Project: Parallel Matrix Multiplication

Due Date: Saturday, November 26, 2022 at **8 PM HKT**. Late submissions will be penalized as per syllabus.

I. Project Instructions

Overview

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

There are three specific objectives to 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 works studying or improving parallel matrix multiplication [1-4] and the design space is quite large. In this project, we narrow down the design space and only pay attention to the basic version of parallel matrix multiplication by dividing the matrix. A practical matrix multiplication 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 divide the matrix for parallelism.
- What work each thread will do.
- How to efficiently merge the final results.

In your project report, please describe the detailed techniques or mechanisms proposed to parallelize the matrix multiplication. 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 used in this project.

Evaluation

We provide 10 test cases for you to test your code. Before submitting your work, please run your `pmm` on the test cases and check if your `pmm` outputs the correct results. 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.

In your project report, summary 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 award 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., pmm.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 multiplication using threads. First, you will recall how to perform a matrix multiplication. Then, you will be given two directions to design your own parallel version of matrix multiplication.

Introduction

For the matrix multiplication, you will be given two matrixes, an $n \times p$ 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 $C = AB$ 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 $j = 1, \ldots, m$.

The matrix multiplication is quite friendly to parallel programming. In this project, we provide you with two 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 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,

$$C = \begin{pmatrix} A_1B \\ A_2B \end{pmatrix}$$

Each sub-matrix multiplication can be computed in a separated thread, i.e., computing A_1B in thread 1 and computing A_2B in thread 2. To fully improve the computation efficiency, you can split rows on the matrix A and split columns on the matrix B.

- 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.

Inputs/Outputs

In this project, your program will read the formatted matrix A and B from the standard input using `scanf()` and output the matrix product to the standard output using `printf()`. First, your program will read 3 integers, `n, p, m` which corresponds to the number of rows of matrix A, the number of columns of matrix A, the number of columns of matrix B. The number of rows of matrix B is the same as the number of columns of matrix A. Then, your program will read two matrixes, matrix A and matrix B. To read the matrix A, your program needs to read `n` lines, where each line contains `p` integers separated by a tab character. Similarly, to read the matrix B, your program needs to read `p` lines, where each line contains `m` integers separated by a tab character.

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 number of rows/columns would **NOT** exceed 4000 (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 multiplication.** Parallelizing the matrix multiplication requires you to think about how to partition the matrix multiplication, so that the computing resources can be fully used. Following the mentioned two directions, 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 determine the number of threads to use.** 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. Then, you are required to create an appropriate number of threads to match the number of CPUs available on whichever system your program is running.

- **How to balance the efficiency of different threads.** During your implementations, you need to think about what works can be done in parallel, and what works must be done serially by a single thread. For example, in the second direction, each sub-matrix product can be computed in parallel, but the sum of them needs to be done in serial. Another interesting issue is that what happens if one thread runs much slower than another? Does the partition scheme give less work to faster threads? This issue is often referred to as the *straggler problem*.

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
├── pmm                     <- A sample solution (executable file).
├── pmm.c                   <- Modify and hand in pmm.c file.
├── README.md
└── tests
    ├── bin
    │   ├── generator.py
    │   ├── generic-tester.py
    │   ├── serialized_runner.py
    │   └── test-pmm.csh
    ├── config
    │   ├── 1.json
    │   ├── ...
    │   └── 10.json
    ├── stdout
    │   ├── 1.out
    │   ├── 1.rc
    │   ├── ...
    │   └── 10.rc
    └── tests-pmm
        ├── 1
        │   └── matrix.txt
        ├── ...
        └── 10
            └── matrix.txt
```

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. Note that the uncompressed project directory has a size of 59M. It takes about a few seconds to unzip the project.zip file on our gateway server. 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
```

A sample pmm is also provided (we only provide a single executable file without source code). This pmm uses pthread to support the parallel matrix multiplication. The pmm adopts a naïve row-wise/column-wise 1D partition scheme that splits the matrix A on rows and splits the matrix B on columns. The pmm uses multiple threads to perform the matrix multiplication.

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

```
$ make run
TEST 1 - toy matrix multiplication (smaller than 10 * 10, 1 sec timeout)
Test finished in 0.010 seconds
RESULT passed
(content removed for brevity)
TEST 10 - large matrix multiplication 5 (smaller than 2000 * 2000, 80 sec
timeout)
Test finished in 43.015 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 pmm with that of the provided one in final report. Note that after building your own pmm (using `make` or `make test`), the provided pmm file will be overwritten. But don't worry, you can always copy it from */public/cs3103/project/pmm*.

2. Writing your pmm program

The pmm.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 pmm is provided for you.

To compile your pmm.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 pmm.

```
$ make
gcc -Wall -Werror -pthread -O pmm.c -o pmm
```

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

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

Note that the `-Werror` compiler flag is specified. It causes all warnings to be treated as build errors. It would be better to fix the compiling issue instead of disabling `-Werror` flag.

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

```
$ ls
Makefile  README.md  pmm  pmm.c  tests
```

If you make some changes in pmm.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 pmm
$ ls
Makefile  README.md  pmm.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/tests-pmm/. 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 - toy matrix multiplication (smaller than 10 * 10, 1 sec timeout)
Test finished in 0.016 seconds
RESULT passed

TEST 2 - small matrix multiplication 1 (smaller than 200 * 200, 1 sec
timeout)
Test finished in 0.022 seconds
RESULT passed

TEST 3 - small matrix multiplication 2 (smaller than 400 * 400, 1 sec
timeout)
Test finished in 0.056 seconds
RESULT passed
```

```
TEST 4 - small matrix multiplication 3 (smaller than 600 * 600, 1 sec
timeout)
Test finished in 0.165 seconds
RESULT passed

TEST 5 - small matrix multiplication 4 (smaller than 800 * 800, 1 sec
timeout)
Test finished in 0.526 seconds
RESULT passed

TEST 6 - large matrix multiplication 1 (smaller than 1000 * 1000, 5 sec
timeout)
Test finished in 1.135 seconds
RESULT passed

TEST 7 - large matrix multiplication 2 (smaller than 1200 * 1200, 15 sec
timeout)
Test finished in 7.173 seconds
RESULT passed

TEST 8 - large matrix multiplication 3 (smaller than 1400 * 1400, 25 sec
timeout)
Test finished in 16.581 seconds
RESULT passed

TEST 9 - large matrix multiplication 4 (smaller than 1600 * 1600, 40 sec
timeout)
Test finished in 22.644 seconds
RESULT passed

TEST 10 - large matrix multiplication 5 (smaller than 2000 * 2000, 80 sec
timeout)
Test finished in 45.642 seconds
RESULT passed
```

The job of those automated scripts is to orderly run your `pmm` on the test cases and check if your `pmm` 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: a toy matrix multiplicaiton. You can use this test case to debug your codes.
- Test case 2-5: the small matrix multiplication. The matrix size in those cases are small, which is smaller than 1000 * 1000, thus the time requirement is only 1 second.
- Test case 6-10: the large matrix multiplication. The matrix size in those cases are all larger than 1000 * 1000, thus the time requirement is loose which is 80 seconds at most.

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.
  - filename: The test case number.
  - matrixsize: The maximum matrix size.
  - timeout: Time limitation (seconds).
  - seed: The seed used to generate the content of input files.
  - 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/matrix.txt (in *tests/tests-pmm/* directory): The test data.

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

```
$ ./pmm < ./tests/tests-pmm/1/matrix.txt > 1.out
```

IV. Reference

[1] Benson A R, Ballard G. A framework for practical parallel fast matrix multiplication[J]. ACM SIGPLAN Notices, 2015, 50(8): 42-53.
[2] Buluç A, Gilbert J R. Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments[J]. SIAM Journal on Scientific Computing, 2012, 34(4): C170-C191.
[3] Buluc A, Gilbert J R. Challenges and advances in parallel sparse matrix-matrix multiplication[C]//2008 37th International Conference on Parallel Processing. IEEE, 2008: 503-510.
[4] Alqadi Z A, Aqel M, El Emary I M M. Performance analysis and evaluation of parallel matrix multiplication algorithms[J]. World Applied Sciences Journal, 2008, 5(2): 211-214.