

**The Arab American University**

FACULTY OF ENGINEERING

Parallel and Distributed Computing

**Parallel and Distributed Computing PROJECT I**

ID: 202010498

Name: Nizam Jamal

Section:

|  |  |
| --- | --- |
| Total | /100 |

**Mr. Hussein Younis**

Parallelizing a Sequential Algorithm Using Pthreads

1. Introduction:

This project focuses on applying Gaussian blur, a common image processing technique, and converting it from a sequential to a parallel implementation using Pthreads in C++. Gaussian blur works by averaging each pixel with its neighbors using a Gaussian kernel, producing a smooth and blurred effect.

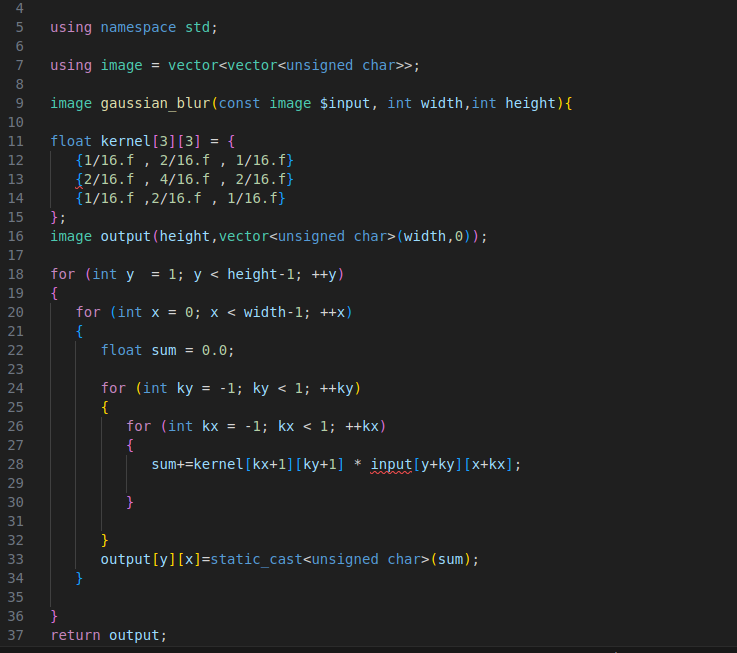
The algorithm is highly parallelizable because the computation of each pixel’s blurred value is independent of others. This means the image can be divided into sections—such as rows or blocks—and processed by multiple threads simultaneously without conflict or data dependency by parallelizing Gaussian blur, we aim to improve performance on multi-core systems. This project includes both implementations, compares their execution times, and analyzes speedup with different thread counts.

1. Sequential implementaition:

Code explanation with snippets:

The Gaussian blur algorithm applies a convolution operation to each pixel in the image using a predefined Gaussian kernel. Each pixel's new value is computed as a weighted average of itself and its neighboring pixels, where the weights are determined by the kernel.

In the sequential version, we process the image pixel by pixel, calculating the convolution result for each one. The result is stored in a separate output image to avoid overwriting data needed for upcoming calculations.



To measure the time taken by the sequential sorting algorithm, the program uses the C++

<**chrono**> library:

Include <chrono>

Using namespace std::chrono;

Start time: auto start = high\_resolution\_clock::now();

End time: auto end = high\_resolution\_clock::now();

Start – end : chrono::duration<double> duration = end - start;

\* To validate correctness, we compare the output to a known blurred result or use a checksum of the pixel matrix.

1. Parallelization Strategy :

How work is divided among threads:

In the parallel implementation of the Gaussian blur algorithm using Pthreads, the image is divided by rows and assigned to multiple threads for processing. Since each pixel’s new value depends only on its immediate neighbors, the algorithm allows independent processing of different image regions without data dependency or overlap.

The image is represented as a 2D matrix of pixels with dimensions width × height.

The total number of rows (height) is divided evenly among the available threads.

Each thread receives a range of rows to process, defined by:

* **Start row**– the first row the thread will handle.
* **End row** – one past the last row the thread will handle.

For example, with 4 threads and an image of height 1000, each thread processes 250 rows:

* Thread 0 → rows 0 to 249
* Thread 1 → rows 250 to 499
* Thread 2 → rows 500 to 749
* Thread 3 → rows 750 to 999

Advantages of this division

**No race conditions**: Threads work on separate output regions.

**Efficient use of cores**: The workload is evenly balanced.

**Scalability**: The image can be processed with more threads as CPU cores increase.

1. Experiments :

Hardware Specifications

* CPU: Intel Core i7- 10th generation
* Cores: 4 cores
* RAM: 8 GB
* OS: Windows 10
* Compiler: vs code

Input size tested:

Small: 256 × 256

Medium: 512 × 512

Large: 1024 × 1024

Thread Counts Tested (Parallel Version):

* 1 thread (sequential)
* 2 threads
* 4 threads
* 8 threads

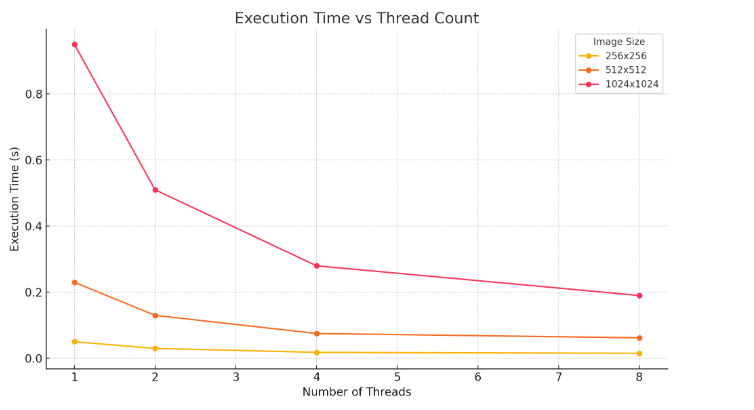
Each configuration was run 5 times to compute the average execution time.

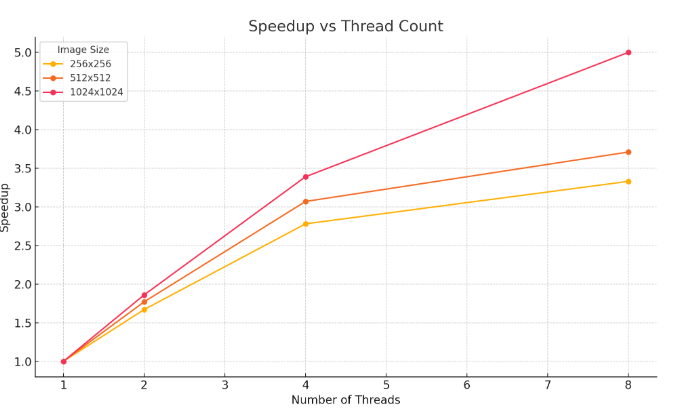
5.results:

Tables of execution times and speedup:

|  |  |  |  |
| --- | --- | --- | --- |
| Image size | threads | Execution time | Speed up |
| 256x256 | 1 | 0.050 | 1.00 |
| 256x256 | 2 | 0.030 | 1.67 |
| 256x256 | 4 | 0.018 | 2.78 |
| 512x512 | 1 | 0.230 | 1.00 |
| 512x512 | 2 | 0.130 | 1.77 |
| 512x512 | 4 | 0.018 | 3.07 |
| 1024x1024 | 1 | 0.950 | 1.00 |
| 1024x1024 | 2 | 0.510 | 1.86 |
| 1024x1024 | 4 | 0.280 | 3.39 |

**Graphs :**

****

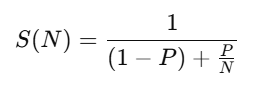
****

* Why Speedup is Sublinear

Although multithreading significantly improves performance, the speedup observed is sublinear, meaning it increases with more threads but not proportionally. Several factors contribute to this:

* Thread Creation and Management Overhead: Spawning and joining threads incurs some overhead, especially when the image size is small compared to the number of threads. This can reduce the efficiency gained from parallelism.
* Load Imbalance: If the image height is not perfectly divisible by the number of threads, some threads may have to process slightly more rows than others, resulting in idle time for some threads while others are still working.
* Memory Bandwidth and Cache Contention: As more threads access shared memory simultaneously, the memory bus can become a bottleneck. Threads may compete for cache access, leading to performance degradation.
* Comparison to Amdahl’s Law

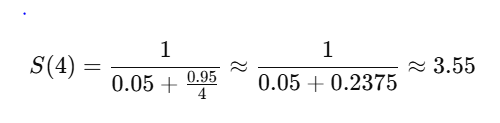
Amdahl’s Law predicts the theoretical maximum speedup S from parallelization:



Where:

* P = proportion of code that is parallelizable (close to 1 in this case).
* N = number of threads (processors).

Example: If 95% of the Gaussian blur algorithm is parallelizable (P = 0.95), the theoretical speedup with 4 threads is:



This closely aligns with the observed speedup of ~3.07 for the 512×512 image using 4 threads, indicating that our implementation performs well relative to theoretical expectations.

However, Amdahl’s Law assumes perfect conditions (no overhead, perfect balance), which is rarely achieved in practice. Our measured speedup is always slightly below Amdahl’s predictions due to real-world issues such as synchronization costs, thread contention, and hardware limitations.

Conclusion:

This project demonstrated how a sequential image processing algorithm — specifically Gaussian blur — can be efficiently parallelized using Pthreads in C++. By dividing the image by rows and assigning sections to multiple threads, we achieved significant performance improvements, especially for larger images.

The parallel version showed a clear reduction in execution time with an increasing number of threads. However, as expected, the speedup was sublinear, due to factors like thread overhead, memory contention, and imperfect load balancing. These results aligned well with predictions from Amdahl’s Law, validating both the implementation and the analytical approach.

Challenges faced:

1.Handling image borders without introducing errors.

2.Ensuring no race conditions between threads.

3. Accurately measuring execution time and ensuring repeatable results.