1 Implementation

1.1 Task Partitioning

My approach to parallelizing the SIFT algorithm combines MPI for coarse-grained parallelism across processes and OpenMP for fine-grained parallelism within each process.

MPI for Octave-Level Parallelism: The overall workload is divided among MPI processes at the octave level. The root process (rank 0) is responsible for reading the input image and broadcasting it to all other processes. A load-balancing algorithm then assigns a specific range of octaves to each MPI process. The computationally intensive tasks of Gaussian pyramid generation, DoG pyramid generation, gradient pyramid generation, and keypoint detection are performed in parallel by each process on its assigned octaves. This is a form of data parallelism, where each process operates on a different subset of the image data (different scales).

OpenMP for Loop-Level Parallelism: Within each MPI process, OpenMP is utilized to parallelize computationally expensive loops in the SIFT pipeline. This allows for efficient use of multiple CPU cores available to each process. The parallelized sections include image resizing, Gaussian blur, DoG pyramid generation, gradient pyramid generation, and keypoint orientation and descriptor generation.

1.2 Scheduling Algorithm

For the different function, different OpenMP scheduling strategies are employed. Some are based on the characteristics of the loop being parallized, and some are choosen after testing different setting.

schedule(static): Used for tasks with relatively uniform workloads, such as in generate dog pyramid and generate gradient pyramid.

schedule(dynamic): Employed for loops where the workload can be unpredictable, such as in find keypoints and the orientation/descriptor generation, to ensure better load balancing among threads.

schedule(guided): Used in gaussian blur as a compromise between the low overhead of static scheduling and the load balancing of dynamic scheduling.

At first, I thought dynamic or guided would always be better to handle the non-uniform workload. However, after some testing, I found that static scheduling is sufficient for my implementation for most cases.

1.3 Techniques to Reduce Execution Time

For some operations, I also used SIMD Vectorization with AVX2. The code leverages these extensions through the use of immintrin.h intrinsics in performance-critical functions such as gaussian blur, generate dog pyramid, and generate gradient pyramid. This brings some visible but not significant performance gain.

1.4 Difficulties Encountered

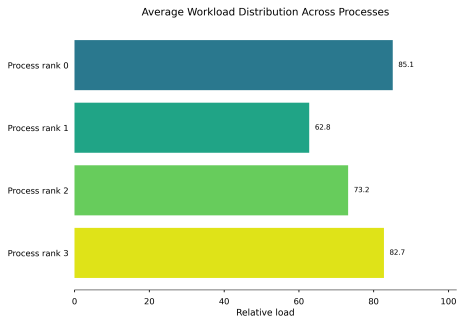
One of the main challenges was handling the MPI data gathering and serialization. Since the Keypoint struct contains std::vector and other non-trivial data types, it cannot be directly sent over MPI. The final way I got through this was to manually serialize the keypoint data into a raw integer array before sending and then de-serialize it back into Keypoint objects on the root process.

Achieving optimal load balance was another difficulty. The initial approach of evenly dividing the octaves among processes resulted in poor performance because the workload per octave is not uniform. After several ineffective attempts, I started to print out more fine-grained balancing statistics and found that my original buggy code often resulted in only half of the processes doing tasks. I also had a version that resulted in every process except one doing the same job. This led to the implementation of the current greedy load-balancing strategy.

2 Analysis

2.1 Load Balance

The implemented load-balancing strategy assigns the more computationally intensive initial octaves to individual processes, while the lighter, later octaves are grouped and allocated to the final process. I have profiled the load balance between processes for some time-consuming steps in the program.



2.2 Scalability

In my implementation, I don't think further scaling the number of nodes would yield significant performance gains. As the number of processes and nodes increases, communication overhead grows, causing the speedup to diminish swiftly.

Operations such as Gaussian blurring, pyramid generation, and keypoint detection are largely "embarrassingly parallel," and thread-level parallelism within a process has much lower overhead compared to network communication. Therefore, increasing the number of CPU cores per process is likely the more effective way to improve performance.

2.3 Others

The performance of the SIFT algorithm also depends on the image content. Images with more features tend to generate more keypoints, which slightly increases the processing time for different stages. As a result, the optimal optimization strategy may vary depending on the image.

3 Conclusion

3.1 What have I learned from this assignment

This assignment provided some experience in parallelizing a complex algorithm using the MPI and OpenMP approaches taught in class.

I learned that to successfully parallelize a program, I must first be thoroughly familiar with its structure and logic. Setting up proper profiling is essential to identify execution bottlenecks. When using MPI, communication overhead must also be carefully considered. Incorporating hybrid techniques such as SIMD can further improve performance, but combining multiple parallelization methods requires attention to how they may interfere with each other's effectiveness.

3.2 Feedback and suggestions

The server getting stuck as the deadline approached was really frustrating. It became difficult to make progress when I couldn't run any tests. Some experiments needed for writing the report were almost impossible to conduct because of the server congestion.

Overall, this assignment felt slightly easier than Assignment 1, at least in terms of passing the baseline. However, the uncertainty about how the performance score would be graded made my friends and me quite stressed, as we didn't know when we could confidently stop working on it before the deadline.