SIT315 M2.T1P: Parallel Matrix Multiplication with pthreads and OMP

Codey Funston | S222250824

Part : Program Parallelisation and Decomposition

A matrix is simply a vector of vectors, and this is how I implemented them in my program. However, it is gets complex if you try and move through a matrix left to right, top to bottom if you want to end at an entry like (3, x) where x is not the last column. Therefore, in my implementation for the *pthreads* section I assigned a set number of rows to each thread as they "requested" it and worked through each column in those rows. This technique was used for both filling and multiplying matrices, and the only shared variable that had to be locked was the index. When the index is shared properly, matrix access can be done simultaneously like in the vector addition program since the entries being accessed are all separate.

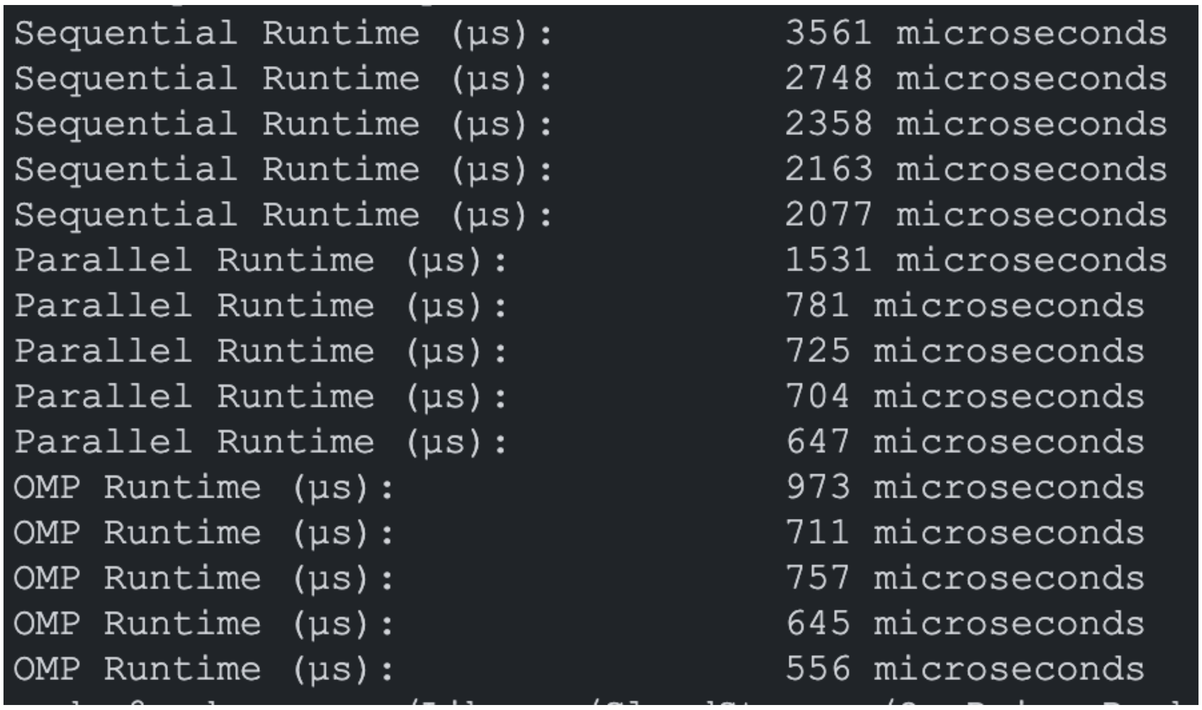
Part 2: Evaluation of pthreads and OMP Performance Against Sequential Version

The programs were tested with matrix dimension sizes of 100, 500, 1000, and 2000. Initially the parallel versions used partition/chunk sizes of $\frac{number\ of\ elements\ in\ matrix}{number\ of\ cores}$, then it was changed to $\frac{number\ of\ elements\ in\ matrix}{20}$ so that that *pthreads* implementation with its continual chunk requests from each thread could be taken advantage of. The terminal output was produced from the shell script below. I set the dimensions in each of the three files, ran this script, then changed the dimensions and repeated.

```
#!/bin/zsh

# Compiles each version.
g++-14 -std=c++11 /Users/codey/od/315/Module2/pt1/seq/Sequential.cpp -o
seqx
g++-14 -std=c++11 /Users/codey/od/315/Module2/pt1/par/Parallel.cpp -o
parx -lpthread
g++-14 -std=c++11 /Users/codey/od/315/Module2/pt1/omp/OMP.cpp -o ompx -
fopenmp

# Run sequential version.
for _ in {1..5}; do
    ./seqx | grep time
done

# Run pthreads version.
for _ in {1..5}; do
    ./parx | grep time
done

# Run OMP version.
for _ in {1..5}; do
    ./ompx | grep time
done
```

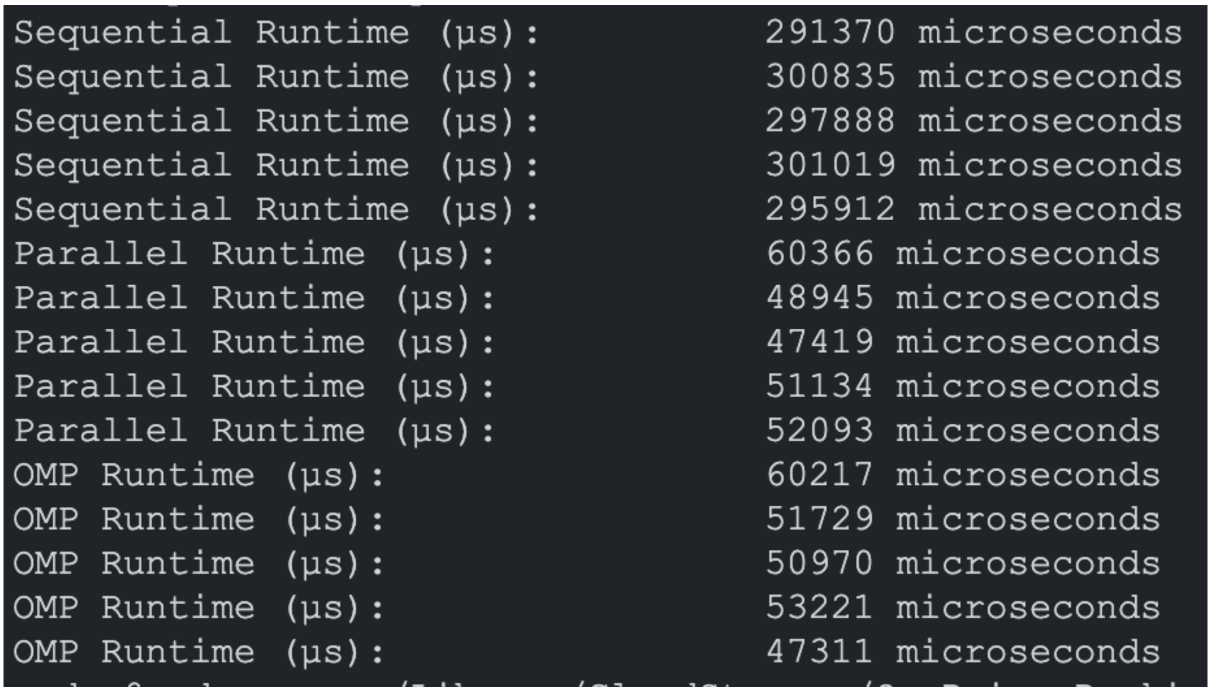
The results for each dimension were:

100.



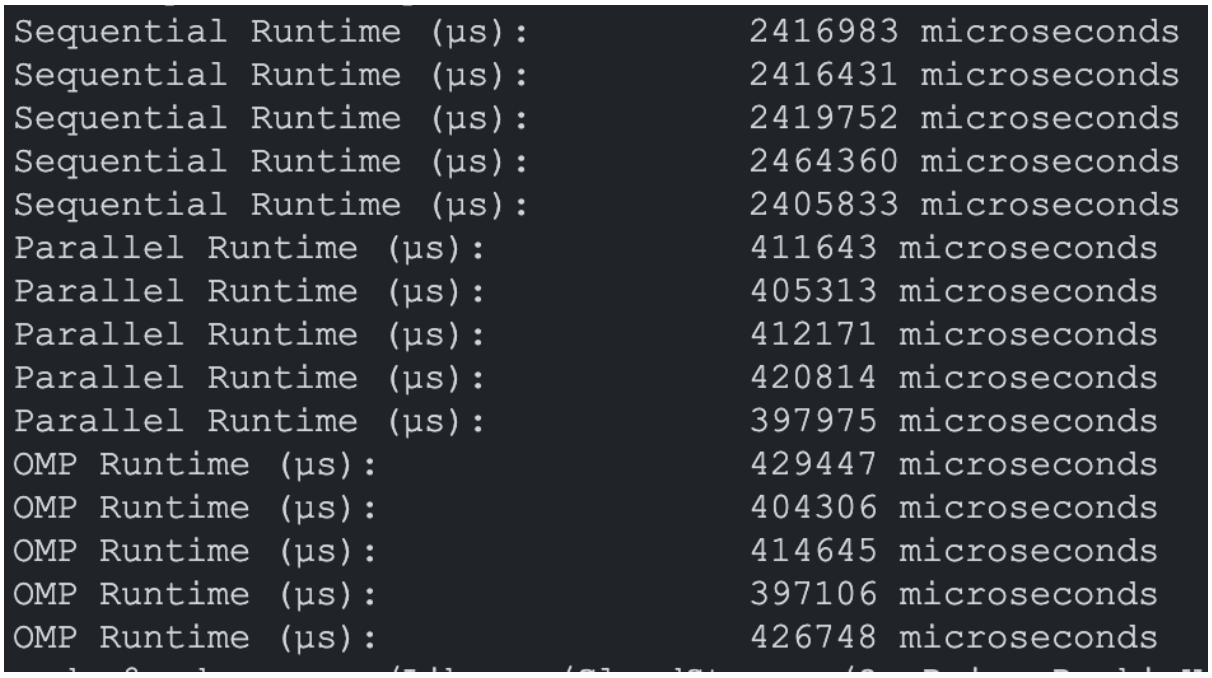
```
Sequential Runtime (µs):          3561 microseconds
Sequential Runtime (µs):          2748 microseconds
Sequential Runtime (µs):          2358 microseconds
Sequential Runtime (µs):          2163 microseconds
Sequential Runtime (µs):          2077 microseconds
Parallel Runtime (µs):            1531 microseconds
Parallel Runtime (µs):            781 microseconds
Parallel Runtime (µs):            725 microseconds
Parallel Runtime (µs):            704 microseconds
Parallel Runtime (µs):            647 microseconds
OMP Runtime (µs):                 973 microseconds
OMP Runtime (µs):                 711 microseconds
OMP Runtime (µs):                 757 microseconds
OMP Runtime (µs):                 645 microseconds
OMP Runtime (µs):                 556 microseconds
```

500.

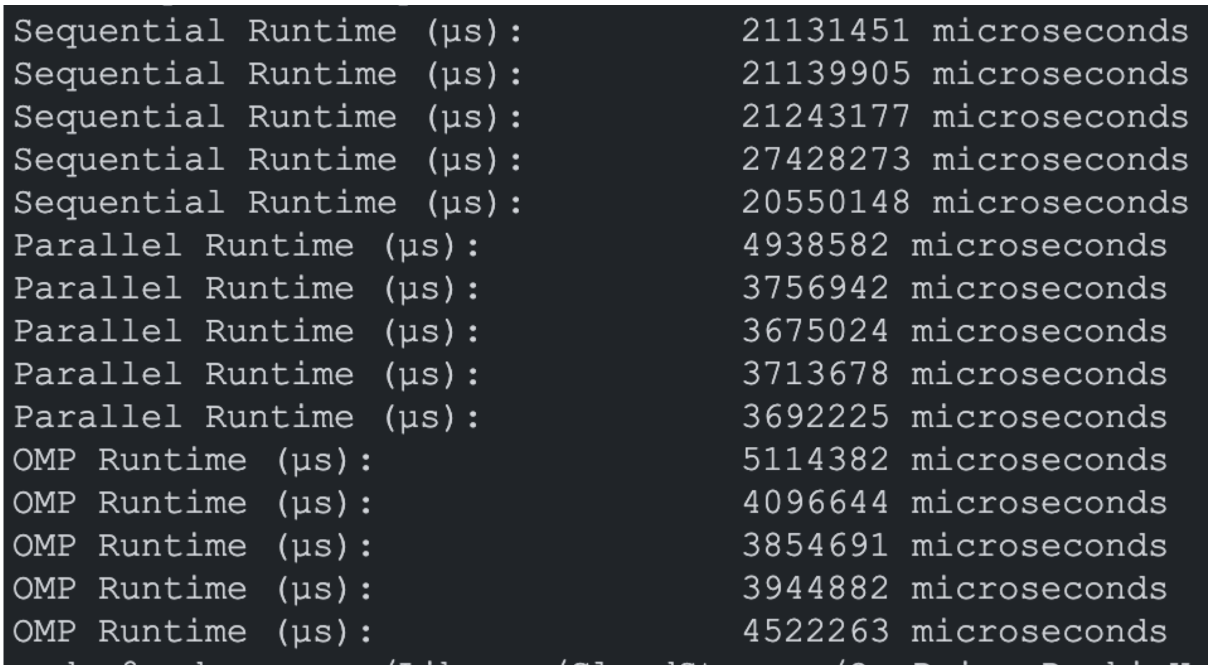


```
Sequential Runtime (µs):          291370 microseconds
Sequential Runtime (µs):          300835 microseconds
Sequential Runtime (µs):          297888 microseconds
Sequential Runtime (µs):          301019 microseconds
Sequential Runtime (µs):          295912 microseconds
Parallel Runtime (µs):            60366 microseconds
Parallel Runtime (µs):            48945 microseconds
Parallel Runtime (µs):            47419 microseconds
Parallel Runtime (µs):            51134 microseconds
Parallel Runtime (µs):            52093 microseconds
OMP Runtime (µs):                 60217 microseconds
OMP Runtime (µs):                 51729 microseconds
OMP Runtime (µs):                 50970 microseconds
OMP Runtime (µs):                 53221 microseconds
OMP Runtime (µs):                 47311 microseconds
```

1000.

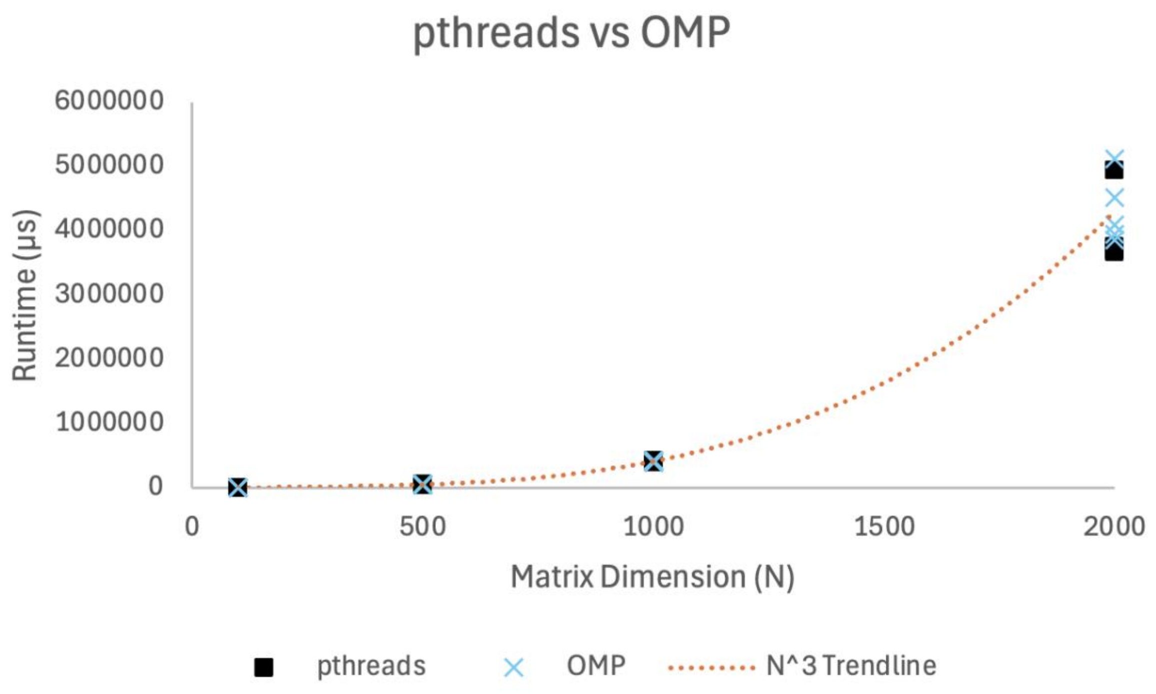

```
Sequential Runtime (µs):          2416983 microseconds
Sequential Runtime (µs):          2416431 microseconds
Sequential Runtime (µs):          2419752 microseconds
Sequential Runtime (µs):          2464360 microseconds
Sequential Runtime (µs):          2405833 microseconds
Parallel Runtime (µs):            411643 microseconds
Parallel Runtime (µs):            405313 microseconds
Parallel Runtime (µs):            412171 microseconds
Parallel Runtime (µs):            420814 microseconds
Parallel Runtime (µs):            397975 microseconds
OMP Runtime (µs):                 429447 microseconds
OMP Runtime (µs):                 404306 microseconds
OMP Runtime (µs):                 414645 microseconds
OMP Runtime (µs):                 397106 microseconds
OMP Runtime (µs):                 426748 microseconds
```

2000.


```
Sequential Runtime (µs):          21131451 microseconds
Sequential Runtime (µs):          21139905 microseconds
Sequential Runtime (µs):          21243177 microseconds
Sequential Runtime (µs):          27428273 microseconds
Sequential Runtime (µs):          20550148 microseconds
Parallel Runtime (µs):            4938582 microseconds
Parallel Runtime (µs):            3756942 microseconds
Parallel Runtime (µs):            3675024 microseconds
Parallel Runtime (µs):            3713678 microseconds
Parallel Runtime (µs):            3692225 microseconds
OMP Runtime (µs):                 5114382 microseconds
OMP Runtime (µs):                 4096644 microseconds
OMP Runtime (µs):                 3854691 microseconds
OMP Runtime (µs):                 3944882 microseconds
OMP Runtime (µs):                 4522263 microseconds
```

The matrix files are much too large to put in a word document, but they can be viewed if necessary, here with N = 100:

https://github.com/cfeng44/SIT-315/tree/main/Module2/pt1/matrix_outputs



This graph shows that the runtime is close to N^3 even when paralellising, this makes sense since in doing so we are not changing the structure of the code ie any data structures or algorithms, we are just assigning threads to do the same thing, only changing the runtime complexity to $O(\frac{1}{c}N^3)$, where c is a constant (5 in this case). Therefore, the program is still in $O(\,N^3)$.

Looking at the results both *pthreads* and OMP achieve a speed up of roughly 5 times. This is as expected given that filling and multiplying matrices is a very parallelisable task. Also I set the OMP version to do dynamic scheduling with the same chunk size and number of threads as in the *pthreads* one to see how they compared. Based on rearranging Amdahl's law tells us that the proportion that is done in parallel with my 11 cores is:

$$S = \frac{1}{(1-P) + \frac{P}{N}}$$

$$5 \approx \frac{1}{(1-P) + \frac{P}{11}}$$

$$5 \approx \frac{1}{\frac{11 - 11P + P}{11}}$$

$$55 - 55P + 5P \approx 11$$

$$\therefore P \approx 0.88$$

When using a smaller number of threads, which was 2, 4, 6, 8 here, the respective speed increases are predicted to be 1.78, 2.94, 3.75, and 4.35. Since the *pthreads* and OMP versions showed very similar results earlier I ran the test with the *pthreads* version with a dimension of 2000 and got the following:

Average runtime from sequential program: 22298590

2 threads: 10217577 = 2.18 speed up
4 threads: 5732758 = 3.89 speed up
6 threads: 4809263= 4.64 speed up
8 threads: 4175345 = 5.34 speed up

The is roughly in line with the theoretical values and makes sense since the program is such that adding more threads is more efficient, especially with a dimension as large as 2000.