CS 565 Multi-threaded Naive Matrix Powers

Charles Recaido

May 2021

1 Background

Lab two is on the multi-threaded naive implementation of square matrix multiplication and matrix powering. The theoretical runtime (or growth) for single threaded naive matrix multiplication is $O(n^3)$. This can easily be understood by analyzing the three for loops in the pseudo code of naive matrix multiplication.

```
squareMatrixMultiply(A, B, dim):
    initialize result
    for i to dim:
        for j to dim:
            for k to dim:
                result[i][j] += A[i][k] * B[k][j]
    return result
```

With multi-threading the theoretical run time for the naive implementation can be as small as $O(n)$. In a perfect simulation for a $n \times n$-dimensional square matrix there would also be n^2-threads. Each thread is responsible for calculating one element of the product matrix and each thread can perform these calculations simultaneously.

```
squareMatrixMultiplyByElement(A, B, dim, row, col):
    initialize result
    for k to dim:
        result[row][col] += A[row][k] * B[k][col]
    return result

squareMatrixMultiplyByRow(A, B, dim, row):
    initialize result
    for j to dim:
        for k to dim:
            result[row][j] += A[row][k] * B[k][j]
    return result

squareMatrixMultiplyByColumn(A, B, dim, col):
    initialize result
    for i to dim:
        for k to dim:
            result[i][col] += A[i][k] * B[k][col]
    return result
```

Of course having n^2 threads is not very realistic when the n-dimension is large due to OS imposed thread limits and actual resources available. A better representation of multi-threaded naive matrix multiplication runtime would be $O(\frac{n^3}{T})$ where T is the number of threads and $1 <= T <= n^2$. A caveat to this and one that will show up later in this report is that having a multi-threaded algorithm is not simple to implement. There is two methods to the madness. One method requires pre-processing. For pre-processing each thread is assigned **specific** jobs (in this case a job is an element, row, or column) to calculate. The second method requires a job queue. Each thread will take a job from the queue to calculate until the queue is empty.

1.1 Pre-processing method

Pro

- No thread communication required to prevent race conditions (a race condition is multiple threads trying to access the same job at the same time). Each thread knows that no other threads will ever attempt the same job.

Con

- There is a considerable amount of pre-processing time required as each thread needs to be assigned a job beforehand.

1.2 Job-queue

Pro

- There is no pre-processing time as jobs don't need to be assigned. Jobs are taken from the queue by a free thread until the queue is empty.

Con

- Thread communication is required. This is because two or more threads may try to access the job queue simultaneously which could result in unwanted behavior. Thus a mutex (similar to a lock) is required each time a thread accesses the queue. Naturally locking then unlocking with a mutex adds time.

Lab two had six different implementations of multi-threaded naive matrix multiplication. The six implementations were divided among contiguous memory and non-contiguous memory. Each memory type had three options where each thread was assigned a single element, an entire row, or an entire column to calculate.

For this project I chose the job queue method. For the static memory by-row and by-column multiplications a simple counter could act as the queue. For static memory by-element I used a queue of indices. As for the linked list example for by-row, by-column, and by-element I used a counter which corresponded to a shift in the linked list to the next product element.

When analyzing my own code for multi-threaded naive matrix multiplication I had additional operations necessary for multi-threading such as if statements to check whether or not any jobs were left to perform, mutexes for various counters and to move about in the linked list structure, and conditional statements to put threads to sleep or wake them up.

My code uses a threadpool (reusing the same threads for multiple jobs (powers)) so it is important to put threads to sleep instead of exiting. Also threads need to sleep rather than waste resources looping if there is no more jobs but some other threads are still working. As an example if I were to multiply a 100 x 100 matrix with 80 threads and I wanted each thread to calculate an entire row. At first 80 threads will take 80 rows. Next 20 threads will take the remaining 20 rows. Thus as this point 60 threads will have no job to do. There are four well known solutions.

- Have those 60 threads loop and do nothing. This is CPU-intensive and not recommended.
- Have those threads exit. This makes sense for only one set of jobs. Should you have multiple sets of jobs a threadpool would be more efficient than creating new threads for every set of jobs.
- Use a conditional wait variable to have threads be put to sleep and woken up when that condition is reached. This is the most common solution when using a threadpool. Semaphores are a similar concept to a conditional wait.
- Put the thread to sleep using a sleep() function. This is not recommended either as it's not precise and can consume valuable time.

2 Results

Comparing all six multi-threaded implementations (Figure 1) it is obvious that threading by-Element was significantly slower than threading by-Row or by-Column. It was mentioned in the previous section that in theory by-Element could be as fast as $O(n)$ and by-Row or by-Column could be as fast as $O(n^2)$. However, to reach these theoretical speeds every job to thread pair would need to be pre-processed and all threads would then need to simultaneously perform these calculations. Additionally for a 1,000 x 1,000 matrix there would need to be 1,000,000 threads for this perfect world.

So why is my code not perfect? This is due to the use of mutexes and if-else statements to keep the integrity of the multiplication problem. By the time a thread could be directed to an element to calculate individually the previous thread would already be done with its element, and the nature of the multiplication becomes more "sequential" rather than "parallel". Each thread in the by-Element implementation had to do 1000 summations but those 1000 summations are extremely quick to perform. Even calculating 100,000 summations is fast for a modern computer. Thus an extremely large dimensional matrix is required to reap the benefits of the by-Element method.

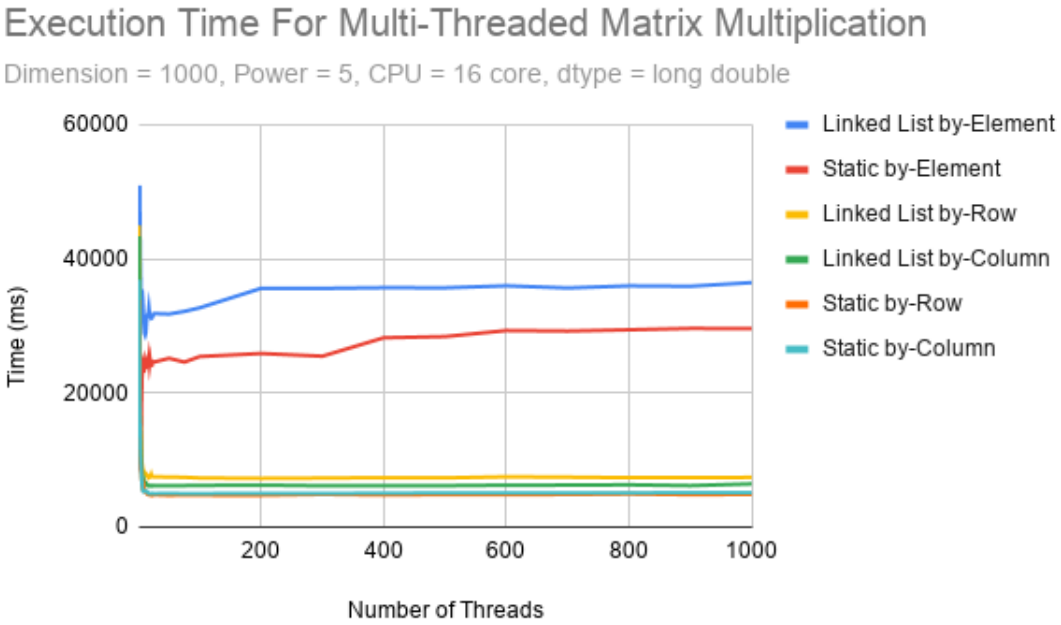

Figure 1: Multi-threaded naive matrix multiplication for 1 to 1000 threads

As shown in Figure 1 all the exciting changes occur when the thread count is small. Zooming in on Figure 1 (Figure 2) a sweet spot in execution seems to be about 7-8 threads. On Figure 1 and Figure 2 it's mentioned that my CPU is 16 core. This is false and my CPU is actually 8 cores with 16 logical processors. Each core uses a concept known as hyperthreading. Well technically only Intel processors have hyperthreading whereas newer AMD processors (Zen architecture) uses Simultaneous Mult-threading (SMT) and I'm using an AMD processor. But hyperthreading and SMT are very similar. SMT allows for two logical processors per physical core. A logical processor can be interrupted, has the ability to execute a thread, and/or can be halted. A pair of logical processors on a core share resources such as the cache, and execution engine.

Going back to Figure 2 I suspect one of two things. Either SMT technology is not as effective for matrix multiplication and 7-8 threads is the sweet spot since I have an 8 core CPU or similar to the by-Element method, the by-Row and by-Column methods are hampered by various mutexes, and if-else statements, just not to the extent as by-Element. Thus to see a 16 thread sweet spot I may need to drastically increase the matrix dimensions. The graph seems to flatline which I attribute to sequential thread execution. Imagine executing multiple threads on a single core. $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5$ is just as fast as $1 \rightarrow 2 \rightarrow 1 \rightarrow 2 \rightarrow 1$. Both the 2-thread and 5-thread are happening sequentially whilst carrying the same sequence of instructions.

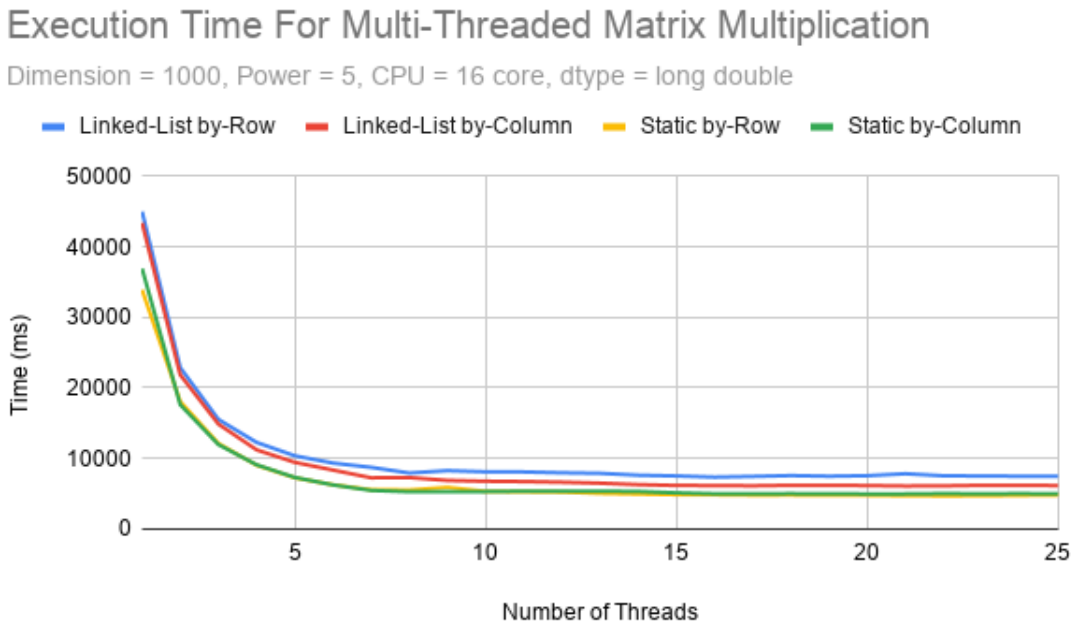

Figure 2: Multi-threaded naive matrix multiplication for 1 to 25 threads

In Figure 2 it is shown that by-Column is slightly faster than by-Row. I must mention that while C performs calculations as row-ordered I designed my matrices to be represented in column order. So my by-Column method is really performing as the standard C by-Row method. So let's pretend I didn't get all funky with my matrix creation. Row order is faster in C because C stores the rows sequentially in memory. If variables are close by in memory then it's more likely they can be stored on the cache together, which means the CPU can access the data it needs much faster. By-row calculations benefit that the sequential memory access is in the inner for loop whereas by-Column has each element separated by an entire row of values and should be considered for the outer for loop. Assuming memory is static for a 1000 x 1000 array the 1st and 2nd column element are 999 spaces apart and the 3rd column element is another 999 spaces, whereas the 1st and 2nd row element and 2nd and 3rd row element are adjacent. Of course a Linked List is a more extreme example of memory separation.

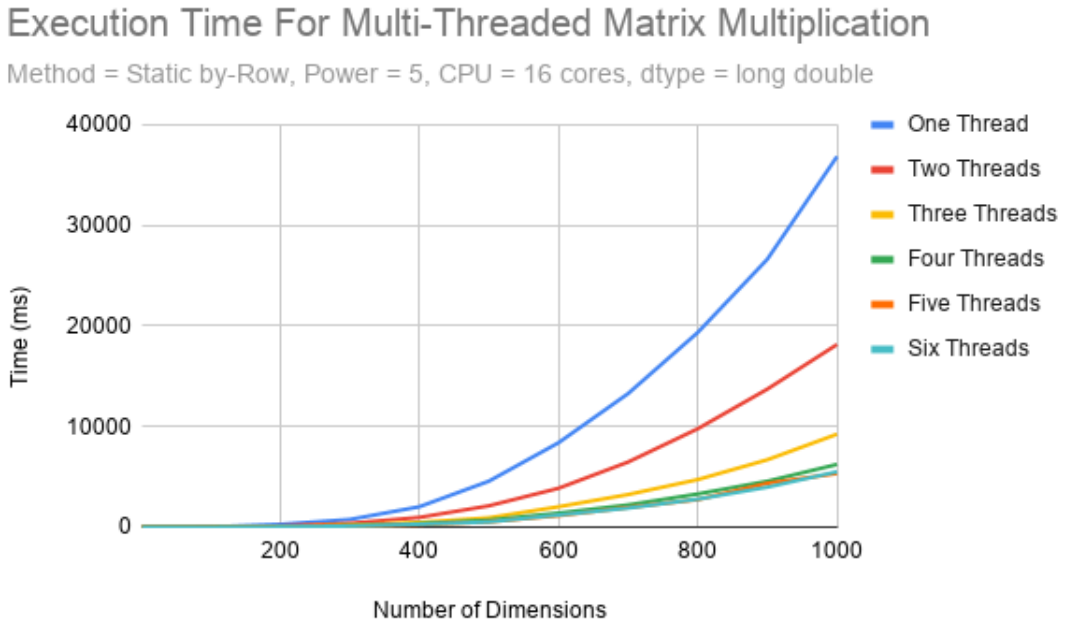

Figure 3: Matrix multiplication with 1 to 8 threads

In Figure 3 we examine the sweet spot of 8 threads. The growth of the single threaded approach was much

faster than that of four or more threads. This is reasonable because given enough number of dimensions the single threaded approach should grow at $O(n^3)$ whereas the multi-threaded options gradually get closer and closer to $O(n^2)$ as the number of threads increase. One thing to consider is that thread creation time is not included in this graph.

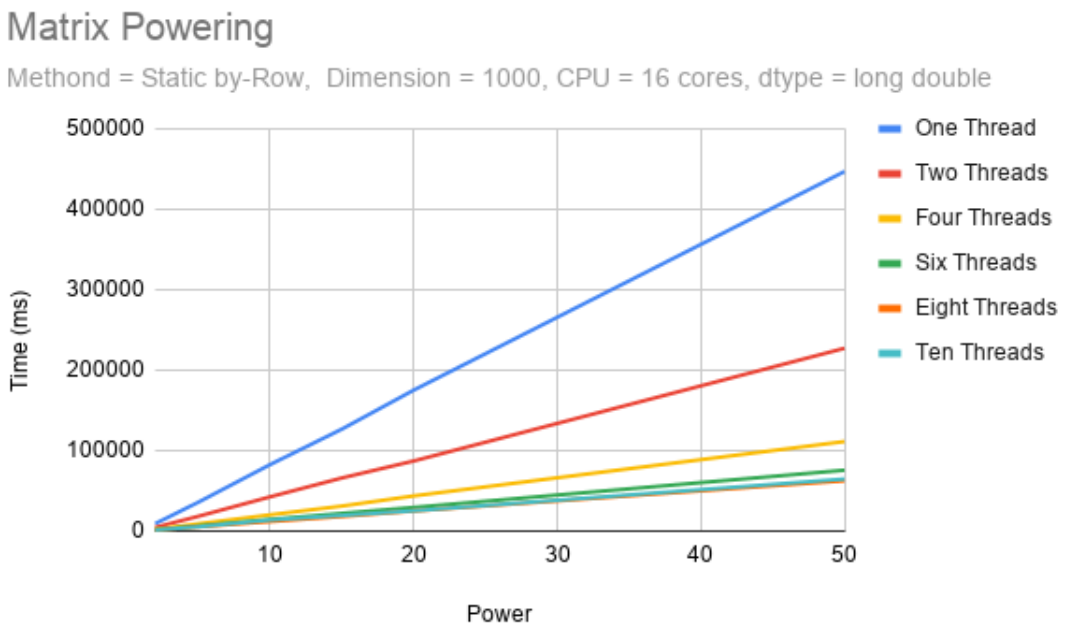

Figure 4: Calculating Matrix Powers for 1 to 8 threads

In figure 4 eight threads still had the fastest times when performing matrix multiplication for several power iterations. A thread pool was maintained so that new threads would not need to be created for each power. Each line is linear because the total number of operations is simply the matrix multiplication times the power. The long double data type was used because powers grow fast. However, long double calculations are much faster when compared to a custom data structure and/or a large number representation in string form.

2.1 Time Functions

In Figure 5 it is shown that time() and gettimeofday() result in similar times whereas clock() is much slower. This is because clock() measures CPU cycles rather than wall time and each thread uses several CPU cycles. Dividing by the CLOCKS_PER_SEC macro translates CPU time to second time. The next step would be to approximate the clock() function output by dividing the time by the number of threads as shown in Figure 6.

time() and gettimeofday() both measure wall time, however, time() is limited to a precision of seconds whereas gettimeofday() has precision in the microseconds. Adjusted clock() time had the smallest times. This is because clock() only measures the CPU cycles relative to the process at hand. A time test was done for a sleep(5) call. The results of the test was that time() and gettimeofday() both returned five seconds whereas clock() returned zero seconds. Overall I found gettimeofday() to be the best timing measurement function for this particular lab.

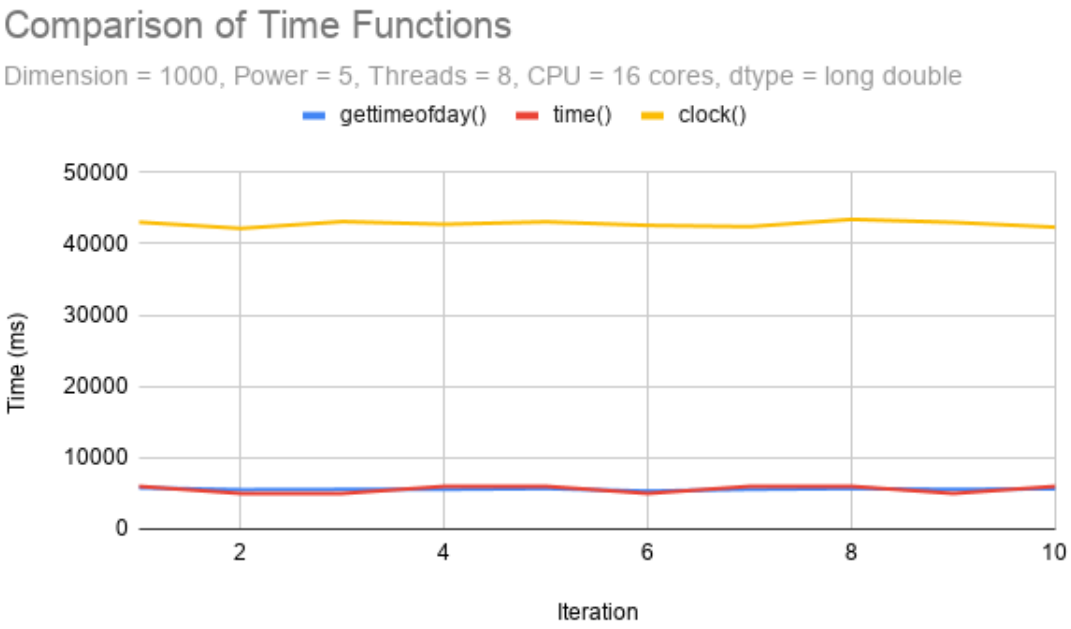

Figure 5: Ten iterations of the same test for each time function

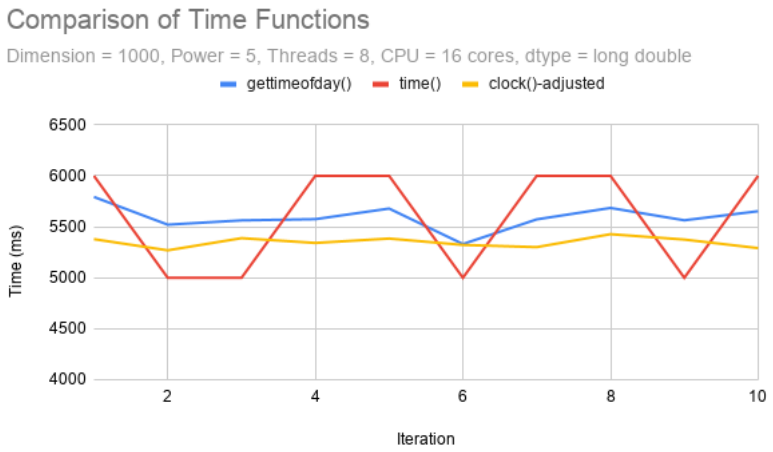

Figure 6: Ten iterations of the same test for each time function

3 Conclusion

This lab had many interesting outcomes such as having more threads than core count wasn't as big as a burden as I expected, however, the thread creation time was not included. Another surprise was that the by-Element performed more poorly than I expected. When designing the linked list script I had to use a lot of mutexes and if-else statements as I was moving about in a global linked list and I didn't want multiple threads trying to move to the same position in the linked list. The most difficult portion of the lab was figuring out thread synchronisation using the conditional_wait, conditional_signal, and conditional_broadcast functions. Compared to single threaded matrix multiplication the results of multi-threading are fantastic and parallel calculations allow for a more efficient workflow. The threadpool was certainly a good concept to focus on and I was able to understand thread communication via signals. If I had more time or was paid to develop a multi-threaded matrix application I would work on the job queue and thread communication as an attempt to make by-Element faster than by-Row or by-Column.