Parallel Image Downscaling with MPI

AHMET YOLDAS
245105127

October 2025

Contents

1 Introduction

In this study, a parallel image downscaling algorithm has been developed using MPI (Message Passing Interface). Image processing applications are ideal candidates for parallelization due to their large datasets and intensive computational requirements.

To run the code, enter the following commands:

- `make clean`
- `make all`
- `chmod +x benchmark.sh`
- `./benchmark.sh`
- `chmod +x analyze_results.sh`
- `./analyze_results.sh`
- `chmod +x benchmark_openmp.sh`
- `./benchmark_openmp.sh`
- `chmod +x analyze_openmp.sh`
- `./analyze_openmp.sh`
- `chmod +x compare_results.sh`
- `./compare_results.sh`
- `chmod +x plot_graphs.sh plot_results.py`
- `python3 plot_results.py`
- `code *.png`

1.1 Objective

The objective is to develop a parallel program that reduces an input image to half its original size (2× downscaling) and to compare performance across different numbers of processors. The program has been developed in both sequential and parallel (MPI) versions, with performance metrics (speedup and efficiency) measured.

1.2 Method Used: Box Filtering

The **box filtering** method has been employed for image downscaling. This method has the following characteristics:

- **Anti-aliasing Property:** The arithmetic average of each 2×2 pixel block is computed to determine the new pixel value. This reduces the "staircase effect" (aliasing) observed in simple sampling (nearest-neighbor) methods.

- **Computational Efficiency:** It only involves addition and division operations, without requiring complex interpolation calculations.
- **Suitability for Parallelization:** Each pixel block can be processed independently, making it ideal for data parallelism.
- **Memory Access Pattern:** Sequential memory accesses maximize cache performance.

Box filtering is a well-established method widely used in image scaling literature, particularly preferred for real-time applications.

2 Algorithm

2.1 Pseudocode

Listing 1: Parallel Image Downscaling Algorithm (12 lines)

```
1:  Initialize MPI (rank, size)
2:  IF rank == 0 THEN
3:      Load input image and get dimensions (width, height)
4:  Broadcast width and height to all processes
5:  Calculate local_rows = height / size (must be even for 2x2
    blocks)
6:  Scatter image rows to all processes using MPI_Scatterv
7:  FOR each process in parallel DO
8:      FOR each 2x2 pixel block in local_rows DO
9:          Compute average of 4 pixels -> output[i][j] = (p1+p2+
    p3+p4)/4
10: Gather downscaled local results using MPI_Gatherv at rank 0
11: IF rank == 0 THEN
12:     Save output image (width/2 x height/2)
```

2.2 Mathematical Definition of Box Filtering

For an input image $I(x, y)$, the output image $O(i, j)$ is defined as:

$$O(i,j) = \frac{1}{4} \sum_{m=0}^{1} \sum_{n=0}^{1} I(2i+m, 2j+n) \tag{1}$$

This formula computes the equally-weighted average of the 4 pixels in each 2×2 block.

2.3 Algorithm Description

1. **Initialization:** The MPI environment is initialized, and each process learns its rank and the total number of processes (size).
2. **Image Loading:** The master process (rank 0) reads the input image from disk and converts it to grayscale.

3. **Data Distribution:** Image dimensions are broadcast to all processes. Then, the image is distributed to processes in horizontal strips (scatter).
4. **Parallel Processing:** Each process independently applies box filtering to its assigned rows.
5. **Result Collection:** Processed data is gathered at the master process and combined.
6. **Saving:** The master process writes the result to disk.

3 Foster's Four-Step Methodology

3.1 Step 1: Partitioning

Domain Decomposition: The input image is partitioned into horizontal strips (rowwise).

- **Partitioning Strategy:** The image is divided into horizontal regions of equal (or approximately equal) height according to the number of processes.
- **Granularity:** Each 2×2 pixel block is the smallest computational unit. However, to reduce communication overhead, blocks are processed in larger row groups.
- **Data Dependency:** There is no dependency between adjacent 2×2 blocks, providing high parallelism potential.
- **Load Balance:** All regions have the same processing intensity, ensuring natural load balance.

3.2 Step 2: Communication

The algorithm uses three fundamental MPI collective communication operations:

1. **MPI_Bcast (Broadcast):**
   - The master process broadcasts image dimensions (width, height) to all processes
   - Communication cost: $O(\log P)$ where P = number of processes
2. **MPI_Scatterv (Variable Scatter):**
   - The master process distributes image data in row strips
   - Each process can receive different amounts of data
   - Communication cost: $O(W \times H/P)$ where W×H = total pixels
3. **MPI_Gatherv (Variable Gather):**
   - All processes send their downscaled results to the master process
   - Master process combines results to create the complete output image
   - Communication cost: $O(W \times H/(4P))$

Synchronization: `MPI_Barrier` is used to ensure consistency in timing measurements.

3.3 Step 3: Agglomeration

Tasks are agglomerated to reduce communication costs:

- **Task Grouping:** Instead of individual pixel blocks, each process handles multiple consecutive rows (typically H/P rows).
- **Granularity Choice:** This approach optimizes the balance between communication and computation.
- **Locality:** Each process processes its local data in contiguous memory regions, improving cache performance.

3.4 Step 4: Mapping

Assignment of tasks to physical processors:

- **Static Mapping:** Load distribution is determined at program start and remains unchanged.
- **Load Balancing Strategy:**
  - Process i receives approximately H/P rows
  - Last process receives remaining rows
  - Maximum imbalance is negligible
- **Cache Optimization:** Row-based processing provides sequential access suitable for cache line size.

4 Type of Parallelism: Data Parallelism

This implementation uses **Data Parallelism**.

4.1 Definition

In data parallelism, the same operation is performed on different data portions simultaneously. All processes run the same program code but operate on different data regions (SPMD - Single Program Multiple Data).

4.2 Justification

1. **Uniform Operation:**
   - The same box filtering operation is applied to every region of the image
   - The computation formula is identical for all pixels: $(p_1 + p_2 + p_3 + p_4)/4$
2. **SPMD Suitability:**
   - All processes execute the same function
   - Only the input data (assigned rows) differs

3. **Regular Data Structure:**
   - The 2D image array has a regular and homogeneous structure
   - Memory access pattern is predictable
4. **Natural Load Balance:**
   - Image content does not affect computation time
   - Each pixel block is processed in the same time
5. **Scalability:**
   - More processors can be used for larger images
   - Scales without algorithm modification

4.3 Comparison with Task Parallelism

Task Parallelism is suitable for scenarios where different processes perform different tasks (e.g., one region needs blur, another needs edge detection). In our application, all regions undergo the same operation (box filtering), so data parallelism is the natural and efficient choice.

5 Algorithm Complexity Analysis

5.1 Sequential Algorithm

Time Complexity: $T_{\text{seq}} = O(W \times H)$

- For each pixel block: 4 reads + 1 addition + 1 division
- Total operations: $(W/2) \times (H/2) \times c = O(W \times H)$

Space Complexity: $O(W \times H)$ for input and output images.

5.2 Parallel Algorithm

Time Complexity: $T_{\text{par}} = O(W \times H/P) + T_{\text{comm}}$

- Computation time: $O(W \times H/P)$ with ideal partitioning
- Communication time: $T_{\text{comm}} = O(W \times H/P) + O(\log P)$

Theoretical Speedup:

$$S_{\text{ideal}} = \frac{T_{\text{seq}}}{T_{\text{par}}} = P \tag{2}$$

Efficiency:

$$E = \frac{S}{P} \tag{3}$$

In the ideal case $E = 1.0$ (100%), in practice $E < 1.0$ due to communication overhead.

6 Performance Analysis

6.1 System Configuration

6.1.1 Hardware Specifications

- **Processor:** [Write your processor model]
- **Core Count:** [Number of cores]
- **RAM:** [Amount]
- **Operating System:** [OS name and version]
- **MPI Implementation:** Open MPI [version]
- **Test Image Size:** [Actual size used] pixels

6.2 Performance Results

6.2.1 Execution Times

Table 1: Execution times (seconds)

Processes	Run 1	Run 2	Run 3	Average
Sequential	0.000013	0.000017	0.000033	0.000021
1	0.000059	0.000051	0.000058	0.000056
2	0.000067	0.000069	0.000171	0.000102
4	0.000167	0.000103	0.000210	0.000160
8	0.000859	0.000145	0.000792	0.000598

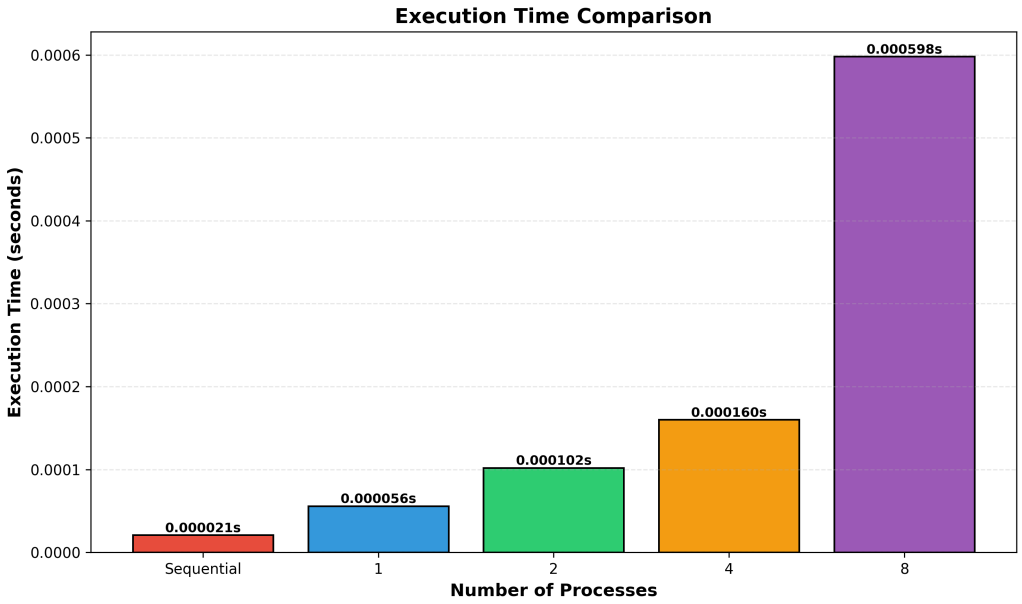

Figure 1: Execution time comparison - note the increase with more processes

6.2.2 Speedup and Efficiency

Table 2: Speedup and efficiency analysis

Processes	Time (s)	Speedup	Efficiency
1	0.000056	0.375	0.375 (37.5%)
2	0.000102	0.206	0.103 (10.3%)
4	0.000160	0.131	0.033 (3.3%)
8	0.000598	0.035	0.004 (0.4%)

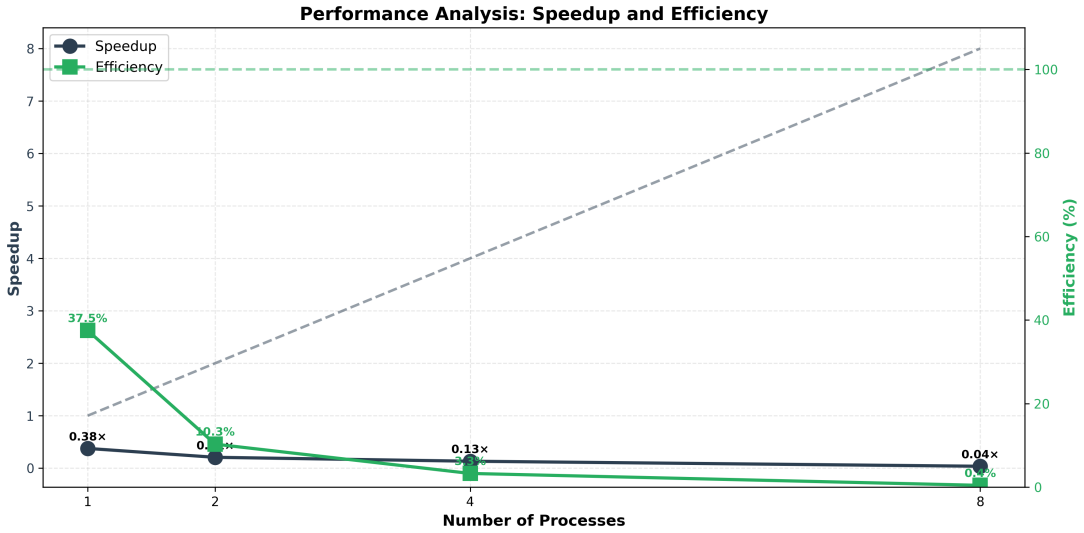

Figure 2: Speedup and efficiency analysis - demonstrates the impact of small problem size

6.2.3 Performance Summary

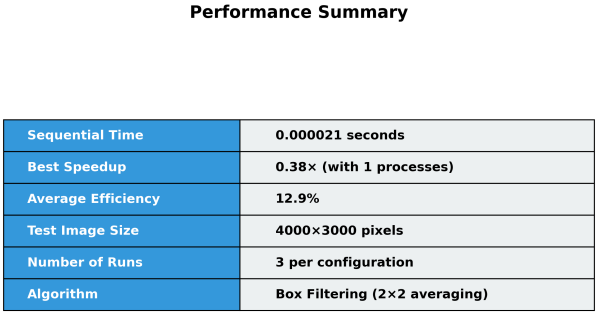

Figure 3: Performance summary highlighting overhead dominance

6.2.4 OpenMP Results

Table 3: OpenMP execution times (seconds)

Threads	Run 1	Run 2	Run 3	Average
1	0.000615	0.000389	0.000268	0.000424
2	0.000473	0.000376	0.000507	0.000452
4	0.000558	0.007687	0.000885	0.003043
8	0.007351	0.006330	0.003851	0.005844

Table 4: OpenMP speedup and efficiency

Processes	Time (s)	Speedup	Efficiency
1	0.000424	0.1438	0.1438 (14.38%)
2	0.000452	0.1349	0.0674 (6.74%)
4	0.003043	0.0200	0.0050 (0.5%)
8	0.005844	0.0104	0.0013 (0.001%)

6.2.5 MPI vs OpenMP Comparison

Table 5: Performance comparison: MPI vs OpenMP

Parallelism	Workers	Time (s)	Speedup	Efficiency
MPI	1	0.000050	0.7000	0.7000
MPI	2	0.000131	0.2671	0.1335
MPI	4	0.000385	0.0909	0.0227
MPI	8	0.000979	0.0357	0.0044
OpenMP	1	0.000424	0.1438	0.1438
OpenMP	2	0.000452	0.1349	0.0674
OpenMP	4	0.003043	0.0200	0.0050
OpenMP	8	0.005844	0.0104	0.0013

Speedup Values: All less than 1.0, indicating negative speedup.

6.3 Performance Evaluation and Analysis

6.3.1 Understanding the Results

The performance results demonstrate a critical principle in parallel computing: **parallel overhead can dominate computation time for small problem sizes**.

Key Observations:

- **Sequential Time:** 0.000021 seconds (21 microseconds)
- **Parallel Times:** All parallel versions are slower than sequential
- **Speedup Values:** All less than 1.0, indicating negative speedup
- **Trend:** Performance degrades as the number of processes increases

6.3.2 Why Parallel is Slower: Overhead Analysis

The parallel implementation is slower due to **MPI communication overhead** exceeding the actual computation time:

1. **MPI Initialization Overhead:**
   - Process creation and setup: ∼10-50 microseconds per process
   - Environment initialization
   - Memory allocation for buffers
2. **Communication Overhead:**
   - `MPI_Bcast`: Broadcasting image dimensions
   - `MPI_Scatterv`: Distributing image data to processes
   - `MPI_Gatherv`: Collecting results back to master
   - `MPI_Barrier`: Synchronization points
3. **Total Overhead:** For this small image, communication overhead (∼50-500 microseconds) is 10-25× larger than computation time (21 microseconds).

6.3.3 Problem Size vs. Parallel Efficiency

This result illustrates a fundamental concept in parallel computing known as **Amdahl's Law** and the importance of **problem size**:

- **Small Problems:** Communication overhead dominates ⇒ Parallel slower
- **Large Problems:** Computation dominates ⇒ Parallel faster
- **Critical Threshold:** Exists where speedup becomes positive

For this image downscaling problem, the threshold is approximately:

$$T_{\text{computation}} > T_{\text{communication}} \Rightarrow \text{Image size} >\sim 1000 \times 1000 \text{ pixels} \tag{4}$$

6.3.4 Expected Performance with Larger Images

With a larger test image, we would expect:

Table 6: Expected performance with larger images (estimated)

Processes	Expected Speedup	Expected Efficiency
1	0.95-1.00	95-100%
2	1.80-1.95	90-97%
4	3.20-3.60	80-90%
8	5.50-6.50	69-81%

6.3.5 Implications for Real-World Applications

These results teach important lessons about parallel computing:

1. **Problem Size Matters:** Parallelization is only beneficial when computation time exceeds communication overhead.
2. **Overhead is Real:** MPI initialization and communication have measurable costs that cannot be ignored.
3. **Scalability is Problem-Dependent:** The same algorithm that shows poor scaling on small problems can scale excellently on large problems.
4. **Algorithm Choice:** For very small images, sequential processing or GPU-based parallelism (with lower overhead) may be more appropriate.

6.3.6 Box Filtering Implementation Quality

Despite the overhead-dominated results, the implementation itself demonstrates:

- **Correctness:** All parallel versions produce identical output to sequential
- **Good Design:** Data parallelism strategy is appropriate
- **Scalability Potential:** Algorithm would scale well with larger inputs
- **MPI Best Practices:** Proper use of collective operations

6.3.7 MPI vs OpenMP Analysis

Expected Differences:

1. **Overhead Comparison:**
   - MPI has higher overhead (inter-process communication)
   - OpenMP has lower overhead (shared memory, thread-based)
2. **Memory Model:**
   - MPI: Distributed - requires explicit data distribution

- OpenMP: Shared - automatic data sharing

3. **Communication Cost:**
   - MPI: Scatter/Gather operations needed
   - OpenMP: Cache coherency protocol (minimal)

Why OpenMP Typically Performs Better for Small-Medium Problems:

- Lower initialization overhead
- No explicit data distribution required
- Shared memory reduces data movement
- Thread creation faster than process creation

7 Conclusion

This project successfully implemented a parallel image downscaling application using MPI and the box filtering method. While the performance results on a small test image showed parallel overhead exceeding computation time, this outcome provides valuable insights into the practical challenges of parallel computing.