CS4473/CS5473 : HW2

Submission guidelines:
You will submit **2 files** in Canvas.

- a PDF report file (filename: HW2_Report.pdf)
- a ZIP file (filename: LastNameFirstName_HW2.zip) containing:
  - Problem_2/
    - parallel_mult_mat_mat.c
    - Makefile
  - Problem_3/
    - encrypt_parallel.c
    - Makefile
  - Problem_4/
    - decrypt_parallel.c
    - Makefile

Notes:

A rubric is available in Canvas under the assignment

We will deduct **5% of the points** of a submission if it fails the auto-grading script due to the non-compliance with the code specifications AND submission guidelines and no matter whether the code contains the correct solution or not.

Test data:
The test data files are available at:
https://sooners-my.sharepoint.com/:f:/g/personal/cpan_ou_edu/EjZ5xZsWLqBHiY1S7R3jDdgBxLM43PoxsLXZelRIGEliMw

Additionally, when working with schooner, you can copy the test data files from the folder: `/home/paulcalle/PP21/HW2/test_data` into your working directory.

Point System: U: 4473, G:5473

Problem 1. [CS4473 and CS5473] (U/G:10 points)

Suppose a serial program takes n^2 microseconds to execute, where n is the problem size. Suppose a parallel program with p processors will take $n^2/p + \log_2(p)$ microseconds. How much will we need to increase the problem size if we increase the number of processors from 4 to 16 so that the efficiency is the same in both cases?

What to submit:
You should submit your calculation in the pdf report

PROBLEMS 2~4: LEARNING OUTCOMES:

For the following 3 problems, you will learn the embarrassingly parallel pattern with 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 partial solutions from every thread into a global solution. It is straightforward to achieve linear scalability for the embarrassingly parallel problems

Problem 2. [CS4473 and CS5473] (U:40 /G:30 points)

Please implement a multi-threaded matrix-matrix multiplication algorithm using OpenMP. Your implementation needs to follow the pseudocode below. You also need to re-use the matrix-vector multiplication code that you implemented in HW1 inside the for loop.

```
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
```

It should be run in this way:

```
parallel_mult_mat_mat file_1.csv n_row_1 n_col_1 file_2.csv n_row_2 n_col_2
num_threads result_matrix.csv time.csv
```

Input matrices dimensions:

Test	rows	columns
1a	100	500
1b	500	1000
2a	1000	800
2b	800	2000
3a	2000	1000
3b	1000	3000
4a	4000	2000
4b	2000	5000

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.

You also provided with four tests with an increasing matrix size.

You need to benchmark the wall-clock computing time of your algorithm using 1, 2, 4, and 8 threads on the three tests. You may use a compute node of Schooner or on a VM from GCP.

Note Autograding: the *test*_output_mat.csv* files are for reference; your values can be in a range of ±1%.

Note schooner(OU supercomputer): When submitting a job, you may wait some time for the results, it can be short or long depending on the availability so **plan ahead**.

Note schooner2(OU supercomputer):
In this course, with your class account in schooner, you also have access to two special queues that may have higher priorities and lower wait time than the normal queue and the debug queue:

This queue contains CPU-only nodes, which should be used in the OpenMP module and the MPI module of this course. Your job can request only up to 2 nodes and can run only for 15 minutes:

```
#SBATCH --partition= parallel_programming_cpu
```

This queue contains GPU nodes, which should be used in the GPU module of this course. Your job can request only up to 1 node and can run only for 15 minutes:

```
#SBATCH --partition= parallel_programming_gpu
```

Note schooner3(OU supercomputer):

Partition(queue parameter)	Time limit
parallel_programming_cpu	15 min
parallel_programming_gpu	15 min
debug	30 min
normal	48 hours
… more	

For more partitions information, you can use see https://www.ou.edu/oscer/support/partitions

Note schooner4(OU supercomputer):
To use make command. You may need to use:

```
module load GCC
```

Note schooner5(OU supercomputer): Supplementary information:

- Running jobs in schooner: https://www.ou.edu/oscer/support/running_jobs_schooner
- Upload and Download from schooner: https://www.ou.edu/oscer/support/file_transfer

What to submit:
You should submit your code. In your report, you should report the wall-clock computing time of the matrix multiplication using 1, 2, 4, and 8 threads and make a 4X4 speedup table and a 4X4 efficiency table. Please comment on the scalability of your parallel code (strongly scalable vs weakly scalable).

Problem 3. [CS4473 and CS5473] (U:50/G:40 points)

The problem is to read in a text and encrypt it using Caesar Encryption algorithm. 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.

You are provided with a starter code and four tests. The output of the tests were obtained with a key of 10.

It should be run in this way:

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

Size of test cases

Test Case	Bytes	Bytes
1	17,610,750	~2^{24}
2	35,221,520	~2^{25}
3	70,443,006	~2^{26}
4	140,886,014	~2^{27}

What to submit:
You should submit your code and a report. The report should 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 a table for the computation time, speedup, and efficiency for all these runs. Is your program strongly scalable or weakly scalable and why?

Problem 4. [Only for students in CS5473] (G: 20 points)

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
```

You are provided with make file, and a test case (key =25)

What to submit:
Your brute-force crack program.

AUTOGRADER:
You should have the minimum following structure:

```
.
├── autograding_all.py
├── test_data/
│   ├── Problem_2/
│   │   ├── test1_input_mat_a.csv
│   │   ├── test1_input_mat_b.csv
│   │   ├── test1_output_mat.csv
│   │   ├── test2_input_mat_a.csv
│   │   ├── test2_input_mat_b.csv
│   │   ├── test2_output_mat.csv
│   │   ├── test3_input_mat_a.csv
│   │   ├── test3_input_mat_b.csv
│   │   ├── test3_output_mat.csv
│   │   ├── test4_input_mat_a.csv
│   │   ├── test4_input_mat_b.csv
│   │   └── test4_output_mat.csv
│   ├── Problem_3/
│   │   ├── test1_text_input.txt
│   │   ├── test1_text_output.txt
│   │   ├── test2_text_input.txt
│   │   ├── test2_text_output.txt
│   │   ├── test3_text_input.txt
│   │   ├── test3_text_output.txt
│   │   ├── test4_text_input.txt
│   │   └── test4_text_output.txt
│   └── Problem_4/
│       └── test1_text_to_decrypt_input.txt
└── submissions/
    └── lastnamefirstname_HW2.zip
        ├── Problem_2/
        │   ├── parallel_mult_mat_mat.c
        │   └── Makefile
        ├── Problem_3/
        │   ├── encrypt_parallel.c
        │   └── Makefile
        └── Problem_4/
            ├── decrypt_parallel.c
            └── Makefile
```

To run use: **python3 autograding_all.py**

You need Python 3. There may be some errors if you haven't installed the required libraries. Then install them.

Output: Two csv files are created:

- Result of test, `grades,csv`, 1: success, 0: failure
- times, `time.csv`.

Additionally, the last lines of stdout will look like this:

```
            P2-T1  P2-T2  P2-T3  P2-T4  P3-T1  P3-T2  P3-T3  P3-T4  P4-T1
walkerjohn    1.0    1.0    1.0    1.0    1.0    1.0    1.0    1.0    1.0
```

Where P is problem T is Test

Running in schooner:

Run the following commands, to install the required libraries:

```
module purge
module load Python/3.8.2-GCCcore-9.3.0
pip install --user numpy
pip install --user pandas
```

A *sample sbatch* `autograder_template.sbatch` file is provided. You may need to use a longer queue.