Project 2: Embarrassingly parallel pattern in OpenMP (100 points)

Submission guidelines:

Please submit **2 files** in Canvas.

- a PDF report file (filename: Project_2_Report.pdf)
- a ZIP file (filename: LastnameFirstname_Project2.zip) containing:
    - Problem_1/
        - `parallel_mult_mat_mat.c`
        - `Makefile`
    - Problem_2A/
        - `parallel_mult_max.c`
        - `Makefile`
    - Problem_2B/
        - `parallel_mult_second_largest.c`
        - `Makefile`
    - Problem_3/
        - `encrypt_parallel.c`
        - `Makefile`
    - Problem_4/
        - `decrypt_parallel.c`
        - `Makefile`

Notes: We will use auto-grading scripts to test your code and check the outputs from your code. So, it is important for you to follow the instructions, including the names of the directories and files. Up to 10 points will be deducted for non-compliance with the code specifications or submission guidelines no matter whether the solution is correct or not. A rubric is available in Canvas under the assignment.

Test data:
The test data are provided on Schooner at `”/home/oucspdnta/PDN2022/test_data/Project_2_Tests/”`. You can copy the test data into your home directory using the following command:
`“cp -r /home/oucspdnta/PDN2022/test_data/Project_2_Tests/ .”`

Project learning outcomes:
The programming problems in this project are designed for you to practice the embarrassingly parallel pattern in OpenMP. This type of problems is easy to parallelize, because the algorithm just needs to divide the problem up, then solve each sub-problem independently using a thread, and finally simply combine the local solutions from every thread into a global solution. It is straightforward to achieve good scalability for the embarrassingly parallel problems.

Problem 1. [40 points for CS4473 and CS5473]

Please implement a multi-threaded matrix-matrix multiplication algorithm using OpenMP. Your implementation needs to follow the pseudo-code below. You also need to re-use the matrix-vector multiplication code that you implemented in Project_1 inside the for loop. All the numbers in the matrices are integers and please use `long int` as the variable type.

```
Input: matrix A, matrix B
Output: matrix C = AB

for each column vector in matrix B do
     multiply matrix A with this column vector
     save the resultant column vector to matrix C
```

Your program should be run using the following command:

```
parallel_mult_mat_mat file_A.csv n_row_A n_col_A file_B.csv n_ro
w_B n_col_B num_threads result_matrix.csv time.csv
```

The input parameters of your programs include:

- `file_A.csv:` input CSV file for the input matrix A
- `n_row_A:` number of rows in the input matrix A
- `n_col_A:` number of columns in the input matrix A
- `file_B.csv:` input CSV file for the input vector B
- `n_row_B:` number of rows in the input vector B
- `n_col_B:` number of columns in the input matrix B
- `result_vector.csv:` output CSV file for the result vector
- `time.csv:` output CSV file to store the runtime information.

You are provided with a 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.

Please benchmark the wall-clock runtime of your program using 1, 2, 4, and 8 threads on the following four tests with increasing sizes of the two input matrices:

Test	**Matrices**	**Rows**	**Columns**
Test 1	A	1000	1000
	B	1000	1000
Test 2	A	1000	1000
	B	1000	2000
Test 3	A	2000	1000
	B	1000	2000
Test 4	A	2000	2000
	B	2000	2000

About using Schooner:
After submitting a job, your job may need to wait in the queue for its turn to run on Schooner. Depending on how busy Schooner is, you may have to wait for some time to get the results. This may cause delay when you debug your code. So, please plan ahead. Please re-visit the Problem 1 in Project 1 for a list of tutorials about using Schooner.

Learning outcome:
This problem requires the joining of the local solutions by concatenation in the embarrassingly parallel pattern. The threads can independently compute the different columns and then place them into the output matrix. The only coordination needed among the threads are the distribution of the columns based on the ranks of the threads.

What to submit:
In the PDF report, please report the wall-clock runtime of your parallel program on the four tests using 1, 2, 4, and 8 threads and make a speedup table and an efficiency table. Please comment on the scalability of your parallel code (strongly scalable vs weakly scalable). If you fail to write a correct and scalable algorithm before the deadline, please explain the likely causes in the report and you will receive points for the error analysis.

Runtime	Test 1	Test 2	Test 3	Test 4
p = 1				
p = 2				
p = 4				
p = 8				

Speedup	Test 1	Test 2	Test 3	Test 4
p = 1				
p = 2				
p = 4				
p = 8				

Efficiency	Test 1	Test 2	Test 3	Test 4
p = 1				
p = 2				
p = 4				
p = 8				

In the ZIP file, submit your code and makefile:

- Problem_1/
    - `parallel_mult_mat_mat.c`
    - `Makefile`

Problem 2. [20 points for CS4473 and CS5473]

In this problem, let us practice the join by reduction in the embarrassingly parallel pattern using the matrix-matrix multiplication result.

Problem 2A. [10 points for CS4473 and CS5473]

Please implement a parallel program to find the maximum value in the output matrix from a matrix-matrix multiplication. To save memory, you cannot compute and store the entire output matrix in the memory and then search for the maximum from the completed output matrix. Instead, you need to find the maximum as each element in the output matrix is computed as shown in the pseudo-code below:

```
Input: matrix A, matrix B
Output: the largest element in the matrix AxB

maximum = 0
for each column vector in matrix B do
     for each row vector in matrix A do
          compute x = the dot product of the two vectors
          if (x > maximum)
               maximum = x

save the maximum to the output file
```

Please distribute the outer for loop across multiple threads. The challenge is how to join the local maximums found by individual threads into a global maximum.

Your program should be run using the following command:

```
parallel_mult_max file_A.csv n_row_A n_col_A file_B.csv n_row_B
n_col_B num_threads result_maximum.csv time.csv
```

Please note the program name is changed to `parallel_mult_max`. The output filename is changed to, `result_maximum.csv`, which saves the maximum found by this program. You may re-use the starter code, the sbatch script, and the tests provided in Problem 1.

<u>Learning outcome:</u>
Maximum is a reduction operator supported by the OpenMP reduction, which makes the join by reduction straightforward to implement. You may look at how the reduction clause is used in the parallel trapezoidal rule algorithm and learn how to apply it here.

What to submit:
In the ZIP file, please submit your code and makefile:

- Problem_2A/
    - `parallel_mult_max.c`
    - `Makefile`

Problem 2B. [10 points for CS4473 and CS5473]

Please implement a parallel program to find the second largest value in the output matrix from a matrix-matrix multiplication. Similar to the problem step in Problem 2A, you cannot compute and store the entire output matrix in the memory and then search for the second largest value from the completed output matrix. Instead, you need to find the second largest value as each element in the output matrix is computed as shown in the pseudo-code below:

```
Input: matrix A, matrix B
Output: the second largest element in the matrix AxB

largest = 0
secondLargest = 0
for each column vector in matrix B do
     for each row vector in matrix A do
          compute x = the dot product of the two vectors
          if (x > largest)
               secondLargest = largest
               largest = x
          else if (x > secondLargest)
               secondLargest = x

save the secondLargest to the output file
```

Please distribute the outer for loop across multiple threads. The challenge is how to join the local second largest values found by individual threads into a global second largest value.

Your program should be run using the following command:

```
parallel_mult_second_largest file_A.csv n_row_A n_col_A file_B.c
sv n_row_B n_col_B num_threads result_second_largest.csv
time.csv
```

Please note the program name is changed to `parallel_mult_second_largest`. The output filename is changed to, `result_second_largest.csv`, which saves the second largest value found by this program. You may re-use the starter code, the sbatch script, and the tests provided in Problem 1.

Learning outcome:
The OpenMP reduction clause does not support finding the second largest value. So, you have to implement the reduction of the local second largest values into a global second largest value. Since the reduction here is computationally inexpensive, it can be done serially without losing much efficiency. Basically, in the parallel region, you can compute the local solutions and save them in some shared variables. Then, after the parallel region, you can sequentially compare the local solutions to find the global solution using the single master thread.

The embarrassingly parallel pattern often uses such serial reduction, but serial reduction decreases the parallel efficiency if the reduction requires a lot of computation. In the upcoming chapters, we will learn how to parallelize the reduction using the map reduce pattern and the tree reduction pattern.

What to submit:
In the ZIP file, please submit your code and makefile:

- Problem_2B/
  - `parallel_mult_second_largest.c`
  - `Makefile`

Problem 3. [40 points for CS4473] and [20 points for CS5473]

The problem is to read in a text and encrypt it using Caesar Encryption algorithm (https://en.wikipedia.org/wiki/Caesar_cipher). The encryption algorithm is very simple. It replaces each character in the text by a certain displacement. For example, if we use a displacement of 3, then every English letter in the text will be replaced by a letter which is down three places from it. The alphabet can be assumed to be placed around a circle. So when we come to the end of the alphabet, we restart from the beginning to find the displacement character. So, in our example, "a" will be replaced by "d", "b" will be replaced by "e", "x" will be replaced by "a", "y" will be replaced by "b" and so on. The number 3 is called the encryption key. The decryption will use the same key but this time we will replace the encrypted text alphabet by going back 3 characters. So "a" will be replaced by "x", "b" will be replaced by "b" and so on. One can find the replacement characters using the modular arithmetic. The alphabet can be coded as 0, 1, 2, …, 25. If k is the key, the letter x will be replaced by x + k mod 26. The decryption will similarly be x – k mod 26.

The Caesar algorithm is an example of a block cipher. A block cipher is an encryption algorithm that takes one block of plain text and converts into encrypted text. Here the block size is one character. Many practical algorithms have a larger block. All block encryption algorithms are embarrassingly parallel, because encryption of any block can be done independent of any other block.

If we store each character in an 8-bit byte, we have 256 different characters. (Some are printable, others are not.) So we can assume that our alphabet contains 256 characters. You can use a key that is between 1 and 255.

Your program should

1. Read in a key and the plain text.
2. Distribute the plain text amongst the threads used almost equally.
3. Let each thread encrypt its portion in parallel.
4. Collates the encrypted text from every thread
5. Save the encrypted text to a file.

Your program should be run in this way:

```
encrypt_parallel key input_text.txt num_threads output_text.txt
time.txt
```

For this problem, you are given a working serial code that can be compiled and executed to convert the input text to the output text. Two tests are provided with the following sizes. The output of the tests were obtained with a key of 10.

Tests	Bytes	Bytes
Test 1	17,198,000	~2^{24}
Test 2	34,396,000	~2^{25}

Learning outcome:

This is another example of join by concatenation in the embarrassingly parallel design. Please notice the common pattern among these examples.

What to submit:

In the report, please describe the algorithm used to solve the problem, including how the data was distributed to the processors. The program should be run using 1, 2, 4, and 8 threads on data of 4 different sizes. For each run, you should time the encryption computation. The report should include 4X4 tables for the computation time, speedup, and efficiency for all these runs. Is your program strongly scalable or weakly scalable and why?

In the ZIP file, please submit your code and makefile:

- Problem_3/
  - `encrypt_parallel.c`
  - `Makefile`

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

Please write a multi-threaded parallel algorithm to crack an encrypted English text using brute-force computation. There are 256 possible keys for the Caesar Encryption algorithm. Among all these possible keys, let us assume that the correct key should yield a text that contains the most occurrence of the word "The" or "the", because the word "the" is the most frequent word in English.

It should be run in this way:

```
decrypt_parallel input_text.txt num_threads key.txt time.txt
```

You are provided with a makefile and a test case (key =25).

Learning outcome:

This is another example of join by reduction in the embarrassingly parallel design.

What to submit:

In the report, please show the parallel computation time, speedup, and efficiency of runs using 1, 2, 4 and 8 threads to crack the provided test case.

In the ZIP file, please submit your code and makefile:

- Problem_4/
  - `decrypt_parallel.c`
  - `Makefile`