

ECE 420 Parallel and Distributed Programming

Lab 1: Matrix Multiplication with Pthreads

Winter 2015

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 requirement for this lab, while Section 3, Section 3.3 and Section 4 introduce some basics related to the working environment and debugging.

1 Background: Matrix Multiplication

Matrix multiplication is a binary operation on two matrices producing a third matrix. Specifically, for an $m \times n$ matrix $A = (a_{ij})$ ¹ 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 ,

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

It is natural to think about speeding up the calculation by simultaneously processing several elements in C . There are different partition strategies for this task. One of the design choices is that we make every thread in charge of calculating several 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.

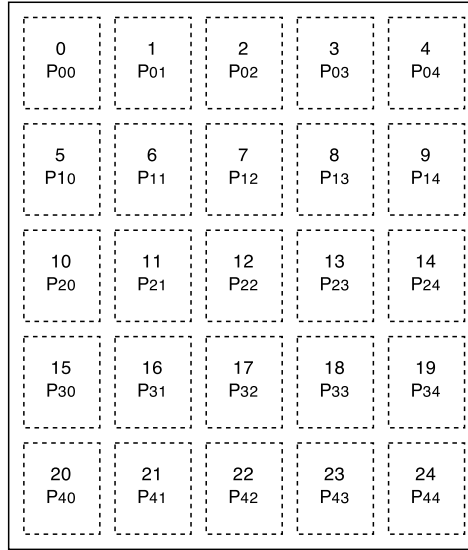


Figure 1: Partitioning a matrix into blocks

To simplify the implementation, we only consider the square matrices, i.e., all A , B and C are $n \times n$ matrices. Furthermore, we only consider the number of threads p by which n^2 is divisible. 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 $y = k \% \sqrt{p}$. 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 Tasks and Requirements

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

Requirements: 1. Use two randomly generated matrices as inputs. For this part, the script is provided to you in “Development Kit Lab 1”. And you can

simply call those functions to load the data into global variables. Refer to the *readme* file for details.

2. Declare a global variable to store the computed result. Also, declare global variables to keep track of the matrix size and the number of running threads.

3. Time measurement should be implemented in your code. Refer to Section 3.3 for details.

In-lab Checking: Please find some way to check the correctness of the results when you finish your program. A serial program is provided to you in “Development Kit Lab 1” to test the correctness of the result. We will ask you to demo your program in lab with some specific number of threads and matrix size. We will also check the time measured. Please have a TA or LI check your work. Remember to bring a printed marking sheet filled with your names and IDs before you ask for checking.

Lab Report: Please discuss the impact of the number of threads and problem size on the speedup and efficiency. You can use figures and tables to present your experimental results. Explain your results and provide insights behind these results.

Please refer to the lab report guide for general guidelines. The marking sheet should be attached to the end of your lab report. Your lab report should be handed in the assignment box on the 2nd floor of ECERF. The due date is two weeks after the first lab, i.e., 4:00 p.m. Wednesday, January 28th sharp.

3 Basics on Compiling and Running Programs under Linux System

3.1 Editing, Compiling and Executing the Code

Pthreads is not a coding language, but an extension package. In this lab, we will use the C language. You can use whatever text editor you like to write the code. Some simple but powerful text editor like VIM 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 with compiling a typical C code in the Linux system. 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 thread to run a function. The function has a special prototypes 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 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 Measuring

The primitive motivation of utilizing the parallel approach is to speed up our program. To find out the real performance and for some evaluation purposes, we need to measure the time consumed by the program. Different to the serial program, 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.² It is not suitable to use

²Actually, 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.

the linux shell command `time` since it will include all the time the program will take in which some portion like the time consumed by the IO is not of our interest. In the parallel program with Pthreads, we can use some function in the `time.h` header to take down the time at both the begin and end of the main calculating segment. Then we can get the time by the difference of those two time.

The header “`timer.h`” has defined a macro for time measuring. Refer to the *readme* and the notes in that file for more detail.

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

4 Debugging and Testing

4.1 Debugging a Parallel Program

Debugging might be one of the toughest part in the parallel programming. There should be thousands of more words than here to go through all the aspects of debugging.

However, never be afraid of the bugs! Be confident in yourself and we can fix everything if we carefully check it. The worst case is only that we check the program line by line with some debugging tools.

On the other hand, although we should be able to fix everything by debugging, it is always better to be more careful in the developing stage and try to prevent the mistakes by good designing.

As for the debugging for parallel program, the challenge is that the thread runs simultaneously and the results are nondeterministic. We can hardly test all the possible situations. Unlike debugging the serial program, debugging a parallel program is state of the 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 smaller number of threads.
3. Only then try the full size.

Note that in our Pthreads program, assigning only one thread to run the program would be a 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 welcomed to share your ideas.

4.2 Basics on Using **gdb** Debugger

In our lab, you can use whatever debugging tool as you like. Here we will introduce the basics on using the **gdb** debugger. **gdb** is a command line based simple but powerful 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 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 multiple thread 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 indexes 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 at a break point of which thread first hits its next breakpoint. This means for example, when you are in Thread 2, after you input the command “c” it could not be in

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 |

Thread 2 when it stops again. Similar situation will occur for “run”, and it will stop in the thread who first hit its breakpoint, rather than Thread 1.

For more commands and information, you can type “help” for more details 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 the parallel programming. It is generally considered to be infeasible to thoroughly test a program of a moderate complexity. However, to be a good programmer, we need to try our best to ensure the quality of the program. In our lab, you only have a minimum requirement on the 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 the parallel programs, some potential errors might

not appear at first in some cases. We might need to test a same case for several times for some kinds of parallel programs.

In a word, testing is a trade-off between time and quality. We always need to try our best to guarantee the desired quality with as less time as possible.

ECE420 Lab 1: Marking Sheet

Names and IDs: _____

In-Lab (40):

Correct Results. 20 _____

Time Measurement. 20 _____

TA Signature: _____

Lab Report (60):

Description of Implementation 10 _____

Testing and Verification 5 _____

Performance Discussion 30 _____

Conclusion and Experiences 5 _____

Presentation 5 _____

Coding Style 5 _____

TA Signature: _____

Total (100): _____