THE ARAB AMERICAN UNIVERSITY

FACULTY OF ENGINEERING
COMPUTER SYSTEMS ENGINEERING DEPARTMENT

Parallel and Distributed Computing Course

Project 1: Parallelizing a Sequential Algorithm Using Pthread

Student: Mohammad Saied Dawood
ID: 202010256
Date: May 28, 2025

1. Introduction

This report analyzes how an image filtering method may be parallelized using POSIX threads (pthreads). The project includes both sequential and parallel implementations of a 3x3 convolution filter, which is a fundamental operation in image processing used for blurring, sharpening, and edge detection.

Algorithm Selection Justification

Image filtering using convolution was chosen for this parallelization study because it demonstrates great parallelization properties:

- Independent Computations: Each output pixel may be computed independently from others, making it intrinsically parallelizable.
- Compute-Intensive Operations: This approach uses many arithmetic operations per pixel, giving enough computational work to justify thread cost.
- Regular Memory Access ways: The approach accesses memory in predictable ways to reduce cache conflicts.
- Scalable load: Divide the picture into sections to evenly spread the burden among threads.

Parallelization Potential

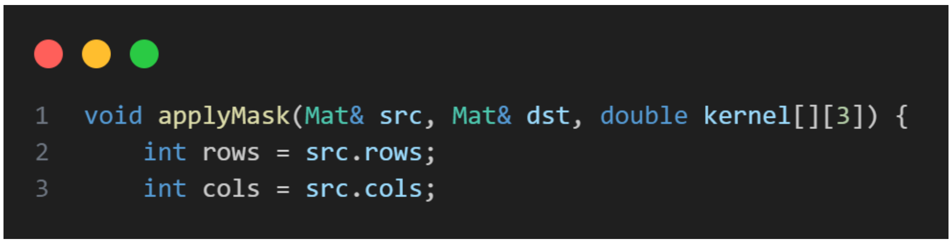
The 3x3 convolution process is excellent for parallelization because:

- No data dependencies exist between output pixels.
- Parallel computing allows for uniform distribution of work across threads.
- Memory access patterns are predictable and optimized.

2. Sequential Implementation

The sequential version processes the image in a single thread, applying the convolution kernel to each pixel in sequence.

Image Processing Pipeline:

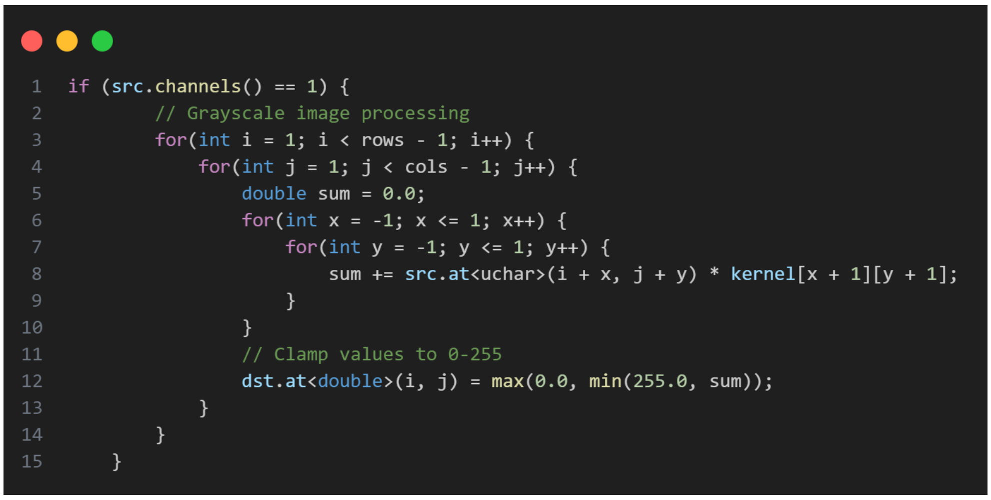


```cpp
void applyMask(Mat& src, Mat& dst, double kernel[][3]) {
    int rows = src.rows;
    int cols = src.cols;
```

Sequential implementation includes:

- Image Reading: Uses OpenCV's {{ imread() }} to load input images in both color and greyscale formats.
- Kernel Implementation: Apply the 3x3 convolution kernel to each pixel using stacked loops.
- Border Handling: Correctly handles edge pixels to avoid out-of-bounds memory access.
- Multi-channel Support: Processes grayscale and color pictures individually.
- Timing Measurement: Uses {{ std::chrono::high_resolution_clock }} for accurate timing.

Code structure:



```cpp
if (src.channels() == 1) {
        // Grayscale image processing
        for(int i = 1; i < rows - 1; i++) {
            for(int j = 1; j < cols - 1; j++) {
                double sum = 0.0;
                for(int x = -1; x <= 1; x++) {
                    for(int y = -1; y <= 1; y++) {
                        sum += src.at<uchar>(i + x, j + y) * kernel[x + 1][y + 1];
                    }
                }
                // Clamp values to 0-255
                dst.at<double>(i, j) = max(0.0, min(255.0, sum));
            }
        }
    }
```

Performance Measurement Methodology

The library in C++ is used to measure the execution time with high-resolution timing:

- The timer begins right before the filtering process begins.
- When the filtering is finished, the timer stops.
- High-precision results are reported in a matter of seconds.

3. Parallelization Strategy

Each thread processes a continuous block of rows in the parallel version, which is implemented in {{parallel_filter_apply.cpp}} and splits the image into horizontal strips. This method divides the burden among several threads by taking advantage of the independence of pixel computations.

Details of Implementation

Thread Management:

Threads are created and managed using POSIX threads, also known as pthreads. Threads are created by the {{pthread_create}} function, and before the program runs, {{pthread_join}} makes sure all threads are finished.

Work Distribution:

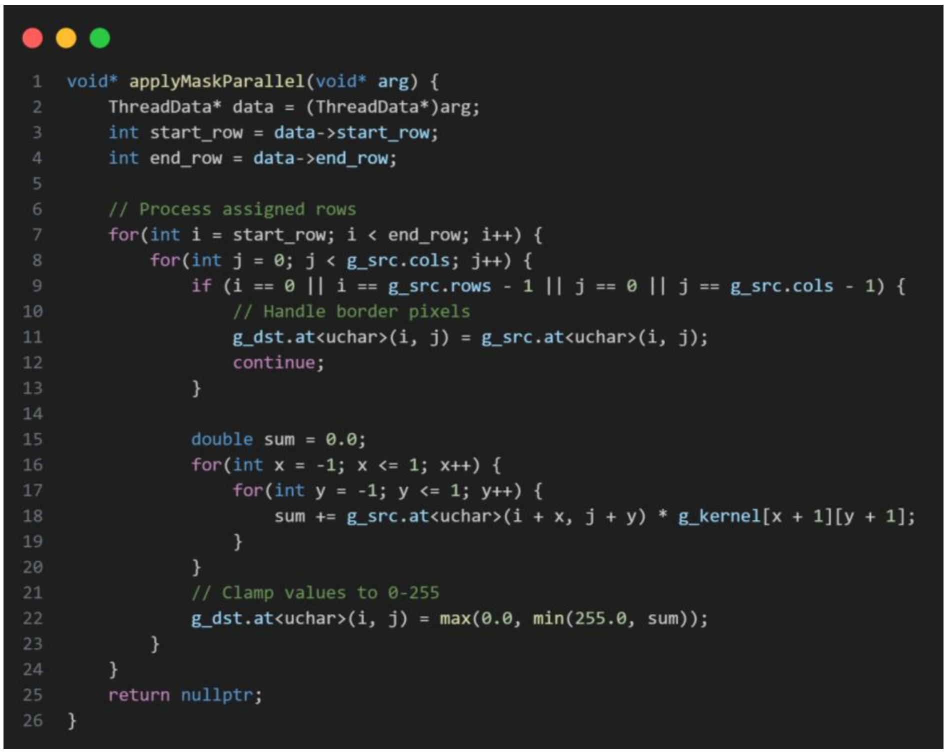
Threads share the image rows equally. When there are N rows and T threads in an image, each thread processes roughly N /T rows.
Any remaining rows are handled by the **LAST** thread to take into consideration situations in which N is not exactly divisible by T.

Synchronization:

Using {{pthread_join}}, the main thread waits for each worker thread to finish, guaranteeing that the output image has been fully processed before saving.

Key Functions:

{{applyMaskParallel}} Is the thread function that applies the 3x3 kernel to every pixel in the designated region while processing a range of rows.

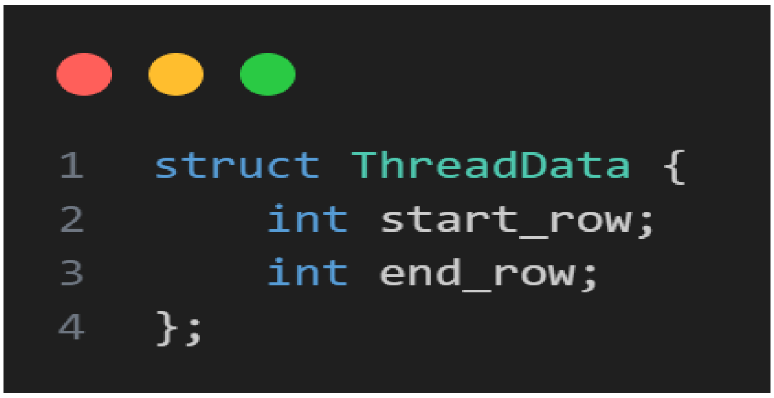


```cpp
void* applyMaskParallel(void* arg) {
    ThreadData* data = (ThreadData*)arg;
    int start_row = data->start_row;
    int end_row = data->end_row;

    // Process assigned rows
    for(int i = start_row; i < end_row; i++) {
        for(int j = 0; j < g_src.cols; j++) {
            if (i == 0 || i == g_src.rows - 1 || j == 0 || j == g_src.cols - 1) {
                // Handle border pixels
                g_dst.at<uchar>(i, j) = g_src.at<uchar>(i, j);
                continue;
            }

            double sum = 0.0;
            for(int x = -1; x <= 1; x++) {
                for(int y = -1; y <= 1; y++) {
                    sum += g_src.at<uchar>(i + x, j + y) * g_kernel[x + 1][y + 1];
                }
            }
            // Clamp values to 0-255
            g_dst.at<uchar>(i, j) = max(0.0, min(255.0, sum));
        }
    }
    return nullptr;
}
```

The start and end row indices for every thread are stored in the {{ThreadData}} struct and are supplied as an argument to applyMaskParallel.



```cpp
struct ThreadData {
      int start_row;
      int end_row;
};
```

Race Condition Avoidance: Because there is no shared writable data and each thread processes a unique set of rows, mutexes and other synchronization primitives are not required.

In order to streamline thread communication, the input image, output image, and kernel are stored in global variables (g_src, g_dst, and g_kernel), respectively.
Except for g_dst, which is where each thread writes to non-overlapping regions, these are read-only for threads.

Global Variables for Thread Communication:

g_src: Source image (read-only access)

g_dst: Destination image (write access, no conflicts due to row partitioning)
g_kernel: Convolution kernel (read-only access)

4. Experiments

Hardware Specifications

- **CPU**: Intel Core i7-10750H @ 2.60 GHz, 6 cores, 12 threads
- **RAM**: 16 GB DDR4-2933 MT/s
- **OS**: Ubuntu 24.04.2 LTS

Test Configuration

Input Images:

- Small: 696×1000 pixels (~0.7 MP)
- Medium: 5184×3054 pixels (~16 MP)
- Large: 14575×8441 pixels (~123 MP)

Thread Configurations:

- 1 thread (sequential baseline)
- 2 threads
- 4 threads
- 8 threads
- 16 threads
- 32 threads

Filter Types Tested:

- Gaussian Blur (Blur_Mask_1.txt)
- Strong Blur (Blur_Mask_Sig16.txt)
- Edge Detection (Edge_Mask.txt)

4. Results

Execution Time Comparison

The following chart shows the execution time for different thread counts across the tested images:

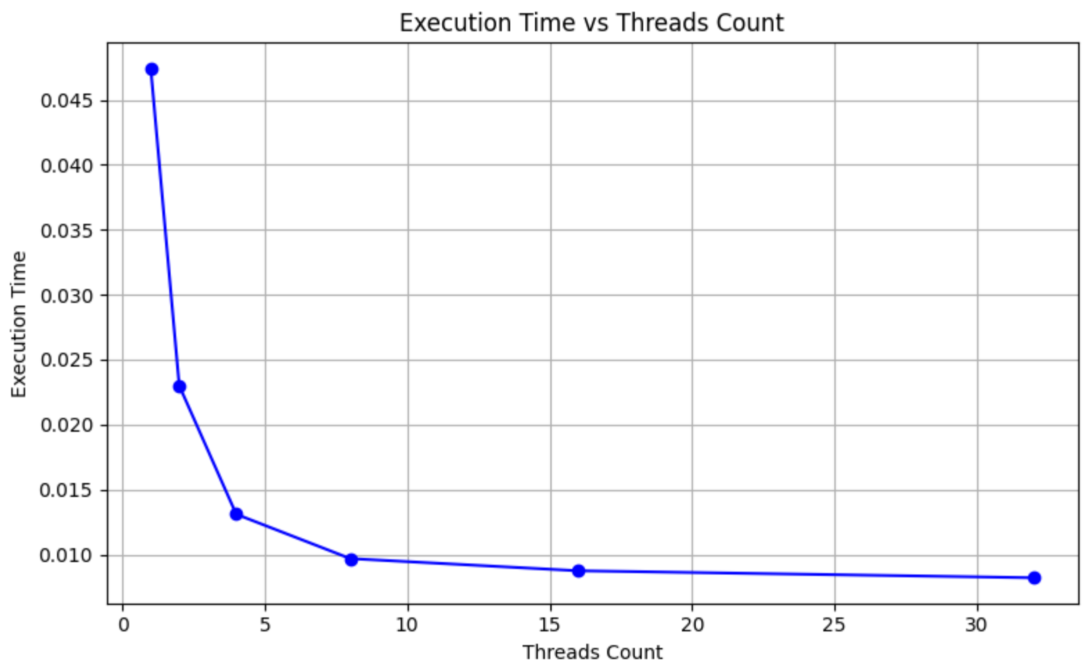


Figure 1:Small input size

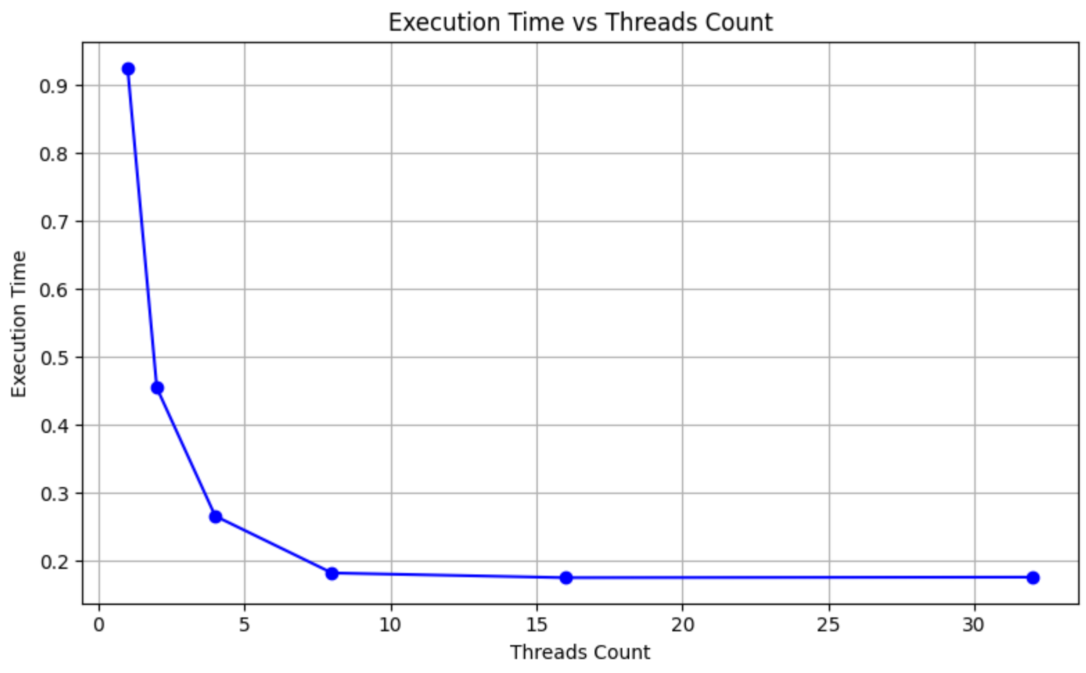


Figure 2:Medium input size

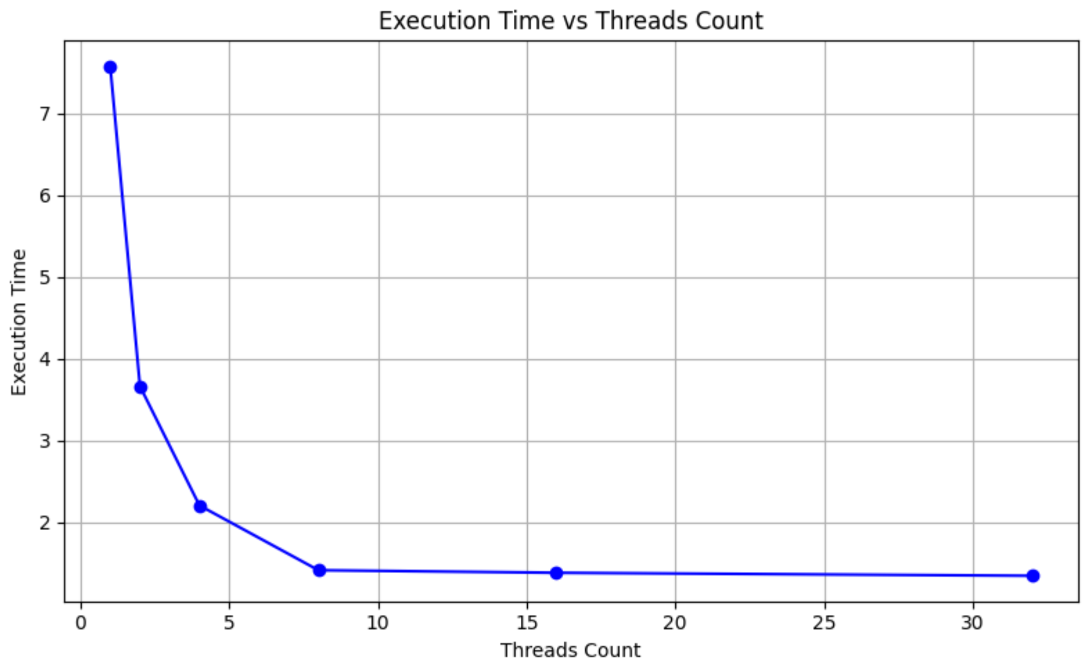


Figure 3:Large input size

Speedup Analysis

The following chart shows the speedup achieved by the parallel implementation relative to the sequential version:

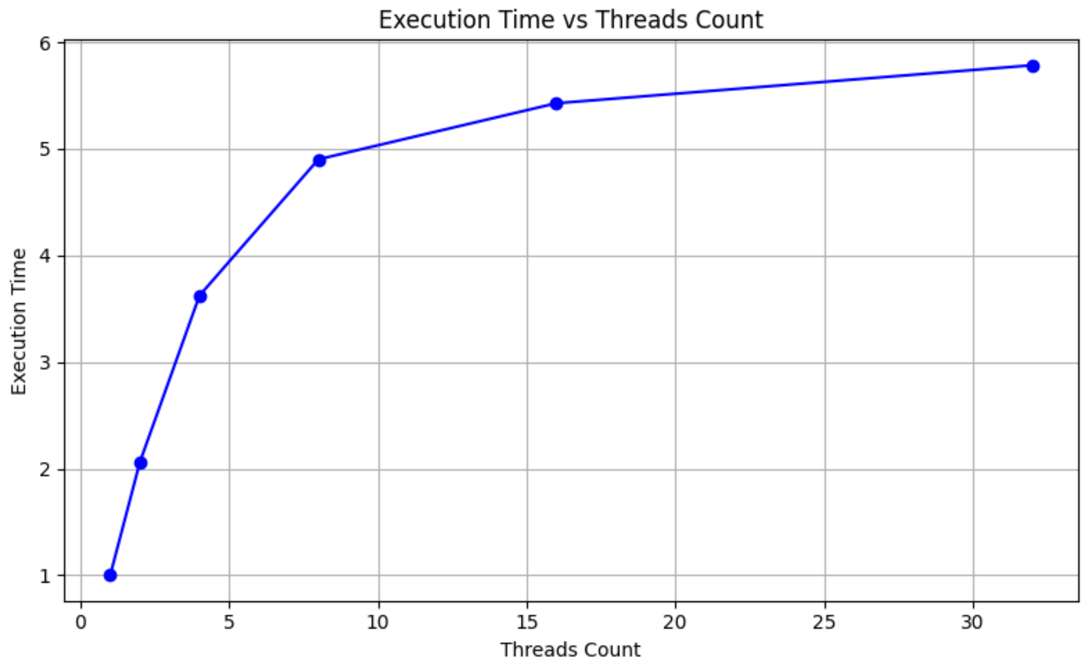


Figure 4: Small input size

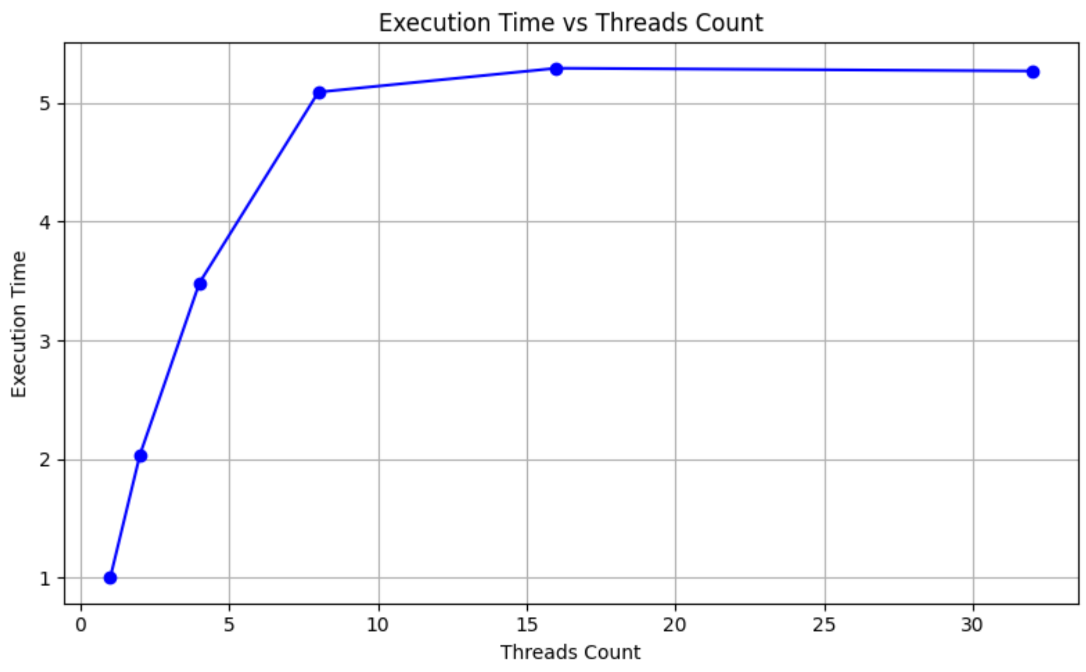


Figure 5: Medium input size

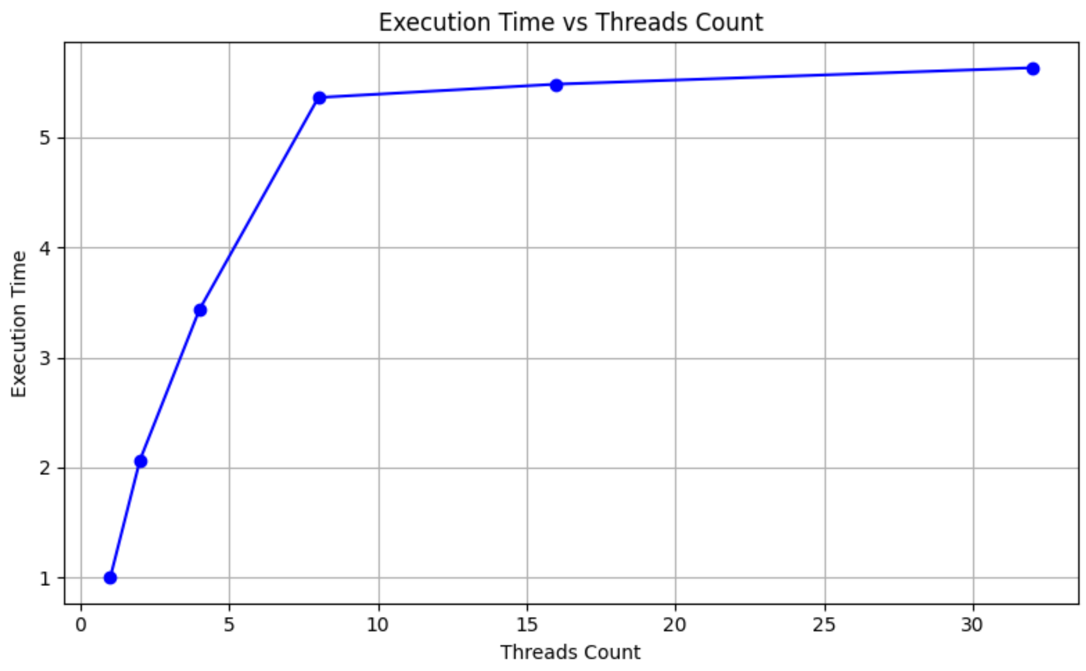


Figure 6: Large input size

6. Discussion

The parallel image filter solution achieves a sublinear speedup due to thread management cost, load imbalance from unequal row distribution, and memory access contention induced by shared access to global matrices. As additional threads are introduced, the benefits lessen as the fixed serial elements of the program become more prominent, in accordance with Amdahl's Law. Additionally, with large thread counts, the workload per thread becomes insufficient to justify the cost, resulting in decreased efficiency. These difficulties explain why execution time does not reduce proportionately with thread count.

Thread Management Overhead: The code utilizes {{pthread_create()}} and {{pthread_join()}} to handle threads, resulting in overhead.
Each thread's task is quite short (a few rows of the image), hence the cost of generating and managing threads might be substantial in comparison to the actual calculation time.
Load Imbalance: Threads are distributed evenly depending on the number of rows, however real work per row may differ.
Border rows (the front and final rows of each chunk) need less calculation due to the border handling condition, however the algorithm does not take this into consideration when distributing work.

7. Conclusion

Parallelizing image processing operations may greatly boost performance, but the gains are strongly dependent on effective workload distribution, thread overhead minimization, and shared memory management. Understanding the limits of parallel speedup, particularly due to Amdahl's Law, is critical when increasing thread counts.

Handling thread synchronization without explicit locking, dealing with load imbalance when image rows are not properly distributed among threads, and minimizing performance decreases caused by memory access contention were all significant issues. Furthermore, managing thread overhead got more difficult as the number of threads rose, particularly for lesser workloads.