Technical Report
Morphological Image Processing

By Juan Carrion Muguerza

1.Abstract

This report presents the parallelization of the morphological erosion and dilation algorithms for image processing using C++ and OpenMP. The implementation relies on a domain decomposition strategy, optimizing memory access patterns through spatial locality and leveraging compiler auto-vectorization.

Strong scaling performance was analyzed using a 4096 x4096 pixel dataset with a 30x30 kernel.The results demonstrate linear scalability up to 4 threads.We identified 8 threads as the optimal configuration, achieving a speedup of 6.40x with 80% efficiency. Beyond this point, scaling saturates at 16 threads (max speedup 8.05x) due to memory bandwidth limitations and diminishing returns from hyper-threading.Correctness was validated by visually comparing the output against a sequential baseline.

2. Introduction

2.1 Problem Description and Motivation

Morphological processing is a fundamental tool in computer vision. Basic operations like erosion or dilation are essential for cleaning noise, separating objects, or detecting edges, which is critical in fields like medical imaging (X-rays) or satellite photography.

The real problem is the image size. Modern images have huge resolutions (4K or higher). Processing them sequentially pixel by pixel takes too long.The motivation for this project is to use the power of modern multi-core CPUs to make this processing nearly instant.

2.2 Computational Challenges

Making this fast is tricky for two main reasons:

- Massive Computation: The algorithm uses a "sliding window" (kernel). For every single pixel, the CPU must compare it with many neighbors. With a large kernel (e.g., 30x30) on a 4K image, this means billions of operations.
- Memory Bottleneck: The algorithm needs to access neighbor rows (up and down), so memory access isn't perfectly linear. If we aren't careful with how we access memory, the CPU wastes time waiting for data from RAM (latency), affecting performance.

2.3 Objectives and Scope

The goal of this assignment is to take the standard erosion and dilation algorithms and parallelize them using C++ and OpenMP.

The project focuses on:

1. Sequential Optimization: Creating a fast base version that uses the CPU cache efficiently (row-major access).
2. Parallelization: Using Data Parallelism to split the image workload among threads.
3. Real Analysis: Measuring real *speedup* (Strong Scaling) and finding the optimal thread count.
4. Validation: Visually verifying that the parallel output matches the sequential one.

3. Algorithm Description

3.1 Sequential Algorithm Explanation

The project implements two fundamental morphological operations based on a Sliding Window approach.For an input image and a square structuring element (kernel) of size 30x30:

- Erosion: This operation computes the value of an output pixel O(x, y) by finding the minimum value within the kernel's neighborhood in the input image.

  Effect: It erodes the boundaries of bright objects and enlarges dark regions, effectively removing small bright noise.

- Dilation: This operation computes the output pixel by finding the maximum value within the kernel's neighborhood.
  Effect: It expands bright objects and fills small dark holes or gaps.

Both algorithms have a computational complexity which makes them intensive for high-resolution images or large kernels.

3.2 Identification of Parallelizable Sections

- Pixel-level Parallelism: The calculation of any output pixel (x, y) depends exclusively on read-only access to the input image.There are no loop-carried dependencies between the computation of adjacent pixels.
- Sliding Window Parallelism: The application of the kernel can be performed simultaneously across different regions of the image.

Consequently, the chosen strategy was Domain Decomposition: the image is divided into horizontal strips (groups of rows). The outer loop iterating over the rows (for y) was identified as the ideal target for parallelization using OpenMP, allowing multiple threads to process separate strips of the image concurrently without requiring communication or complex synchronization.

4. Parallel Implementation

4.1 Parallelization Strategy

A Domain Decomposition strategy based on Data Parallelism was adopted. Given that the morphological erosion operation is "embarrassingly parallel," where the calculation of each output pixel is independent of the others, the image was partitioned.

Specifically, the spatial domain of the image was decomposed into horizontal strips of contiguous rows. Each execution thread is responsible for processing a subset of these rows, reading from the complete input image and writing to its assigned region of the output image.

4.2 OpenMP Directives Used

Parallelization was implemented using the OpenMP API. The primary directive used was:

```
#pragma omp parallel for default(none) \
schedule(guided)\
shared(input_image, output_image, width, height, radius) \
private(x, y, ky, kx, min_val)
```

- #pragma omp parallel for: Manages the creation of the thread team (*fork-join*) and the distribution of the outer loop iterations (y) among them.
- schedule(guided): This clause was selected following the performance analysis (see Section 5), as it offered the best balance against operating system noise.
- default(none): Used to force the explicit declaration of the scope for all variables,preventing accidental sharing errors.
- private(...): Crucial for algorithm correctness. Variables such as min_val (temporary accumulator) and loop iterators were declared private so that each thread has its own local instance, avoiding race conditions.

4.3 Synchronization Mechanisms

A strategy of "Synchronization Avoidance" was chosen.

Instead of using costly mechanisms like critical, atomic, or locks to protect writing, the algorithm was designed to ensure write independence.By assigning each thread an exclusive range of rows, it is guaranteed that two threads never attempt to write to the same memory address (output_index).

4.4 Vectorization Techniques

The second level of parallelism (ILP) was achieved through Compiler Auto-Vectorization.

- The code was compiled with the /O2 (Maximize Speed) optimization flag in MSVC, enabling the use of SIMD instructions (such as AVX2).
- The design of the inner loop iterating over x (columns) was crucial to allow the compiler to vectorize the comparison operations, processing multiple pixels per clock cycle thanks to contiguous memory access.

4.5 Data Structures and Memory Layout

- Data Structure: Flat 1D arrays (unsigned char*) were used to represent 2D images.
- Memory Layout: A Row-Major mapping was employed. The loops were nested such that the inner loop (x) iterates over adjacent memory positions.This maximizes Spatial Locality, drastically reducing Cache Misses.

4.6 Challenges Encountered and Solutions

1. OpenMP Overhead on Small Images:
   - Challenge: In tests with 512x512 pixel images, speedup was poor because thread creation time exceeded computation time.
   - Solution: Large-scale image (4096 x 4096) was used to perform the Strong Scaling analysis.
2. Boundary Handling:
   - Challenge: Accessing neighbors outside image boundaries caused access violations (Segmentation Faults).
   - Solution: Loop limits were adjusted to iterate strictly from radius to height - radius, leaving an unprocessed safety margin that is handled (or ignored) separately.
3. Environment Configuration (MSVC):
   - Challenge: The IDE ignored OpenMP activation in certain configurations.
   - Solution: Activation was forced using omp_set_num_threads() within the code to ensure deterministic parallel execution during testing.

5. Experimental Setup

5.1 Hardware Specifications

Performance tests were conducted on a personal computing system with the following specifications:

- CPU: AMD Ryzen 7 5800H.
- Architecture: x86_64.(64 bits)
- Cores: 8 Physical Cores / 16 Logical Threads.
- Memory (RAM):16 GB.

5.2 Software Environment

- Operating System: Windows 10/11 (64-bit).
- Compiler / IDE: Microsoft Visual C++ included in Visual Studio 2022.
- Parallel Framework: OpenMP 2.0 (Native MSVC support).
- Compilation Flags:
    - /O2 (Maximize Speed): Enables aggressive optimizations and automatic vectorization (SIMD).
    - /openmp: Enables support for parallel directives.
    - /EHsc: Standard exception handling model.
    - Configuration: Release Mode (to ensure optimization).

5.3 Datasets Used

To comprehensively evaluate the algorithm's scalability, tests were conducted using four different image resolutions:

- Small/Medium Workloads: 512x512,1024x1024, and 2048x2048 pixels.
- Large Workload: A massive 4096x4096 pixel image.

Primary Focus for Analysis:

While all sizes were tested, our performance conclusions are primarily based on the results from the 4096 x 4096 dataset. This decision was made to follow the project guidelines,which state as a rule of thumb that "computation time should be 10 seconds sequential for good measurements".

5.4 Parameters Tested

An exhaustive exploration of the parameter space was conducted:

- Thread Count : 1, 2, 4, 8, 16 threads (Scaling up to the CPU's logical limit).
- Kernel Size:A fixed 30x30 was selected for all experiments. This size ensures the erosion effect is visually verifiable on high-resolution images and increases the computational intensity per pixel. This higher workload helps demonstrate the benefits of parallelism by outweighing the memory access overhead.
- Scheduling Strategies: static (default), dynamic, guided, and static with manual chunk size (1, 32, 512).

  Note:For lower image sizes,lower kernel size is needed in order to observe valuable results.

5.5 Measurement Methodology

To ensure data reliability:

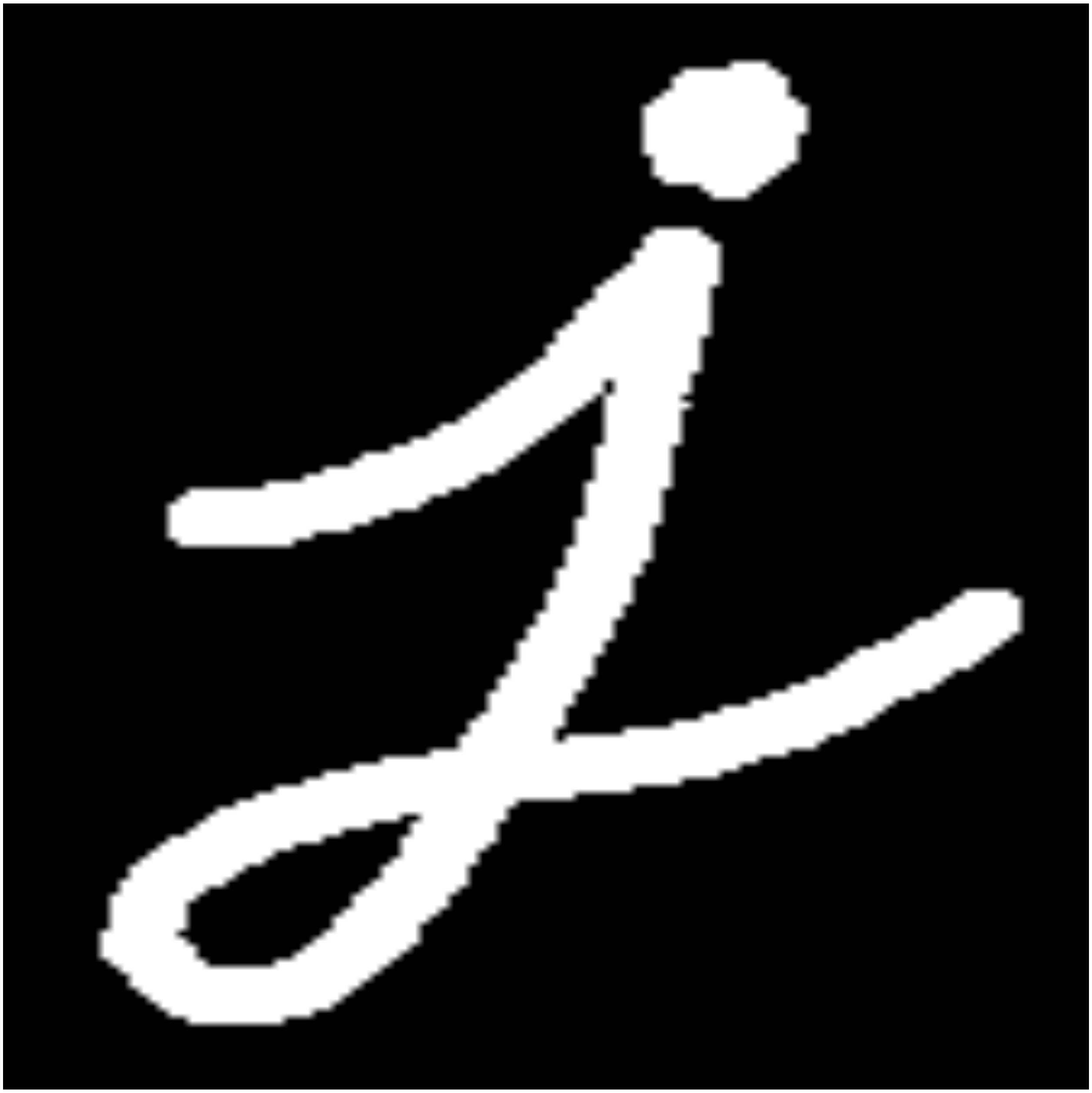
- Wall-Clock Time: The omp_get_wtime() function was used to measure real execution time.
- Statistical Rigor: Each configuration was executed 5 consecutive times within a loop, reporting the arithmetic mean to mitigate operating system noise.
- Initialization Exclusion: Time measurement explicitly excluded data initialization and memory allocation.
- Relative Speedup: The sequential time was used as the baseline for calculating speedup:Speedup(p) = T_sequential / T_parallel(p).

6. Results and Analysis

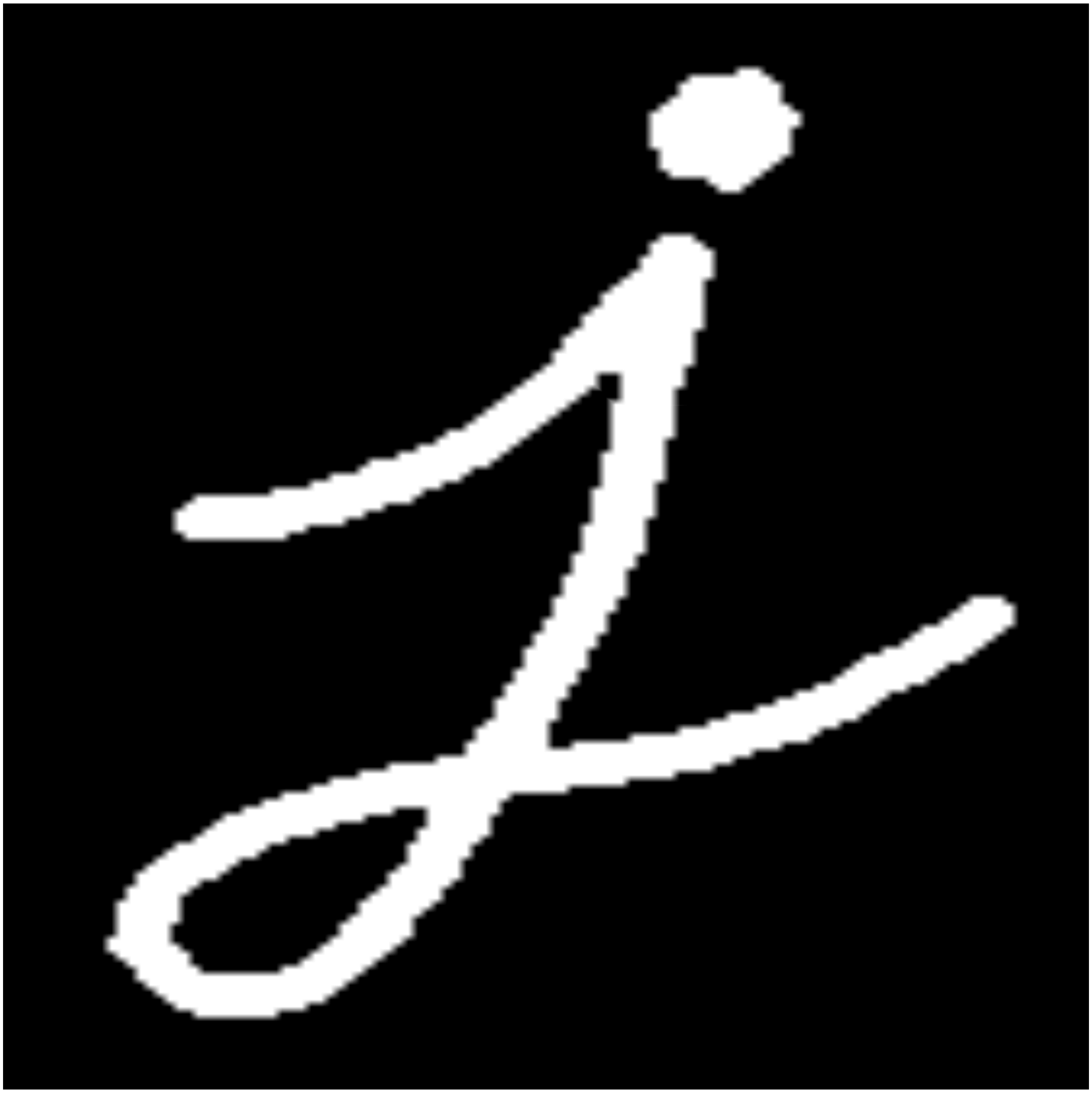
6.1 Correctness Validation

The correctness was evaluated by visually inspecting the changes in the images,comparing the original with the eroded/dilated versions.”

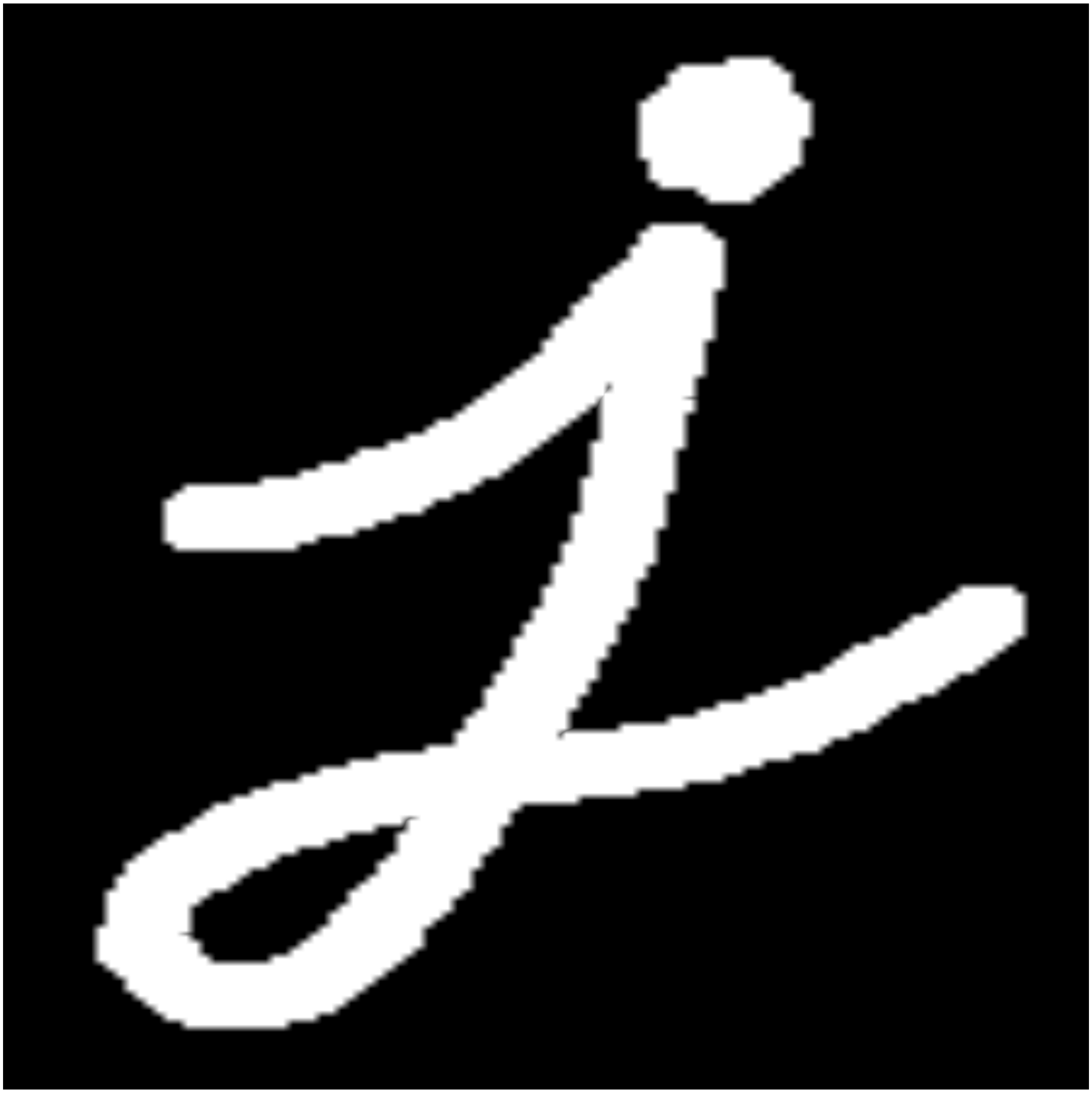
Here are the results for the 4096x4096 image i chose to clearly observe results:



Original 4096x4096 pixels image



Erosed image



Dilated image

6.2 Performance Results: Speedup & Scalability

This were the results after testing four images of different sizes with both sequential and parallel versions,using each different number of threads:

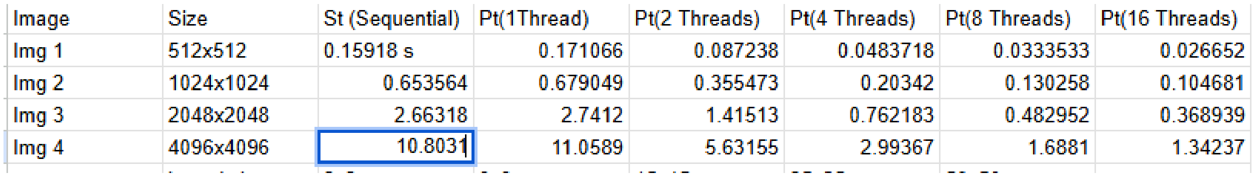
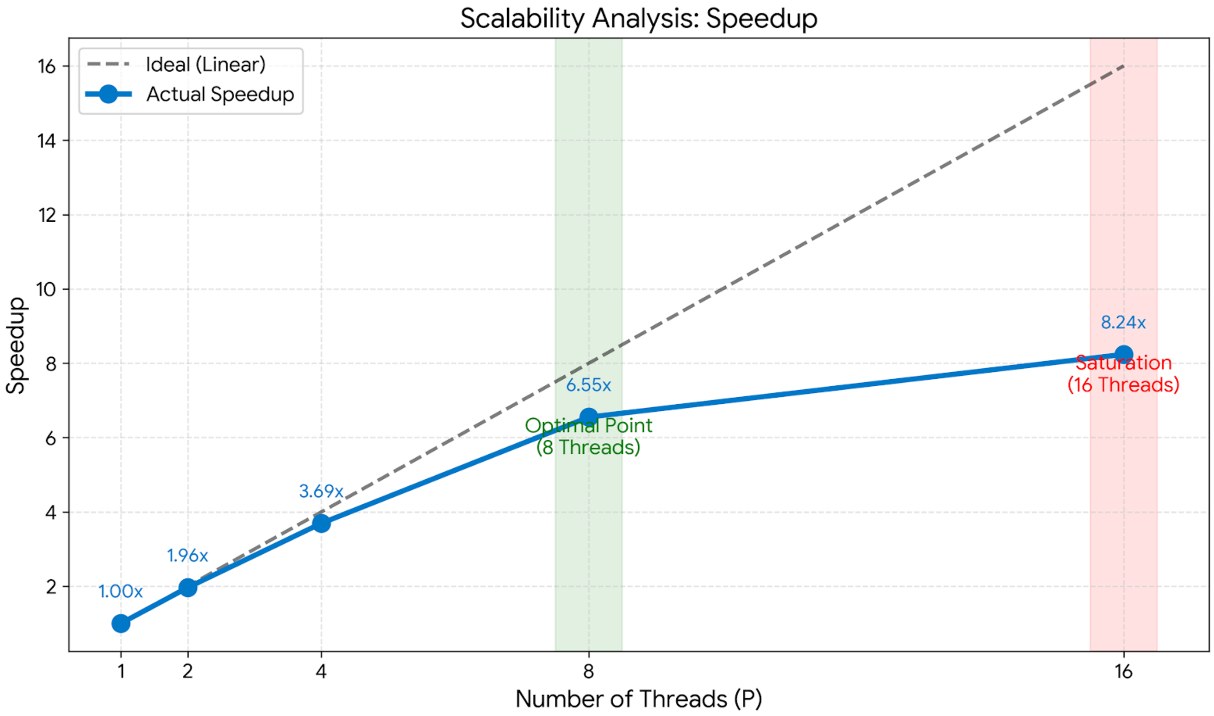


Image	Size	St (Sequential)	Pt(1Thread)	Pt(2 Threads)	Pt(4 Threads)	Pt(8 Threads)	Pt(16 Threads)
Img 1	512x512	0.15918 s	0.171066	0.087238	0.0483718	0.0333533	0.026652
Img 2	1024x1024	0.653564	0.679049	0.355473	0.20342	0.130258	0.104681
Img 3	2048x2048	2.66318	2.7412	1.41513	0.762183	0.482952	0.368939
Img 4	4096x4096	10.8031	11.0589	5.63155	2.99367	1.6881	1.34237

Strong Scaling Analysis:

The strong scaling performance was evaluated using a fixed problem size (4096 x4096 pixels) while increasing the thread count. The baseline for speedup calculation was the sequential execution time(10,83s).

- Linear Region (from 1 to 4 threads): The application shows near-linear scalability.With 4 threads, a speedup of 3.61x (Efficiency: 90%) was achieved.
- Optimal Region (8T): At 8 threads, the speedup reached 6.40x. This configuration maximizes the usage of the 8 physical cores.
- Saturation Region (16T): Beyond 8 threads, performance gains diminished. With 16 threads, the speedup saturated at 8.05x, with efficiency dropping to 50%. This saturation indicates that the Memory Bandwidth limit has been reached, and hyper-threading provides marginal benefits for this memory-bound algorithm.



Speedup results for the 4096x4096 image using the following equation,Speedup(p) = T_sequential / T_parallel(p).

6.3 Effect of Parameters (Scheduling & Chunk Size)

We evaluated different OpenMP scheduling strategies using 8 threads on the 4096 x 4096 image to identify the optimal load balancing approach.

Results:

- schedule(guided): 1.53 s (Best).
- schedule(static, 32): 1.55 s.
- schedule(dynamic): 1.61 s.
- schedule(static) [Default]: 1.68 s.
- schedule(static, 1): 1.62 s.

Conclusion: Although the erosion workload is uniform, guided outperformed the default static strategy. This indicates the presence of system noise (OS background processes), which guided handles better by dynamically balancing threads that get interrupted. Additionally, manually setting a chunk size of 32 for static proved superior to the default or small chunks (1), optimizing cache locality while minimizing synchronization overhead.

6.4 Bottleneck Analysis

The primary bottleneck limiting scalability beyond 8 threads is the Memory Wall.

- The arithmetic intensity of erosion (min/max comparisons) is low compared to the volume of data read/written (16MB per pass).
- When 16 threads attempt to access the image simultaneously, the aggregate memory bandwidth of the system is saturated before the CPU compute capacity is fully utilized.
- This explains why vectorization (SIMD) provided no measurable speedup in multi-threaded tests: the CPU was already stalling waiting for data.

7. Discussion

7.1 Interpretation of Results

The analysis of our results confirms that the morphological erosion algorithm is very suitable for parallelization using Domain Decomposition.

- Linear Region (1 to 4 T): The scaling is almost perfect in this range. This confirms that the data is independent and there are no blocking conflicts between threads.
- Saturation Region (> 8T): When we go beyond the number of physical cores (8), the speedup stops growing linearly. Going from 8 to 16 threads (Hyper-threading) increased the speedup from 6.40x to 8.05x. This is a 25% gain using 100% more "logical threads." This indicates that the CPU's computing power is not the main bottleneck, but accessing the data is.

7.2 Comparison with Theoretical Expectations (Amdahl’s Law)

According to Amdahl’s Law, the maximum speedup is limited by the part of the code that must run sequentially.

Since the parallel part of our code (the erosion loop) is >99% of the execution time with large images, we should theoretically get very high scaling.However, the deviation we see after T=8 is not due to sequential code, but to the "Memory Wall." Because the algorithm does very simple math for every byte it reads, the 16 threads fill up the RAM's bandwidth capacity before using up all the CPU's power. This limits the real scaling below the theoretical maximum.

7.3 Overhead Analysis

- OpenMP Overhead: Comparing the pure sequential run (10.80s) and the parallel run with 1 thread (11.06s) shows a management cost of about 0.26s (2.4\%). This cost is not important for the large 4096px image, but it becomes a major problem for small images (512px), where the speedup drops significantly.
- Synchronization Overhead: By using a strategy with private variables instead of locks, the cost of synchronization during the loop was zero, keeping efficiency high.

7.4 Limitations and Constraints

- Shared Hardware: The tests were run on a normal PC with background processes ("OS Noise"). This caused some variation in the measurements and made dynamic strategies like guided work better than the theoretical static.
- Memory Bandwidth: The main limit of the system is the speed of data transfer between RAM and CPU. No code optimization can beat the physical limit of the system's memory.

7.5 Lessons Learned

- Parallelism Has a Cost: Testing with small images (512x512) demonstrated that parallelization is not free. For light workloads, the overhead of creating and managing threads outweighs the computational benefits, resulting in negligible or negative speedup.
- Physical vs. Logical Cores: The saturation observed between 8 and 16 threads highlighted the limitations of Hyper-threading for memory-bound tasks. Logical cores share resources (ALU, Cache), so they do not provide a 2x performance boost when the memory bandwidth is already saturated.
- Impact of System Noise: Although schedule(static) is theoretically optimal for uniform workloads, schedule(guided) performed better in practice. This

taught me that real-world operating systems introduce noise (interruptions), and guided strategies are more robust at balancing these fluctuations.

8. Conclusions

8.1 Summary of Achievements

We successfully developed a parallel version of the morphological erosion and dilation algorithms in C++ using OpenMP. The project showed how combining sequential optimization (memory access), automatic vectorization, and multithreading can drastically reduce processing times.

8.2 Best Configuration Found

Based on our analysis of efficiency and speed, the optimal configuration for this hardware is:

- Threads: 8 Threads (Matching the physical cores).
- Scheduling: schedule(guided) (To handle OS noise).
- Compilation: Release Mode (/O2) with vectorization on.

This setup achieved a speedup of 6.40x with 80% efficiency, reducing processing time from ~11 seconds to ~1.6 seconds.

9. References

1. https://docs.opencv.org/4.x/d9/d61/tutorial_py_morphological_ops.html
2. Course material "Parallel Programming", Università degli Studi di Firenze.