A Comparison of Sequential and Parallel Implementations in Binary Mathematical Morphology

Marco Billi
Università degli studi di Firenze
marco.billi@edu.unifi.it

Abstract

This work presents a performance evaluation of sequential and parallel implementations of binary mathematical morphology operations, including dilation and erosion. The parallel implementation is developed using OpenMP to exploit multi-core CPU architectures. Benchmarking was conducted on images of varying sizes to assess speedup and scalability. Results show that the OpenMP-based parallel version significantly reduces execution time compared to the sequential baseline, especially for larger images and structuring elements. The findings highlight the effectiveness of parallelization in improving the computational efficiency of morphological processing.

1. Introduction

Binary mathematical morphology is a widely used framework in image processing for analyzing and manipulating the structure of binary shapes within an image. Fundamental operations such as dilation, erosion, opening, and closing play a crucial role in tasks like noise reduction, shape detection, and object segmentation. Although these operations are algorithmically simple, their application to large images or datasets can be computationally intensive, particularly when processing must be done in real-time or with limited resources.

In many cases, especially in academic or experimental settings, binary images used for morphological processing are synthetically generated. This approach allows full control over image characteristics (such as size, shape complexity, and noise level) enabling more systematic benchmarking of algorithmic performance. In this work, all input images are programmatically generated to ensure consistency and reproducibility of the experiments.

The main objective of this study is to compare the performance of sequential and parallel implementations of basic morphological operations. The sequential version serves as a reference point, while the parallel version is implemented using OpenMP, a widely adopted API for multi-threading in shared-memory systems. Through a series of tests on synthetic images of varying sizes and complexity, the efficiency and scalability of the two approaches are analyzed and discussed.

2. Implementations

The implementation of the morphological operations has been carried out in C++, with a clear separation between image management, structuring element definition, and algorithmic processing. This section describes the internal data structures used for image representation, the generation of synthetic input images, and the handling of the structuring element. The codebase is organized into section components in main.cpp file[1]:

- **Image and Structuring Element section:** Handles creation, shape rendering, and saving of binary images and structuring elements.
- **Morphology section:** Contains both sequential and OpenMP-parallelized implementations of dilation, erosion, opening and closing operations.
- **Benchmark section:** Executes timed tests on images, collecting performance data.

2.1. Image Representation and Generation

Binary images are represented using a custom `struct`, which stores pixel data as a one-dimensional array of `uint8_t`, along with metadata such as width and height. To simplify image export, the `stb_image_write.h` header from the STB library is used to write PNG files. All image data is treated as binary, with pixels having a value of either `0` (background) or `255` (foreground).

To ensure repeatability and full control over the input complexity, all test images are synthetically generated via code. Each image has fixed dimensions (chosen at runtime),

[1] All source code is available at https://github.com/billimarco/Morphological_Image_Processing

and the background is assumed to be black. Foreground elements (lines, rectangles, or circles) are drawn in white. This controlled and consistent generation process enables reliable benchmarking across both sequential and parallel implementations.

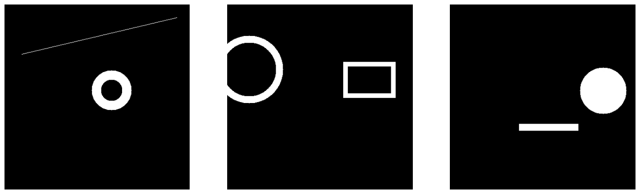


Figure 1. Sample of random generated 400x400 images

2.2. Structuring Element

The structuring element is implemented as a regular, two-dimensional binary mask using `std::vector<std::vector<int>>`, where a value of `1` indicates an active element and `0` indicates an inactive one. To avoid complications at the algorithmic level, only regular and symmetric structuring elements (e.g., square or disk-shaped) are considered in this study. This assumption simplifies the application of morphological operations and the treatment of boundary conditions.

2.3. Sequential Implementations

For each morphological operation, two sequential implementations have been developed. Both produce equivalent results, but differ in how the structuring element is handled during the computation.

The **first version** follows a direct and exhaustive approach: for each pixel in the image, the entire structuring element is scanned. At each position, the algorithm checks whether the corresponding neighborhood in the image satisfies the condition imposed by the morphological operation. This method is conceptually simple and adheres closely to the mathematical formulation, but may introduce inefficiencies when the structuring element contains a significant number of inactive elements.

The **second version** improves performance by precomputing the coordinates of all active positions within the structuring element. These positions are stored as a list of relative offsets with respect to the anchor point. During processing, only the active offsets are applied to the current pixel position, which reduces the number of unnecessary memory accesses and conditional checks. This optimization proves especially beneficial in cases where the structuring element is sparse or contains symmetrical patterns.

The **third version** also precomputes the coordinates, as in the second version, but it processes the image by dividing it into equally sized tiles and performing computations tile by tile.

In both cases, additional routines are provided to handle multiple images in sequence. These routines simply iterate over a collection of input images and apply the corresponding operation to each one individually. This separation is useful for later applying coarse-grain parallelism across image batches.

2.4. Parallel Implementations with OpenMP

The parallel implementations leverage the independence of pixel-wise computations using the `OpenMP` API. Since each output pixel depends only on a fixed local neighborhood in the input image, a natural and efficient approach is to apply the `#pragma omp parallel for schedule(static)` directive to the outermost loop that traverses the image space. This strategy allows multiple threads to simultaneously compute disjoint subsets of the output image without the need for synchronization.

Parallelizing over the structuring element, on the other hand, would not be effective. This inner loop typically involves a limited number of iterations and often results in memory-bound access patterns. The additional overhead of managing parallel execution at this level would outweigh any potential performance gain.

When processing batches of images, further parallelism is introduced by applying an outer `parallel for` over the image list. Since each image is processed independently, this strategy leads to excellent scalability on systems with many cores. The combination of intra-image and inter-image parallelism ensures efficient resource utilization across a wide range of use cases, from single large images to large sets of medium-sized inputs.

The key difference between Version 2 and Version 3 lies in memory access efficiency. While both versions precompute the coordinates of structuring elements to avoid redundant calculations, Version 3 additionally divides the image into equally sized tiles and processes each tile independently. This tiling strategy improves cache locality, leading to better performance due to reduced memory access latency and increased data reuse within the cache.

2.5. Use of stop conditions and OpenMP Compatibility

In both sequential versions of the morphological operations, the use of stop conditions within the nested loops that scan the structuring element represents a deliberate performance optimization. When a condition is met that determines the final outcome for a given pixel (e.g., a mismatch or early fulfillment of the operation's criteria), the loop is immediately terminated. This avoids unnecessary iterations, reducing memory accesses and improving overall computational efficiency, especially for large structuring elements or high-resolution images.

These stop conditions are fully compatible with OpenMP parallelization. Since each thread processes a different pixel (or a different image), and the loops that include these conditions are local to each thread's execution, there are no side effects or shared state that could lead to race conditions. As a result, OpenMP can safely parallelize the outer loop over image pixels or images, while preserving the optimization benefits of early exits inside the inner loops.

In Version 1, an additional early-exit condition is included within the loop, allowing the iteration to terminate as soon as the result is determined. This early termination reduces the number of iterations and can significantly improve performance. In contrast, Versions 2 and 3 lack this optimization, as they rely on a foreach loop that checks each offset individually and uses continue to skip unnecessary positions, which is less efficient due to the absence of loop truncation.

3. Methodology

To assess and compare the performance of the three algorithmic versions developed for morphological operations, I designed a controlled set of benchmarks focusing on erosion and dilation. These operations are fundamental in image processing and are particularly sensitive to data access patterns, memory usage, and loop efficiency.

While erosion and dilation are analyzed individually due to their foundational role and direct computational impact, opening and closure are composite operations, defined as sequences of erosion followed by dilation, and vice versa. From a performance perspective, these operations are effectively combinations of the base algorithms, and their behavior is largely determined by the performance characteristics of the underlying erosion and dilation steps.

My evaluation is structured around two main parameters:

- **Image size**: We selected three representative resolutions to evaluate scalability and cache behavior:
  - 400×400 – small dataset
  - 1600×1600 – medium dataset
  - 6400×6400 – large dataset
- **Structuring element**: Two commonly used structuring elements were chosen to reflect different neighborhood complexities:
  - 3x3 disk (cross)
  - 11x11 disk (circle)

These configurations allow us to assess how each algorithm responds to increasing data volume and structural complexity.

All tests are conducted on synthetic binary images where the black background dominates, simulating a sparse foreground scenario. This setup highlights optimizations like early loop termination and cache locality, and reveals how each algorithm handles redundant computations over uniform regions.

Performance is measured using multi-threaded executions under different thread counts. Speedup is evaluated both on per-operation mean time and total execution time with multi-threading over images, allowing us to isolate the effects of algorithmic changes and parallel scheduling strategies.

4. Experiments and Results

The experiments were conducted on a system equipped with an **Intel Core i7-8700 processor**. This processor, based on the Coffee Lake architecture, features 6 physical cores and supports Hyper-Threading, enabling the execution of 12 threads simultaneously. The system was configured with the g++ compiler, utilizing optimization level 02 to enable vectorization, ensuring efficient execution of the algorithmic versions.

Each operation was performed on a dataset of 50 randomly generated images to ensure robust and statistically significant results. The following observations emerge from figures 2 and 3:

- **Similarity between V2 and V3:** Across all benchmark scenarios, versions V2 and V3 show very similar speedup behavior. Although V3 introduces a tiling strategy to improve cache locality, this optimization does not lead to substantial performance differences compared to V2. This indicates that the benefit of tiling, while conceptually sound, may not be impactful under the tested conditions.
- **Impact of image size:** Increasing image resolution from 400×400 to 6400×6400 results in a slight improvement in speedup. However, this improvement is relatively modest when compared to the effect of the structuring element. Larger images do introduce more computation, but this does not always translate to a proportional gain in parallel efficiency.
- **Structuring element influence:** The size and complexity of the structuring element play a more critical role in parallel performance. A 11×11 disk structuring element generates a significantly greater computational load than a 3×3 disk, leading to better utilization of threads and higher speedup. This demonstrates that computational density is a key factor in parallel performance scaling. This behavior aligns more closely with Gustafson's Law, which argues that increasing problem size allows parallel portions of the code to dominate execution time, making parallelization more effective even in the presence of some sequential overhead.

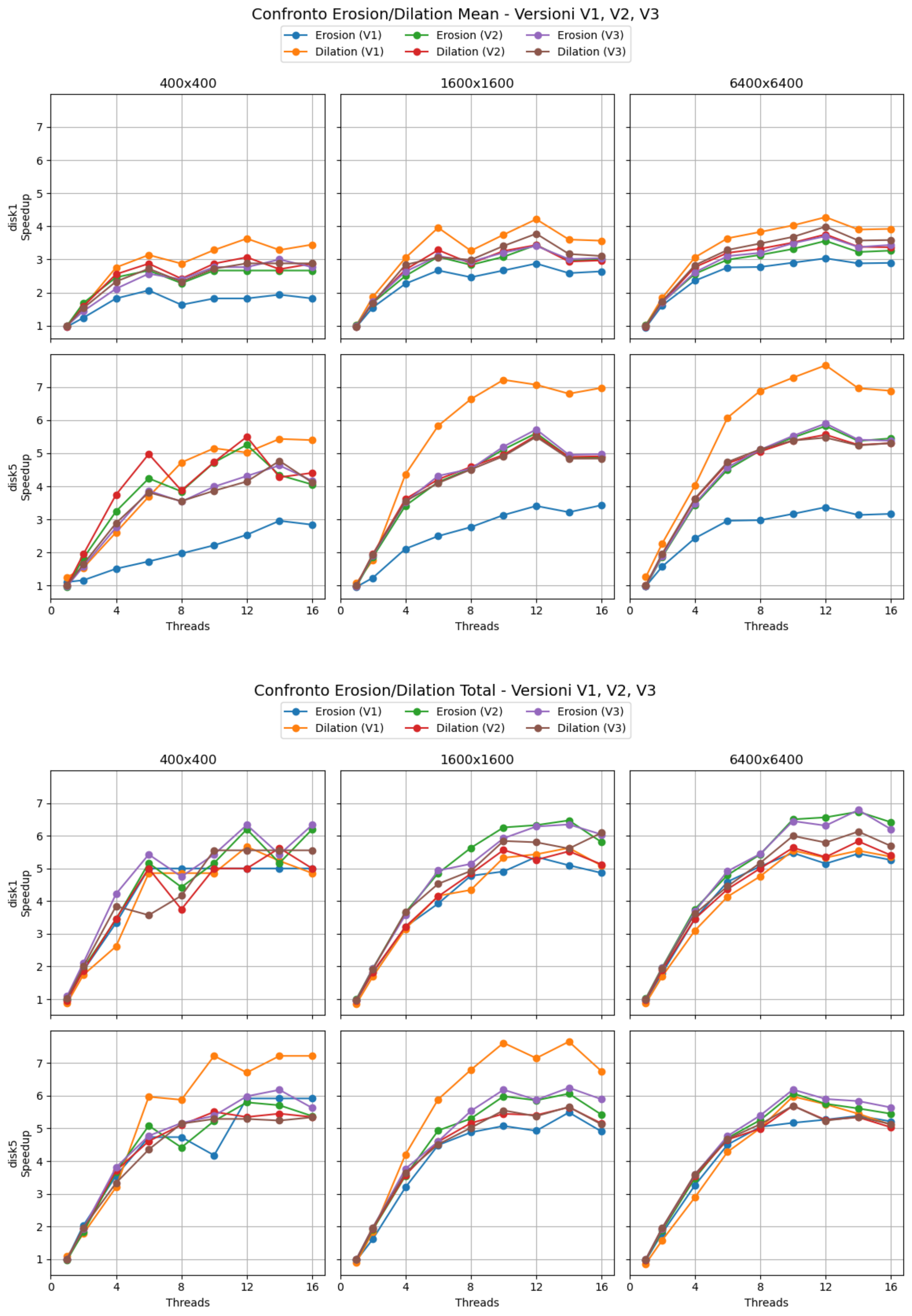
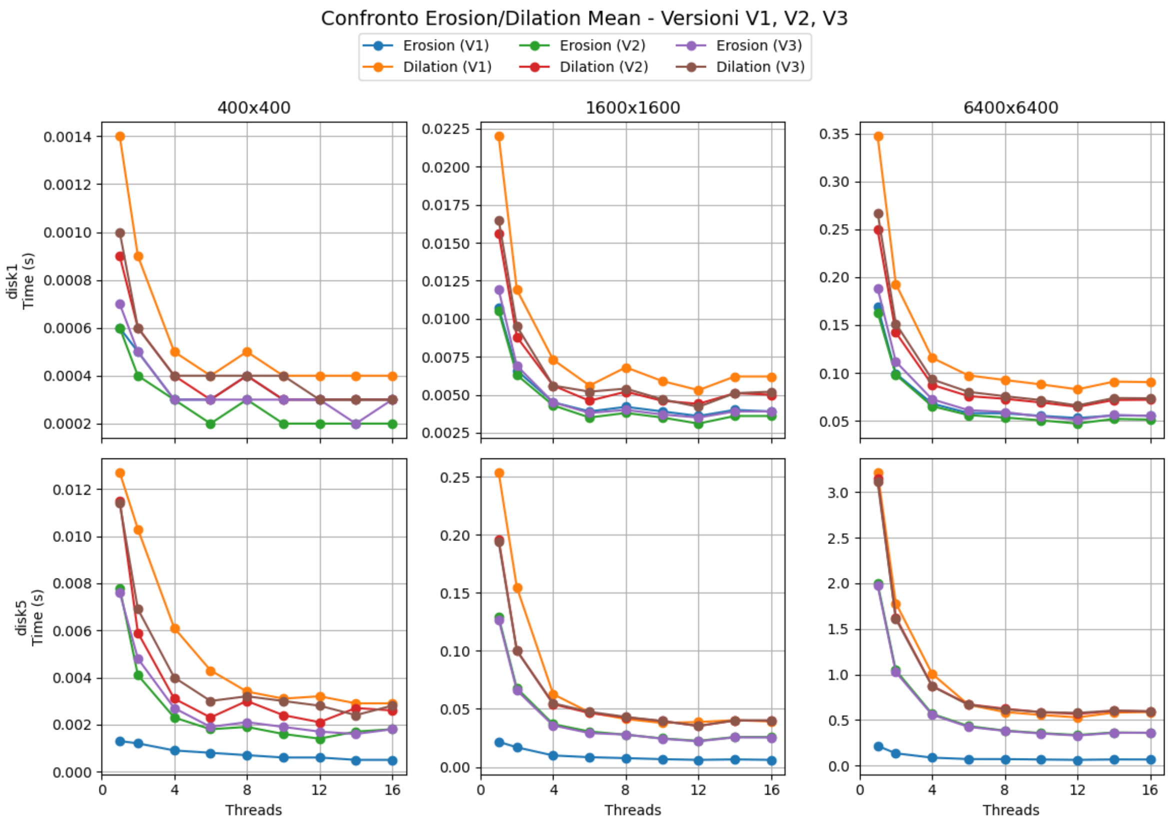


Figure 2. Comparison of speedup results across all configurations.



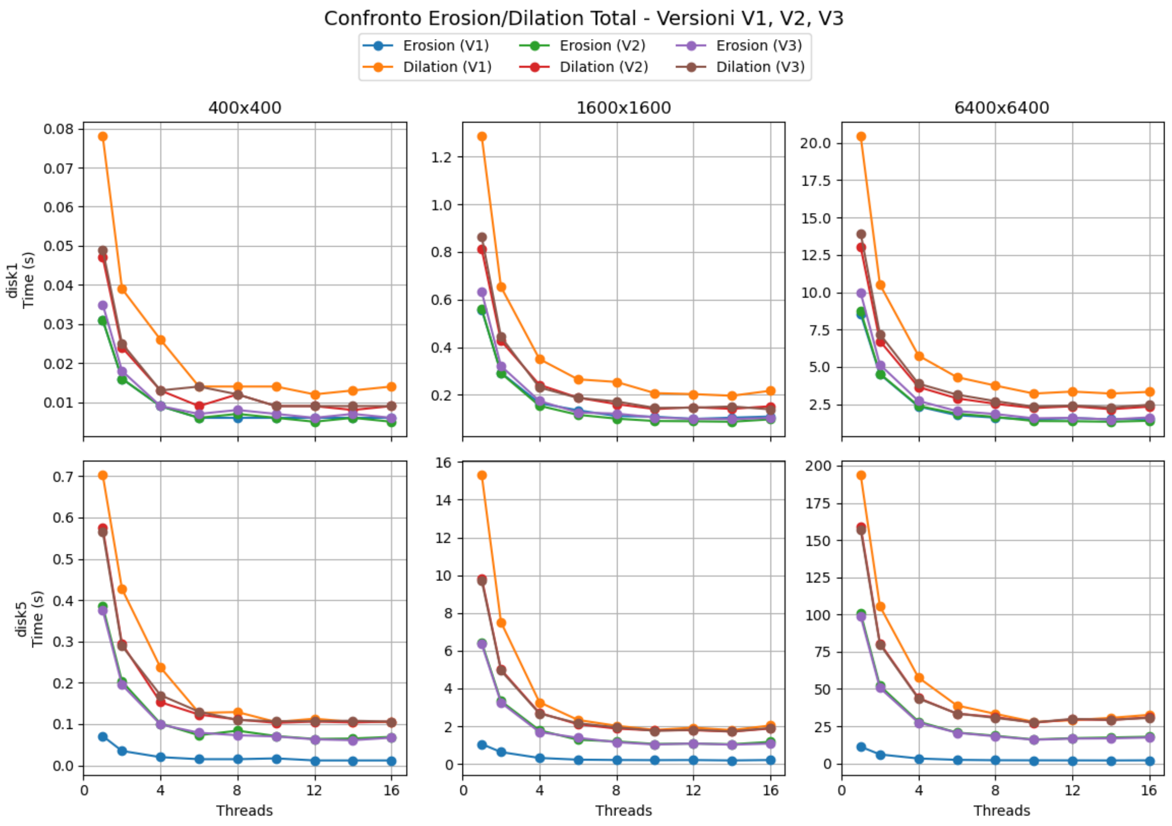


Figure 3. Comparison of times results across all configurations.

- **Effect of inter-image parallelism:** When morphological operations are parallelized across a set of images rather than within a single image, we observe a marked increase in speedup. This strategy effectively distributes the workload across threads with minimal synchronization overhead, making it particularly useful in batch processing scenarios.

- **Erosion vs. Dilation in V1:** In version V1, erosion shows significantly lower speedup than dilation. This is because erosion benefits from early termination conditions in the loop, allowing it to skip unnecessary computations when the background is predominantly black. While this improves erosion's absolute runtime, it also reduces the relative gain from parallelization, as fewer operations remain to be executed concurrently. This phenomenon is clearly visible in figure 3.

- **Thread scalability:** Execution times decrease significantly with the number of threads, particularly up to six, which corresponds to the number of physical cores on the Intel Core i7-8700 processor used in our tests. Beyond this point, as hyperthreading comes into play, the performance gain becomes marginal and gradually stabilizes. This suggests that the parallel workload benefits primarily from true core-level parallelism, and adding logical threads does not yield substantial additional performance due to resource contention and limited parallel workload remaining.

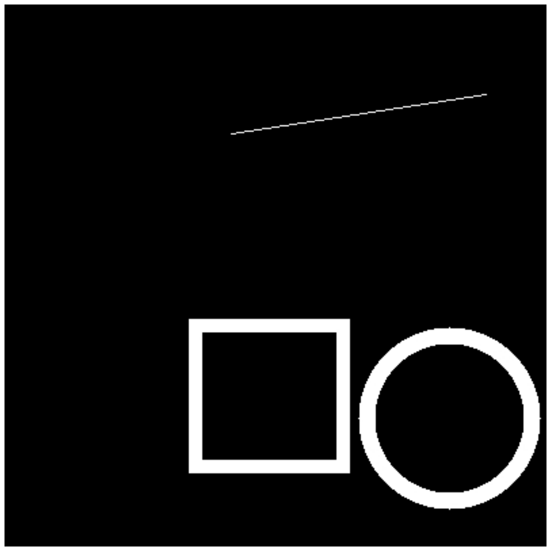
5. Conclusion

Through systematic benchmarking of three algorithmic versions for erosion and dilation, we observed key insights into the trade-offs between simplicity, memory locality, and scalability. The baseline version (V1), despite being the least complex, performs well in absolute terms due to early loop termination but benefits less from parallelization. Versions V2 and V3, though more complex and cache-aware, show similar speedup patterns, suggesting that for small-to-medium structuring elements, tiling does not offer substantial gains.

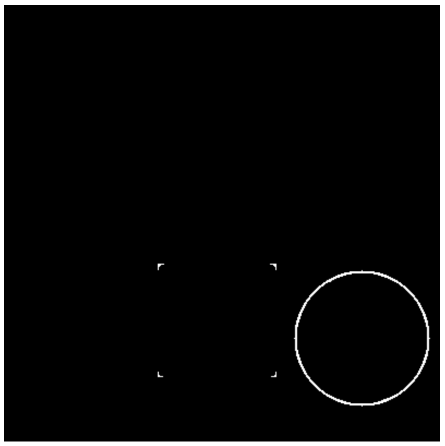
Moreover, the experiments highlight that the size of the structuring element has a more pronounced impact on speedup than image resolution alone. Finally, inter-image parallelism emerges as an effective strategy, delivering higher throughput with minimal synchronization overhead.

These findings provide practical guidance for optimizing morphological operations, especially when processing large batches of images or using high-core-count CPUs.

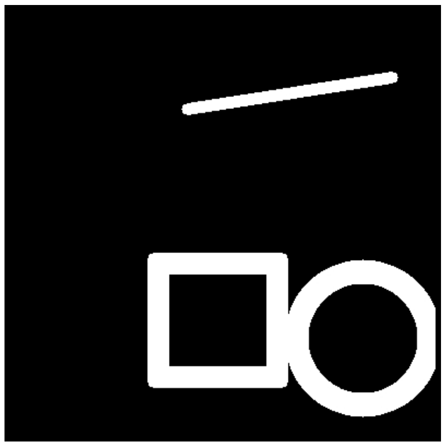
A. Visualization of morphological operations



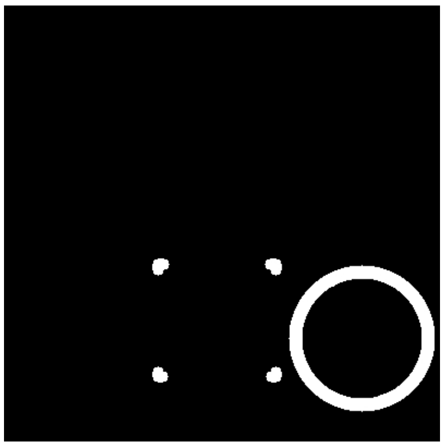
(a) Basis



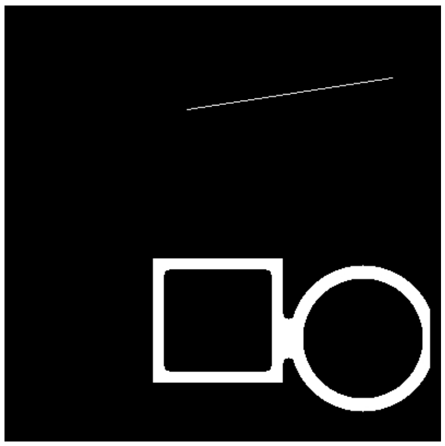
(b) Erosion



(c) Dilation



(d) Opening



(e) Closing

Figure 4. Comparisons of morphological operations of an image with a disk 11x11