

ECE 420 Parallel and Distributed Programming

Lab 1: Matrix Multiplication with Pthreads

Winter 2024

In this lab, we will implement a parallel program for matrix multiplication with Pthreads. This lab manual mainly consists of three 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. Section 5 provides the marking guideline and rubric.

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 $\forall i$ and j ,¹

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

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 rows/columns is not a multiple of the number of threads). Specifically, we can divide C into blocks of similar sizes and assign the elements in each block (or a submatrix) to a thread, as shown in Fig. 1.

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

0 P ₀₀	1 P ₀₁	2 P ₀₂	3 P ₀₃	4 P ₀₄
5 P ₁₀	6 P ₁₁	7 P ₁₂	8 P ₁₃	9 P ₁₄
10 P ₂₀	11 P ₂₁	12 P ₂₂	13 P ₂₃	14 P ₂₄
15 P ₃₀	16 P ₃₁	17 P ₃₂	18 P ₃₃	19 P ₃₄
20 P ₄₀	21 P ₄₁	22 P ₄₂	23 P ₄₃	24 P ₄₄

Figure 1: Partitioning a matrix into blocks

To simplify the implementation, we only consider the square matrices, i.e., A , B and C are all $n \times n$ matrices. Furthermore, we only consider the number of threads p by which n^2 is **divisible** and we only consider **square** numbers for p . Then, the matrix C is divided into an array of $\sqrt{p} \times \sqrt{p}$ blocks. 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 Moreover, the thread of rank k mapped to block P_{xy} contains the elements c_{ij} for $\forall i, j$ such that

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

and

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

² “ $\lfloor \cdot \rfloor$ ” 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. For VM setup, you must generate an RSA key pair and email your public key to the LI and TAs. They will provide you access to a VM cluster. To ensure that you can verify your code on the VM before submission, please do this as soon as possible.
2. Before submission, you should ensure that your code can properly compile and execute on the master node of the VM.
3. Ensure 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).
4. For marking purposes, you must use the IO functions in “lab1_IO.c” for loading the input matrices and saving the output results.
5. Optimize your code as much as possible. You will be graded on the speedup achieved. Deductions may also be applied if your program takes too long to run.
6. **Hint:** Implement a serialized version of the matrix multiplication program first. From here, you can identify the sections that should be parallelized and add the necessary multithreading code.

7. Ensure that your code is properly submitted on eClass by complying with the requirements in the submission section below.

Submission:

One member in each team is required to submit a zip file to eClass before the submission deadline. While we do not enforce a naming convention for the submissions, a good template is “user**_lab1.zip” (where ** is the user number). The zip file should contain the following:

1. “Makefile”: By executing the `make` command, the solution executable named “main” should be generated. Please do not use any optimization flags (i.e. no `-O3`).
2. Solution source files: You should include all source files necessary to compile your solution executable. Note that you do not need to include “matrix-gen.c”.

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 you finish 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 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 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 third one 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_idx=1; /*Here we want to pass the rank ``1`` to the
```

```
thread*/
/*...some other code*/
pthread_create(&thread_handle, NULL,
threadfunc, (void*) thread_idx);
```

Note that we need to cast the type “int” into “void*” for the desired argument we want to pass 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” 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 include 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 `time.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`.

³Actually, 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.

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.

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: 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 4.2 shows the typical commands for debugging a serial program. Note that to set the break points, you can either indicate the function name or the line number of the code. “run” will start running the program and possible 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 4.2 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 thread. 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]”.

Note that when you are inside a thread, command 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 could be not in Thread 2 when it stops again. A similar situation will occur for “run”, and it will stop in the thread which first hits its breakpoint, rather than Thread 1.

For more commands and information, you can type “help” for more details

Table 2: gdb Thread Commands

Commands	Usage	Example
<code>info thread</code>	show the running threads	
<code>thread</code>	switch into another thread	thread 2

or check it out 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 programmer, 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.

5 Marking Guideline

5.1 Marking Session

Each group is required to present a short demo of their code within an appointed in-lab time-slot. Each demo consist of the components:

1. **Demo:** Upload and compile your eClass submission and verification code (provided during the marking session) in your VM. Demonstrate that your code works for different matrix sizes and with different thread counts, as specified by the marker (a TA or LI). This includes verifying the correctness of your results and checking the speedup achieved.

2. **Presentation:** Provide a short (2-minutes) verbal explanation on the implementation design.
3. **Group Response:** The marker will ask some questions to the group. Marks will be assigned to the group based on your collective response to these questions.
4. **Individual Response:** The marker will ask some questions to each group member. Marks will be assigned to each individual based on their individual response (without assistance from other group members) to these questions.

To expedite the marking process, please rehearse your demo beforehand. Be aware that:

1. You should practice the process of loading, unzipping, and compiling your eClass submission in the VM in an efficient manner.
2. The marking process is timed. While there should be ample time for the demo, the marker may cut you off if the demo runs over the time limit.
3. The questions are sampled from a pool, and may be different for different groups.
4. The contents of the questions asked by the marker may include, but are not limited to, explaining observations from the results, predicting program performance under certain scenarios, potential improvements on the existing solution, describing certain components within the solution code, defining concepts (learned in the lecture) that are relevant to this lab, etc.

5.2 Marking Rubric

VM setup and successful code compilation:	1
Correct results under multithreading:	1
Time measurement and reasonable speedup:	1
Group response to questions:	1
Individual response to questions:	1
Total:	5