Project: Parallel Matrix Operations

Due Date: Monday, November 27, 2023 at **8 PM HKT**. Late submissions will be penalized as per syllabus.

I. Project Instructions

Overview

Matrix operations including multiplication or addition are basic mathematical operations that are widely applied in many fields, including scientific computing and pattern recognition. In this project, you will implement a parallel version of matrix operation called `pmo`.

There are three specific objectives for this project:

- To familiarize yourself with the Linux pthreads library for writing multi-threaded programs.
- To learn how to parallelize a program.
- To learn how to program for performance.

Project Organization

Each group should do the following pieces to complete the project. Each piece is explained below:

- **Design** **30 points**
- **Implementation** **30 points**
- **Evaluation** **40 points**

You're required to submit a project report which concisely describes your design, implementation, and experimental results.

Design

There are plenty of scenarios that work with parallel matrix operations. In this project, we narrow the working scenario and only pay attention to calculating the result of an expression of matrix operations (less than 10 operations). A practical calculator that achieves a better performance will require you to address (at least) the following issues (see part *II. Project Description* for more details):

- How to parallelly perform the matrix operations.
- How to parallelly calculate the expression.

In your project report, please describe the detailed techniques or mechanisms proposed to parallelize the matrix operations. List and describe all the functions used in this project.

Implementation

Your code should be nicely formatted with plenty of comments. Each function should be preceded by a header comment that describes what the function does. The code should be easy to read, properly indented, employ good naming standards, have good structure, and should correctly implement the design. Your code should match your design. Extern libraries of matrix multiplication are **NOT** allowed to be used in this project.

Evaluation

We provide 10 test cases for you to test code. Before submitting your work, please run your `pmo` on the test cases and check if your `pmo` outputs the correct results. A time limitation is set for each test case, and if this limit is exceeded, your test will fail. For each test case, your grade will be calculated as follows:

- ***Correctness***. Your code will first be measured for correctness, ensuring that it outputs the correct results. You will receive full points if your solution passes the correctness tests performed by the test script. You will receive **zero points** for this test case if your code is buggy.
- ***Performance***. If you pass the correctness tests, your code will be tested for performance; The test script will record the running time of your program for performance evaluation. Shorter time/higher performance will lead to better scores.

Your project report needs to summarize and analyze the results. You can also compare your solution with the provided baseline implementation.

Tips: Keep a log of the work you have done. You may wish to list optimizations you tried, what failed, etc. Keeping a good log will make it easy to put together your final write-up.

Bonus

You're encouraged to be creative and innovative, and this project awards bonus points (**up to 10 points**) for additional and/or exemplary work.

- New ideas/designs are welcome to fully explore the parallelism of matrix multiplication. Please comprehensively illustrate your algorithms in your report.
- To encourage healthy competition and desire to improve, we will provide a daily update scoreboard to show scores and running time for each group. Additional larger test cases will be used to test your solution for scalability.

Further details will be posted on Canvas soon.

Language/Platform

The project should be written in ANSI standard C.
This project can be done on Linux (recommended), MacOS, or Windows using Cygwin. Since grading of this project will be done using the gateway Linux server, students who choose to develop their code on any other machine are strongly encouraged to run their programs on the gateway Linux server before turning it in. There will be no points for programs that do not compile and run on the gateway Linux server, even if they run somewhere else.

Handing In

The project can be done individually, in pairs, or in groups, where each group can have a maximum of three members. **All students are required to join one project group in Canvas**: "People" section > "Project Groups" tab > Project 1–Project 50. Contact TA to add additional groups if necessary. Self sign-up is enabled for these groups. Instead of all students having to submit a solution to the project, Canvas allows **one person** from each group to submit on behalf of their team. If you work with partner(s), both you and your partner(s) will receive the **same grade** for the entire project **unless** you explicitly specify each team member's contribution in your report. Please be sure to indicate who is in your group when submitting the project report.

Before you hand in, make sure to add the requested identifying information about your project group, which contains the project group number, full name and e-mail address of each group member.

When you're ready to hand in your solution, go to the course site in Canvas, choose the "Assignments" section > "Project" group > "Project" item > "Start Assignment" button and upload your files, including the following:

1) A PDF document which concisely describes your design, implementation, and experimental results; If you are working in a team, please also describe each team member's contribution.
2) The source file, i.e., pmo.c;

Academic Honesty

All work must be developed by each group separately. Please write your own code. **All submitted source codes will be scanned by anti-plagiarism software**. If the code does not work, please indicate in the report clearly.

Questions?

If you have questions, please first post them on Canvas so others can get the benefit of the TA's answer. Avoid posting code that will give away your solution or allow others to cheat. If this does not resolve your issue, contact the TA (Mr. WU Shangyu<shangyuwu2-c@my.cityu.edu.hk>).

Acknowledgements

This project is modified from the OSTEP book written by Remzi and Andrea Arpaci-Dusseau at the University of Wisconsin. This free OS book is available at http://www.ostep.org. Automated testing scripts are from Kyle C. Hale at the Illinois Institute of Technology.

Disclaimer

The information in this document is subject to change **with** notice via Canvas. Be sure to download the latest version from Canvas.

II. Project Description

For this project, you will implement a parallel version of matrix operations using threads. First, you will recall how to perform the basic matrix operations and how to calculate. Then, you will be given some basic ideas to design your own parallel version of matrix operations.

Introduction

Matrix Addition/Subtraction

For the matrix addition or subtraction, you will be given two $n \times m$ matrixes, the matrix A and the matrix B,

$$A = \begin{pmatrix} a_{11} & \cdots & a_{1m} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nm} \end{pmatrix}, B = \begin{pmatrix} b_{11} & \cdots & b_{1m} \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{nm} \end{pmatrix}$$

The matrix addition or subtraction is $C = A \pm B$ is defined to be a $n \times m$ matrix,

$$C = \begin{pmatrix} c_{11} & \cdots & c_{1m} \\ \vdots & \ddots & \vdots \\ c_{n1} & \cdots & c_{nm} \end{pmatrix}$$

Such that

$$c_{ij} = a_{ij} \pm b_{ij}$$

for $i = 1, \dots, n$ and $j = 1, \dots, m$.

Matrix Multiplication

For the matrix addition or subtraction, you will be given two matrixes, a $n \times p$ matrix A and a $p \times m$ matrix B,

$$A = \begin{pmatrix} a_{11} & \cdots & a_{1p} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{np} \end{pmatrix}, B = \begin{pmatrix} b_{11} & \cdots & b_{1m} \\ \vdots & \ddots & \vdots \\ b_{p1} & \cdots & b_{pm} \end{pmatrix}$$

The matrix product $C = A \cdot B$ is defined to be a $n \times m$ matrix.

$$C = \begin{pmatrix} c_{11} & \cdots & c_{1m} \\ \vdots & \ddots & \vdots \\ c_{n1} & \cdots & c_{nm} \end{pmatrix}$$

Such that

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{ip}b_{pj} = \sum_{k=1}^{p} a_{ip}b_{pj}$$

for $i = 1, \ldots, n$ and $j = 1, \ldots, m$.

Expression of Matrix Operations

An expression of matrix operations is a combination of a series of matrix operations, such as $A + B \cdot C + D$. Similar to the arithmetic operations, there is an order between matrix operations, the matrix multiplications have higher priority than the matrix additions/subtractions. In the above example, $B \cdot C$ is first calculated, then the matrix additions are calculated.

The basic calculation of an expression can be done in two steps. Since there are only two types of operator priorities, i.e., multiplication vs. addition/subtraction, you can first get all multiplications in the expression, and parallelly calculate their results. Then, you can calculate the results of additions/subtractions based on the multiplication results.

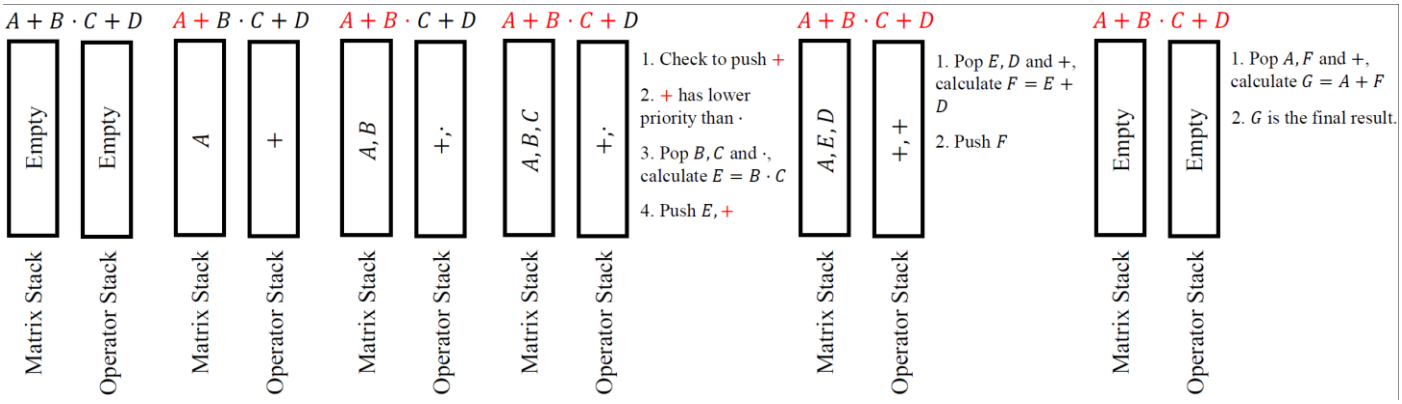

Figure 1. An example of calculating an expression.

Another advanced calculation of an expression can be implemented by two stacks, i.e., an operator stack and a matrix stack. Using the same example of $A + B \cdot C + D$ shown in Figure 1, initially, two stacks are empty. Calculating the expression from left to right, we first push the first matrix A into the matrix stack and the first operator into the operator stack. Then, we add the next matrix into the matrix stack, where A, B are in the matrix stack. Before pushing the operator, we compare the pushing operator with the operator on the top of the operator stack. If the pushing operator has higher priority than the top operator, we push the operator. Otherwise, we need to do the matrix multiplications first. After pushing C into the matrix stack and $\cdot$ into the operator stack, we compare $+$ with the top operator $\cdot$. Since $+$ has lower priority, we need to do $B \cdot C$ then push the result into the matrix stack. Finally, we repeat popping an operator from the operator stack, and two matrixes from the matrix stack until there is only one matrix, which is the final result.

Ideas

The matrix operations are quite friendly to parallel programming. In this project, we provide you several possible simple directions to explore the parallelism of matrix multiplication.

- Row-wise/Column-wise 1D Partition.

The first direction is to split the matrix into sub-matrixes on the rows/columns. Taking row-wise 1D partition on matrix multiplication as an example, the matrix A is a 4×2 matrix, and the matrix B is a 2×2 matrix, so the matrix product C is a 4×2 matrix.

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \\ a_{41} & a_{42} \end{pmatrix} \times \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \\ c_{31} & c_{32} \\ c_{41} & c_{42} \end{pmatrix}$$

To improve the parallelism, the matrix A can be split into two sub-matrixes on rows, called A_1 and A_2.

$$A = \begin{pmatrix} A_1 \\ A_2 \end{pmatrix}, A_1 = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}, A_2 = \begin{pmatrix} a_{31} & a_{32} \\ a_{41} & a_{42} \end{pmatrix}$$

Thus, the matrix product C can be computed parallelly by simultaneously performing two sub-matrixes multiplication,

$$C = \begin{pmatrix} A_1 B \\ A_2 B \end{pmatrix}$$

Each sub-matrix multiplication can be computed in a separate thread, i.e., computing $A_1 B$ in thread 1 and computing $A_2 B$ in thread 2. To fully improve the computation efficiency, you can split rows on the matrix A and split columns on the matrix B.

Similar partition also can be applied to the matrix additions/subtractions.

- 2D Partition.

The second direction is to split the matrix both on rows and columns. For example, the matrix A is a 4×4 matrix and the matrix B is also a 4×4 matrix, so the matrix product C is a 4×4 matrix.

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \times \begin{pmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ c_{31} & c_{32} & c_{33} & c_{34} \\ c_{41} & c_{42} & c_{43} & c_{44} \end{pmatrix}$$

The matrix A can be split into 4 blocks, i.e.,

$$A = \begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix}$$

$$A_1 = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}, A_2 = \begin{pmatrix} a_{13} & a_{14} \\ a_{23} & a_{24} \end{pmatrix}, A_3 = \begin{pmatrix} a_{31} & a_{32} \\ a_{41} & a_{42} \end{pmatrix}, A_4 = \begin{pmatrix} a_{33} & a_{34} \\ a_{43} & a_{44} \end{pmatrix}$$

The same partition can be performed on the matrix B.

$$B = \begin{pmatrix} B_1 & B_2 \\ B_3 & B_4 \end{pmatrix}$$

Thus, the matrix product C can be computed by the following steps.

$$C = \begin{pmatrix} C_1 & C_2 \\ C_3 & C_4 \end{pmatrix} = \begin{pmatrix} A_1B_1 + A_2B_3 & A_1B_2 + A_2B_4 \\ A_3B_1 + A_4B_3 & A_3B_2 + A_4B_4 \end{pmatrix}$$

Each A_iB_j is the sub-matrix product and can be computed simultaneously. Finally, the sub-matrix products need to be merged as the above. If you directly store the sub-matrix product in the corresponding positions in the matrix C, you need to be aware of the potential data consistency issue.

Similar partition also can be applied to the matrix additions/subtractions.

- Parallel calculation of expression.

  In the expression that contains multiple matrix operations, you can use threads to perform the matrix operations that has the same priority concurrently. For example, to calculate a more complex expression $A + B + C \cdot D \cdot E + F \cdot G + H$. We can split this expression into four parts, i.e., $A + B$, $C \cdot D \cdot E$, $F \cdot G$, H. Each part can be calculated independently with one thread. Finally, we can use a thread to merge all results. This process can be implemented by a producer-consumer model, where the producer splits the expression and consumers do the calculations of each part. After all consumers finish its calculations, a thread can be used to merge all results.

Inputs/Outputs

In this project, your program will read a line of string corresponding to the expression. Then you will read several matrixes. For each matrix, your program will read 2 integers, `n, m` which corresponds to the number of rows of matrix, the number of columns of matrix. Then, your program will read `n` lines, where each line contains `m` integers separated by a tab character. The number of matrixes is determined by the expression.

Your program is required to outputs the matrix C. The outputs don't need to be written into a file. Each line outputs a row of the matrix C separated by a 'tab' character.

The evaluation adopts pipes in Linux to input data, so you don't need to handle the file processing. The max number of matrixes in the expression will **NOT** exceed 10. The number of rows/columns would **NOT** exceed 1000 (including additional large test cases).

Challenges

Doing so effectively and with high performance will require you to address (at least) the following issues:

- **How to parallelize the matrix operations.** Parallelizing the matrix operations requires you to think about how to partition the matrix operations, so that the computing resources can be fully used. Following the mentioned first two ideas, you can design your own partition schemes. The input matrix varies from different size, different dense/sparse degree. Your schemes should take all those factors into consideration.
- **How to split the expression for parallel calculation.** The matrix operations in the expression can be calculated independently as mentioned in the last ideas. The splitting scheme determines the amount of

computations of each thread. However, unbalanced splitting cannot make full use of all threads. Your schemes should be a balance splitting scheme. Besides, on Linux, the determination of the number of threads may refer to some interfaces like `get_nprocs()` and `get_nprocs_conf()`; You are suggested to read the man pages for more details.

To understand how to make tackle these problems, you should first understand the basics of thread creation, and perhaps locking and signaling via mutex locks and condition variables. Review the provided materials and read the following chapters from OSTEP book carefully in order to prepare yourself for this project.

- Intro to Threads
- Threads API
- Locks
- Using Locks
- Condition Variables

III. Project Guidelines

1. Getting Started

The project is to be done on the CSLab SSH gateway server, to which you should already be able to log in. As before, follow the same copy procedure as you did in the previous tutorials to get the project files (code and test files). They are available in */public/cs3103/project/* on the gateway server. `project.zip` contains the following files/directories:

```
/project
├── Makefile
├── pmo_std            <- A sample solution (executable file).
├── pmo.c              <- Modify and hand in pmo.c file.
├── README.md
└── tests
    ├── bin
    │   ├── gen_data.sh
    │   ├── generator.py
    │   ├── generic-tester.py
    │   └── test-pmo.sh
    └── config
        ├── 1.json
        ├── ...
        └── 10.json
```

Start by copying the provided files to a directory in which you plan to do your work. For example, copy */public/cs3103/project/project.zip* to your home directory, extract the files from the ZIP file with the `unzip` command. After the unzip command extracts all files to the current directory, change to the project directory and take a look at the directory contents:

```
$ cd ~
$ cp /public/cs3103/project/project.zip .
$ unzip project.zip
$ cd project
$ ls
```

You need to run the following commands to generate input data and standard outputs.

```
$ make gen_data
$ make gen_ans
```

A sample `pmo_std` is also provided (we only provide a single executable file without source code). This `pmo` uses pthread to support the parallel matrix multiplication. The `pmo_std` is a naïve baseline scheme that uses one thread to split the expression and another thread to calculate the results of sub-expression.

You can run and test `pmo_std` by using the `make test_std` command.

```
$ make test_std
TEST 1 - small matrix addition (smaller than 100 * 100, 1 sec timeout)
Test finished in 0.009 seconds
RESULT passed
(content removed for brevity)
TEST 10 - complex matrix operations 3 (smaller than 1000 * 1000, 10 sec timeout)
Test finished in 4.679 seconds
RESULT passed
```

You can also regard this one as a baseline implementation for performance evaluation, which means you can compare the execution time of your `pmo` with that of the provided one in final report. Note that after building your own `pmo` (using `make` or `make test`), a `pmo` file will be created.

2. Writing your `pmo` program

The pmo.c is the file that you will be handing in and is the only file you should modify. Write your code from scratch to implement this parallel version of matrix multiplication. Again, it's a good chance to learn (as a side effect) how to use a proper code editor such as `vscode`[1,2].

3. Building your program

A simple makefile that describes how to compile `pmo` is provided for you.

To compile your pmo.c and to generate the executable file, use the `make` command within the directory that contains your project. It will display the command used to compile the `pmo`.

```
$ make
gcc -Wall -pthread -g -O3    pmo.c   -o pmo
```

If everything goes well, there would an executable file `pmo` in it:

```
$ ls
Makefile  README.md  pmo  pmo_std pmo.c  tests
```

If you make some changes in pmo.c later, you should re-compile the project by running `make` command again.

To remove any files generated by the last `make`, use the `make clean` command.

```
$ make clean
rm -f pmo
$ ls
```

[1] Visual Studio Code, https://code.visualstudio.com/

[2] C/C++ for Visual Studio Code, https://code.visualstudio.com/docs/languages/cpp

```
Makefile  README.md  pmo_std pmo.c  tests
```

4. Testing your C program

We also provide 10 test cases for you to test your code. You can find them in the directory tests/stdin/. The makefile could also trigger automated testing scripts, type `make run` (run testing only) or `make test` (build your program and run testing). The test cases would be different during our final evaluation.

```
$ make test
TEST 0 - clean build (program should compile without errors or warnings)
Test finished in 0.183 seconds
RESULT passed

TEST 1 - small matrix addition (smaller than 100 * 100, 1 sec timeout)
Test finished in 0.009 seconds
RESULT passed

TEST 2 - small matrix multiplication (smaller than 100 * 100, 1 sec
timeout)
Test finished in 0.008 seconds
RESULT passed

TEST 3 - small matrix subtraction (smaller than 100 * 100, 1 sec timeout)
Test finished in 0.007 seconds
RESULT passed

TEST 4 - matrix additions/subtractions (smaller than 500 * 500, 1 sec
timeout)
Test finished in 0.087 seconds
RESULT passed

TEST 5 - consecutive matrix multiplications (smaller than 500 * 500, 1 sec
timeout)
Test finished in 0.437 seconds
RESULT passed

TEST 6 - combination of matrix additions and matrix multiplications
(smaller than 500 * 500, 1 sec timeout)
Test finished in 0.202 seconds
RESULT passed
```

```
TEST 7 - combination of matrix additions and matrix multiplications 2
(smaller than 500 * 500, 1 sec timeout)
Test finished in 0.272 seconds
RESULT passed

TEST 8 - complex matrix operations (smaller than 1000 * 1000, 10 sec
timeout)
Test finished in 5.379 seconds
RESULT passed

TEST 9 - complex matrix operations 2 (smaller than 1000 * 1000, 10 sec
timeout)
Test finished in 7.180 seconds
RESULT passed

TEST 10 - complex matrix operations 3 (smaller than 1000 * 1000, 10 sec
timeout)
Test finished in 4.598 seconds
RESULT passed
```

The job of those automated scripts is to orderly run your `pmo` on the test cases and check if your `pmo` perform the matrix multiplication correctly. TEST 0 (available for `make test`) will fail if your program is compiled with errors or warnings. Time limitation is set for each test case, and if this limit is exceeded, your test will fail. Besides, the script will record the running time of your program for performance evaluation. Shorter time/higher performance will lead to better scores.

Below is a brief description of each test case:

- Test case 1-3: the small matrix operation including one matrix addition, or subtraction, or multiplicaiton. You can use this test case to debug your codes.
- Test case 4-5: multiple matrix operations. There are more than 3 matrixes in the expression. There is only one type of operator in the expression (addition and subtraction are regareded as the same operator). The matrix size in those cases are smaller than 500 * 500, thus the time requirement is only 1 second.
- Test case 6-7: combination of matrix operations. There are more than 3 matrixes and multiple types of operators in the expression. The matrix size in those cases are also smaller than 500 * 500, thus the time requirement is only 1 second.
- Test case 8-10: complex matrix operations. There are more than 5 matrixes and multiple types of operators in the expression. The matrix size in those cases are smaller than 1000 * 1000, thus the time requirement is loose and set to 10 seconds.

Each test consists of 5 files with different filename extension:

- n.json (in *tests/config/* directory): The configuration of test case n.
  - binary: Indicates the data type of input and output is binary.
  - timeout: Time limitation (seconds).
  - seed: The seed used to generate the content of input files.
  - params: Including the maximum matrix size and the expression.
  - description: A short text description of the test.
- n.rc (in *tests/stdout/* directory): The return code the program should return (usually 0 or 1).
- n.out (in *tests/stdout/* directory): The standard output expected from the test.
- n/input.txt (in *tests/stdin/* directory): The test data.

You can also run and test your `pmo` manually. For example, to run your `pmo` to perform the toy example and save the results as 1.out, enter:

```
$ ./pmo < ./tests/stdin/1/input.txt > 1.out
```

Then, you can use the following command to compare your results and the standard outputs.

```
$ diff 1.out ./tests/stdout/1.out
```

If your results are correct, nothing will show in the terminal.