CMSC5702 (Advanced Topics in Parallel / Distributed Systems)

Assignment 1: Parallel Programming with Sobel Filter

Objective

Apply the Sobel filter for edge detection on **large square grayscale images** using MPI and OpenMP, and compare the performance of shared-memory vs distributed-memory parallelization. Optionally, include a blurring step to reduce noise before applying the Sobel operator.

This assignment allows you to practice:

- Sequential programming
- OpenMP parallelization on shared-memory systems
- MPI parallelization on distributed-memory systems
- Performance measurement, verification, and speedup analysis

---

Problem Description

The **Sobel filter** computes approximate image gradients to detect edges. For a grayscale image (I):

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * I, \quad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} * I$$

The **gradient magnitude** at each pixel is:

$$G = \sqrt{G_x^2 + G_y^2}$$

Optional **blurring step**: compute the local average of pixels in a small neighborhood (e.g., 3×3) before applying the Sobel operator to reduce noise.

Part 1: Sequential Version

- Implement a standard sequential C program to apply the Sobel operator on a grayscale image.
- If including blurring, apply an average filter first (e.g., 3×3 kernel) before Sobel.

Question: Explain the algorithm you implemented and if you applyed blurring, explain the blurring method.

Your Answer:

My implementation follows a two-stage process.

First, I implemented the optional **blurring step** to reduce image noise before edge detection. The method I used is the 3×3 **average filter**, which is also called a mean filter. As shown in my `blur_filter` function, this filter works by sliding a 3x3 window over the image. For each pixel, it calculates the average value of all 9 pixels in its 3×3 neighborhood (summing them up and dividing by 9.0) and uses this average as the new value for that pixel.

Second, I apply the **Sobel operator** to the *blurred* image. This is implemented in my `sobel_filter` function. This algorithm calculates two gradients for each pixel by convolving the image with two 3x3 kernels:

- G_x: Using the 3x3 Sobel kernel to detect horizontal changes.
- G_y: Using the 3x3 Sobel kernel to detect vertical changes.

Finally, the magnitude of the gradient is computed using the formula $G = \sqrt{G_x^2 + G_y^2}$ to get the final pixel intensity for the edge-detected image. In my implementation, for pixels on the 1-pixel border of the image, I set their output value to 0.0, as the 3x3 kernel cannot be fully applied there.

In [1]:
```
!pip install pandas
```

```
Requirement already satisfied: pandas in /opt/anaconda3/envs/hwpytorch/lib/p
ython3.12/site-packages (2.2.3)
Requirement already satisfied: numpy>=1.26.0 in /opt/anaconda3/envs/hwpytorc
h/lib/python3.12/site-packages (from pandas) (2.0.1)
Requirement already satisfied: python-dateutil>=2.8.2 in /opt/anaconda3/env
s/hwpytorch/lib/python3.12/site-packages (from pandas) (2.9.0.post0)
Requirement already satisfied: pytz>=2020.1 in /opt/anaconda3/envs/hwpytorc
h/lib/python3.12/site-packages (from pandas) (2024.2)
Requirement already satisfied: tzdata>=2022.7 in /opt/anaconda3/envs/hwpytor
ch/lib/python3.12/site-packages (from pandas) (2024.1)
Requirement already satisfied: six>=1.5 in /opt/anaconda3/envs/hwpytorch/li
b/python3.12/site-packages (from python-dateutil>=2.8.2->pandas) (1.16.0)
```

In [2]:
```
!pip install matplotlib
```

```
Requirement already satisfied: matplotlib in /opt/anaconda3/envs/hwpytorch/l
ib/python3.12/site-packages (3.9.2)
Requirement already satisfied: contourpy>=1.0.1 in /opt/anaconda3/envs/hwpyt
orch/lib/python3.12/site-packages (from matplotlib) (1.3.1)
Requirement already satisfied: cycler>=0.10 in /opt/anaconda3/envs/hwpytorc
h/lib/python3.12/site-packages (from matplotlib) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in /opt/anaconda3/envs/hwpy
torch/lib/python3.12/site-packages (from matplotlib) (4.55.0)
Requirement already satisfied: kiwisolver>=1.3.1 in /opt/anaconda3/envs/hwpy
torch/lib/python3.12/site-packages (from matplotlib) (1.4.7)
Requirement already satisfied: numpy>=1.23 in /opt/anaconda3/envs/hwpytorch/
lib/python3.12/site-packages (from matplotlib) (2.0.1)
Requirement already satisfied: packaging>=20.0 in /opt/anaconda3/envs/hwpyto
rch/lib/python3.12/site-packages (from matplotlib) (24.1)
Requirement already satisfied: pillow>=8 in /opt/anaconda3/envs/hwpytorch/li
b/python3.12/site-packages (from matplotlib) (10.4.0)
Requirement already satisfied: pyparsing>=2.3.1 in /opt/anaconda3/envs/hwpyt
orch/lib/python3.12/site-packages (from matplotlib) (3.2.0)
Requirement already satisfied: python-dateutil>=2.7 in /opt/anaconda3/envs/h
wpytorch/lib/python3.12/site-packages (from matplotlib) (2.9.0.post0)
Requirement already satisfied: six>=1.5 in /opt/anaconda3/envs/hwpytorch/li
b/python3.12/site-packages (from python-dateutil>=2.7->matplotlib) (1.16.0)
```

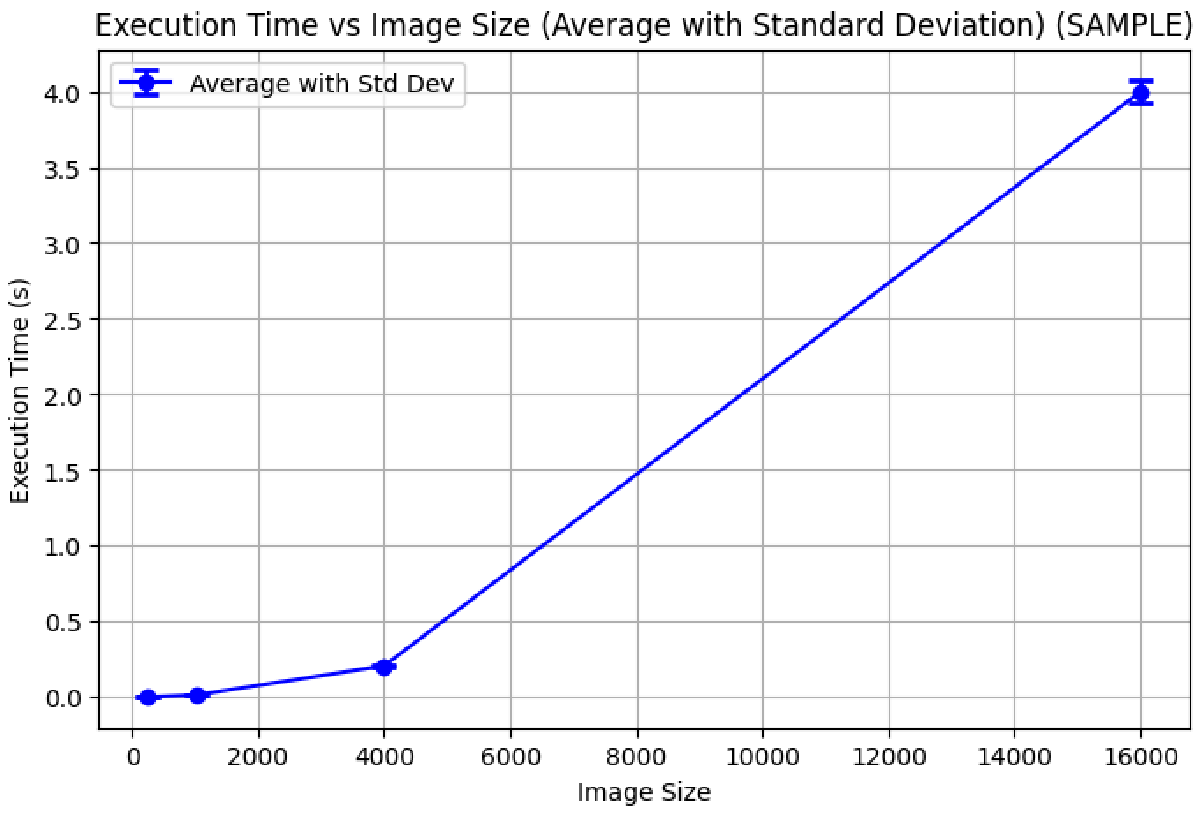
In [3]:
```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

seq_df = pd.read_csv("sequential_times.csv")
```

In [4]:
```
seq_df
```

Out[4]:

	image_size	run1_time	run2_time	run3_time
0	256	0.000933	0.001047	0.000888
1	1024	0.015544	0.014636	0.014826
2	4000	0.210384	0.194488	0.215198
3	16000	4.079864	3.984604	3.932855

In [5]:
```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

# Assuming your data is in a DataFrame called df
# Calculate average and standard deviation
seq_df['average_time'] = seq_df[['run1_time', 'run2_time', 'run3_time']].mea
seq_df['std_dev'] = seq_df[['run1_time', 'run2_time', 'run3_time']].std(axis

# Plot with error bars showing standard deviation
plt.figure(figsize=(8,5))
plt.errorbar(seq_df['image_size'], seq_df['average_time'],
             yerr=seq_df['std_dev'],
             marker='o',
             color='blue',
             label='Average with Std Dev',
             capsize=5,  # Adds caps to the error bars
             capthick=2,
             elinewidth=2)

plt.xlabel('Image Size')
plt.ylabel('Execution Time (s)')
plt.title('Execution Time vs Image Size (Average with Standard Deviation) (S
plt.legend()
plt.grid(True)
plt.show()

# Optional: Print the calculated values
print("Data with averages and standard deviations:")
print(seq_df[['image_size', 'average_time', 'std_dev']])
```



```
Data with averages and standard deviations:
   image_size  average_time   std_dev
0         256      0.000956  0.000082
1        1024      0.015002  0.000479
2        4000      0.206690  0.010838
3       16000      3.999108  0.074570
```

Part 2: OpenMP Programming

- Use#pragma omp parallel for collapse(2) for nested loops.
- Experiment with schedule(static) and other scheduling clauses.
- Process large images in tiles (e.g., 512×512 or 1024×1024) to reduce memory bandwidth issues.

- [Bonus Point] Optionally include a blurring step before the Sobel operator to reduce noise.

Question: Explain the algorithm you implemented and if you applied blurring, explain the blurring method.

Your Answer:

For the OpenMP implementation, my algorithm follows the same two-stage process as the sequential version, but the parallelization strategy is optimized for cache performance.

I implemented the **blurring step**. The method is the 3×3 **average filter**, where each pixel's new value is the average of its 9-pixel neighborhood (summing them up and dividing by 9.0) to reduce noise.

The core of my parallel implementation is a **tiling (or block-based) decomposition**.Here is the logic:

1. **Tiling:** Instead of parallelizing the image row by row, I divide the entire image into small, fixed-size square blocks.

2. **Parallelism:** I use OpenMP's `#pragma omp parallel for collapse(2)` directive. This directive assigns *entire tiles* to different threads. The `collapse(2)` tells OpenMP to merge the two outer loops (the one iterating tile rows and the one iterating tile columns) into one large loop, which provides many independent tasks (the tiles) for the threads to work on.

This tiling method is applied to both functions:

- First, `blur_filter_tiled` runs, with threads processing different tiles to create the blurred image.
- Second, `sobel_filter_tiled` runs on the blurred image, using the exact same parallel tiling strategy. Inside each tile, the standard Sobel logic is applied (convolving with Gx and Gy kernels, then calculating the magnitude $G = \sqrt{G_x^2 + G_y^2}$).

Processing the image in tiles ensures that the data a thread is working on (a single tile) is small enough to fit well within the CPU caches, which reduces memory bandwidth issues and improves performance, especially on large images.

```
In [6]: omp_df = pd.read_csv("openmp_times.csv")
```

```
In [7]: omp_df.head()
```

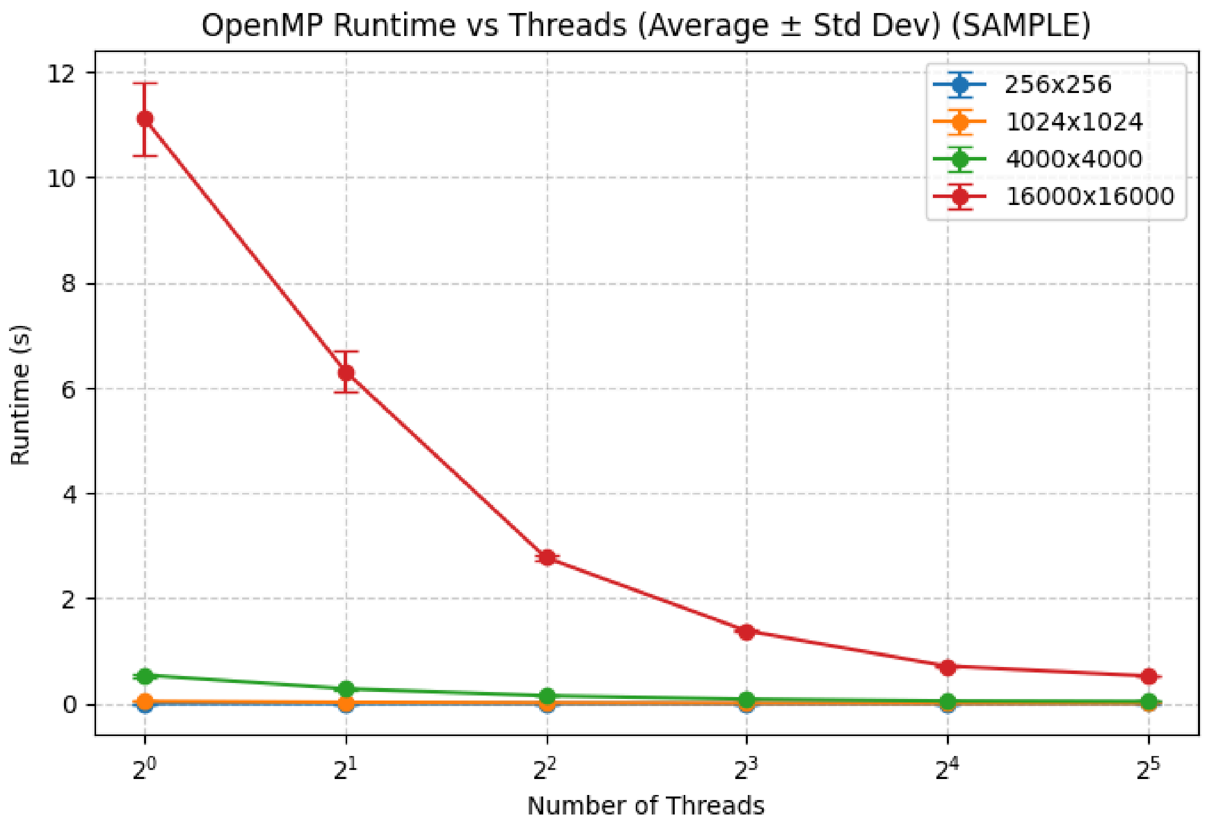
Out[7]:

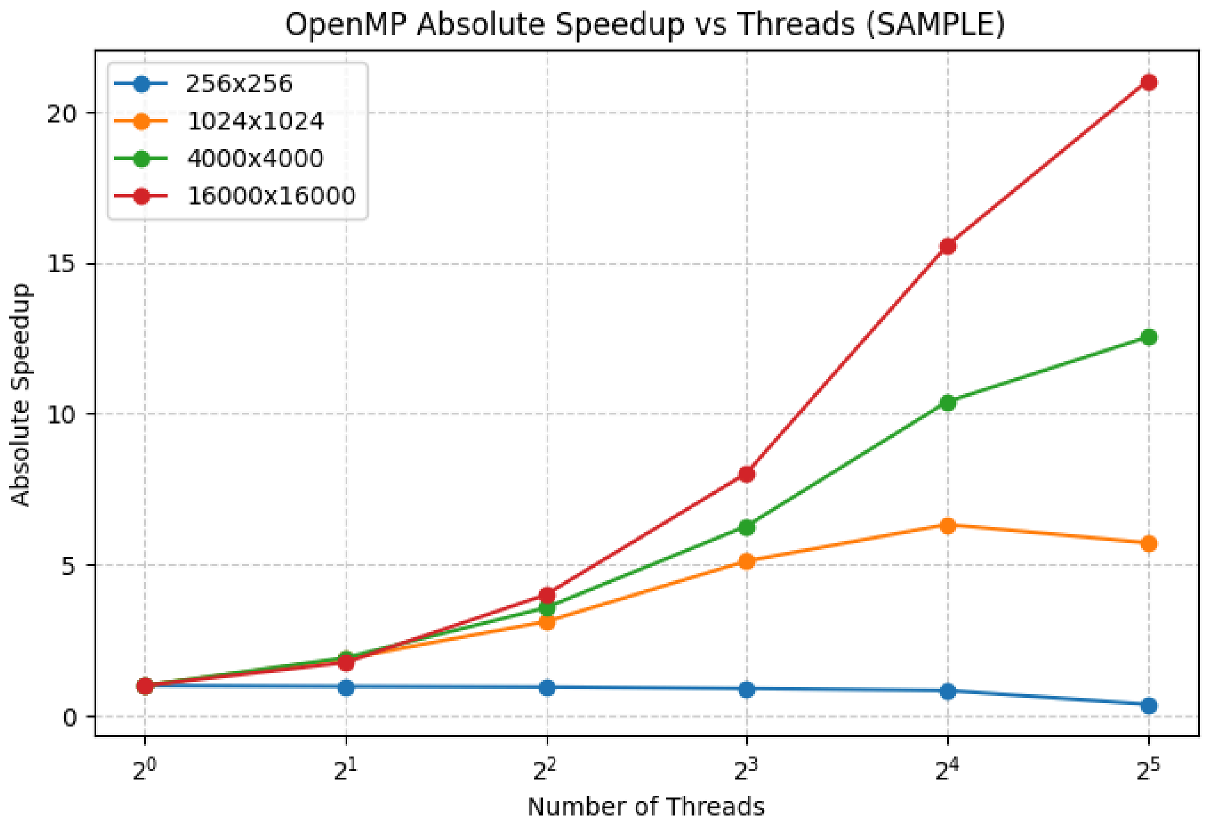
	image_size	threads	run1_time	run2_time	run3_time
0	256	1	0.002602	0.002662	0.002512
1	256	2	0.002633	0.002648	0.002686
2	256	4	0.002697	0.002705	0.002729
3	256	8	0.002934	0.002855	0.002830
4	256	16	0.003115	0.003054	0.003121

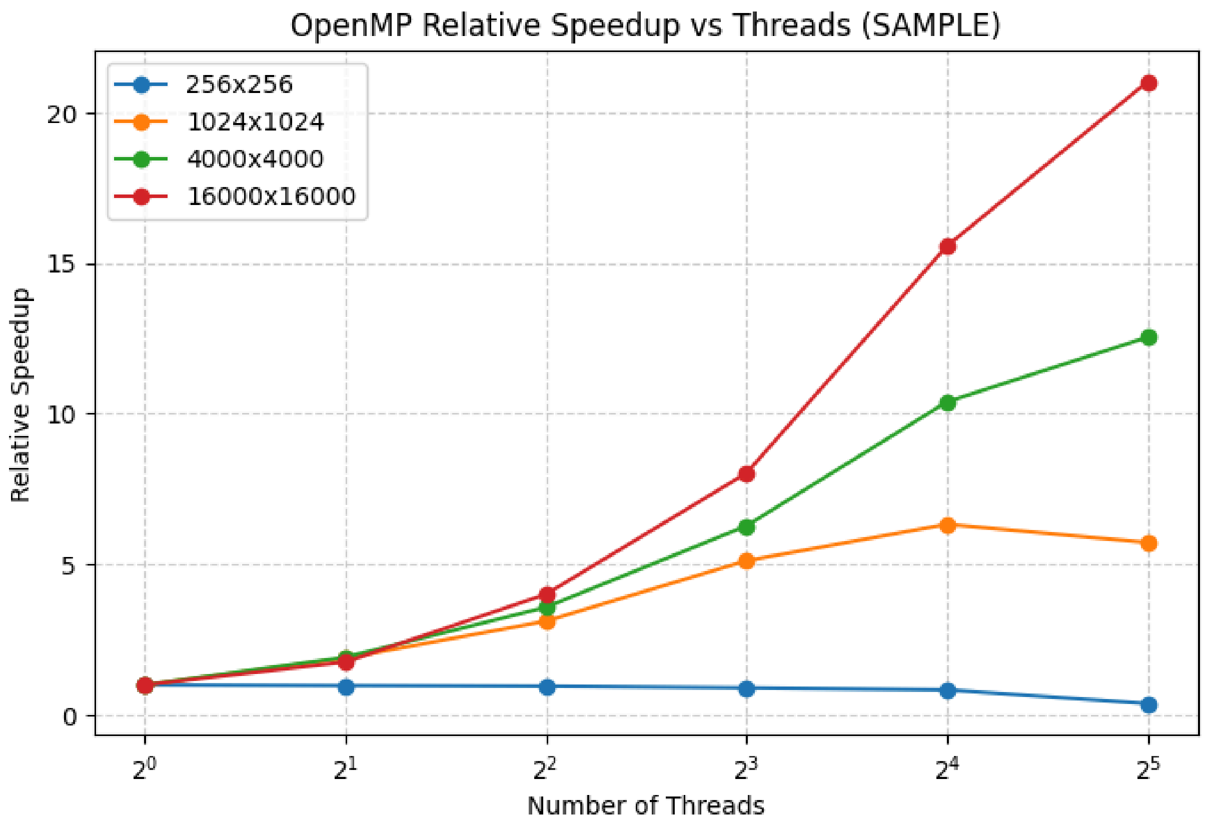
In [8]:

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

# Load CSV if not already loaded
# omp_df = pd.read_csv("openmp_timings.csv")

# 1️⃣ Compute average and standard deviation across runs
omp_df['average_time'] = omp_df[['run1_time','run2_time','run3_time']].mean(
omp_df['std_dev'] = omp_df[['run1_time','run2_time','run3_time']].std(axis=1

# Prepare columns for speedups
omp_df['absolute_speedup'] = 0.0
omp_df['relative_speedup'] = 0.0

# 2️⃣ Compute absolute and relative speedup
# Absolute speedup: vs sequential execution (threads=1)
# Relative speedup: vs OpenMP 1-thread execution (same in this dataset)
for size in omp_df['image_size'].unique():
    # Sequential time (threads=1)
    seq_time = omp_df[(omp_df['image_size']==size) & (omp_df['threads']==1)]

    subset_idx = omp_df[omp_df['image_size']==size].index
    omp_df.loc[subset_idx, 'absolute_speedup'] = seq_time / omp_df.loc[subse
    omp_df.loc[subset_idx, 'relative_speedup'] = seq_time / omp_df.loc[subse

# 3️⃣ Plot runtime with error bars
plt.figure(figsize=(8,5))
for size in sorted(omp_df['image_size'].unique()):
    subset = omp_df[omp_df['image_size']==size]
    plt.errorbar(subset['threads'], subset['average_time'],
                 yerr=subset['std_dev'],
                 marker='o', label=f'{size}x{size}', capsize=5)

plt.xlabel('Number of Threads')
plt.ylabel('Runtime (s)')
plt.title('OpenMP Runtime vs Threads (Average ± Std Dev) (SAMPLE)')
plt.xscale('log', base=2)
plt.xticks([1,2,4,8,16,32])
plt.grid(True, which='both', linestyle='--', alpha=0.6)
plt.legend()
plt.show()

# 4️⃣ Plot absolute speedup
```

```
plt.figure(figsize=(8,5))
for size in sorted(omp_df['image_size'].unique()):
    subset = omp_df[omp_df['image_size']==size]
    plt.plot(subset['threads'], subset['absolute_speedup'], marker='o', labe

plt.xlabel('Number of Threads')
plt.ylabel('Absolute Speedup')
plt.title('OpenMP Absolute Speedup vs Threads (SAMPLE)')
plt.xscale('log', base=2)
plt.xticks([1,2,4,8,16,32])
plt.grid(True, which='both', linestyle='--', alpha=0.6)
plt.legend()
plt.show()

# 5️⃣ Plot relative speedup (same as absolute here)
plt.figure(figsize=(8,5))
for size in sorted(omp_df['image_size'].unique()):
    subset = omp_df[omp_df['image_size']==size]
    plt.plot(subset['threads'], subset['relative_speedup'], marker='o', labe

plt.xlabel('Number of Threads')
plt.ylabel('Relative Speedup')
plt.title('OpenMP Relative Speedup vs Threads (SAMPLE)')
plt.xscale('log', base=2)
plt.xticks([1,2,4,8,16,32])
plt.grid(True, which='both', linestyle='--', alpha=0.6)
plt.legend()
plt.show()

# 6️⃣ Optional: Print the full dataframe
print("OpenMP data with runtime, std_dev, absolute and relative speedups:")
print(omp_df[['image_size','threads','average_time','std_dev','absolute_spee
```


OpenMP Runtime vs Threads (Average ± Std Dev) (SAMPLE)
Runtime (s)
Number of Threads
2^0
2^1
2^2
2^3
2^4
2^5
0
2
4
6
8
10
12
256x256
1024x1024
4000x4000
16000x16000


OpenMP Absolute Speedup vs Threads (SAMPLE)
Absolute Speedup
Number of Threads
2^0
2^1
2^2
2^3
2^4
2^5
0
5
10
15
20
256x256
1024x1024
4000x4000
16000x16000


OpenMP Relative Speedup vs Threads (SAMPLE)
256x256
1024x1024
4000x4000
16000x16000
Relative Speedup
0
5
10
15
20
2^0
2^1
2^2
2^3
2^4
2^5
Number of Threads

```
OpenMP data with runtime, std_dev, absolute and relative speedups:
    image_size  threads  average_time   std_dev  absolute_speedup  \
0          256        1      0.002592  0.000075          1.000000   
1          256        2      0.002656  0.000027          0.976026   
2          256        4      0.002710  0.000017          0.956340   
3          256        8      0.002873  0.000054          0.902193   
4          256       16      0.003097  0.000037          0.837029   
5          256       32      0.006793  0.000634          0.381551   
6         1024        1      0.045112  0.001055          1.000000   
7         1024        2      0.023913  0.002435          1.886519   
8         1024        4      0.014483  0.000557          3.114776   
9         1024        8      0.008809  0.000231          5.121358   
10        1024       16      0.007126  0.000275          6.330963   
11        1024       32      0.007880  0.000090          5.724915   
12        4000        1      0.545054  0.034005          1.000000   
13        4000        2      0.284085  0.021516          1.918630   
14        4000        4      0.152469  0.007123          3.574859   
15        4000        8      0.086843  0.005820          6.276291   
16        4000       16      0.052449  0.002981         10.392076   
17        4000       32      0.043478  0.001073         12.536413   
18       16000        1     11.111987  0.696742          1.000000   
19       16000        2      6.314521  0.390359          1.759751   
20       16000        4      2.775402  0.044862          4.003739   
21       16000        8      1.384710  0.007167          8.024778   
22       16000       16      0.714140  0.014279         15.559964   
23       16000       32      0.528825  0.008575         21.012585   

    relative_speedup  
0           1.000000  
1           0.976026  
2           0.956340  
3           0.902193  
4           0.837029  
5           0.381551  
6           1.000000  
7           1.886519  
8           3.114776  
9           5.121358  
10          6.330963  
11          5.724915  
12          1.000000  
13          1.918630  
14          3.574859  
15          6.276291  
16         10.392076  
17         12.536413  
18          1.000000  
19          1.759751  
20          4.003739  
21          8.024778  
22         15.559964  
23         21.012585  
```

# Part 3: MPI Programming

- Choose a domain decomposition strategy, such as row-wise decomposition, column-wise decomposition, etc., decomposition. Your choice will affect performance, and better-designed decomposition and communication patterns will receive higher marks.
- For each process, you may need to include some approaches to allow seamless computation for the edges.
- [Bonus Point] You can include If a parallelized blurring step and you may need to try to optimize the communications.
- Verify your implementation using small test patterns (5×5 or 10×10) and visually inspect sample images for correctness.

In [9]:
```
mpi_df = pd.read_csv("mpi_times.csv")
```

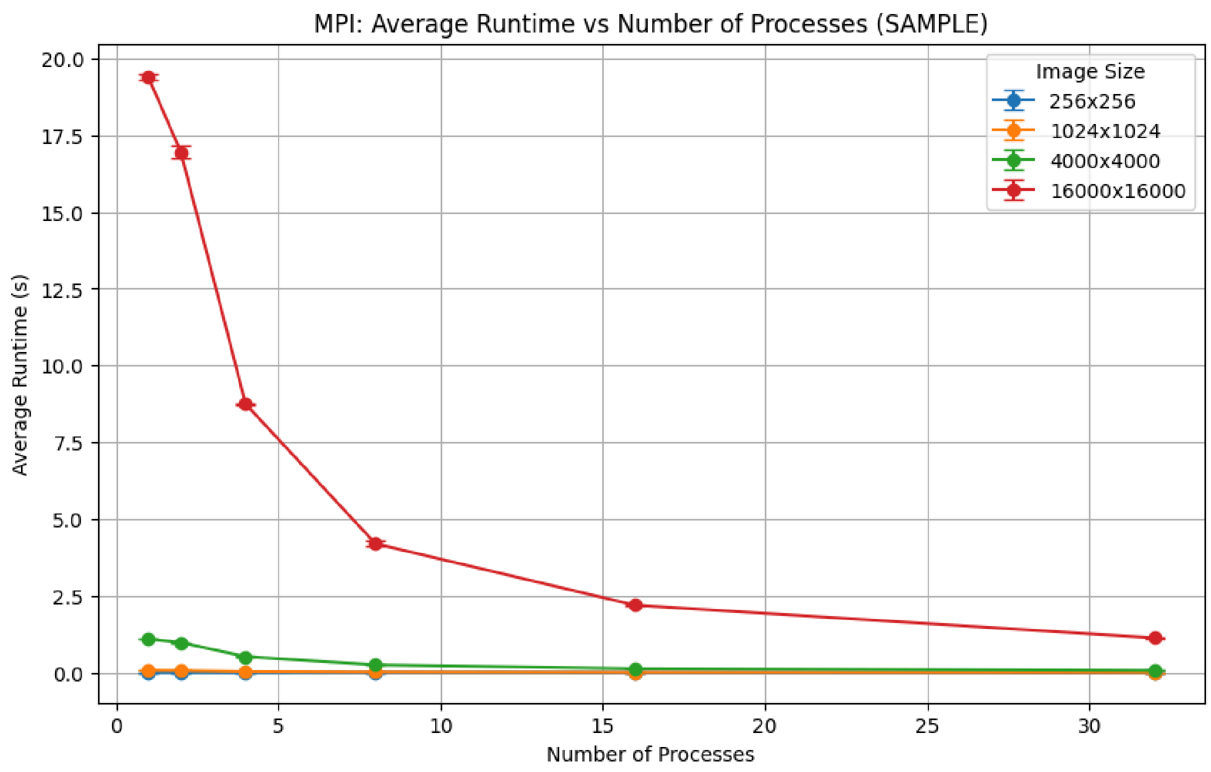
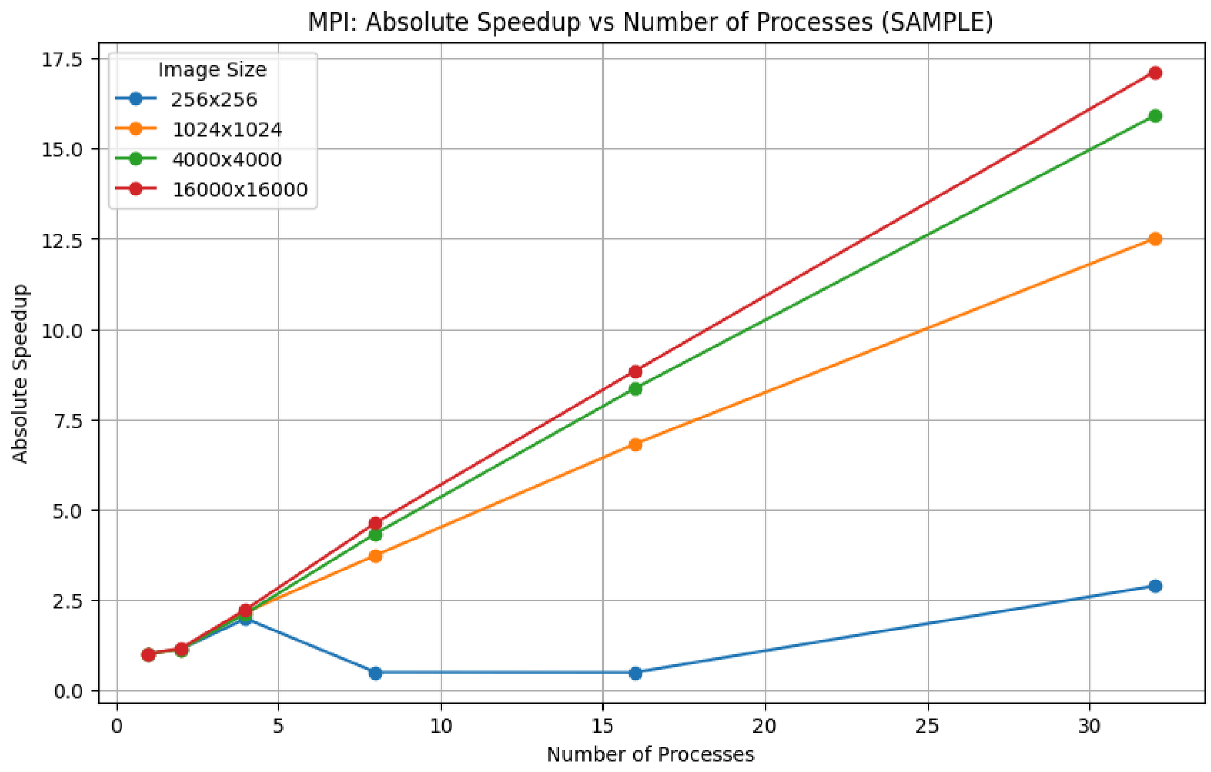
In [10]:
```
mpi_df.head()
```

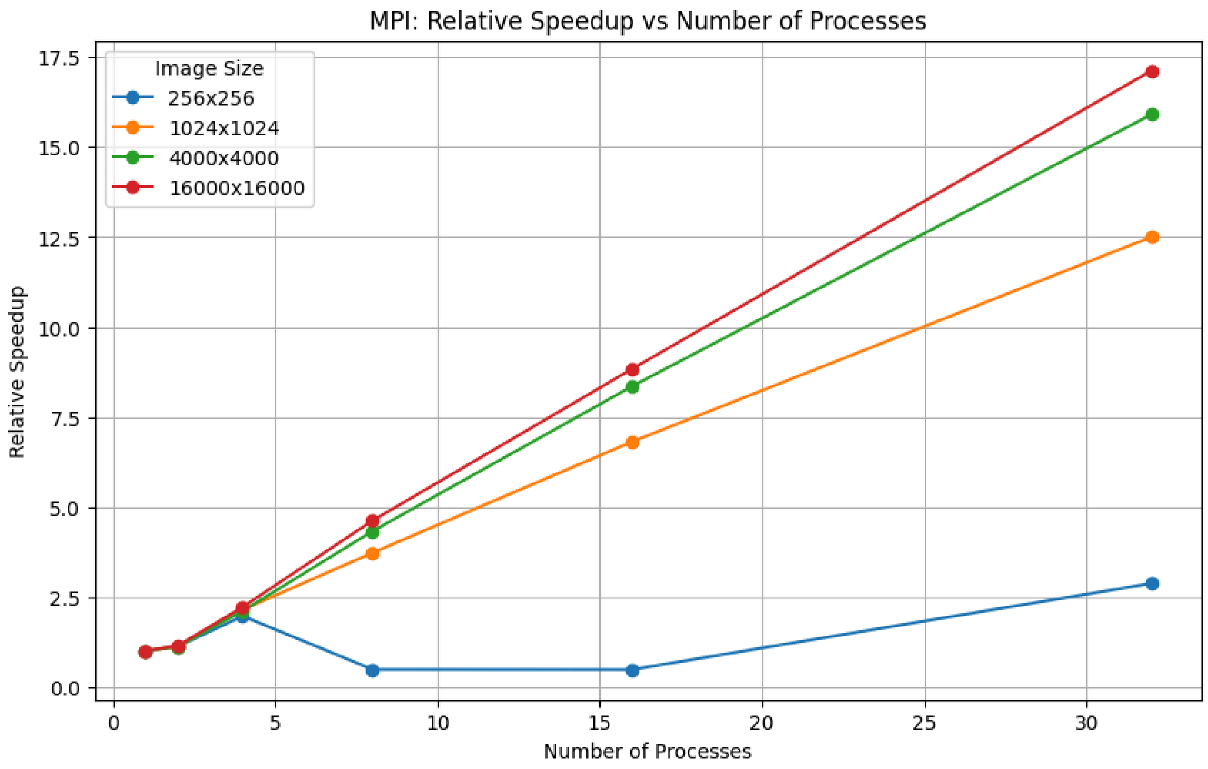
Out[10]:

	image_size	nodes	processes	run1_time	run2_time	run3_time
0	256	1	1	0.004858	0.004847	0.004859
1	256	1	2	0.004317	0.004334	0.004342
2	256	1	4	0.002223	0.002965	0.002210
3	256	4	8	0.010011	0.009881	0.010153
4	256	4	16	0.010022	0.009914	0.010408

In [11]:
```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np


# Compute average and standard deviation of runtime
mpi_df['average_time'] = mpi_df[['run1_time', 'run2_time', 'run3_time']].mea
mpi_df['std_dev'] = mpi_df[['run1_time', 'run2_time', 'run3_time']].std(axis

# --- Compute Absolute and Relative Speedup from CSV ---

# Absolute speedup: sequential = 1 process on 1 node
absolute_speedup = {}
for img_size in mpi_df['image_size'].unique():
    # Find sequential runtime: 1 node, 1 process
    seq_time = mpi_df[(mpi_df['image_size']==img_size) &
                      (mpi_df['nodes']==1) &
                      (mpi_df['processes']==1)]['average_time'].values[0]
    absolute_speedup[img_size] = seq_time

# Add speedup columns
mpi_df['absolute_speedup'] = mpi_df.apply(lambda row: absolute_speedup[row['

# Relative speedup: 1 process on 1 node runtime as reference
relative_speedup = {}
```

```
for img_size in mpi_df['image_size'].unique():
    ref_time = mpi_df[(mpi_df['image_size']==img_size) &
                      (mpi_df['nodes']==1) &
                      (mpi_df['processes']==1)]['average_time'].values[0]
    relative_speedup[img_size] = ref_time

mpi_df['relative_speedup'] = mpi_df.apply(lambda row: relative_speedup[row['

# --- Plots ---

# Runtime with error bars
plt.figure(figsize=(10,6))
for img_size in sorted(mpi_df['image_size'].unique()):
    df_img = mpi_df[mpi_df['image_size']==img_size]
    plt.errorbar(df_img['processes'], df_img['average_time'], yerr=df_img['s
                 marker='o', capsize=5, label=f'{img_size}x{img_size}')
plt.xlabel('Number of Processes')
plt.ylabel('Average Runtime (s)')
plt.title('MPI: Average Runtime vs Number of Processes (SAMPLE)')
plt.legend(title='Image Size')
plt.grid(True)
plt.show()

# Absolute speedup
plt.figure(figsize=(10,6))
for img_size in sorted(mpi_df['image_size'].unique()):
    df_img = mpi_df[mpi_df['image_size']==img_size]
    plt.plot(df_img['processes'], df_img['absolute_speedup'], marker='o', la
plt.xlabel('Number of Processes')
plt.ylabel('Absolute Speedup')
plt.title('MPI: Absolute Speedup vs Number of Processes (SAMPLE)')
plt.legend(title='Image Size')
plt.grid(True)
plt.show()

# Relative speedup
plt.figure(figsize=(10,6))
for img_size in sorted(mpi_df['image_size'].unique()):
    df_img = mpi_df[mpi_df['image_size']==img_size]
    plt.plot(df_img['processes'], df_img['relative_speedup'], marker='o', la
plt.xlabel('Number of Processes')
plt.ylabel('Relative Speedup')
plt.title('MPI: Relative Speedup vs Number of Processes')
plt.legend(title='Image Size')
plt.grid(True)
plt.show()

# Optional: print dataframe
print(mpi_df[['image_size','nodes','processes','average_time','std_dev','abs
```


MPI: Average Runtime vs Number of Processes (SAMPLE)
Average Runtime (s)
Number of Processes
Image Size
256x256
1024x1024
4000x4000
16000x16000
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
0
5
10
15
20
25
30

MPI: Absolute Speedup vs Number of Processes (SAMPLE)
Absolute Speedup
Number of Processes
Image Size
256x256
1024x1024
4000x4000
16000x16000
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
0
5
10
15
20
25
30


MPI: Relative Speedup vs Number of Processes
Image Size
256x256
1024x1024
4000x4000
16000x16000
Relative Speedup
Number of Processes
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
0
5
10
15
20
25
30

```
    image_size  nodes  processes  average_time   std_dev  absolute_speedup  \
0          256      1          1      0.004855  0.000007          1.000000   
1          256      1          2      0.004331  0.000013          1.120911   
2          256      1          4      0.002466  0.000432          1.968640   
3          256      4          8      0.010015  0.000136          0.484740   
4          256      4         16      0.010115  0.000260          0.479963   
5          256      4         32      0.001689  0.000492          2.874852   
6         1024      1          1      0.073846  0.001352          1.000000   
7         1024      1          2      0.066292  0.001806          1.113961   
8         1024      1          4      0.034436  0.000432          2.144431   
9         1024      4          8      0.019822  0.000114          3.725473   
10        1024      4         16      0.010837  0.000087          6.814069   
11        1024      4         32      0.005906  0.000102         12.504318   
12        4000      1          1      1.095608  0.007461          1.000000   
13        4000      1          2      0.979258  0.000564          1.118815   
14        4000      1          4      0.519954  0.013461          2.107125   
15        4000      4          8      0.253195  0.005454          4.327131   
16        4000      4         16      0.131151  0.001393          8.353769   
17        4000      4         32      0.068871  0.001714         15.908118   
18       16000      1          1     19.388482  0.104251          1.000000   
19       16000      1          2     16.949818  0.223479          1.143876   
20       16000      1          4      8.740856  0.008950          2.218144   
21       16000      4          8      4.196811  0.094911          4.619813   
22       16000      4         16      2.194286  0.025026          8.835897   
23       16000      4         32      1.132161  0.026099         17.125203   

    relative_speedup  
0           1.000000  
1           1.120911  
2           1.968640  
3           0.484740  
4           0.479963  
5           2.874852  
6           1.000000  
7           1.113961  
8           2.144431  
9           3.725473  
10          6.814069  
11         12.504318  
12          1.000000  
13          1.118815  
14          2.107125  
15          4.327131  
16          8.353769  
17         15.908118  
18          1.000000  
19          1.143876  
20          2.218144  
21          4.619813  
22          8.835897  
23         17.125203  
```

In [ ]: