

# Lab 5 : Threaded Matrix Multiply

September 14, 2016

## Objective :

- Lab 5 is intended to give you some experience with programming using pthreads.

- **The Assignment: Threading Matrix Multiply**

In this assignment you are to write a version of matrix multiply that uses threads to divide up the work necessary to compute the product of two matrices. There are several ways to improve the performance using threads. You need to divide the product in row dimension among multiple threads in the computation. That is, if you are computing  $A * B$  and  $A$  is a  $10 \times 10$  matrix and  $B$  is a  $10 \times 10$  matrix, your code should use threads to divide up the computation of the 10 rows of the product which is a  $10 \times 10$  matrix. If you were to use 5 threads, then rows 0 and 1 of the product would be computed by thread 0, rows 2 and 3 would be computed by thread 1,...and rows 8 and 9 would be computed by thread 4.

- If the product matrix had 100 rows, then each "strip" of the matrix would have 20 rows for its thread to compute. This form of parallelization is called a "strip" decomposition of the matrix since the effect of partitioning one dimension only is to assign a "strip" of the matrix to each thread for computation. Note that the number of threads may not divide the number of rows evenly. For example, if the product matrix has 20 rows and you are using 3 threads, some threads will need to compute more rows than others. In a good strip decomposition, the "extra" rows are spread as evenly as possible among the threads. For example, with 20 rows and 3 threads, there are two "extra" rows ( $20 \bmod 3$  is 2). A good solution will not give both of the extra rows to one thread but, instead, will assign 7 rows to one thread, 7 rows to another, and 6 to the last. Note that for this assignment you can pass all of the  $A$  and  $B$  matrix to each thread.

- What you need to do

Your program must conform to the following prototype:

```
my_matrix_multiply -a a_matrix_file.txt -b b_matrix_file.txt -t  
thread_count
```

where the `-a` and `-b` parameters specify input files containing matrices and `thread_count` is the number of threads to use in your strip decomposition.

The input matrix files are text files having the following format. The first line contains two integers: rows columns. Then each line is an element in row major order. Lines that begin with "#" should be considered comments and should be ignored. Here is an example matrix file

```
3 2
# Row 0
0.711306
0.890967
# Row 1
0.345199
0.380204
# Row 2
0.276921
0.026524
```

This matrix has 3 rows and 2 columns and contains comment lines showing row boundaries in the row-major ordering.

- Your assignment will be graded using a program that generates random matrices in this format (print-rand-matrix.c). You will need to write a function that can read matrix files in this format and to do the argument parsing necessary so that the prototype shown above works properly.
- Your program will need to print out the result of  $A * B$  where A is contained in the file passed via the -a parameter and B is contained in the file passed via the -b parameter. It must print the product in the same format (with comments indicating rows) as the input matrix files: rows and columns on the first line, each element in row-major order on a separate line.
- Your solution will also be timed. If you have implemented the threading correctly you should expect to see quite a bit of speed-up when the machine you are using has multiple processors. For example, on some machine you may witness

```
my_matrix_multiply -a a1000.txt -b b1000.txt -t 1
```

completes in 8.8 seconds when a1000.txt and b1000.txt are both 1000 x 1000 matrices while applying

```
my_matrix_multiply -a a1000.txt -b b1000.txt -t 2
```

the time drops to 4.9 seconds (where both of these times come from the "real" number in the Linux time command).

Note this method of timing includes the time necessary to read in both A and B matrix files. For smaller products, this I/O time may dominate the total execution time. So that you can time the matrix multiply itself during development. A C function has been provided (c-timer.c) that

returns Linux epoch as a double (including fractions of a second) and it may be used to time different parts of your implementation. For example, using this timing code, the same execution as shown above completes in 7.9 seconds with one thread, and 4.1 seconds with two threads. Thus the I/O (and any argument parsing, etc.) takes approximately 0.8 seconds for the two input matrices.

- Hint: Refer to [http://www.cs.ucsb.edu/~rich/class/cs170/labs/mmult/what\\_i\\_did.html](http://www.cs.ucsb.edu/~rich/class/cs170/labs/mmult/what_i_did.html)