CS/EE 451 PHW2

Name: Jack Li
ID: 1321-1056-14

P1a

Running on Mac OS once per p value, where p x p is the number of threads

For p = 1:

```
lordchingiss@MacBook-Pro PHW2 % ./p1a
Execution time = 2007.956288 sec
C[100][100]=879616000.000000
```

For p = 2:

```
lordchingiss@MacBook-Pro PHW2 % ./p1a 2
Execution time = 482.460627 sec
C[100][100]=879616000.000000
```

For p = 4:

```
lordchingiss@MacBook-Pro PHW2 % ./p1a 4
Execution time = 239.191346 sec
C[100][100]=879616000.000000
```

For p = 8:

```
lordchingiss@MacBook-Pro PHW2 % ./p1a 8
Execution time = 398.903798 sec
C[100][100]=879616000.000000
```

For p = 16:

```
lordchingiss@MacBook-Pro PHW2 % ./p1a 16
Execution time = 417.738786 sec
C[100][100]=879616000.000000
```

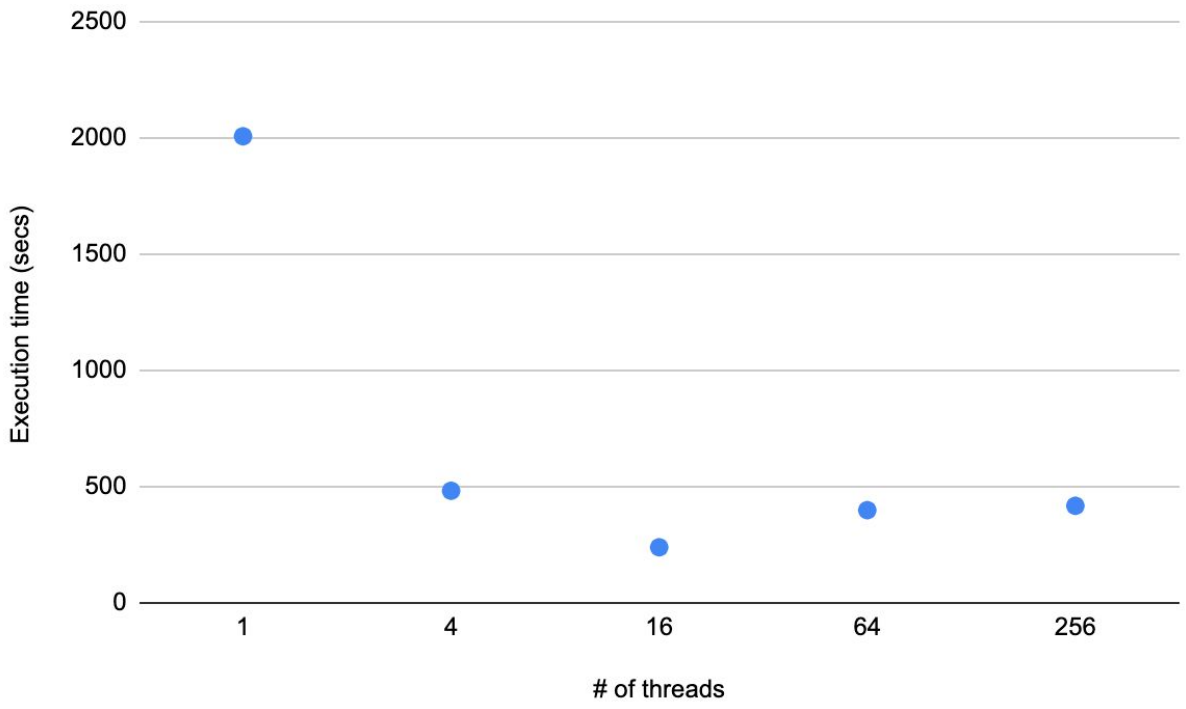

Figure 1a: Plot of execution times vs # of threads for local output variables

From the above figure, the optimal # of threads for parallel MM of 4K x 4K is 16. Notice that using too many threads actually hurts performance due to the large overhead of thread management.

I partitioned the computation works by having thread(i, j), 0 <= i, j < n work on blocks of size n/p. More specifically, thread(i, j) is responsible for C(i * n/p, j * n/p) to C((i + 1)* n/p - 1, (j + 1)* n/p - 1).

P1b

Running on Mac OS once per p value, where p x p is the number of threads

For p = 1:

```
lordchingiss@MacBook-Pro PHW2 % ./p1b
Execution time = 168.666530 sec
C[100][100]=879616000.000000
```

For p = 2:

```
lordchingiss@MacBook-Pro PHW2 % ./p1b 2
Execution time = 208.039217 sec
C[100][100]=879616000.000000
```

For p = 4:

```
lordchingiss@MacBook-Pro PHW2 % ./p1b 4
Execution time = 207.306426 sec
C[100][100]=879616000.000000
```

For p = 8:

```
lordchingiss@MacBook-Pro PHW2 % ./p1b 8
Execution time = 217.451583 sec
C[100][100]=879616000.000000
```

For p = 16:

```
lordchingiss@MacBook-Pro PHW2 % ./p1b 16
Execution time = 206.069416 sec
C[100][100]=879616000.000000
```

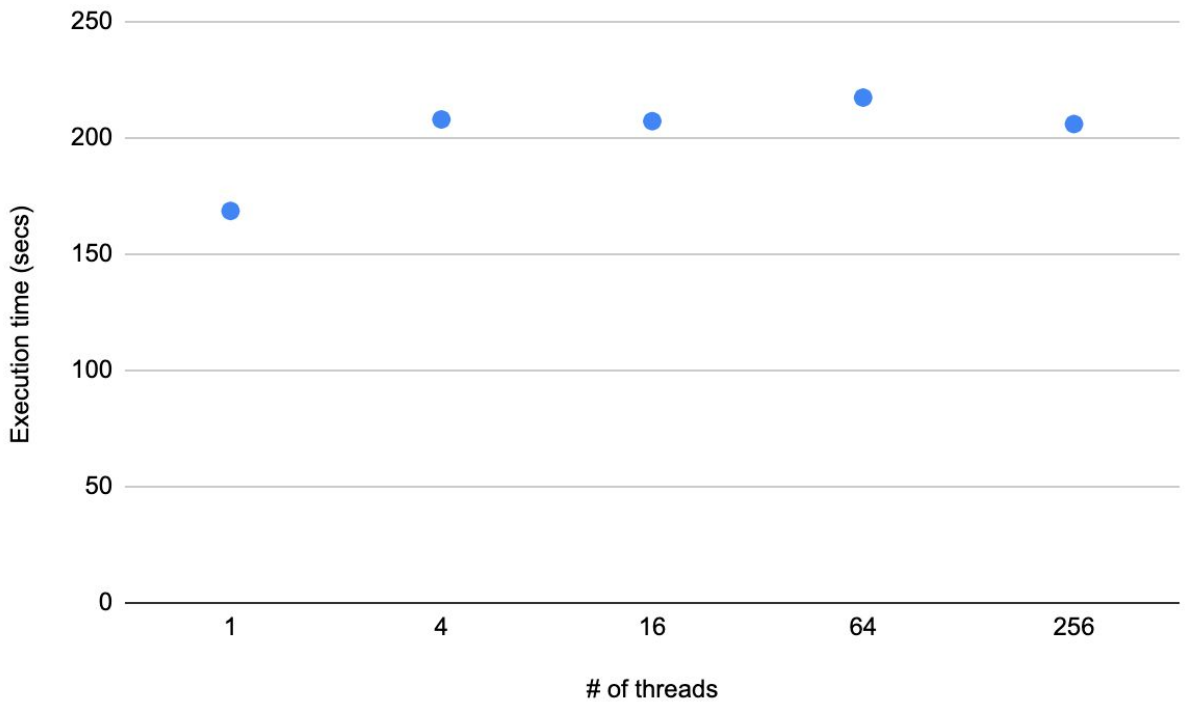

Figure 1b: Execution time vs. # of threads for shared output variables

The single-threaded version runs the fastest in this case but still around the same as the other thread sizes. Whenever a lock mechanism is needed to protect a critical section, using multiple threads shouldn't improve performance since multiple threads are contending for the lock.

To distribute the work among threads, each thread was assigned a block of size n/p x n/p in input A. Each thread would update C(i, k) for k = 0 to n - 1 by multiplying A(i, j) with B(j, k).

Overall, the execution times for 1b were fasting than their 1a counterparts, in terms of # of threads. Another difference is the shape of the plots; 1a is an upwards pointing parabola while 1b is a downwards pointing parabola. This is related to how multi-threading has improved serial execution of MM with local output variables but the opposite can be said for MM with shared output variables. Analyzing the serial code of both versions, one explanation for why using shared output variables is faster is that all matrices are accessed in row-major order and is thus more cache friendly than the local output variable counterpart (B is accessed in column-major order).

P2

Running on Mac OS once per p value, where p is the number of threads

Serial execution (p = 1 by default):

```
lordchingiss@MacBook-Pro PHW2 % ./p2
Execution time = 0.831262 sec
```

For p = 4:

```
lordchingiss@MacBook-Pro PHW2 % ./p2 4
Execution time = 0.262389 sec
```

For p = 8:

```
lordchingiss@MacBook-Pro PHW2 % ./p2 8
Execution time = 0.168278 sec
```

Running with 4 threads is about 3.2x faster than the serial version while running with 8 threads is about 5x faster.

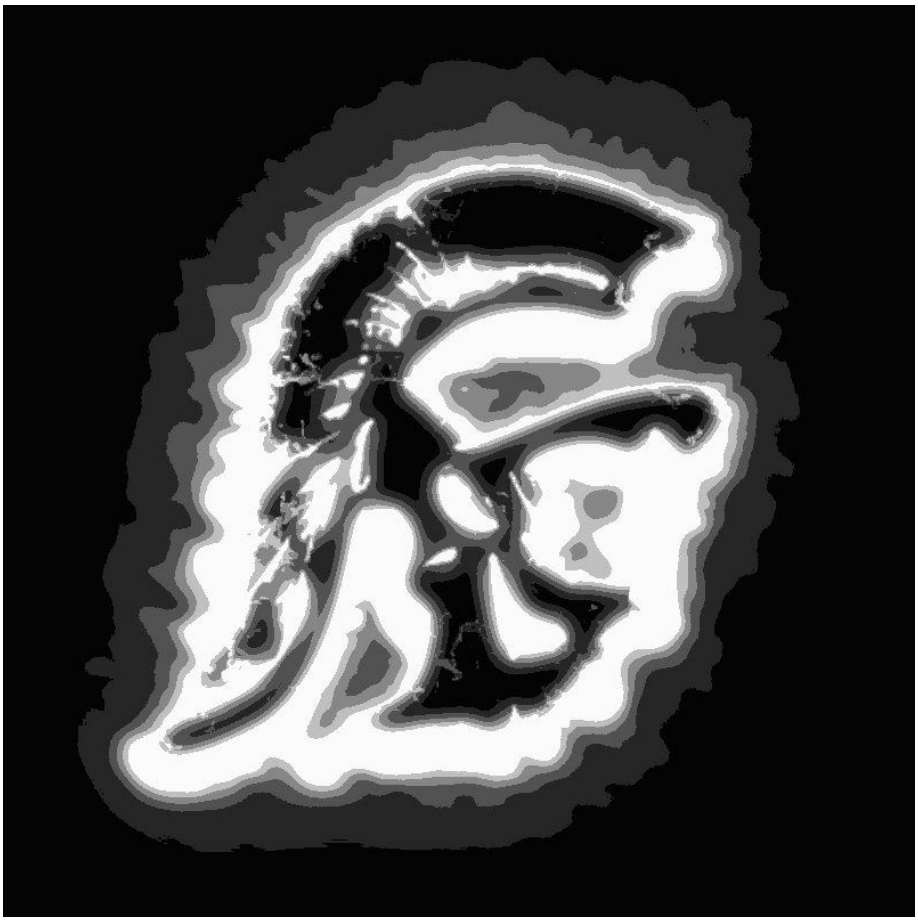

Figure 2: Output image