EE 352 Lab 4 – Going Parallel

1 Introduction

In this lab you will create a parallelized matrix multiply algorithm using the PTHREAD (Posix Threads) library in C/C++ and compare the performance for different parallelization strategies and number of threads/processors.

2 What you will learn

This lab is intended to give you an introduction to PTHREAD programming and expose you to some of its intricacies. You will also learn how to distribute the workload and data to multiple threads as well as ensuring synchronization with mutex/locks. Specifically, after completing this lab you will…

1. Be able to use the fundamental functionality of the PTHREAD threading library
2. Gain an appreciation for the effects of the memory hierarchy on parallel programs
3. Gain an appreciation for the cost of synchronization when shared variables are written
4. Understand the different locking mechanisms and their advantages.

3 Background Information and Notes

PTHREAD Programming: Many good tutorials exist for PTHREAD programming on the Internet. These can be found by searching for the key phrase: "pthread programming". Two of the better resources are at: https://computing.llnl.gov/tutorials/pthreads/ and http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html. Read through these websites focusing on the following skills and examples:

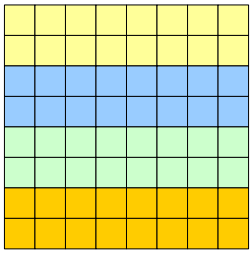
- Creating PTHREADS
- Terminating PTHREADS
- Compiling and Linking
- Mutex/lock declaration, initialization, locking, unlocking, and destruction

If you prefer man pages, you may see http://opengroup.org/onlinepubs/007908799/xsh/pthread.h.html for a reference.

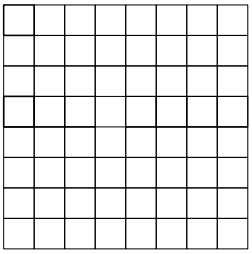
Parallelizing Matrix Multiplication: Traditional matrix multiplication of an NxN square matrix is achieved by taking the inner/dot product of an entire row and column as shown with the code below:

```
for(i=0; i < N; i++)
  for(j=0; j < N; j++)
    for(k=0; k < N; k++)
      C[i][j] = C[i][j] + A[i][k] * B[k][j];
```

Many parallelization strategies exist when trying to decompose a program in parallelizable threads. One of the simplest techniques is to have each thread perform the same task on different subsets of the data. This is referred to as data-level parallelism. For matrix multiply, we must compute the inner product for each C[i][j] element. Thus, we can distribute portions of the C matrix to each thread and let it compute its subset of elements (note that A and B may be shared amongst all the threads). This data parallel approach can easily be implemented by choosing one of the 'for' loops and rather than having a single thread iterate from 0 to N-1, distribute that range to all the threads (i.e. each thread handles N/T iterations of the loop where T = # of threads). Depending on which loop we choose, the data in certain matrices will be distributed amongst the threads. Figure 1 below illustrates this data distribution depending on which loop is parallelized.

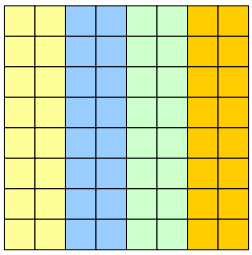


A and C matrices

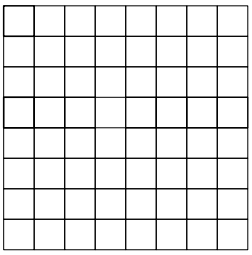


B Matrix
Shared amongst all

Parallelizing the outer loop (i-variable) = Distributing responsibility for subset of **rows** of A and C to each thread. All columns of B are shared amongst all threads.

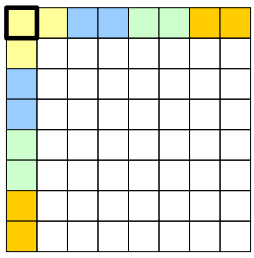


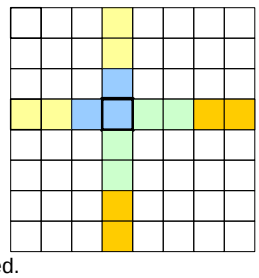
B and C matrices



A Matrix
Shared amongst all

Parallelizing the middle loop (j-variable) = Distributing responsibility for subset of **columns** of B and C to each thread. All **rows** of A are shared.





All matrices shared.

Parallelizing the inner loop (k-variable) = Each C[i][j] element is calculated with the help of each thread (the inner product is distributed).

Figure 1

To "distribute" the range, each thread has an id argument that is passed to it [i.e. a number between 0 and T-1]. You can use this number to set the for loop iteration range. Note: N must be a multiple of T for this to work correctly.

Synchronization Issues: For the first two parallelization schemes, there are no elements that are read/modified/written by multiple threads (i.e. each C[i][j] is written by only one thread and A and B are shared but are only read). In this case, we do not need to worry about synchronization/locking. However, in the case of parallelizing the inner (k) loop, each C[i][j] is read/modified/written by EACH thread and thus we need to worry about "atomic" updates of those variables. Thus you will need to use lock/mutex variables. We can create one mutex/lock PER C[i][j] entry (i.e. have a matrix of locks corresponding to the data matrices). To do this, we simply declare:

```
pthread_mutex_t Cpi_locks[SIZE][SIZE];
```

Each mutex/lock needs to be initialized (which is done in the `main()` function) via the `pthread_mutex_init()` function before being used and then destroyed with `pthread_mutex_destroy()` when the code is finished. As the code runs you can call the `pthread_mutex_lock()` and `pthread_mutex_unlock()` functions to ensure only one threads enters the critical section at a time.

When performing the locking we can lock the C[i][j] element EACH time we update it in each thread by simply putting a lock/unlock pair around the body of the triple-nested loop (i.e. C[i][j] = C[i][j] + A[i][k] + B[k][j]). However, this may lead to poor performance because each thread will contend for the one lock on EACH update (i.e. each item will be locked and unlocked N times). You should think about a way to let each thread aggregate its computations to a private variable and then aggregate the results from each thread (i.e. lock and unlock T times).

4 Procedure and Program Requirements

You will create separate version of the basic matrix multiply program to explore the different parallelization schemes: outer loop (rows), middle loop (columns), and inner loop with different locking mechanisms. You will also perform some experiments understand the difference between mutex locks and spin locks. For each program, we will provide code that computes the answer in a set of reference matrices and then calls the code that you will modify to complete the parallelized multiply. We will compare the results to ensure you implemented your parallelized multiply code correctly.

Follow the instructions below to complete the lab.

a. Read completely and **understand** the previous section on Background Notes and Information.

b. You will conduct your experiments and run your programs on one of two servers: parallel03.usc.edu or parallel04.usc.edu. Accounts have been created for you (if you have not used the machines before) using your current USC username and an initial password of your 10-digit student ID. After your first logon, you can use the '`passwd`' command to change your password as desired (this is highly recommended since your 10-digit USC ID is not very private). If you have used these machines before you can simply logon with your current credentials.

c. SSH into your account on parallel03.usc.edu or parallel04.usc.edu using your USC username and provided password. If you like, you may start a VNC session via `vncserver` and then use any VNC Viewer application to remotely control you GNOME desktop on parallel03 or parallel04.

d. Create an ee352 directory: "mkdir ee352" and change to that directory, "`cd ee352`".

e. Copy the skeleton files and makefile in a provided tarball by executing the command: "`cp ~redekopp/rel/matmult.tar ~/ee352`"

f. Untar the tarball via: `tar xvf matmult.tar`

g. **Take a tour of the source code skeleton.**

- Open and edit the program `matmult_par_row.c` in your favorite text editor (emacs, vi, etc.).
- At the top you will find #define's for the SIZE of the matrix.
- Next, you will find global declarations of the number of threads (we will make this a command line argument so you can vary it easily) and the matrices. Note that global variables are visible and shared amongst all threads.
- You will then find your matrix multiply function. Each thread will execute this function concurrently. The argument passed to it will be the threaded (a number between 0 and NUM_THREADS-1). We first cast that argument from a void * to a normal integer (PTHREADs requires the argument to a thread be of type void * which is a 64-bit type on these machines, so we just cast it to a 32-bit integer).
- Finally, you will see the set of nested loops that implement the matrix multiply. THIS WILL NEED TO BE MODIFIED BY YOU SO EACH THREAD ONLY DOES ITS PORTION OF THE WORK. Currently it is setup so each thread does the entire multiply (which is not correct).
- Below the mult_thread() function you will see the main() function. Examine this code and notice how threads are created and joined. Notice that we spawn only N-1 threads and use the main function itself to call mult_thread, so that the main thread acts as the last multiplier thread. Also, notice the error checking loops at the end of the main function that compare the reference answer to your computed answer. You should not need to modify the main thread other than changing some printf's that mention "row" parallelization to mention "column" or "inner-loop" parallelization.

h. Change/add to the matmult() function so that each thread only computes the result on a subset of rows of the C matrix by using the thread id (tid variable) to determine which rows are traversed by each thread. While there are a few alternatives for how each thread can select which rows it is responsible for, choose a solution that matches the picture in Figure 1 for how we would like rows distributed.

i. To compile our program we will use the Intel C++ compiler (an alternative to gcc/g++ that is specifically designed to take advantage of certain Intel processor features). Compile your program with:

```
'icpc -o matmult_par_row matmult_par_row.c  -lpthread'
```

Note 1: 'icpc' is the analog of 'g++' and stands for '**I**ntel **C** **P**lusplus **C**ompiler'. For, C programs, you can use 'icc' which is analog of 'gcc'.

Note 2: Notice that we must include the PThread library by using the option '`-lpthread`'.

j. Run your program first for 1 thread and then for 2, 4, and 8 threads respectively, recording the execution time (printed in units of microseconds) for each run. Later you will make a graph of the run-time vs. number of threads, so be sure you write the results down. Also, recall that the number of threads is provided as a command line argument, so your command sequence should be:

```
> ./matmult_par_row 1
> ./matmult_par_row 2
> ./matmult_par_row 4
> ./matmult_par_row 8
```

Ensure that the error count for your program is 0. If it is non-zero you have made a mistake in your data distribution and should go back and check your work.

Important: The servers you are running on are time-shared systems, meaning other students could be running their code at the same time you are. If multiple students are running their programs and using 8 of the cores each, your run-time numbers will be skewed. You can see who is running processes on the machine via the 'top' command in Linux ('q' will quit the program). This will show you a list of processes and how much CPU time they are taking. Try to find a time when no other users are using the machine and then run your tests. You should probably run each test 2-3 times to make sure your numbers are consistent.

k. Once you have got your row-parallel code working, open the column parallel implementation, 'matmult_par_col.c' and modify the program so that the columns are distributed to each thread (i.e. parallelize the middle-loop).

l. Compile and run your column-parallel solution for 1, 2, 4, and 8 threads again recording the execution times for each run. Again ensure the error count is 0. Mentally compare the execution times for the row vs. column parallel implementations.

m. Now open the inner-loop parallel code, 'matmult_par_inner.c'. Change the inner (k) loop to be distributed amongst each thread. Notice that in each iteration of the inner loop, we must gain the lock before we updated the C matrix entry.

n. Compile and run your inner loop parallel solution for 1, 2, 4, and 8 threads again recording the execution times for each run. These could take quite a while to run due to the locking overhead (up to 5 minutes or so, so don't be alarmed if you don't see much response for a while). Again ensure the error count is 0.

o. To demonstrate the need for locking, comment out the 'lock' and 'unlock' function calls in the inner-loop so that there is no protection when updating C[i][j]. Recompile and re-run your program and check if errors occur. Then uncomment those lines since they are necessary.

p. In addition to the '`pthread_mutex_t`' type, the PTHREAD library has another lock type called '`pthread_spinlock_t`'. Let's experiment using these in place of the '`pthread_mutex_t`' variables. First, modify the `C_lock` declaration at the top of the file to read:

```
pthread_spinlock_t C_lock[SIZE][SIZE];
```

Then, replace all '`pthread_`**`mutex`**`_xxx`' function calls with '`pthread_`**`spin`**`_xxx`'

Recompile and re-run your program again using 1, 2, 4, and 8 threads and note the execution times. Which locking mechanism performs best in this program? **Going forward, use the best performing lock-type for the rest of this lab.**

q. As discussed in the Background Information section, this current inner-loop implementation is locking on each iteration, meaning each time through the inner-loop the threads are contending for the lock and the sequentializing access. This is a prime reason that our solution is poor. Think about how you can reduce the amount of locking (shared-updates) that takes place and modify the matmult() function to implement your ideas. You will still need locks since the inner-loop is being parallelized but you should be able to reduce the locking to only once per thread. Test your solution (ensure the error count is 0) and see if you can reduce the execution times for the various number of threads. Record your execution times for 1, 2, 4, and 8 threads.

5 Review

1) Write out your expression/formula for calculating the start value of the for loop index for a given thread in terms of N, T, and tid.

2) Briefly explain how you reduced the locking in procedure part 4q.

3) Create an Excel XY plot of the table above using only the data from the row-parallel, column-parallel, and the results of your code from 4q, showing speedup (1 thread exec. time / n thread exec. time…note your first data point should always be 1 since n=1) on the vertical axis and number of threads on the horizontal axis. Because some

sets of results may be large compared to others, you may change the vertical axis to a log scale to make the difference between each data set more visually distinct.

4) Ideally, when parallelizing code on t processors, we should see a speed up of approximately t times (i.e. linear speedup vs. number of threads). For each data set, at what number of threads does this fail to hold by a significant amount (e.g speedups of 3.7 using 4 threads still demonstrates a high-degree of linear speedup while 2.8 using 4 threads does not)? Think about what might cause such performance degradation.

5) In certain situations, speedup of a program parallelized amongst t threads can be even more than t-times (called super-linear speedup). You should see this occur in at least one of your data sets? If so suggest why this might occur? (Hint: consider the memory hierarch and provide as much specific insight as you are able.)

6) Use the Internet to do some research as to the difference between the different lock types used in this lab: pthread_mutex_t and pthread_spinlock_t. Explain the main functional differences in your own words and when you may want to use one vs. the other using the mutex vs. spinlock performance used in this lab as an example.

7) Which parallelism method yielded the best execution times and explain why it was the best both in the context of parallelism and memory hierarchy effects.

6 Lab Report

Name: _____ Score: _____
Due: _____

(Detach and turn this sheet along with any other requested work or printouts)

1. Turn in a hardcopy of your completed inner-loop parallel reduced locking approach source code file (from section 4q).

2. Complete the table below for your program runs.

# of Threads	Row-Parallel	Column-Parallel	Inner-Loop Parallel (Mutex Lock each iteration)	Inner-Loop Parallel (Spinlock each iteration)	Inner-Loop Parallel Best locking scheme + your reduced locking approach.
1					
2					
4					
8					

3. Include answers to all the review questions including a printout of your Excel XY plot.