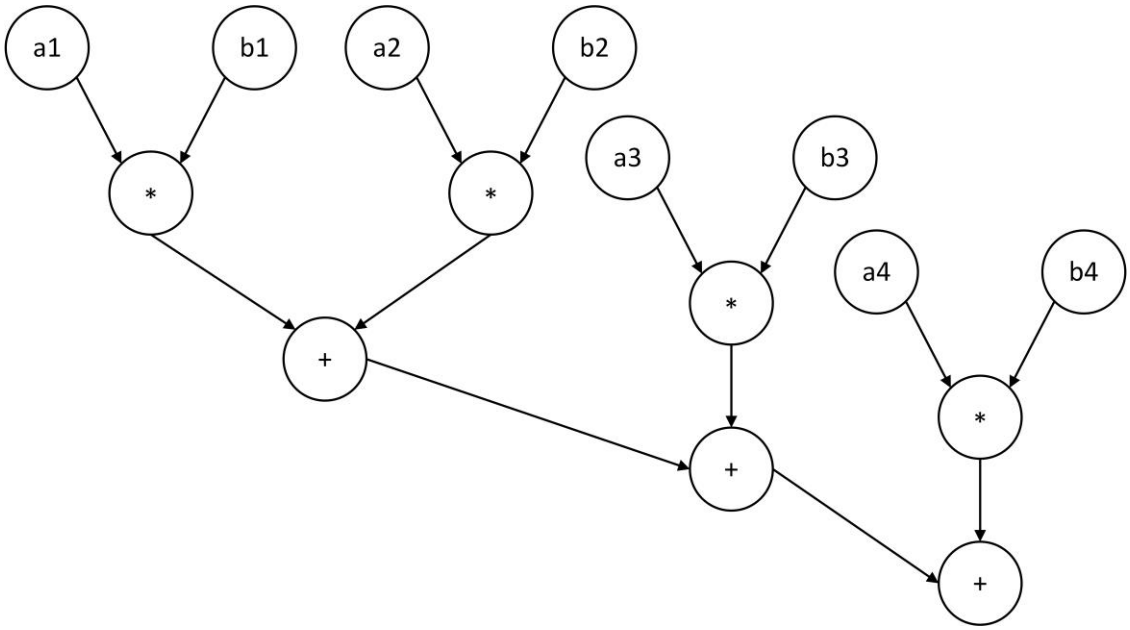
HPC | HW2 | Alireza Rafiei

1. Work/Depth Model (10 pts)

1.1 Draw the DAG of algorithm to compute vector dot product for vectors of length 4 via sequential algorithm. What are the work and depth for this case? Generalize the derivation to vector of size n. What is the average available parallelism?

To calculate the dot product of two vectors A=[a1,a2,a3,a4] and B=[b1,b2,b3,b4], the sequential algorithm would perform the following steps:
Compute a1×b1, a2×b2, 3a3×b3, and 4a4×b4 sequentially, and
sum all the results: (a1×b1)+(a2×b2)+(a3×b3)+(a4×b4) sequentially.
The Directed Acyclic Graph (DAG) for this algorithm would look like this:



Work and Depth for Vectors of Length 4

Work: The "work" is the total number of operations performed. For vectors of length 4, there are 4 multiplications and 3 additions, totaling **7** operations.

Depth: The "depth" is the length of the longest chain of dependent operations. The longest chain involves one multiplication and three additions, making the depth **4**.

Generalization to Vectors of Size n

Work: For vectors of size n, we would have n multiplications and $n-1$ additions, giving a total work of $n+(n-1)=$ **$2n-1$** operations.

Depth: For vectors of size n, the longest chain of operations would include one multiplication and $n-1$ additions, making the depth **n**.

Average Available Parallelism

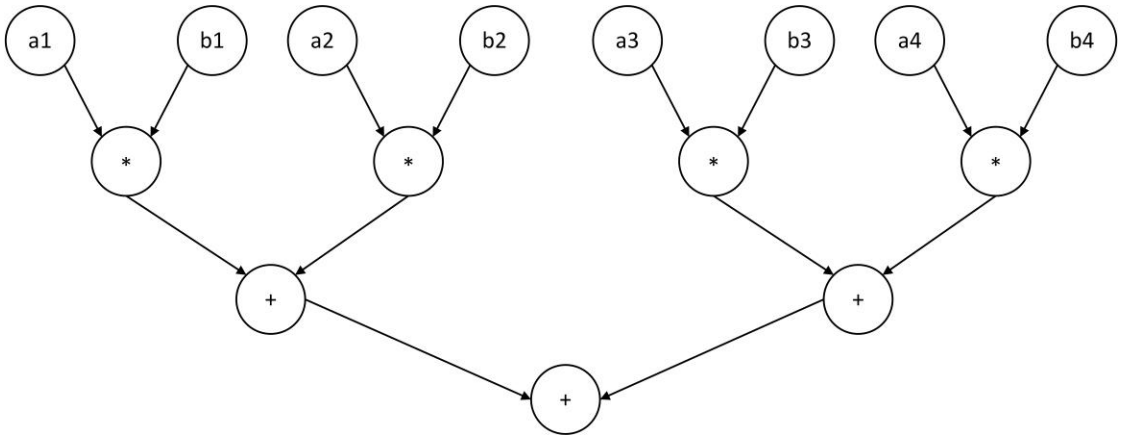
The average available parallelism is calculated as the total work divided by the depth. For vectors of size *n*, this would be:

Average Available Parallelism = $\frac{2n-1}{n} = 2 - \frac{1}{n}$ for n = 4: $\frac{7}{4}$

As *n* gets large, the average available parallelism approaches 2, indicating that, on average, about two operations can be performed in parallel.

1.2 Draw the DAG of algorithm to compute vector dot product for vectors of length 4 by summing in parallel. What are the work and depth for this case, then generalize the derivation to vector of size n. What is the average available parallelism?

The DAG for this algorithm would look like this:



Work and Depth for Vectors of Length 4

Work: Similar to the sequential algorithm, there are 4 multiplications. However, the summing can be done in parallel, requiring only 2 additions to sum *s*1 and *s*2, and 1 addition to sum *s*1+*s*2 to get the final result. So, the total work is 4+2+1= **7** operations.

Depth: The longest chain of dependent operations starts with a multiplication and then involves two additions. Therefore, the depth is 1+2= **3**.

Generalization to Vectors of Size n

Work: The work remains the same as in the sequential algorithm, which is **$2n-1$** operations (since we still need *n* multiplications and $n-1$ additions).

Depth: The depth of the operation is significantly reduced when using a parallel summing algorithm. For a vector of size *n*, the depth is $\log_2(n)$ for the summing part, plus 1 for the multiplication. Therefore, the depth is **$\log_2(n)+1$**.

Average Available Parallelism

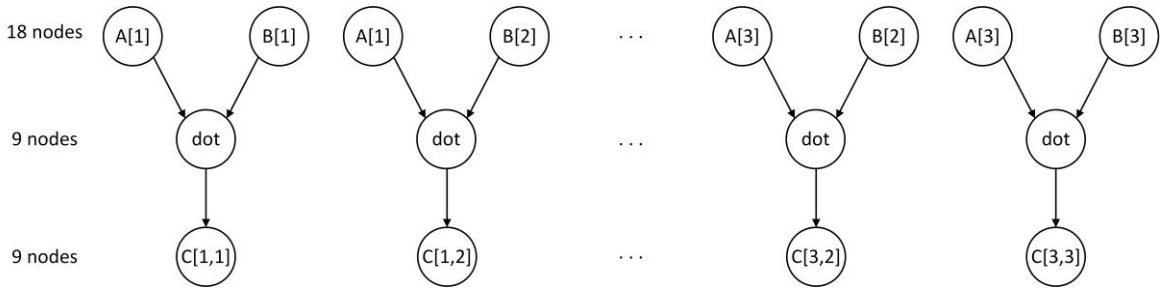
The average available parallelism is calculated as the total work divided by the depth. For vectors of size *n*, this would be:

Average Available Parallelism = $\frac{\boldsymbol{2n-1}}{\boldsymbol{log_2(n)+1}}$ for n = 4: $\frac{7}{3}$

As *n* gets large, the depth grows logarithmically, making the average available parallelism increase. This indicates that the parallel algorithm can effectively utilize more parallel resources as the size of the vector increases.

1.3 Extending 1.2, draw the DAG for a parallel square matrix-matrix multiplication with three rows and three columns. (Assume algorithm uses a simple double loop over the elements of the output array, and the vector dot-product routine from 1.2.) Derive the depth and work for the 4 × 4 case. Generalize your derivation to the case of multiplying two matrices of size n × n.

The DAG for this algorithm would look like this:



Depth and Work for 4 × 4 Case

Work: For a 4×4 matrix multiplication, we have 16 dot products, and each dot product involves 7 operations (from the previous question). Therefore, the total work is 116×7=**112** operations.

Depth: In a 4×4 case, each dot product has a depth of 2. Since the dot products themselves can be computed entirely in parallel, the depth of the entire operation is the same as the depth of a single dot product, which is **2**.

Generalization to n × n Matrices

Work: For $n \times n$ matrices, we have n^2 dot products, and each dot product has a work of $2n-1$ operations. So, the total work is $n^2 \times (2n-1) = \boldsymbol{2n^3-n^2}$.

Depth: If we consider the ideal number of parallel processors, the depth for the parallel square matrix-matrix multiplication would be independent of n and a constant of **2**. Otherwise, each dot product for an $n \times n$ matrix has a depth of $\log_2(n)+1$, so the depth of the entire operation is the same as the depth of a single dot product, which is $\boldsymbol{\log_2(n)+1}$.

By generalizing to $n \times n$ matrices, we can see that the algorithm is highly parallelizable, particularly as *n* grows, which is reflected in the relatively low depth compared to the amount of work done.

1.4 Given the following task DAG, what are the work, depth, and average available parallelism? What is the maximum achievable speed up? What is the minimum number of processes needed to achieve maximum speed up? What is the maximum speed up with a) 2, and b) 8 processors?

Work and Depth

Work: The "work" is the total number of operations performed. For the given task DAG, there are **14** operations.

Depth: The "depth" is the length of the longest chain of dependent operations. For the given task DAG, the longest chain is **7**.

Average Available Parallelism

The average available parallelism is calculated as the total work divided by the depth. For the given task DAG, this would be:

Average Available Parallelism = $\frac{14}{7} = 2$

The maximum achievable speed up is $\frac{14}{7} = 2$

The minimum number of processes needed to achieve maximum speed up is 8 as just the dependent tasks to the first node can be parallelized

The maximum speed up

a) 2 processors: 1.5 as $\frac{1}{\frac{6}{15}+\frac{(1-\frac{6}{15})}{2}} = \frac{15}{10}$

b) 8 processors: 2 as it provides the maximum speed up

2. Parallel Algorithm Analysis (10 pts)

2.1 As presented in Week 5 Lecture 1, we can compute the cost or T_{all} by multiplying the number of processors with the parallel time. Re-analyze the algorithm in problem 1.3 with matrix size NxN and processor count of p for cost, speed up, efficiency, and overhead (as expressions of N and p). Is the algorithm cost optimal?

I presume that computations are uniformly allocated across processors and the processors are completely ideal, so:

Sequential time $T_s = N \times N \times N = N^3$

Parallel time $T_p = \frac{N}{\sqrt{p}} \times \frac{N}{\sqrt{p}} \times N = \frac{N^3}{p}$

Speedup $S = \frac{T_s}{T_p} = p$

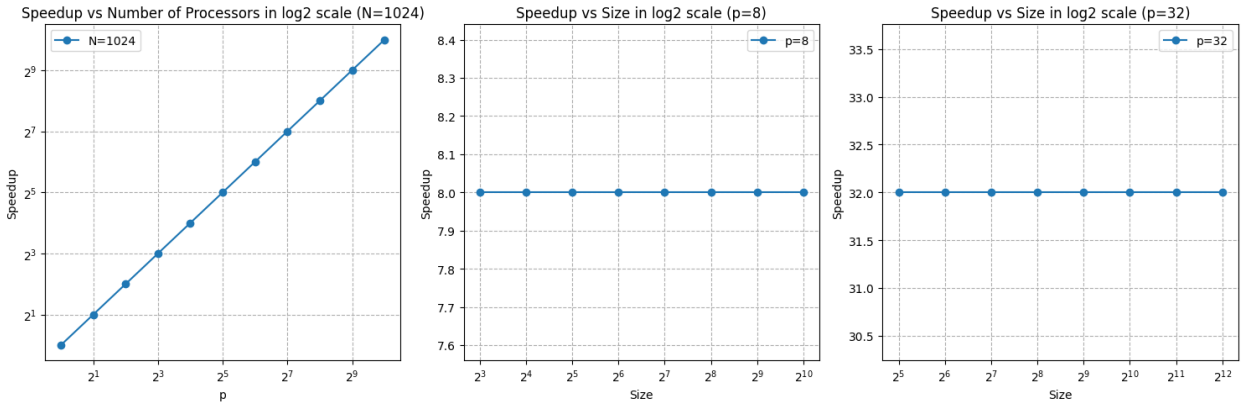
Efficiency $E = \frac{S}{p} = 1$

Overhead $F = pT_p - T_s = 0$

An algorithm is cost-optimal if the total cost of solving a problem in parallel is asymptotically the same as solving it serially. Given our computations, the cost of the parallel algorithm is $O(N^3)$ and the serial cost is also O(N3). Thus, the parallel matrix-matrix multiplication algorithm is cost-optimal.

2.2 Let N = 1024, plot the speed up of the algorithm in 2.1 as p increases from 1 to 1024 in powers of 2. Next fix p to 8, plot the speed up of the algorithm as N increases from 8 to 1024. Repeat the second plot for p = 32 and N increases from 32 to 4096.

The code is avaiable in q2_2.py.



2.3 Repeat the second plot for p = 32 and N increases from 32 to 4096. Instead of the algorithm in 1.3, assume the algorithm uses sequential vector dot product and parallelized by the outer loop (over rows of the first matrix). Analyze the algorithm with matrix size NxN and processor count of p for cost, speed up, efficiency, and overhead (as expressions of N and p). Is the algorithm cost optimal?

I presume that computations are uniformly allocated across processors and the processors are completely ideal, so:

Sequential time $T_s = N \times N = N^2$

Parallel time $T_p = \frac{N}{\sqrt{p}} \times \frac{N}{\sqrt{p}} = \frac{N^2}{p}$

Speedup $S = \frac{T_s}{T_p} = p$

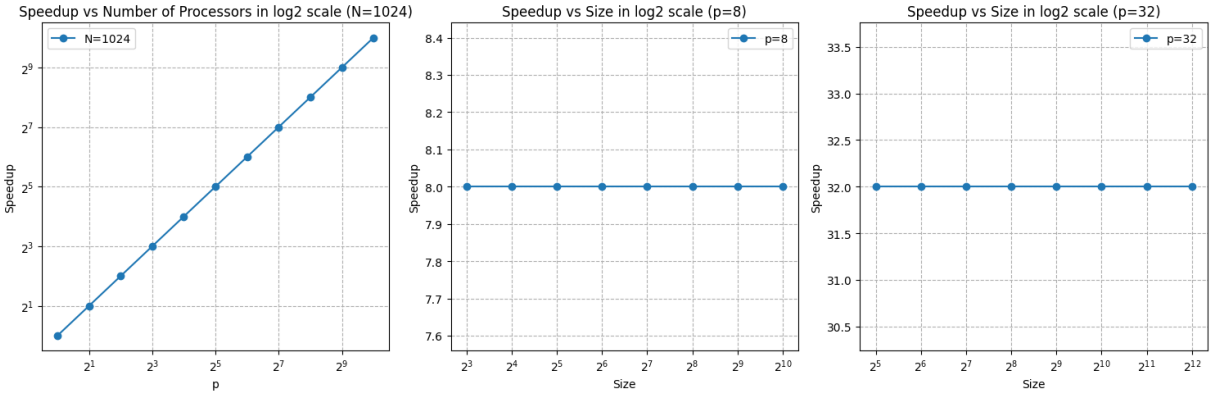
Efficiency $E = \frac{S}{p} = 1$

Overhead $F = pT_p - T_s = 0$

The algorithm is cost-optimal.

2.4 Using the algorithm in 2.3, create the speed up plots in 2.2

The code is avaiable in q2_4.py.



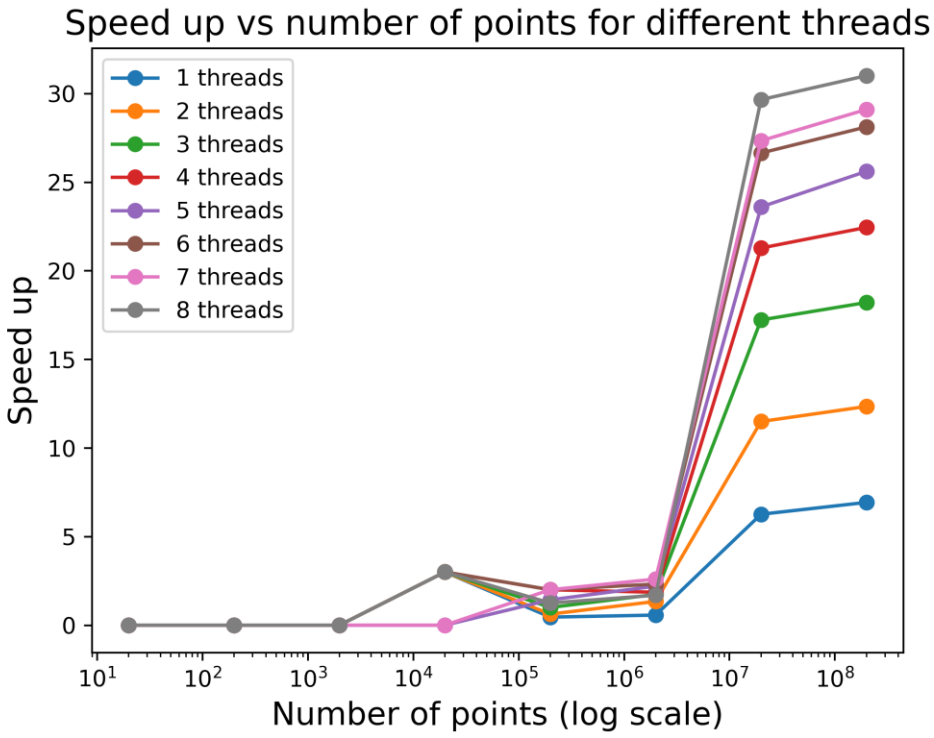
3 Compute π (10 points for code, 5 points for write-up)

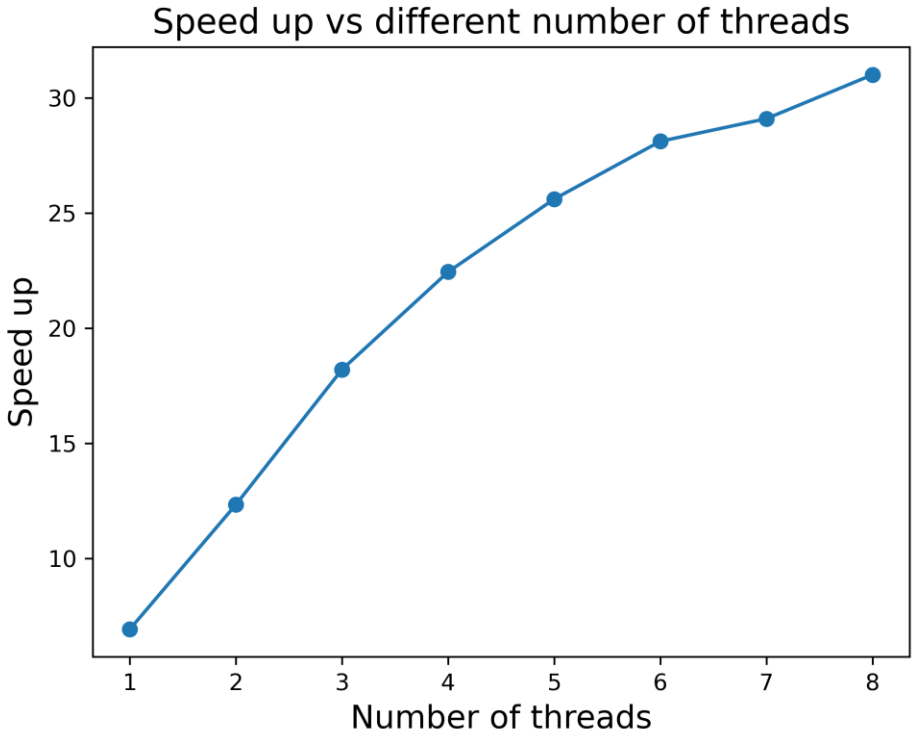
3.1 Using your previous sequential Monte Carlo code to estimate PI. Parallelize it using OpenMP. Recall that random number generation needs to be parallelized. Compare the error with the value of $\pi = \cos^{-1}(-1.0)$.

The parallelized Monte Carlo code for estimating PI is available in the q3.c script. This code estimates PI using varying numbers of points, ranging from 20 to 20000000 with a stride of 10, and different numbers of threads, ranging from 1 to 8 with a stride of 1. The estimated values of PI and associated errors, corresponding to different combinations of points and threads, are stored in the pi_results.csv file. For the maximum values of both the number of points (n) and the number of threads (p), the error rate is 0.004082%, and the runtime is 4.669 seconds.

3.2 What is the speedup? Plot the strong scaling (as a function of number of threads) and the weak scaling (as a function of the number of points) for p from 1 to 8.

The weak and strong scaling plots are presented below. These plots show the scaling behavior with respect to varying numbers of points, ranging from 20 to 20000000 with a stride of 10, and different numbers of threads, ranging from 1 to 8 with a stride of 1. The first plot illustrates how the speedup varies as the number of points increases for different thread counts. In my experiment, the speedup starts to significantly increase from 2000 data points onward, and after reaching 2000000 points, the growth becomes dramatic. The second plot examines how the time required changes as the number of threads increases, while keeping the number of points fixed at its highest value. As can be observed, this relationship is approximately linear.


Speed up vs number of points for different threads
1 threads
2 threads
3 threads
4 threads
5 threads
6 threads
7 threads
8 threads
Speed up
Number of points (log scale)


Speed up vs different number of threads
Speed up
Number of threads

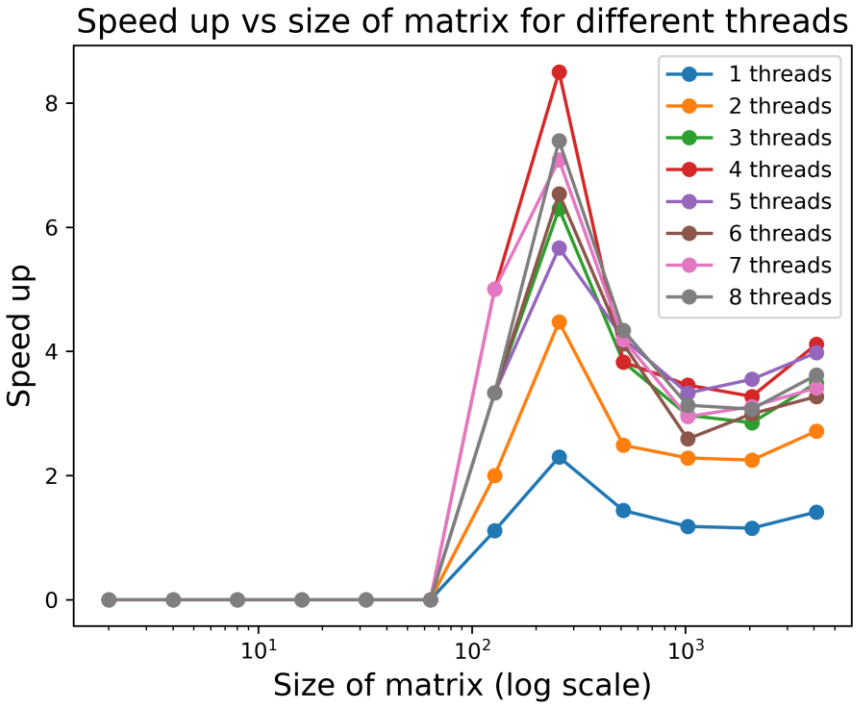
4 Matrix Multiplication (10 points for code, 5 points for writeup)

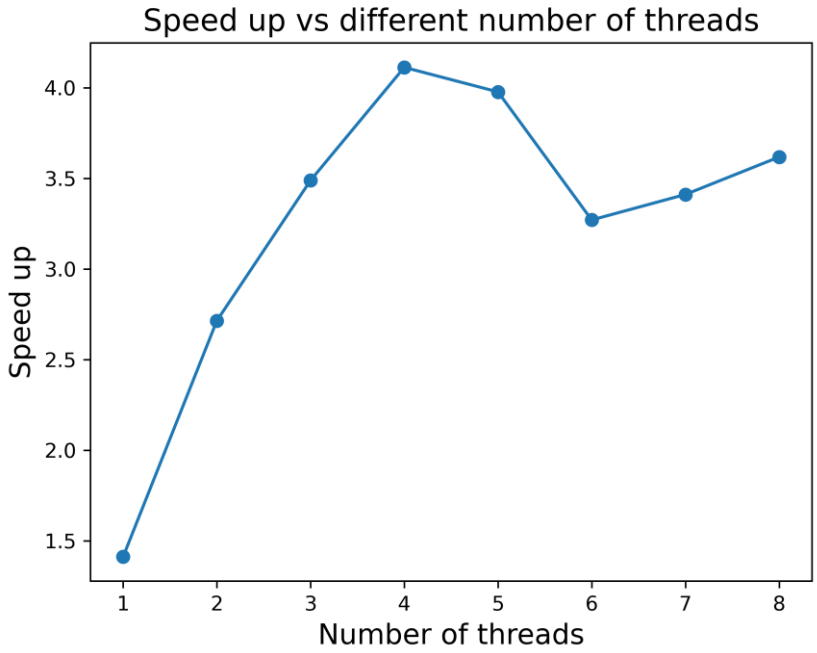
4.1 What is the speedup? Plot the strong scaling (as a function of number of threads) and the weak scaling (as a function of the number of points) for p from 1 to 8.

The parallelized matrix manipulation code is available in the q4.c script. This code calculates matrix manipulation with a user input matrix size or automatically ranging from 2 to 4096 with a stride of powers of 2. The time taken for these calculations based on different matrix sizes and the number of threads are stored in the matrix_results.csv file. For the maximum values of both size and the number of threads the runtime is 529.408 seconds.

4.2 Plot the strong scaling (as a function of number of threads) and the weak scaling (as a function of the number of points) for p from 1 to 8. Use any optimization strategy that you can find, to run your code faster

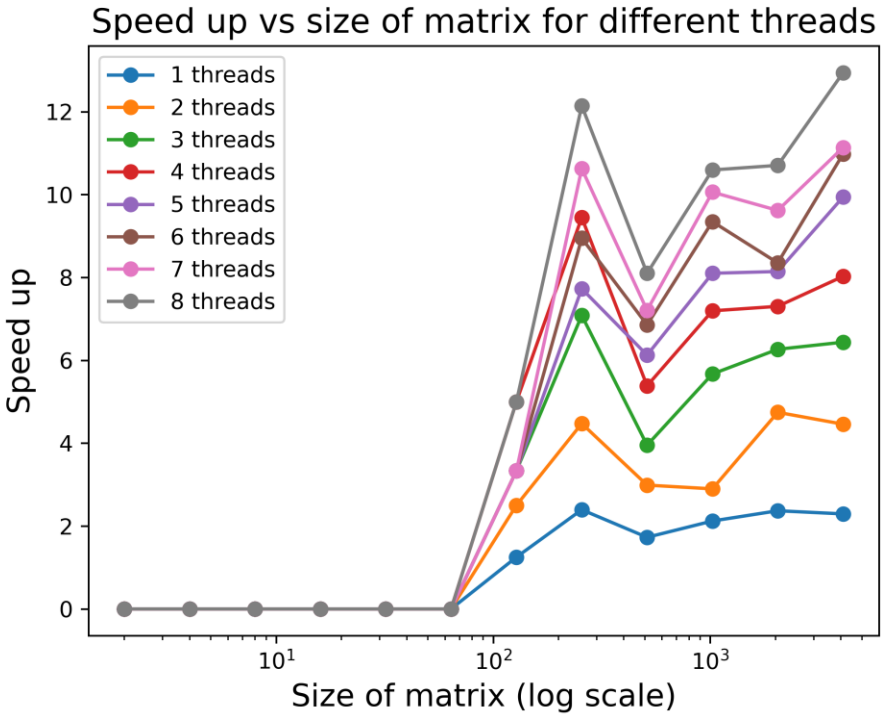
The weak and strong scaling plots are presented below. These plots show the scaling behavior with respect to varying numbers matrix sizes, ranging from 2 to 4096 with a stride of^2, and different numbers of threads, ranging from 1 to 8 with a stride of 1. The first plot illustrates how the speedup varies as the size increases for different thread counts. In my experiment, the speedup starts to significantly increase from the matrix size of 256, followed by a drop, and steady growth. The second plot examines how the time required changes as the number of threads increases, while keeping the matrix size fixed at its highest value.

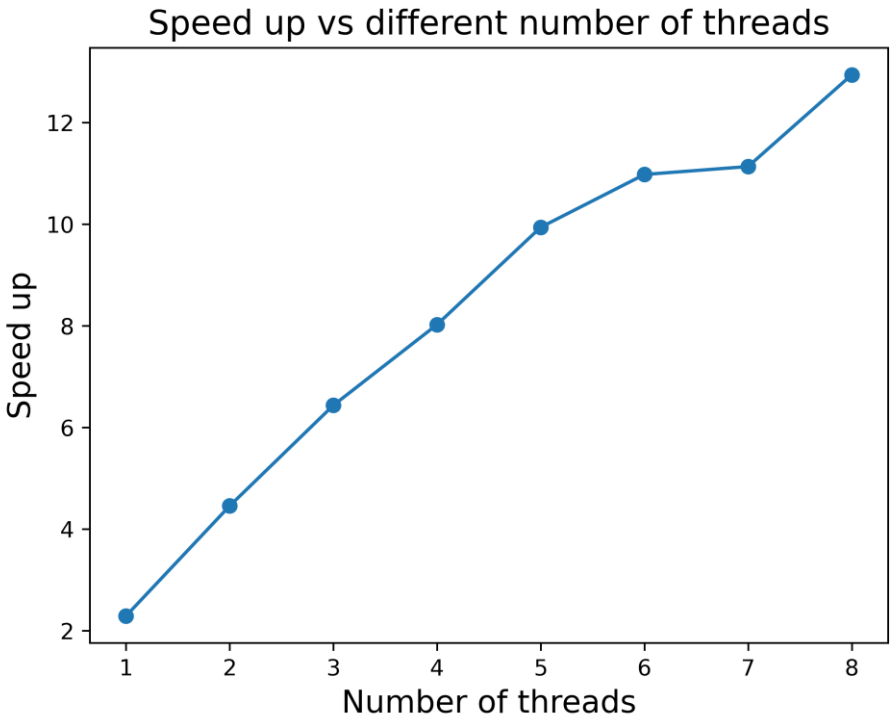




4.3 You are free to use the full OpenMP API, adjust your matrix multiplication strategy, and use any optimization strategy you can find. In other words, write the fastest matrix multiplier.

For this question, I combined SIMD instructions with the full OpenMP API approach to write the fastest matrix multiplier. We can see a considerably higher speed up for the algorithm in the following figures. The main code is available in q4_3.c




Speed up vs different number of threads
Speed up
Number of threads
2
4
6
8
10
12
1
2
3
4
5
6
7
8