Parallel Programming HW2 Report

資工四 B10705009 邱一新

1. My Implementation

1.1 Hybrid Parallel Model: MPI + OpenMP

My implementation adopts a hybrid parallel model, combining MPI for inter-process task distribution with OpenMP for intra-process, shared-memory parallelism on a per-node basis.

Within each MPI process, image processing operations are aggressively parallelized using OpenMP. For nested loops iterating over image pixels, which are common throughout the SIFT algorithm, I employed the directive
`#pragma omp parallel for collapse(2)`.
This flattens the nested loops into a single larger iteration space, allowing OpenMP's scheduler to distribute the workload more flexibly and efficiently across available threads. For simpler single-layer loops, I used `#pragma omp simd` to explicitly enable vectorization and leverage hardware-level parallelism.

1.2 Parallelized Output Handling

This multi-threaded strategy was extended to the final I/O operations as well. Since the number of key points can reach tens of thousands, each associated with a 128-dimensional descriptor, a parallel output method was required.
To prevent interleaved or truncated lines caused by concurrent writes, each thread first constructs its full output line within a private `stringstream`. Once the line is complete, it is written to the final output stream in a single, atomic operation, ensuring data integrity.

1.3 MPI-Based Task Decomposition

For inter-process parallelism, I implemented a task-decomposition strategy based on the SIFT algorithm's Octave structure.
A key dependency exists in the initial `gaussian_pyramid` construction, as each blurred image depends on its predecessor. This serial dependency makes it inherently difficult to parallelize across processes without introducing complex data exchanges. Therefore, the `gaussian_pyramid` is computed by every process.

Once all nodes have the complete pyramid, the main parallel workload begins. I adopted an interleaved assignment of Octaves across processes: a process with rank `i` handles Octaves `i`, `i+size`, `i+2*size`, and so on.
This approach was chosen over simple block assignment to achieve better load balancing,

as earlier (higher-resolution) Octaves are much more computationally expensive than later ones. Interleaving distributes both heavy and light workloads more evenly across all processes. Each process then performs full SIFT computations, including DoG generation, gradient computation, orientation assignment, and descriptor extraction, for its assigned Octaves. Finally, all results are gathered back to Rank 0 for consolidated I/O.

1.4 Abandoned Spatial Decomposition Approach

A major challenge emerged during an alternative spatial decomposition attempt, where the image was partitioned into horizontal strips, each handled by a different process. This approach ultimately fail (in my implementation) for two main reasons.

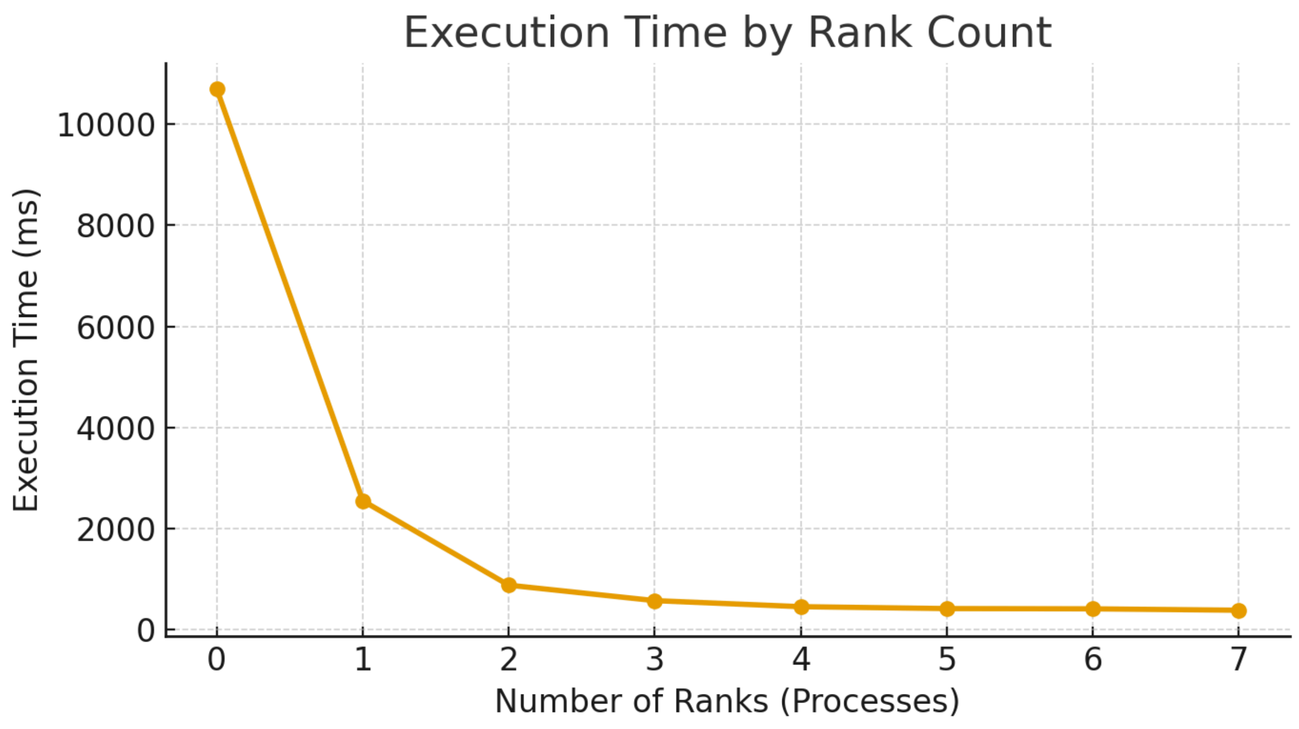
First, the `gaussian_pyramid` construction required frequent halo exchanges between neighboring processes to share boundary pixels. Without this, blurring operations produced data contamination (“dirty information”) at strip boundaries, propagating errors across subsequent stages.
Second, mapping locally detected keypoints back to global coordinates was error-prone, often producing missing or duplicated keypoints, which degraded overall accuracy to about 80%.

Given these severe correctness and implementation challenges, I abandoned the spatial decomposition design. The final Octave-based task decomposition, though it incur sequential computation of Gaussion pyramid and imperfect load balancing, provides both correctness and accuracy, making it the preferred solution.

2. Analysis

2.1 Plots to show the load balance between processes

The commend I used: `srun -A ACD114118 -N 1 -n 8 -c 6 --time=00:03:00 ./hw2 ./testcases/06.jpg ./results/06.jpg ./results/06.txt`



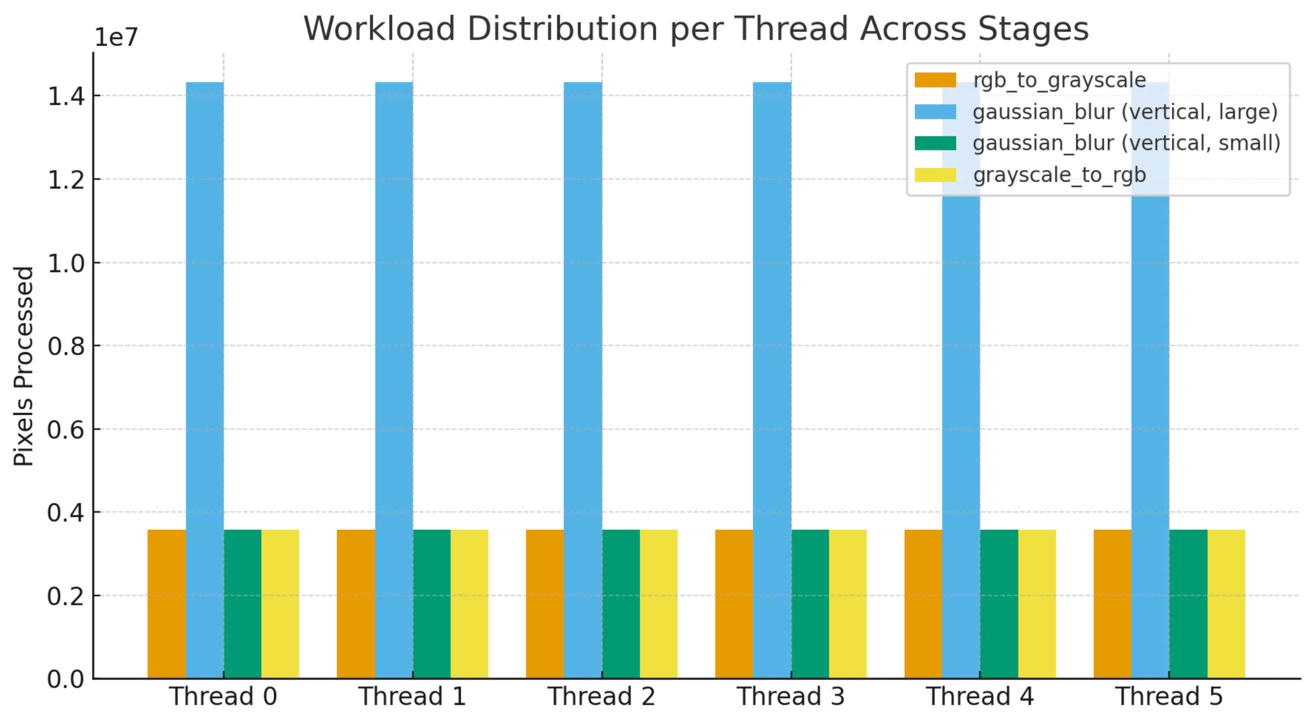
output.png

The inter-process load balancing was not very effective. This is an inherent limitation of the octave-based parallelization strategy, since the computational cost varies significantly across different octaves. Because of this imbalance, assigning entire octaves to each process cannot evenly distribute the workload.

A better approach would be to divide the image into multiple strips and assign them to different processes. This way, every process would handle approximately the same number of pixels, achieving better balance. However, implementing such a strip-based decomposition was too complex within the scope of this project. Given my workload from other courses, I was unable to complete that version successfully.

2.2 Plots to show the load balance between threads

The commend I used: `srun -A ACD114118 -N 1 -n 1 -c 6 --time=00:03:00 ./hw2./testcases/06.jpg ./results/06.jpg ./results/06.txt`



output (1).png

When running with a single MPI process (-n 1) and multiple OpenMP threads (-c 6), I analyzed how well image-processing functions were parallelized. Because OpenMP introduces a synchronization barrier during data aggregation, it was difficult to measure execution time per thread directly. Instead, I calculated the number of tasks assigned to each thread.

Since each task involves simple and uniform computations, this metric reliably reflects the load balance. As shown in the plots, every thread receives almost the same number of tasks, demonstrating perfect load balancing within a process

2.3 Scalability Analysis

Number of nodes / Number of processes per node
These two aspects share the same underlying behavior, as both increase the total number of MPI processes. The main difference is that adding nodes increases inter-node communication costs (IPC), but this impact is minimal in my implementation. The fundamental limitation is that there are only eight octaves in total, and each process is responsible for at least one. Since I did not further partition the image, scalability saturates once the number of processes exceeds eight — additional processes will simply remain idle.

Number of CPU cores per process
The number of CPU cores per process directly affects scalability. It determines how many threads can be used for pixel-level parallelism, and because image data often involves millions of pixels, increasing the number of threads consistently improves performance.

3. Conclusion

This assignment gave me a much clearer understanding of how to approach performance optimization and a more tangible sense of Amdahl’s Law. I realized that parallelization is not simply about adding “for” loops with parallel directives everywhere, but about carefully analyzing performance bottlenecks and identifying the parts of the program that truly dominate the computation time.

Although this project did not particularly deepen my understanding of image processing itself, it allowed me to practice analyzing an unfamiliar algorithm and exploring ways to optimize it. I believe this experience has strengthened my problem-solving skills — though perhaps at some cost to my physical health.