.sbatch`
    - `autograder_project_3.py`
    - `autograder_problem_3_2.py`
    - `autograder_problem_3_3.py`
    - `autograder_problem_3_4.py`
  - `autograder_base.py`

To Run the Autograder:

- **Running on Schooner:**
  - First, make sure to edit the sbatch file where indicated. You will have to edit your username, and the directories of where the script is located, and where to output the stdout and stderr files.

  - You can also edit the exact python script that the sbatch job will call, and thus indicate which problem to grade, or to grade all of them

  - To submit the job, run the command:
    `sbatch autograder_project_3.sbatch`

  - To see the status of the job, run the command:
    `squeue –u oucspdn###`

  - Once the job completes, it will return the output and error messages in two files:
    `autograding_########_stderr.txt`
    `autograding_########_stdout.txt`
    (Where the #s are replaced with the job number)
- **Running Locally:**
  - For Windows users, we recommend using Windows Subsystem for Linux (WSL) with Ubuntu to run code from the command-line.
  - Make sure that your file structure is correct, otherwise the autograder will not be able to find your files.
  - Navigate to the location of autograding_individual_1b.py, and then run either:
    `python autograder_problem_3_#.py`
    To run the autograder on just one individual problem
    or
    `python autograding_project_3.py`
    To run the autograder on all problems
  - This will then test your program against the provided test data, and will output your score for each test, where a 1.0 for each test indicates that the test succeeded, and a 0.0 means the test failed.

Problem 1. [20 points for CS4473] and [Not required for CS5473]

Suppose that a and b are two arrays with 100 integers. Explain why each of the following loops can or cannot be parallelized with a parallel for directive.

a)
```
# pragma omp parallel for
for (int i = 0; i < 100; i++){
     if ( i % 2 == 0 )
          b[i] = a[i];
     else
          a[i] = b[i];
}
```

b)
```
int x;
# pragma omp parallel for
for (int i = 0; i < 100; i++){
     if ( b[i] > a[i] ){
          x = a[i];
          a[i] = b[i];
          b[i] = x;
     }
}
```

c)
```
# pragma omp parallel for
for (int i = 0; i < 100; i++){
     for(int j = i+1; j < 100; j++)
          a[i] += a[j];
}
```

d)
```
for (int i = 0; i < 100; i++){
     # pragma omp parallel
     for(int j = i+1; j < 100; j++)
          a[i] += b[j];
}
```

e)
```
# pragma omp parallel for
for (int i = 1; i < 99; i++){
     if( i % 2 == 0 ){
          a[i] = (a[i-1] + a[i+1]) / 2;;
     }
}
```

What to submit:
The explanations in the report.

Problem 2. [50 points for CS4473 and CS5473]

You will find a text file called *GRCh38_latest_rna.fna* from this website under *RefSeq Transcripts* and *GRCH38*: https://www.ncbi.nlm.nih.gov/genome/guide/human/. This file contains the DNA sequences of all genes in the human genome.

The *GRCh38_latest_rna.fna* is quite large and may take up to 30 minutes to download. If you download this from the `/home/oucspdnta/current/test_data/Project_3_Tests/` folder on Schooner, please login using the data transfer node *dtn2.oscer.ou.edu* instead of *schooner.oscer.ou.edu*. Other smaller files are also provided in the `Project_3_Tests/` folder for you here in order for you to test your code quickly.

Below are the first 3 lines in each file provided:

```
>NM_000014.6 Homo sapiens alpha-2-macroglobulin (A2M), transcript variant 1, mRNA
GGGACCAGATGGATTGTAGGGAGTAGGGTACAATACAGTCTGTTCTCCTCCAGCTCCTTCTTTCTGCAACATGGGGAAGA
ACAAACTCCTTCATCCAAGTCTGGTTCTTCTCCTCTTGGTCCTCCTGCCCACAGACGCCTCAGTCTCTGGAAAACCGCAG
```

The entry for a gene starts with a line that contains the name and a description of the gene. This line starts with a “>” character. After this, there are many lines containing the DNA sequence of this gene. This DNA sequence contains only 4 types of characters, A, G, T and C.

It is very useful to compute the tetranucleotide frequencies (TF) of a gene. A tetranucleotide is a DNA sequence window of length 4. There are a total of 256 (4^4) tetranucleotides. Each tetranucleotide can be converted to an integer index between 0 and 255 by assuming A=0, C=1, G=2, and T=3 in a quaternary numeral system.

The TF of a gene can be stored and retrieved using an integer array of length 256. A gene’s TFs can be counted by a moving window of size 4 over each DNA character. The index for a tetranucleotide may be calculated by computing:

```
idx = Window[0]*64 + Window[1]*16 + Window[2]*4 + Window[3]
```

Window[i] returns A, C, G or T, which can be converted to 0, 1, 2 or 3, respectively. For example, the index for the tetranucleotide of “TGGA” can be computed as:

```
Idx(“TGGA”) = Window[0]*64 + Window[1]*16 + Window[2]*4 + Window[3]
            = ‘T’*64 + ‘G’*16 + ‘G’*4 + ‘A’
            = 3*64 + 2*16 + 2*4 + 0
            = 232
```

The pseudocode below shows you how to compute the TFs from a gene:

```
Input: A DNA sequence of length N for a gene
Output: The TF of this gene, which is an integer array of length
256

For each i between 0 and N-4:
     Get the substring from i to i+3 in the DNA sequence
     This substring is a tetranucleotide
     Convert this tetranucleotide to its array index, idx
     TF[idx]++
```

Please parallelize the computation of the average tetranucleotide frequencies of all the genes in the human genome provided in GRCh38_latest_rna.fna. The average TF of the human genome should be saved as an array by the index order of the tetranucleotides (i.e., the first number is the average frequency of the tetranucleotides "AAAA" and the last is for the tetranucleotides "CCCC"). Below is the pseudo-code.

```
Input: A list of DNA sequences for all genes in a genome
Output: The average TF of all input genes, which is a double
array of length 256

For each gene in the list:
     Compute this gene's TF
     Add this gene's TF to the running total TF

Compute the average TF from the running total TF
```

Your program should be run using the following command:

```
compute_average_TF GRCh38_latest_rna.fna average_TF.csv time.csv
num_threads
```

The input parameters of your programs include:

- `GRCh38_latest_rna.fna:` input DNA sequence
- `average_TF:` average TF of all genes in the human genome
- `time.csv:` output CSV file to store the runtime information.
- `num_threads:` number of threads

You are provided with reference code, starter code, a make file to compile the starter code, and a sbatch script to run the executable on the test case with 8 threads on Schooner. If you run make Exp1 or make Exp2, you compile one at a time. If you want to compile all, use make. The expected output is also provided for you to check your solution. All the benchmarking experiments below should be performed using 8 threads on the provided human genome file. The Python file: *2_algorithm.py* contains the algorithm needed to solve this problem. Please reference that when you implement your C program. You can print out the data structures used in the C starter code and the Python script and compare them with the provided input and output to understand the algorithms. The provided Makefile file only works with those files, please update the Makefile for the autograder and the submission.

Experiment 1:
Please use the default(none) in your variable scope clause and use the default schedule. When multiple threads increment the running total TF with their genes' TF, there will be a race condition. Mutual exclusion is needed to avoid the race condition. You learned three mutual exclusion techniques: critical directive, atomic directive, and an array of locks. Please try each mutual exclusion technique and time their parallel runtime using 8 threads. Report the comparison result and explain why the three mutual exclusion techniques generated the performance that you observed here.

Experiment 2:
Please select the quickest algorithm from Experiment 1 and try to speed it up by improving its load balancing. When you work on the Experiment 1, please just use the default schedule. You learned three schedules: static, dynamic and guided. Please try each schedule and time their parallel performance. Report the benchmarking results and explain the results of the three schedules.

<u>Learning outcome:</u>

This problem is designed for you to learn mutual exclusion and load balancing. You will also gain some experience in optimizing the performance of a parallel algorithm. It involves a lot of iterative improvements by experimentation and benchmarking. You should not aim to develop a parallel algorithm with all the bells and whistles in the first try. Instead, you should develop a basic parallel algorithm that works and then try to incrementally improve it using the real-world tests as the guide.

What to submit:

In the report, please describe and discuss your results from Experiments 1 and 2.

In the ZIP file, submit your code and makefile:

- Problem_2/
  - `compute_average_TF_Exp1_critical.c`
  - `compute_average_TF_Exp1_atomic.c`
  - `compute_average_TF_Exp1_locks.c`
  - `compute_average_TF_Exp2_schedule.c`
  - `Makefile`

Problem 3. [30 points for CS4473 and CS5473]

In the previous problem, we have developed a parallel algorithm to compute the average TF of the human genome. Computing the average TF is straightforward to parallelize, because we just need to update a running total TF in the for loop and it is trivial to compute the average from the running total TF. In this problem, please compute the **median** TF. The provided Makefile file only works with those files, please update the Makefile for the autograder and the submission.

```
Input: A list of DNA sequences for all genes in a genome
Output: The median TF of all input genes, which is a double
array of length 256

For each gene in the list:
     Compute this gene's TF
     Save it to an 2D array containing rows for genes and
     columns for tetranucleotides

For each tetranucleotide in the 2D array:
     Sort the frequencies of this tetranucleotide in all genes.
     Find the median frequency
```

Your program should be run using the following command:

```
compute_median_TF GRCh38_reduced_rna.fna median_TF.csv time.csv
num_threads
```

The input parameters of your programs include:

- `GRCh38_reduced_rna.fna:` input DNA sequence
- `median_TF:` median TF of all genes in the human genome
- `time.csv:` output CSV file to store the runtime information.
- `num_threads:` number of threads

You are provided with reference code, starter code, a make file to compile the starter code, and a sbatch script to run the executable on the test case with 1, 2, 4 and 8 threads on Schooner. The expected output is also provided for you to check your solution. Please benchmark the scalability of your parallelization, 4 and 8 threads. The Python file: *3_algorithm.py* contains the algorithm needed to solve this problem. Please reference that when you implement your C program.

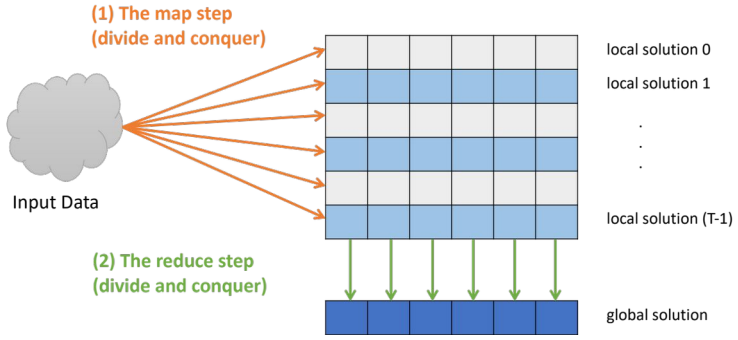
Note that you **should not** use `GRCh38_latest_rna.fna` for this problem. The expected time will be much too great for the amount of data. Please use a reduced input file,

`GRCh38_reduced_rna.fna`, for this problem. With this reduced file, it should take around 10 minutes to run serially.

Experiment 1:
The pseudo-code contains two for loops. Please just parallelize the first for loop and run the second for loop serially. Please benchmark the runtime of this baseline algorithm. In the report, please report the runtime, speedup and efficiency using 1, 2, 4, and 8 threads and discuss the results and the parallel design used here.

Experiment 2:
To improve the parallel scalability, please parallelize both the first for loop and the second for loop in the pseudo-code. You need to use work sharing constructs so that both for loops can be executed by the same team of threads. Please benchmark the runtime of this improved algorithm. In the report, please report the runtime, speedup and efficiency using 1, 2, 4, and 8 threads and discuss the results and the parallel design used here.

Learning outcome:



This problem is designed for you to learn the map and reduce pattern. Please recognize the map step and the reduce step used here.
In our parallel algorithm for computing the average TF in Problem 2, we have essentially performed the reduce step serially using the embarrassingly parallel pattern. This is fine, because the reduce step for computing the average is computationally too trivial to parallelize.

However, in this problem, the computation of the medians for all tetranucleotides is computationally expensive, so we should parallelize it. In the map reduce pattern, we exploit the parallelization opportunity presented by the fact that the reduce step is performed independently for many elements in an array of intermediate result. The map reduce pattern won't be applied if the immediate results cannot be represented in an 2D array that enables parallel reduction column-wise. In the next module, we will learn a tree reduction pattern as a different parallel reduction approach.

What to submit:

In the report, please report and discuss the results from Experiment 1 and Experiment 2.

In the ZIP file, submit your code and makefile:

- Problem_3/
    - `compute_median_TF_Exp1_baseline.c`
    - `compute_median_TF_Exp2_mapreduce.c`
    - `Makefile`

Problem 4. [Not required for CS4473] and [20 points for CS5473]

Please learn the k-means clustering algorithm from https://en.wikipedia.org/wiki/K-means_clustering. Here, you need to parallelize a k-means clustering algorithm to cluster a large number of points in a two-dimensional space into K clusters. The distance between a centroid and a point is measured by their Euclidean distance.

Below is a pseudo-code for a serial implementation of the k-means clustering.

```
Input: A list of data points
Output: K centroids

Initialize the K centroids at the provided initial coordinates.

While not converged (i.e. the average moving distance of all the centroids in the last iteration is more than 1.0)
     1) For every point, assign it to its nearest centroid (for loop 1)
     2) For every centroid, compute its new location as the geometric center of its assigned data points and then compute the moving distance by which it is moved from the previous location (for loop 2)
     3) Compute the average moving distance by which all the centroids are moved in this iteration.
```

The k-means clustering is considered to be converged if the average Euclidean distance by which the centroids are moved in an iteration is less than 1.0 (i.e. the update did not move the centroids by an average distance more than 1.0).

You are provided with the following input files:

- Three different input files: points_1M.csv, points_4M.csv and points_16M.csv, with 1 million, 4 million, and 16 million points respectively. All the points are provided in a table of two columns; first column for the x-coordinates and the second column for the y-coordinates.
- The initial coordinates of the K centroids in the initial_centroids.csv file. This minimizes the randomness of the results for auto-grading. In this exercise, the K is fixed at 16 centroids

You are also provided with the following files:

- A Make file for compiling all the required source codes. The flag -lm is necessary because of library math. You may use it for function pow
- An example sbatch file for submitting a single benchmarking job on Schooner
- A starter code that contains all the I/O and program parameters

Your program should be run as:

```
kmeans_parallel n_points points.csv n_centroids
initial_centroid_values.csv final_centroid_values.csv time.csv
num_threads
```

The input parameters of your programs include:

- `n_points`: the number of points (1000000, 4000000 or 16000000)
- `points.csv`: The csv files with the points
- `n_centroids`: number of centroids
- `initial_centroid_values.csv`: csv file with initial centroids
- `final_centroid_values.csv`: After applying the algorithm, the position of the centroids in csv format
- `time.csv`: time that took to run the parallelization part
- `num_threads`: number of threads used

First, please implement a serial k-means clustering algorithms following the serial pseudo-code. To prevent anyone from simply modifying some k-means clustering C programs online, we expect all students to follow closely the logic outlined in the pseudo-code.

Second, please parallelize the two for loops as shown in the pseudo-code. Please follow the coding requirements below:

- In all the OpenMP parallel directives, please specify the variable scope clauses [*i.e.*, default(none), shared(), private(), and reduction()]to explicitly declare the scope of every variable. The parallelization overhead can be used by using private() to create private copies of variables for every thread, instead of declaring private variables inside the parallel regions.
- Please use work sharing constructs to minimize the parallelization overhead of repeatedly forking and joining threads. You should use the omp parallel directive before the while loop and then use the omp for directives before the two for loops. You may also need to use other work sharing directives, such as single and barrier.

You should optimize your parallel algorithm and improve its scalability by trying the following:

- Please try using the mutual exclusion or the map reduce pattern.

- Please optimize the OpenMP schedule clause to improve the load balancing.
- Please consider false sharing when updating different elements in an array.

Finally, please benchmark the parallelization performance of your code using 1 thread, 4 threads, and 16 threads on the three different input files with 1 million, 4 million, and 16 million data points. And make a 3X3 speedup table and a 3X3 efficiency table for each parallelization implementation.

Learning outcome:

This is a non-trivial algorithm for you to practice all the OpenMP parallelization techniques that you learned in this module.

What to submit:

In the report, please report and discuss all the optimizations that you have done in the second step above and report the benchmarking results.

In the ZIP file, submit your final parallel code and makefile:

- Problem_4/
    - `kmeans_clustering.c`
    - `Makefile`