Automatic Parallelization Proposal

November 30, 2025

Submitted to:

Dr. Islam Tharwat

Eng. Hamdy Osama

Project Team

Ahmed Mohamed Al Amin	**22P0137**
Mohamed Ahmed Abdelhamid	**22P0287**
Seif Elhusseiny	**22P0215**
Youssef Moustafa Elsayed	**22P0047**

1 Introduction

1.1 Motivation

Median filtering is a widely used image-processing technique for removing impulsive noise while preserving important image features such as edges and structural details. It is applied in fields including medical imaging, satellite imaging, computer vision, and real-time video enhancement. However, the algorithm is computationally expensive because it requires extracting a neighborhood window around every pixel in the image and performing a sorting operation to compute the median value.

For high-resolution images such as 4096×4096, this results in more than sixteen million sorting operations, making sequential implementations slow and unsuitable for large-scale or time-critical applications. As modern processors increasingly rely on multi-core architectures and SIMD units, there is a growing need to adapt traditional algorithms to fully exploit available parallel hardware.

Automatic parallelization techniques aim to accelerate sequential programs by identifying independent operations and distributing them across processing units with minimal programmer intervention. Median filtering represents an ideal candidate for such techniques due to its inherent pixel-level parallelism.

1.2 What Has Been Done Before

Many researchers have focused on accelerating the median filter because of its high computational cost, especially on large images. Earlier work mainly optimized sequential implementations by reducing the number of comparisons or by using specialized data structures. However, as multi-core processors became standard, parallel approaches gained more attention.

A common method is to use OpenMP to parallelize the outer loops of the filter, since each pixel can be processed independently. This technique has shown significant speedups by distributing pixel computations across multiple CPU threads. Other studies explored MPI-based parallelization, where the image is divided into blocks and distributed across multiple processes, allowing large images to be processed in parallel on multi-node systems. These tools aim to automatically parallelize independent loop iterations without manual restructuring. However, the median filter remains challenging for fully automated compiler transformations due to the sorting step, which often requires manual guidance or domain-specific optimization.

More recent work has applied SIMD and SPMD approaches, such as Intel's ISPC, to exploit data-level parallelism inside the sorting and window-collection operations. These approaches allow multiple window elements to be processed simultaneously using vector instructions.

GPU-based methods (e.g., CUDA, OpenCL) have also been used to achieve real-time performance, but they require specialized hardware and are outside the scope of this project. Most CPU-based parallel implementations consistently demonstrate that median filtering is highly parallelizable, particularly at the pixel and loop levels, making it an ideal target for automatic parallelization techniques such as OpenMP and MPI.

2 Problem Definition

2.1 Description of the Median Filtering Problem

Median filtering is a nonlinear image-processing technique used primarily for removing impulsive noise (such as salt-and-pepper noise) from digital images while preserving important edges. The algorithm operates by replacing each pixel with the median value of its surrounding neighbors inside a fixed-size window. For an image of resolution $W \times H$, and a window size $k \times k$, the median filtering operation for a pixel at position (x, y) is defined as:

$$O(x,y) = \text{median}\{I(x+i, y+j) \mid -r \leq i, j \leq r\} \tag{1}$$

where:

- $k = 2r + 1$ is the window size,
- I is the input image,
- O is the filtered output image,
- the window contains k^2 pixel values.

This requires, for *every* pixel, extracting its neighborhood, storing these values in a temporary array, sorting them, and selecting the middle value. When applied across millions of pixels, this computation becomes extremely expensive.

2.2 Computational Challenges in Sequential Execution

Sequential median filtering has two main computational challenges:

2.2.1 High Number of Window Operations

For a 4096×4096 image:

- Total pixels ≈ 16.7 million
- Each pixel requires reading **25 neighbor values** (for a 5×5 window)

This results in:

$$16,777,216 \text{ window extractions and sorting operations}$$

2.2.2 Sorting as the Dominant Bottleneck

For each pixel, the algorithm sorts **25 integers**. Even though 25 is small, sorting is still computationally heavy, and doing it millions of times leads to substantial execution time.

The time complexity is:

$$O(W \times H \times k^2 \log k^2)$$

For a 5×5 window:

$$O(16,777,216 \times 25 \log 25)$$

Sorting becomes the main bottleneck because:

- It runs **millions** of times
- It involves repeated memory allocations and comparisons
- It prevents real-time or large-scale processing in the sequential version

2.3 Why the Problem is Suitable for Parallelization

Median filtering is an *embarrassingly parallel* problem because:

- Each pixel's output depends **only** on its local neighborhood.
- **No synchronization** is needed between pixels.
- No pixel's computation affects another's.
- Each pixel can be processed independently in parallel.

Thus, the double loop structure:

```
for (y = ... )
  for (x = ... )
```

contains inherently parallel iterations. This makes the algorithm ideal for automatic parallelization using techniques such as:

- **OpenMP** (parallelizing the outer loops).
- **MPI** (splitting the image into blocks processed by distributed processes).

2.4 Well-Chosen Example

Consider a pixel at location (x, y) with a 5×5 neighborhood. Suppose the local window values are:

```
10, 12, 14, 200, 13
11, 12, 14, 199, 15
13, 255, 12, 12, 11
10, 14, 13, 12, 12
11,  9, 15, 14, 13
```

This window contains **several noise spikes** such as 199, 200, and 255. The algorithm performs:

1. **Window collection** $\rightarrow$ 25 values.
2. **Sorting** $\rightarrow$ Sorted list might be:

   ```
    9, 10, 10, 11, 11, 12, 12, 12, 12, 12,
   13, 13, 13, 13, 14, 14, 14, 14, 15, 15,
   199, 200, 255
   ```

3. **Choose the median** (13th element) $\rightarrow$ **13**.

Thus, the noisy pixel is replaced by **13**, effectively removing impulse noise.

While this is easy for one pixel, the sequential algorithm must repeat this process:

- For **every pixel**
- For a 4100×4100 region
- Performing **tens of millions** of sorting operations

This demonstrates why a sequential implementation is slow and why parallelization is necessary.

3 Proposed Evaluation

The goal of the evaluation phase is to quantitatively measure the performance improvements achieved by applying automatic parallelization techniques primarily using OpenMP and MPI to the median filtering algorithm implemented in C/C++. The evaluation will compare the execution time, scalability, and efficiency of the sequential implementation against its parallelized versions under several controlled and repeatable benchmarks.

3.1 Experimental Setup

The experiments will be conducted on a Linux-based environment with a multi-core CPU that supports thread-level parallelism. The implementation will be compiled using `g++` with optimization flags (e.g., `-O2` and `-O3`).

We will evaluate three implementations:

1. **Baseline Sequential Implementation** (no parallelism)
2. **OpenMP Implementation** (`parallel for` + `collapse(2)`)
3. **MPI Implementation** (image subdivision across processes) *(if included)*

This setup allows us to isolate the effects of parallelization and analyze speedup and scaling behavior.

3.2 Benchmark Inputs

To ensure meaningful and consistent results, we will use standardized input sizes such as:

- 4096×4096 image (primary benchmark)
- 2048×2048 image (medium size)
- 8192×8192 image (stress test)

Randomized grayscale images will be generated using `rand()` to produce high-variance pixel values. This ensures a realistic workload and avoids optimization bias from uniform or trivial inputs.

3.3 Performance Metrics

We will evaluate the parallelized implementations based on the following metrics:

1. Execution Time

Measured using:

- Linux `time` command
- `chrono` library timing inside C++ code

The primary metric will be **total wall-clock time** of the median filter.

2. Speedup

$$\text{Speedup} = \frac{T_{\text{sequential}}}{T_{\text{parallel}}}$$

This shows how much faster the parallel version is compared to the baseline.

3. Efficiency

$$\text{Efficiency} = \frac{\text{Speedup}}{\text{Number of Threads (Cores)}}$$

This metric helps evaluate how well the algorithm utilizes the available cores.

4. Scalability

We will test the OpenMP implementation with varying thread (OpenMP automatically generates threads) counts:

- 1, 2, 4, 8, 16, and 32 threads

Scalability graphs will show how performance changes as thread count increases.

5. CPU Utilization and Workload Distribution

Using `perf` to monitor:

- CPU cycles
- Cache misses
- Branch mispredictions
- Context switches
- Instructions per cycle (IPC)

This helps understand how parallelization interacts with CPU architecture.

3.4 Profiling Methodology

To identify bottlenecks and measure where time is spent, we will use:

1. gprof

- Compile with `-pg`
- Generate call graphs
- Identify the most expensive functions
- Confirm that sorting is the dominant cost

This will guide optimization efforts.

2. perf

We will use commands such as:

```
perf stat ./program
perf record ./program
perf report
```

These will measure:

- Instruction counts
- Cache behavior
- Parallel overhead
- Load/store costs

This profiling validates whether the parallelization effectively reduces the bottleneck.

3.5 Correctness Verification

Parallelism must not change the algorithm's output. We will verify correctness by:

- Comparing sequential output and parallel output byte-for-byte.
- Running multiple random images.
- Checking edge cases (borders, uniform images, highly noisy images).

Only if the output is 100% identical will the performance results be considered valid.

3.6 Expected Results

Based on the nature of the median filter and the independence of pixel computations, we expect:

- Significant reduction in execution time using OpenMP.
- Near-linear speedup for moderate thread counts.
- Larger images to benefit more from parallelization.
- MPI version to show improvement for extremely large images or multi-node systems.
- Sorting to remain the primary bottleneck despite parallelism, as identified via profiling tools (`gprof`, `perf`).

4 Conclusion

This project aims to evaluate how automatic parallelization using OpenMP and optionally MPI can significantly accelerate the computationally expensive median filtering algorithm. By comparing the sequential and parallelized versions through benchmarking and profiling, we expect to demonstrate notable reductions in execution time and improved scalability. The findings will highlight the practical value of automatic parallelization techniques in optimizing real-world, computation-intensive applications.