ECE 420 Parallel and Distributed Programming
Lab 1: Matrix Multiplication with Pthreads

Winter 2023

In this lab, we will implement a parallel program for matrix multiplication with Pthreads. This lab manual mainly consists of two parts: Section 1 and Section 2 introduce the background of the problem and the requirements for this lab, while Section 3 and Section 4 provide some help regarding the working environment and debugging.

1 Background: Matrix Multiplication

Matrix multiplication is a binary operation on two matrices to produce a third matrix. Specifically, for an $m \times n$ matrix $A = (a_{ij})$ and an $n \times k$ matrix $B = (b_{ij})$, the product $A \cdot B$ is an $m \times k$ matrix $C = (c_{ij})$, where for some i and j,[1]

$$c_{ij} = \sum_{r=0}^{n-1} a_{ir} \cdot b_{rj}. \tag{1}$$

It is natural to think about speeding up the calculation by simultaneously processing several elements in C by several threads. We can make every thread in charge of calculating a certain group of c_{ij}. To balance the load, we can assign the same number of elements in C to each thread (or about the same number if the number of elements is not a multiple of the number of threads). Specifically, we can divide C into blocks of similar sizes and assign the elements of each block (or a submatrix) to a thread, as shown in Fig. 1.

[1] All our indices start from 0 to respect the C convention.

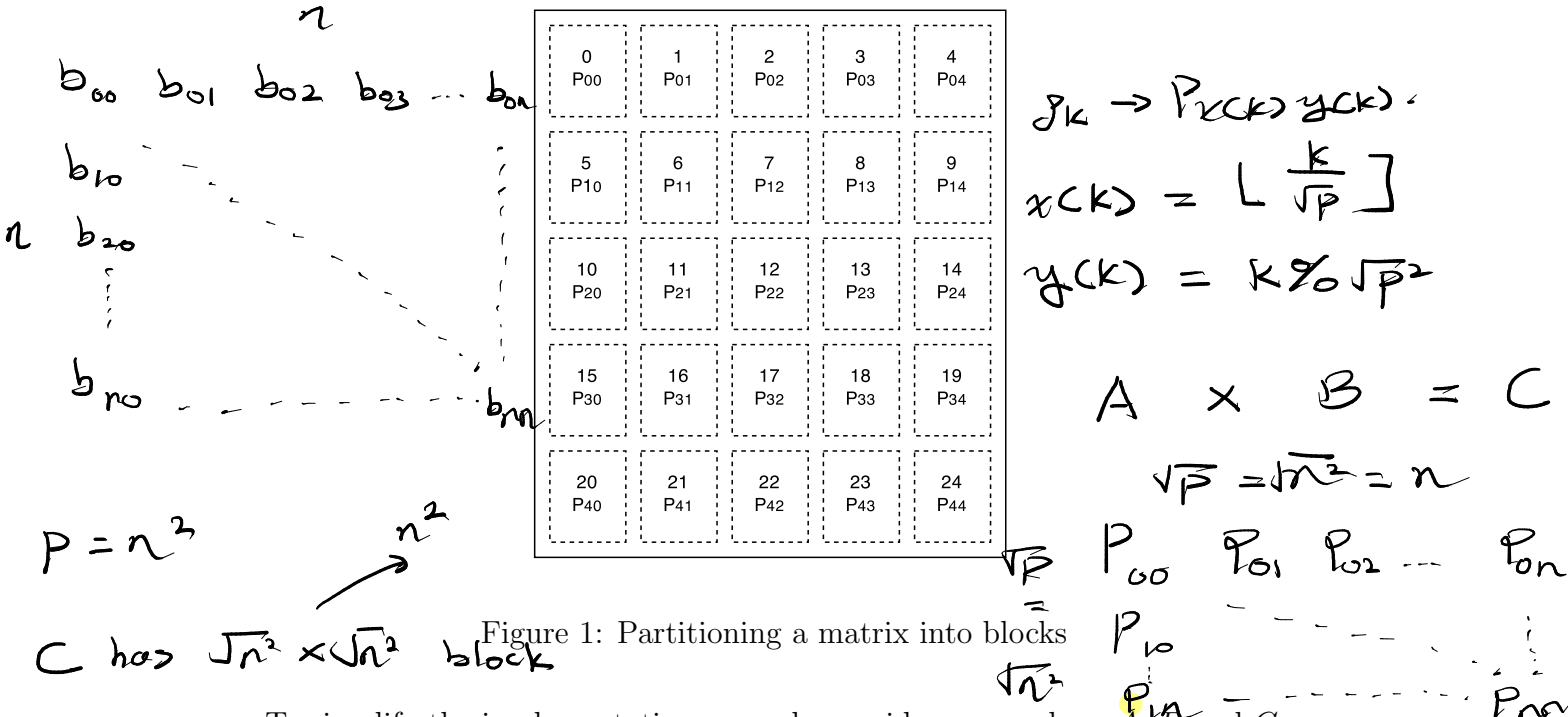

Figure 1: Partitioning a matrix into blocks

To simplify the implementation, we only consider cases where A, B and C are all $n \times n$ square matrices. Furthermore, we only consider cases where the number of threads, p, is both a **square number** and a **factor of n^2**. Here, the matrix C is divided into an array of $\sqrt{p} \times \sqrt{p}$ blocks. We denote the block in the x^{th} row and the y^{th} column as P_{xy}. For a thread of rank k, $0 \leq k \leq p-1$, we can map it to the block P_{xy}, where $x = \lfloor \frac{k}{\sqrt{p}} \rfloor$ and $y = k\%\sqrt{p}$.[2] After this mapping, the thread of rank k is responsible for computing a group of c_{ij} elements whose i and j fulfill the conditions:

$$\frac{n}{\sqrt{p}}x \leq i \leq \frac{n}{\sqrt{p}}(x+1) - 1$$

and

$$\frac{n}{\sqrt{p}}y \leq j \leq \frac{n}{\sqrt{p}}(y+1) - 1.$$

[2] "$\lfloor\rfloor$" returns the floor; "%" is modulus (returns the remainder after division).

2 Tasks and Requirements

Task:

Implement a shared-memory parallel program for matrix multiplication using Pthreads with the block partition strategy described in Section 1.

Instructions:

1. Use the scripts in "Development Kit Lab 1" to generate input data, load data and save result. Refer to the *readme* file for details on how to use them.
2. The number of threads should be passed as the only command line argument to your program.

Requirements:

1. Ensure proper submission of your work by complying with the requirements in the submission section below.
2. For system usage, you must first have your VM cluster set up with the LI and TA and be able to remotely access the VM cluster.
3. On the master node of the VM cluster, you should ensure that your code can properly compile and execute.
4. Check that your implementation yields correct results for different matrix sizes and with different thread counts (so long as both parameters adhere to the constraints mentioned in Section 1).
5. Time measurement should be implemented in your code and saved through the provided IO functions.

Submission:

Each team is required to submit a zip file to eClass by the submission deadline. The zip file should be named "StudentID.zip", where "StudentID" is the Student ID of **one** of your group members (it does not matter which member, though this should be consistent for all submissions throughout the course). The zip file should contain the following two folders:

1. "Code": this folder should contain all the codes necessary to compile the "main" executable, including:

   (a) "Makefile": with which, the solution *executable* named "main" should be generated in the same folder of "Makefile" after executing the "make" command;

   (b) Other source files necessary to compile your solution "main".

   DO NOT include any compiled executable files or the input/output data file.

2. "Members": this folder should contain a single text file *named* "members.txt", listing the student IDs of ALL group members, with each student ID occupying one line.

Note: you MUST use the file names suggested above. File names are case-sensitive. You MUST generate the required zip file by directly compressing all these folders, rather than compressing a parent folder containing the aforementioned folders.

Example:

Consider that a lab group consists of the students Alice (Student ID 1234567), Bob (Student ID 7654321), and Charlie with (Student ID 4352617). The group has decided that Charlie is to handle their lab submission:

Charlie should submit a zip file named "4352617.zip", containing the following:

1. "Code" folder with the "Makefile" and all necessary source files. By executing the sequence of terminal commands: "unzip 4352617.zip", "cd Code/" and "make", the executable "main" should be generated.

2. "Members" folder containing the "members.txt" file. The "members.txt" file should have three lines: the first line being "1234567", the second line being "4352617", and the third line being "7654321". Note the Student IDs do not need to be presented in a particular order.

Hints:

The correctness of your results can be checked by the script "serialtester.c" in the "Development Kit Lab 1"

3 Basics on Compiling and Running Programs on Linux Systems

3.1 Editing, Compiling and Executing the Code

Pthreads is not a programming language, but an extension package. In this lab, we will use C. You can use whatever text editor you like to write the code. Some simple text editor like VIM, emacs and gedit will be good enough. To use the Pthreads package, you need to include the header file "pthread.h" in your code:

```
#include <pthread.h>
```

After completing your code, you need to compile it to generate the executable file. It is more or less the same as compiling a typical C program in Linux. For example, if the code file name is "demo.c", in the terminal, supposing the current path is your code folder, the command to compile will be

```
$gcc -g -Wall -o demo demo.c -lpthread
```

"-g" will generate the necessary information for the debugger. "-Wall" will turn on all the warnings. "-o demo" specifies the output file path and name. "demo.c" is the source code file. "-lpthread" tells the compiler to link the Pthreads library. Once it is successfully compiled and error free. You can execute the code by

```
$./demo <possible command line parameters>
```

3.2 Starting and Terminating a Thread

The Pthreads library can create threads to run a function. The function has a special prototype for the Pthreads. It has a "void*" return type and the argument is a void pointer "void*". For example, supposing the name of the function is "threadfunc", the prototype should be

```
void* threadfunc (void* arg_p)
```

In fact we can pass whatever argument through this pointer "arg_p". In this lab, we only focus on the single program multiple data scheme, so typically we will pass the rank into the thread function. Inside the thread function, we need to cast the void type pointer back to the desired type before we can get access to those arguments.

Pthreads uses "pthread_t" data structure to store the thread information and

handle them. We need to assign each thread to an individual "pthread_t" object. Same as all the other variables in C language, we need to declare the "pthread_t" objects before we use them.

To start a thread running a specific function, we use `pthread_create`. The syntax is

```
int pthread_create (
pthread_t* thread_p,
const pthread_attr_t* attr_p,
void* (*start_routine)(void*),
void* arg_p)
```

"thread_p" is the pointer of the handle we assign to the thread. We don't use the second attribute in the lab. The "start_routine" is the function we want the thread to run. The "arg_p" is the pointer to the argument we want to pass to the thread function. Say if we want to start a thread running the function "threadfunc" with an assigned rank "1", the following code will do so

```
pthread_t thread_handle;/*Declare the object before you use
it*/
int thread_indx=1;/*Here we want to pass the rank ``1'' to the
thread*/
/*...some other code*/
pthread_create(&thread_handle, NULL,
threadfunc, (void*) thread_indx);
```

Note that we need to cast the type "int" into "void*" for "thread_indx" to pass it to the thread function.

We use `pthread_join` to wait for the thread to stop in our program and collect the returned arguments by the threads. The syntax is

```
pthread_join(pthread_t thread_p,
void** ret_val_p)
```

In our example, if we simply want to wait for the thread function "threadfunc", with the handle "thread_handle" while ignoring the returning value, we can use

```
pthread_join(thread_handle, NULL);
```

3.3 Time Measurement

The motivation of utilizing the parallel approach is to speed up our program. To find out the real performance and for evaluation purposes, we need to measure the time consumed by the program. Different from serial programs, it will make no sense to use the `clock` function in C since we are more interested in the total elapsed time, not the CPU time.[3] It is not suitable to use the linux shell command `time` since it will record the entire program execution time. This includes time for operations that are not of our interest, such as time consumed by I/O. In the parallel program with Pthreads, we can use some function in the `timer.h` header to record the time at both the start and end of the main calculating segment. Then we can get the time as the difference between those two checkpoints.

The header "timer.h" has defined a macro for time measurement. Refer to the *readme* and the notes in that file for more details.

In Lab 1, to measure the time in our program, it is necessary to record the start time right before you create the threads and record the end time right after you stop all the threads by `pthread_join`.

4 Debugging and Testing

4.1 Debugging a Parallel Program

Debugging might be one of the toughest part in parallel programming. However, never be afraid of bugs! Be confident in yourself and we can fix everything if we check carefully. The worst case is only that we check the program line by line with some debugging tool. Nevertheless, although we should be able to fix everything by debugging, it is always better to be more careful in the development stage and try to prevent the mistakes through good design.

For debugging parallel programs, the challenge is that the threads run simultaneously and the results are nondeterministic. We cannot test all the possible situations. Unlike debugging a serial program, debugging a parallel program is an art.

[3] Actually, we can expect that the parallel program will take more CPU time than the serial one. The parallel version cannot shrink the necessary calculation. To the contrary, it will introduce some overhead and other cost which the serial program will not contain.

One possible approach would be

1. Write your code so that it can run in serial: perfect that first.

2. Deal with communication, synchronization and deadlock on a small number of threads.

3. Only then should you increase the number of threads.

Note that in our Pthreads program, assigning only one thread to run the program would be an efficient way to run it in a serial manner. Also, you don't have to follow this approach. You can come up with your own better strategies and you are always welcome to share your ideas.

4.2 Basics on Using `gdb` Debugger

In our lab, you can use whatever debugging tools you like. Here we will introduce the basics on using the `gdb` debugger. `gdb` is a simple, but powerful, command line based debugger.

You can launch the `gdb` by the "gdb" command in the terminal. If you want to debug the executable program "demo", you can type

```
$ gdb demo -tui
```

Note that you need to compile the code with the "-g" flag to link the executable code to the source to use the debugger. The "-tui" option will launch a simple GUI.

Table 1 shows the typical commands for debugging a serial program. Note that to set the breakpoints, you can either indicate the function name or the line number of the code. "`run`" will start running the program and command line arguments can be set after it. The index for the breakpoints are generated by the `gdb`, you can use "`info b`" to check the index and delete the corresponding breakpoint with "`delete`".

As for debugging a program with multiple threads, Table 2 shows the basic commands. When you want to debug a multithreaded program, you need to set break points inside the thread function first. The program will stop at the breakpoint in one of the threads. You can then check the current running threads with "`info thread`". There will be a "*" before the active thread. You can also find the indices and you can switch to other threads by the command "`thread [thread index]`".

Table 1: Basic `gdb` Commands

Commands	Usage	Example
`b`	set the break point	b main; b 41
`info b`	list break points	
`delete`	delete break point	delete 2
`run`	run the program	run [args]
`n`	step to the next statement	
`s`	step into the function	
`c`	continue running	
`p`	display variable value	p V
`set`	set variable value	set V=3

Table 2: `gdb` Thread Commands

Commands	Usage	Example
`info thread`	show the running threads	
`thread`	switch into another thread	thread 2

Note that when you are inside a thread, commands like "`n`", "`s`" etc. will only influence the current thread and all the other threads will do nothing. However, when you use "`c`", all the threads will run simultaneously, and it will stop on the breakpoint of whichever thread that first hits its next breakpoint. This means for example, when you are in Thread 2, after you input the command "`c`" it is possible to be in another thread when the program stops again. A similar situation will occur for "`run`", as the program will stop in the thread which first hits its breakpoint, rather than Thread 1.

For more commands and information, you can type "`help`" for details or check online.

4.3 Testing Your Program

Testing is always an important procedure of coding and even tougher than debugging, especially for parallel programming. It is generally considered to be infeasible to thoroughly test a program of moderate complexity. However, to be a good pro-

grammer, we need to try our best to ensure the quality of the program. In our lab, we only have a minimum requirement on testing, i.e., to guarantee the correctness of your program.

Since we cannot cover all the possible inputs, we need to carefully choose the testing cases and justify the correctness as much as possible. Due to the nondeterministic property of parallel programs, some potential errors might not appear at first in some cases. We might need to test the same case several times for some parallel programs.

A Appendix: Marking Guideline

System usage (cloud setup, remote access):	1
Correct results under different thread count settings:	2
Time Measurement Implementation:	1
Notable Speedup From Multithreading:	1
Lab Report:	0 (Not required for this lab)
Total:	5