PARALLEL EDGE DETECTION

Rebecca Abraham

CSE 633 - Spring 2025

Prof. Russ Miller

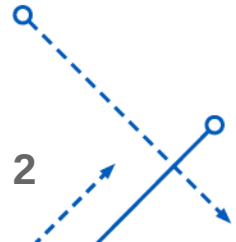
University at Buffalo The State University of New York

Introduction

- **Edge Detection in Image Processing**
  - What is Edge Detection?
  - Sobel Edge Detection?
- **Challenges with Sequential Computation**:
  - Time Complexity: For large images, the computational complexity of applying the Sobel operator to each pixel becomes significant
  - Bottlenecks: Processing time is dominated by the CPU speed and memory bandwidth
  - Scaling: As image size increases, execution time increases linearly, leading to poor performance for larger datasets
- **Need for Parallel Computing**:
  - The concept of parallel computing helps address these challenges by dividing the workload among multiple processors
  - Parallel computing accelerates computationally expensive tasks using Message Passing Interface (MPI)

Goal: *Speed up the edge detection process by parallelizing the Sobel operator using MPI (Message Passing Interface)*

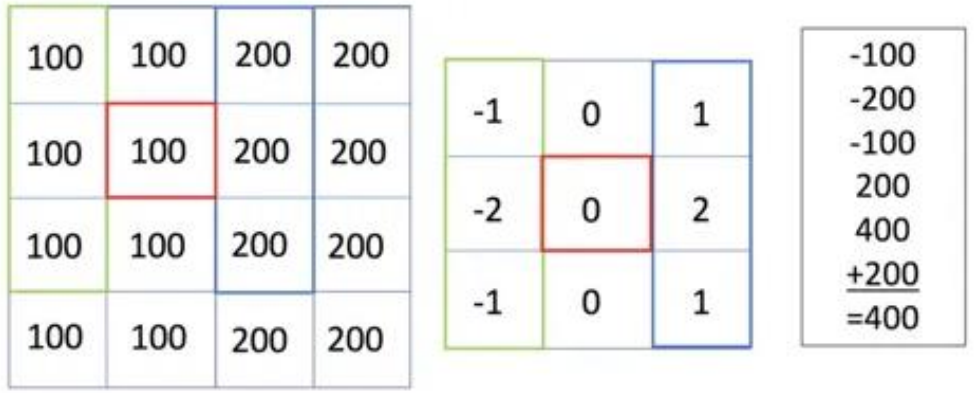
- MPI enables multiple processes to run in parallel across different nodes or cores, communicating via messages
- MPI is ideal for distributed-memory systems, allowing each process to operate on a portion of the image and share results



Sobel Operator

$$\mathbf{G}_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{A}$$

Pre-Convoluted Kernel (A is a matrix containing the image data)



Kernel Convolution: The bigger the value at the end, the more noticeable the edge will be.

$$|G| = \sqrt{Gx^2 + Gy^2}$$

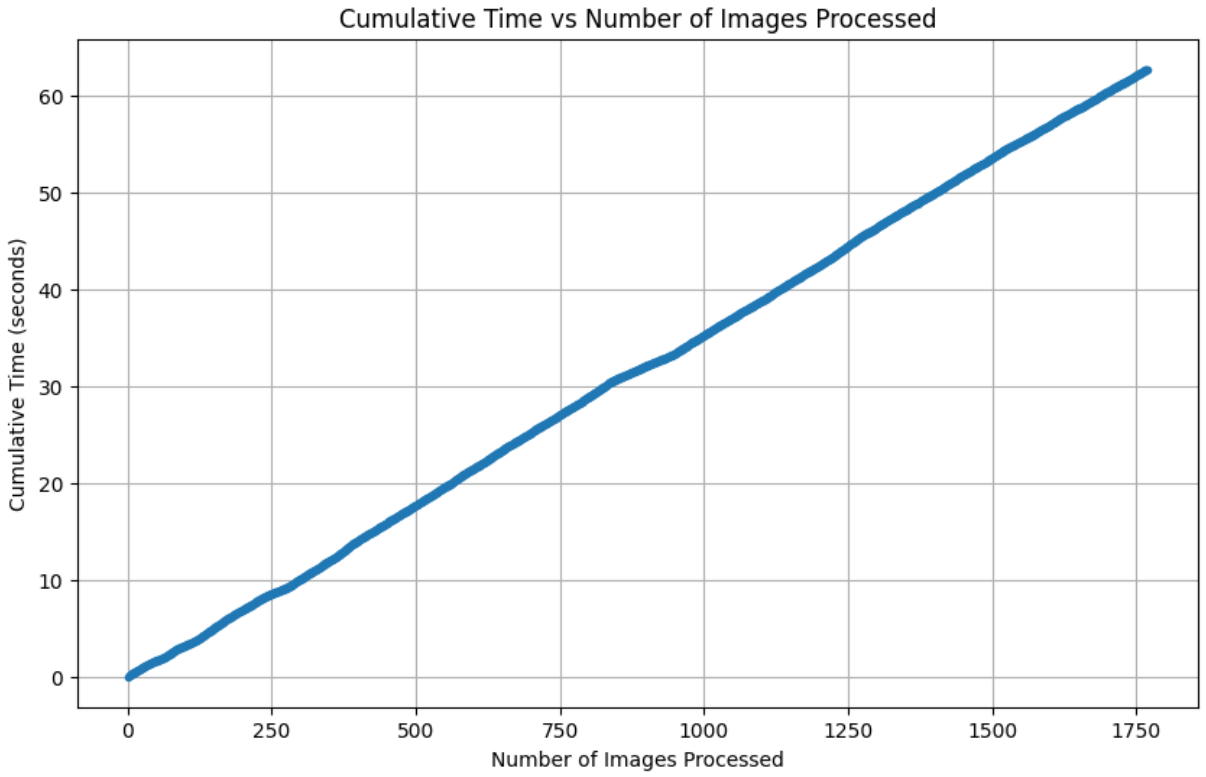
Sequential Edge Detection

- **Overview of Sequential Execution**:
  - The image is processed pixel by pixel using the Sobel operator
  - The Sobel filter calculates the gradient magnitude for each pixel, producing an edge-detected image
- **Code Walkthrough**: Brief steps of code for sequential edge detection
- **Performance in Sequential Computing**:
  - Sequential computation performs the task without distributing any processing
  - Performance is tied to the processor's speed

```
// Sobel filter function
void applySobel(const unsigned char* grayImage, unsigned char* edgeImage, int width, int height) {
    // Sobel operator kernels
    int gx[3][3] = {
        {-1, 0, 1},
        {-2, 0, 2},
        {-1, 0, 1}
    };

    int gy[3][3] = {
        {-1, -2, -1},
        {0, 0, 0},
        {1, 2, 1}
    };

    // Apply Sobel filter (edge detection)
    for (int y = 1; y < height - 1; y++) {
        for (int x = 1; x < width - 1; x++) {
            int grad_x = 0;
            int grad_y = 0;

            // Apply Sobel kernels (convolution operation)
            for (int ky = -1; ky <= 1; ky++) {
                for (int kx = -1; kx <= 1; kx++) {
                    int pixel = grayImage[(y + ky) * width + (x + kx)];
                    grad_x += gx[ky + 1][kx + 1] * pixel;
                    grad_y += gy[ky + 1][kx + 1] * pixel;
                }
            }

            // Calculate gradient magnitude and clamp to [0, 255]
            int edge_value = std::sqrt(grad_x * grad_x + grad_y * grad_y);
            edgeImage[y * width + x] = std::min(255, edge_value);
        }
    }
}
```

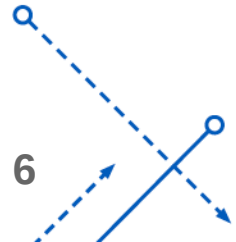
Sequential Performance Metrics

```
=== Performance Metrics ===
Total Images Processed: 1768
Total Execution Time: 71.3722 seconds
Average Time per Image: 0.0354552 seconds
Throughput: 24.7716 images per second
All images processed successfully!
```



Parallel Edge Detection Using MPI

- **How Parallel Edge Detection Works**: **Data Decomposition**:
  - The image is divided into smaller sub-images, each processed by a separate MPI process
  - Each process applies the Sobel operator to its portion of the image
- **Communication Between Processes**:
  - After processing, the results (edges) from all the processes are gathered and combined to form the final output image
- **Code Walkthrough**:
  - Initialization of MPI and process allocation
  - Distribution of image data to different processes
  - Synchronization and collection of results



Parallel Edge Detection Using MPI

```cpp
int main(int argc, char *argv[]) {
    MPI_Init(&argc, &argv);
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    char hostname[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(hostname, &name_len);

    if (rank == 0) {
        std::cout << "Total MPI Processes Running: " << size << std::endl;
    }

    // Print rank and hostname to see where each process is running
    std::cout << "Process " << rank << " is running on node " << hostname << std::endl;


    double start_time = MPI_Wtime();

    // Collect image paths
    std::vector<std::string> all_image_paths;
    if (rank == 0) {
        std::string datasetPath = "dataset/images";
        for (const auto& entry : fs::directory_iterator(datasetPath)) {
            if (entry.is_regular_file()) {
                std::string path = entry.path().string();
                if (path.find(".jpg") != std::string::npos || path.find(".png") != std::string::npos)
                    all_image_paths.push_back(path);
                }
            }
        }

        // Ensure output directory exists
        fs::create_directories("edges_output");
    }
```

```cpp
    // Broadcast number of images
    int num_images = 0;
    if (rank == 0) num_images = all_image_paths.size();
    MPI_Bcast(&num_images, 1, MPI_INT, 0, MPI_COMM_WORLD);

    std::vector<std::string> local_images;

    if (rank == 0) {
        // Dsitribute the images per process
        int images_per_proc = (num_images + size - 1) / size;

        for (int proc = 0; proc < size; proc++) {
            int start_idx = proc * images_per_proc;
            int end_idx = std::min(start_idx + images_per_proc, num_images);  // To make sure each process

            if (proc == 0) {
                for (int i = start_idx; i < end_idx; i++) {
                    local_images.push_back(all_image_paths[i]);
                }
            } else {
                int count = end_idx - start_idx;
                MPI_Send(&count, 1, MPI_INT, proc, 0, MPI_COMM_WORLD);

                for (int i = start_idx; i < end_idx; i++) {
                    int length = all_image_paths[i].length() + 1;
                    MPI_Send(&length, 1, MPI_INT, proc, 1, MPI_COMM_WORLD);
                    MPI_Send(all_image_paths[i].c_str(), length, MPI_CHAR, proc, 2, MPI_COMM_WORLD);
                }
            }
        }
    } else {
        int count;
        MPI_Recv(&count, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        for (int i = 0; i < count; i++) {
            int length;
            MPI_Recv(&length, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

Parallel Edge Detection Using MPI

```
MPI_Barrier(MPI_COMM_WORLD);

double processing_start = MPI_Wtime();

for (const auto& imagePath : local_images) {
    int width, height, channels;
    unsigned char* data = stbi_load(imagePath.c_str(), &width, &height, &channels, 1);

    if (!data) {
        std::cerr << "Rank " << rank << " - Error loading image: " << imagePath << std::endl;
        continue;
    }

    unsigned char* resized_data = new unsigned char[NEW_WIDTH * NEW_HEIGHT];
    stbir_resize_uint8(data, width, height, 0, resized_data, NEW_WIDTH, NEW_HEIGHT, 0, 1);
    stbi_image_free(data);

    unsigned char* edge_data = new unsigned char[NEW_WIDTH * NEW_HEIGHT]();

    for (int y = 1; y < NEW_HEIGHT - 1; y++) {
        for (int x = 1; x < NEW_WIDTH - 1; x++) {
            int gx = 0, gy = 0;

            for (int ky = -1; ky <= 1; ky++) {
                for (int kx = -1; kx <= 1; kx++) {
                    int pixel = resized_data[(y + ky) * NEW_WIDTH + (x + kx)];
                    gx += kx * pixel;
                    gy += ky * pixel;
                }
            }

            int edge_value = std::sqrt(gx * gx + gy * gy);
            edge_data[y * NEW_WIDTH + x] = std::min(255, edge_value);
        }
    }
```

```
    std::string filename = fs::path(imagePath).filename().string();
    std::string outputPath = "edges_output/sobel_" + filename;

    stbi_write_jpg(outputPath.c_str(), NEW_WIDTH, NEW_HEIGHT, 1, edge_data, 90);

    delete[] resized_data;
    delete[] edge_data;
}

double processing_end = MPI_Wtime();
double processing_time = processing_end - processing_start;
double max_processing_time;
MPI_Reduce(&processing_time, &max_processing_time, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);

double end_time = MPI_Wtime();
double total_parallel_time = end_time - start_time;

if (rank == 0) {
    double sequential_time = processImagesSequentially(all_image_paths);
    std::cout << "Total parallel execution time: " << total_parallel_time << " seconds.\n";
    std::cout << "Sequential execution time: " << sequential_time << " seconds.\n";
    std::cout << "Observed speedup: " << sequential_time / total_parallel_time << "x\n";

    // Output to file for plotting
    std::ofstream output_file("performance_data_biggerdataset.csv", std::ios::app);
    output_file << size << "," << sequential_time << "," << total_parallel_time << ","
                << sequential_time / total_parallel_time << "\n";

    // Calculate Amdahl's and Gustafson's speedup
    double amdahl_speedup = calculateAmdahlSpeedup(sequential_time, total_parallel_time, size);
    double gustafson_speedup = calculateGustafsonSpeedup(sequential_time, total_parallel_time, size);

    std::cout << "Amdahl's Speedup: " << amdahl_speedup << "x\n";
    std::cout << "Gustafson's Speedup: " << gustafson_speedup << "x\n";
}
```

Slurm Script

- **Configuration used for Sobel Parallel Program**:
  - Script with parameters –node and –ntasks changed for different values
  - Table in Slide 10 with configs run

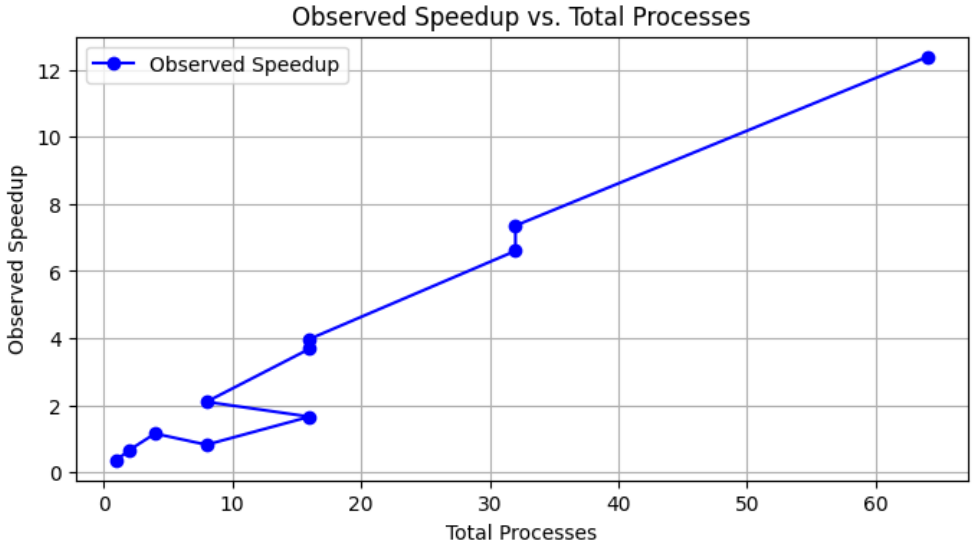
```
#!/bin/bash
#SBATCH --cluster=ub-hpc
#SBATCH --partition=general-compute
#SBATCH --qos=general-compute
#SBATCH --account=cse633
#SBATCH --time=00:40:00
#SBATCH --nodes=8
#SBATCH --ntasks-per-node=8
#SBATCH --mem=10GB
#SBATCH --mail-type=END
#SBATCH --mail-user=ra65@buffalo.edu
#SBATCH --output=sobelmpi.out
#SBATCH --job-name=sobelmpi
#
```

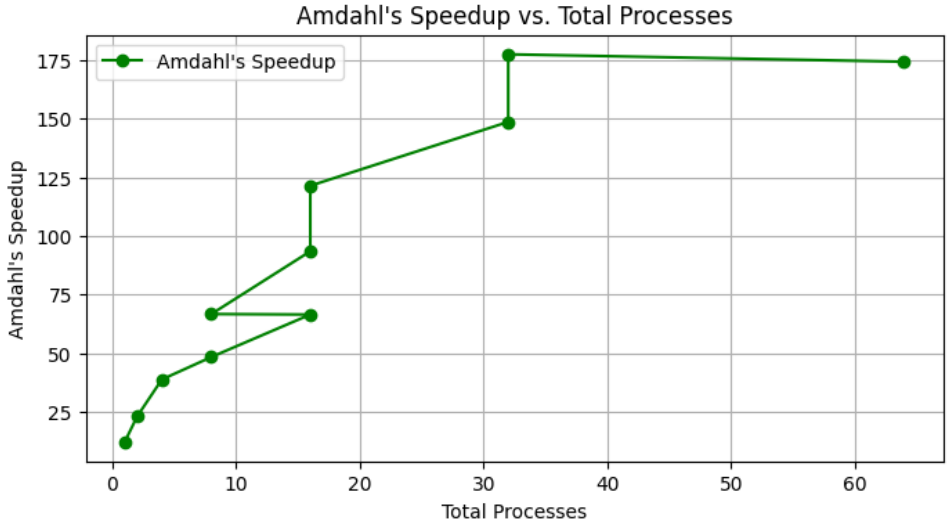
Performance Analysis

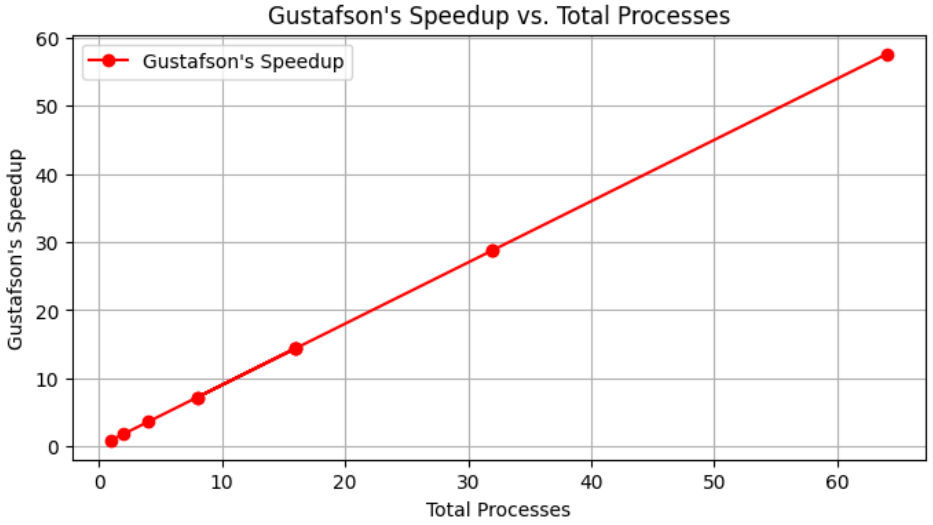
Nodes (N)	Processes per Node (P/N)	Total Processes (P)	Total Execution time
1	1	1	**34.1823 secs**
1	2	2	**19.2634 secs**
1	4	4	**10.9802 secs**
1	8	8	**12.5769 secs**
1	16	16	**6.28501 secs**
2	4	8	**6.75986 secs**
2	8	16	**3.96967 secs**
4	4	16	**4.77486 secs**
4	8	32	**2.88434 secs**
8	4	32	**3.09034 secs**
8	8	64	**1.60407 secs**

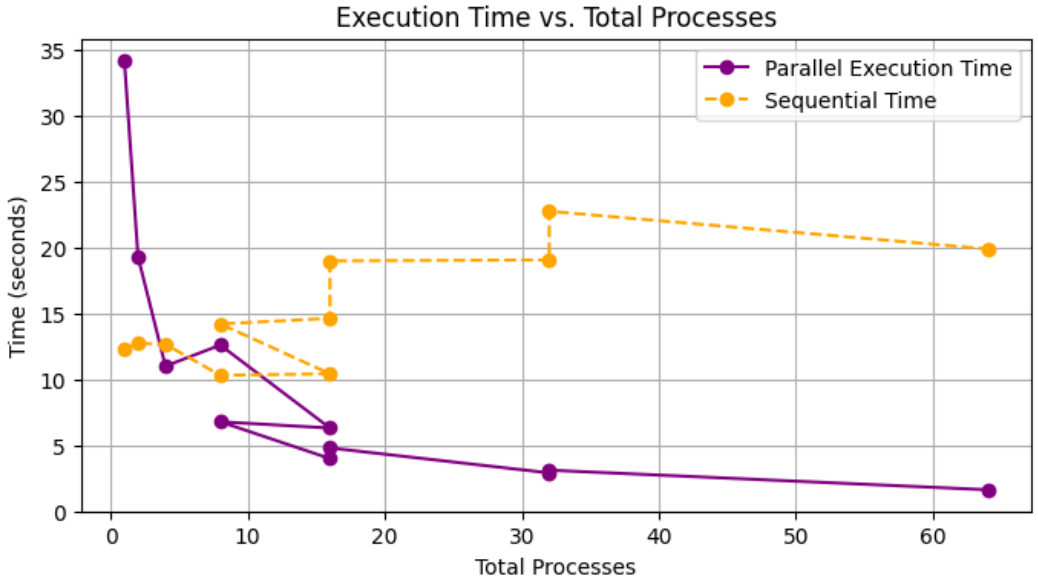
Performance Analysis

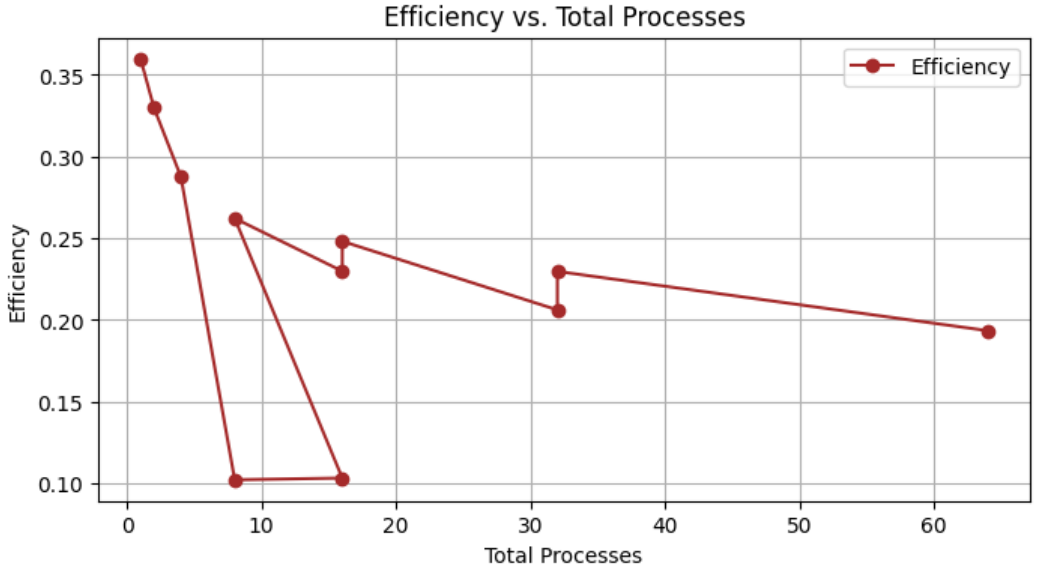
$$S_{\text{observed}} = \frac{T_{\text{sequential}}}{T_{\text{parallel}}}$$

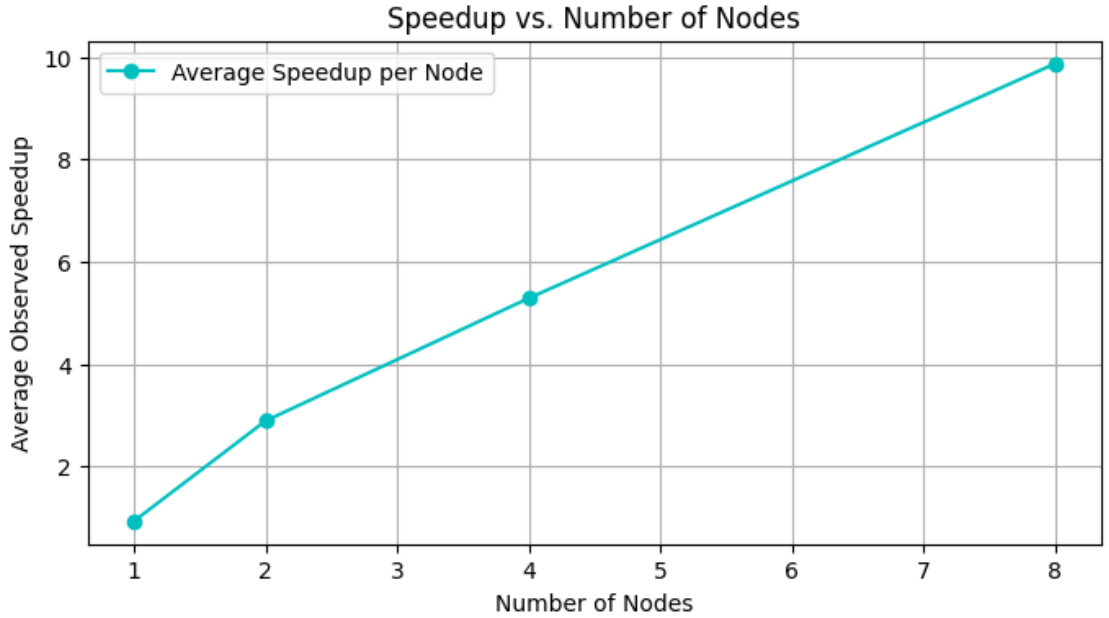







Execution Time vs. Total Processes
Parallel Execution Time
Sequential Time
Time (seconds)
Total Processes
0
5
10
15
20
25
30
35
0
10
20
30
40
50
60

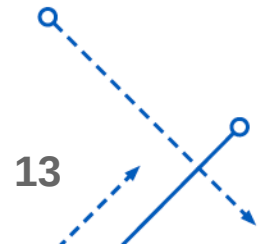

Efficiency vs. Total Processes
Efficiency
Efficiency
Total Processes
0.10
0.15
0.20
0.25
0.30
0.35
0
10
20
30
40
50
60


Speedup vs. Number of Nodes
Average Speedup per Node
Average Observed Speedup
Number of Nodes
2
4
6
8
10
1
2
3
4
5
6
7
8

Summary of Results Above

- Parallel program follows master-worker setup
- Setup not the best way to learn parallel computing
- Amdahl's and Gustafson's Speedup hardcoded for theoretical understanding
- Hence updated the setup in phase 2

Updated The Program Setup Post Midterm



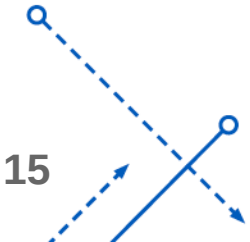
University at Buffalo The State University of New York

POST MIDTERM

MPI + OpenMP

Approach #1:

- Combined MPI inter-process distribution
- Added OpenMP thread-level acceleration
- Multi-threading of MPI process (controlled in SLURM Script)
- Different images are processed by different ranks
- Use ***#pragma omp parallel for collapse(2***) directive to parallelize the nested loops



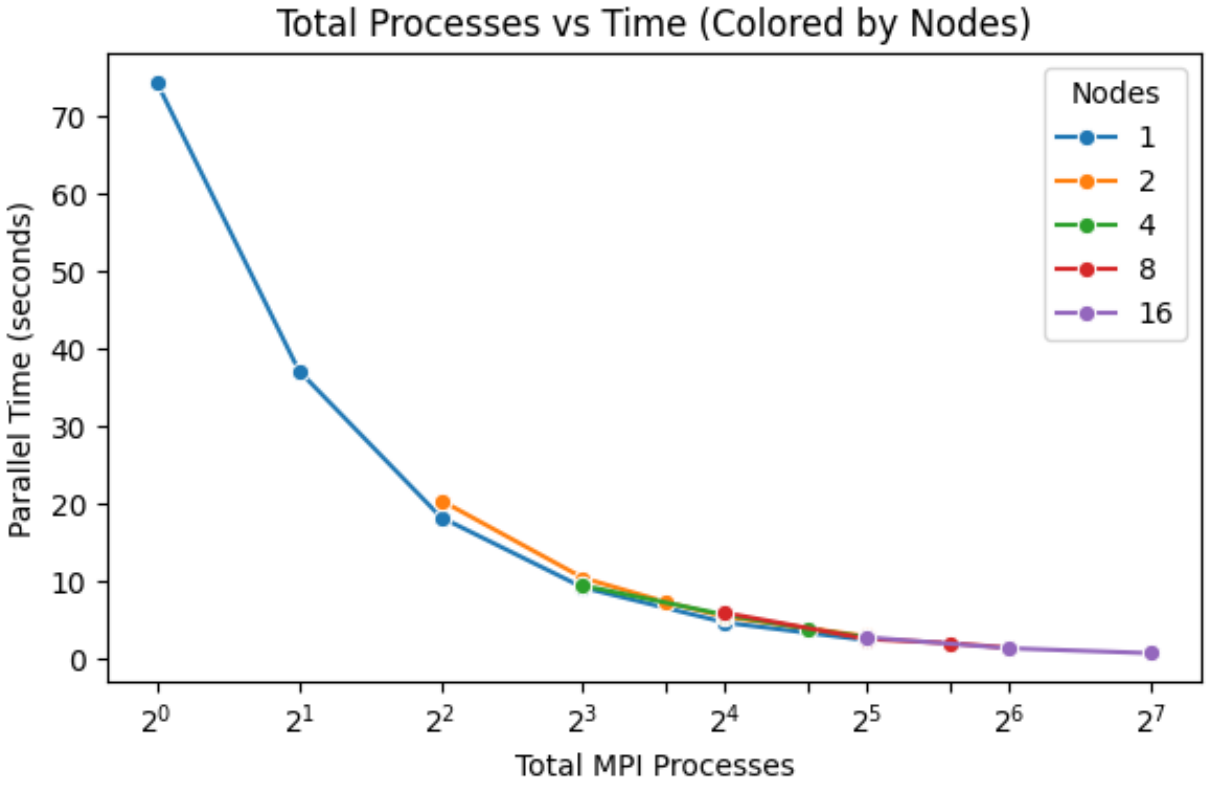
Slurm Configurations

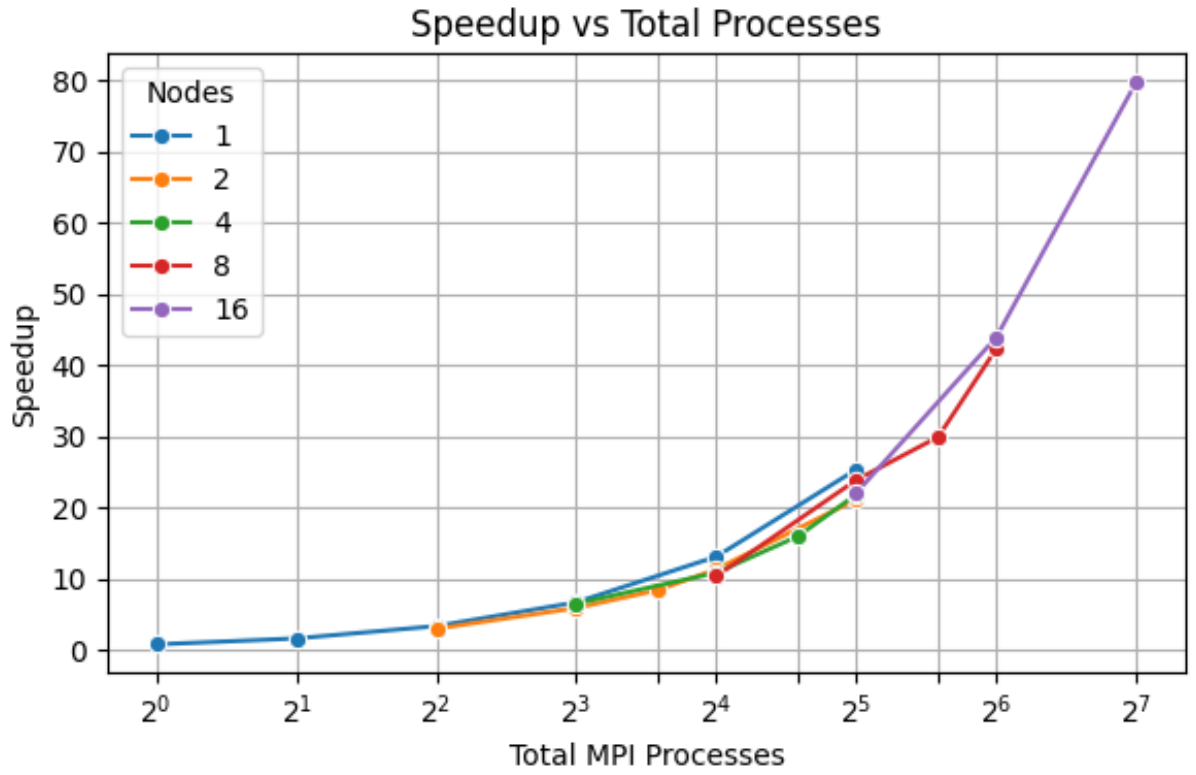
Approach #1: Config Table

Nodes (N)	Processes per Node (P/N)	Total Processes (P)	ntasks	ntasks-per-node	total_parallel_time_seconds
1	1	1	1	1	74.3479
1	2	2	2	2	37.1944
1	4	4	4	4	18.3074
1	8	8	8	8	9.24701
1	16	16	16	16	4.7412
1	32	32	32	32	2.45626
2	2	4	4	2	20.4809
2	4	8	8	4	10.4802
2	6	12	12	6	7.36726
2	8	16	16	8	5.47671
2	16	32	32	16	2.9443
4	2	8	8	2	9.5019
4	4	16	16	4	5.72176
4	6	24	24	6	3.89099
4	8	32	32	8	2.8501
8	2	16	16	2	5.98394
8	4	32	32	4	2.60923
8	6	48	48	6	2.08199
8	8	64	64	8	1.47336
16	2	32	32	2	2.8219
16	4	64	64	4	1.41931
16	8	128	128	8	0.781925

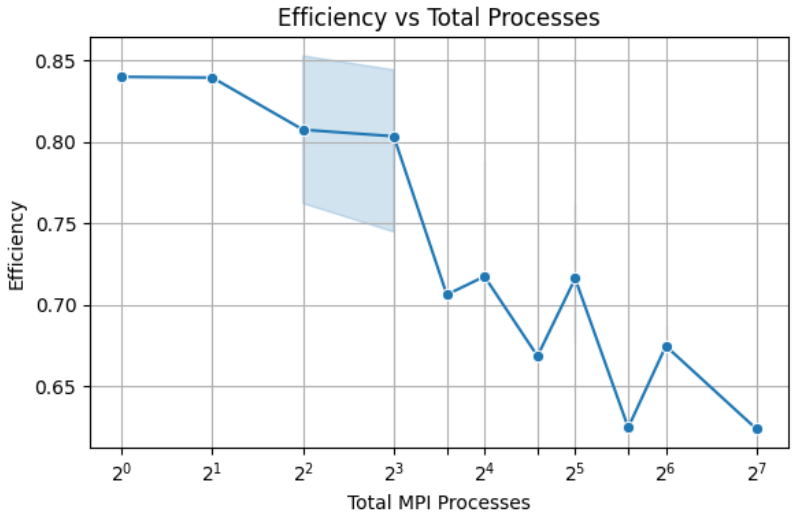
Performance Analysis

Approach #1

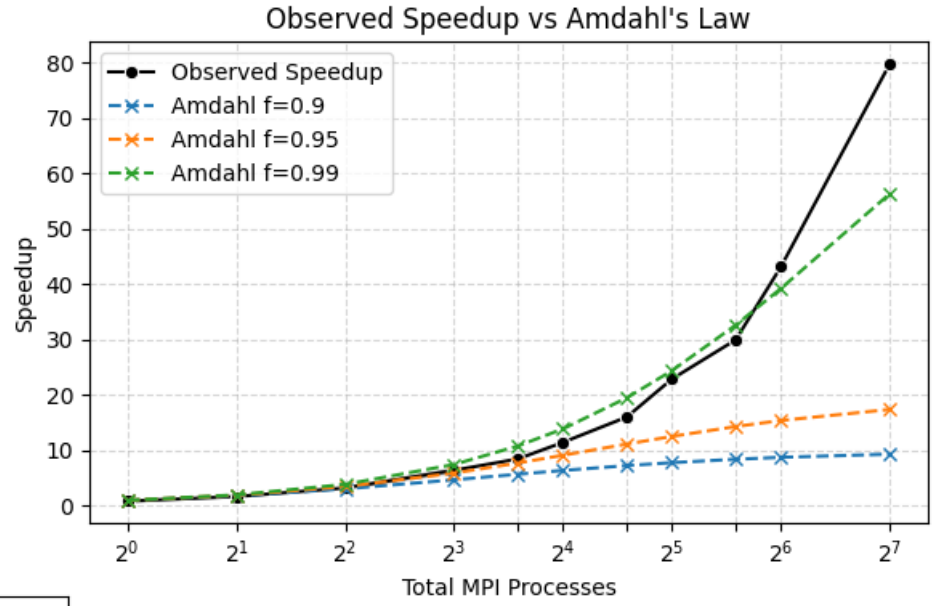


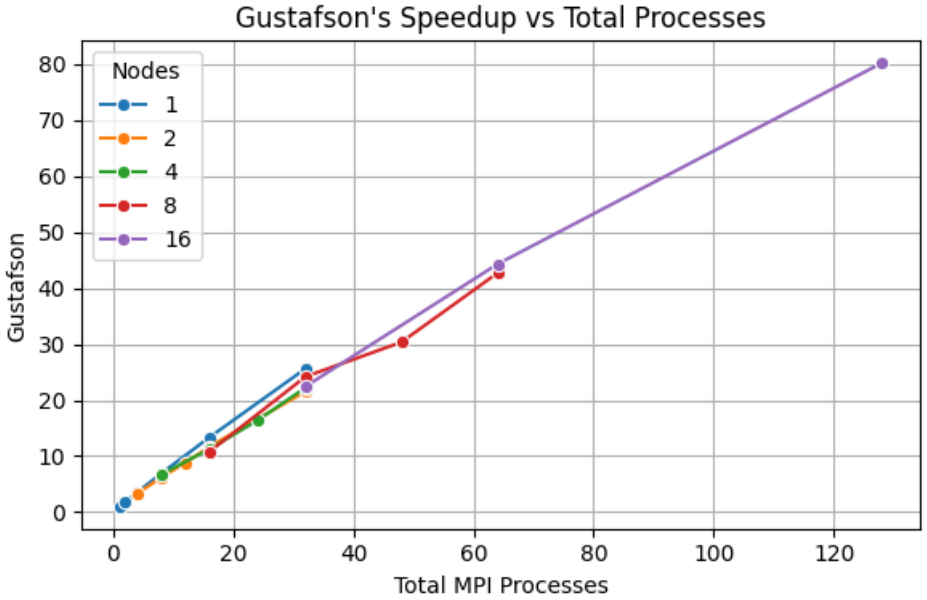


Performance Analysis



Approach #1

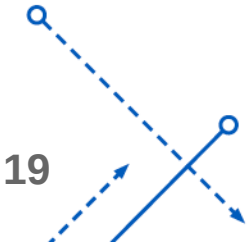




MPI + OpenMP Approach

Approach #2:

- Using a tiling strategy
- Parallelization happens at the tile level with ***#pragma omp parallel for***
- Improved cache locality
- Optimized Sobel implementation



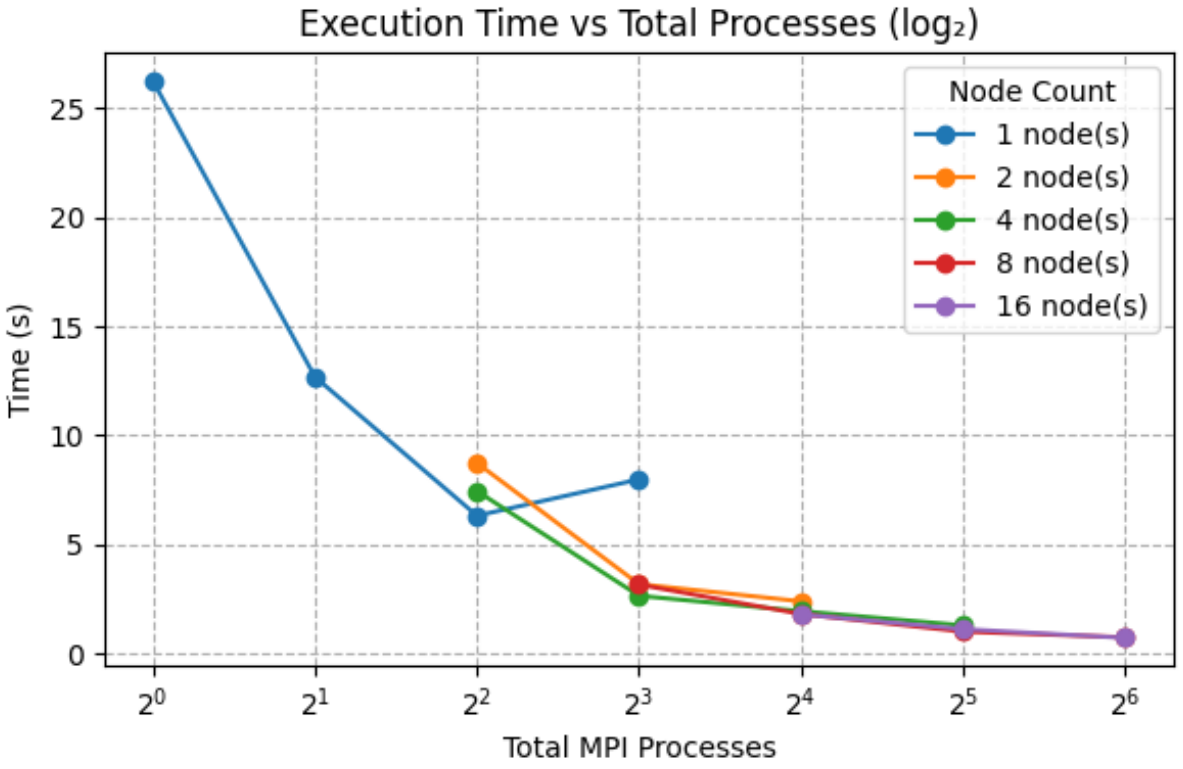
Slurm Configurations

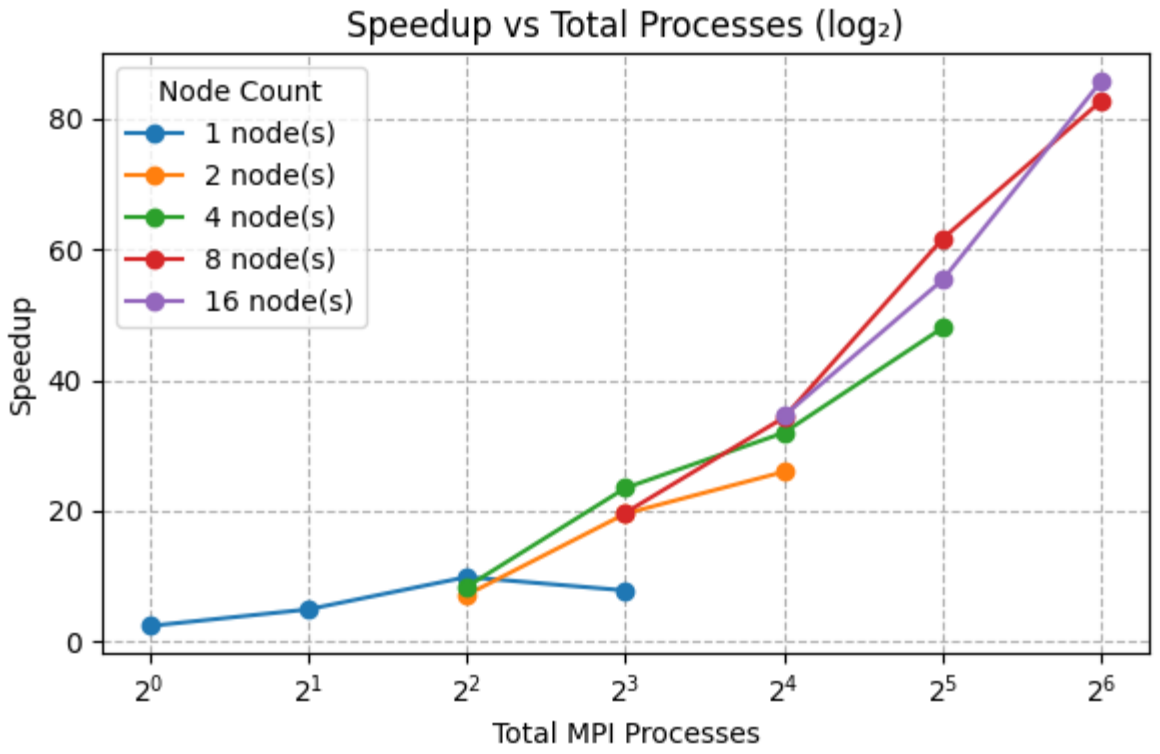
Approach #2: Config Table

ntasks	ntasks-per-node	cpus-per-task	OMP_NUM_THREADS	Total Cores	total_parallel_time_seconds
1	1	1	1	1	26.2397
2	2	1	1	2	12.686
4	4	1	1	4	6.32322
4	2	2	2	8	8.74173
4	1	4	4	16	7.45413
8	8	1	1	8	7.98683
8	4	2	2	16	3.1891
8	2	4	4	32	2.65627
16	8	1	1	16	2.402
16	4	2	2	32	1.95275
8	1	8	8	64	3.16731
16	1	8	8	128	1.80725
16	2	4	4	64	1.82037
32	4	2	2	64	1.01261
32	8	1	1	32	1.30007
32	2	4	4	128	1.12771
64	4	2	2	128	0.727688
64	8	1	1	64	0.754323

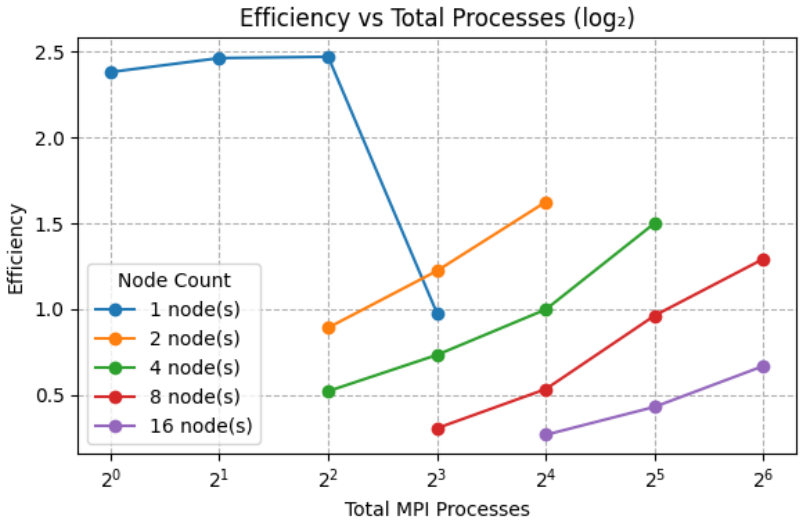
Performance Analysis

Approach #2

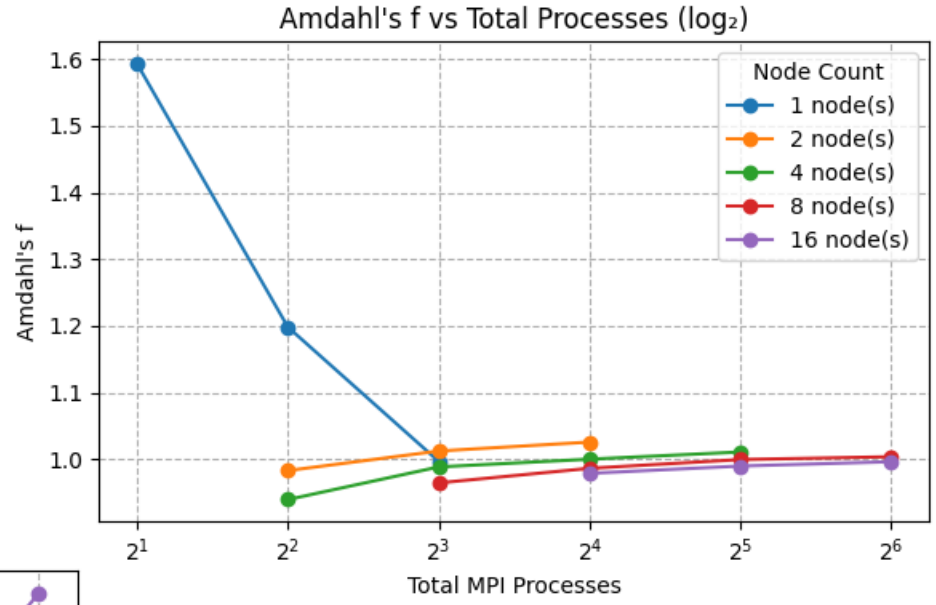


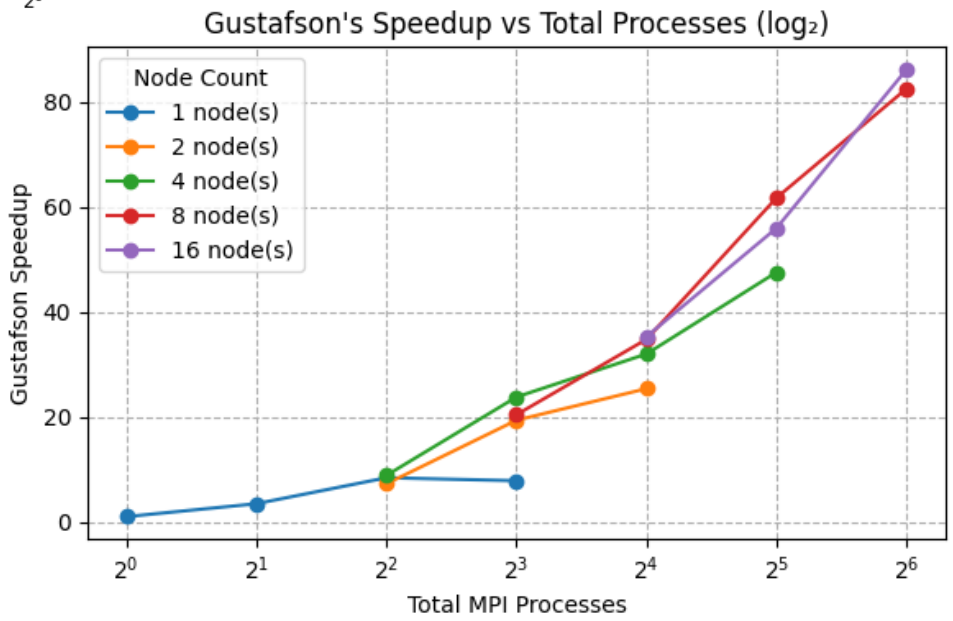


Performance Analysis



Approach #2





Final Notes

- Successful implementation of hybrid parallelism with MPI and OpenMP
- Performance metrics showed good scaling, and code mostly parallelized
- Future improvements/experiments:
  - Use more data
  - Scale to more number of nodes i.e. 32, 64 and more if possible
  - Use CUDA and process the computation over GPU’s

End Result

References

- https://medium.com/@twinnroshan/understanding-and-implementing-edge-detection-in-c-with-sobel-operator-31159f26587c
- https://aryamansharda.medium.com/how-image-edge-detection-works-b759baac01e2
- https://www.geeksforgeeks.org/what-is-edge-detection-in-image-processing/
- https://medium.com/lcc-unison/applying-sobel-filter-for-image-processing-using-parallel-computing-d1eae128b4e
- https://medium.com/data-science/sobel-operator-in-image-processing-1d7cdda8cadb
- https://www.geeksforgeeks.org/sobel-edge-detection-vs-canny-edge-detection-in-computer-vision/
- https://buffalo.app.box.com/s/vb6lkxg72jgekuyo5xbps7xesfj076ok

University at Buffalo The State University of New York

THANK YOU!