SIFT Feature Extraction: Performance Analysis Report

Name and Student ID

- **Name:** Chin-Hui Chu
- **Student ID:** r12944041

Implementation Details

How do you partition the task?

Our implementation utilizes a hybrid parallel programming model combining MPI for distributed-memory parallelism across processes and OpenMP for shared-memory parallelism across threads within a single process. The task is partitioned in two main stages:

1. **MPI-level (Data Parallelism):** The initial and most computationally intensive stage, Gaussian pyramid generation, is parallelized using a **2D block decomposition**. The input image is divided into a grid of rectangular sub-images (blocks). Each MPI process is assigned one block, making it responsible for all computations within that spatial region.
2. **OpenMP-level (Task Parallelism):** After keypoint candidates are identified (this part of the pipeline runs on the root process), the task of calculating keypoint orientations and descriptors is parallelized using OpenMP. The list of keypoint candidates is treated as a "bag of tasks". Each thread dynamically pulls a candidate from this list, computes its orientations and descriptors, and repeats until all candidates are processed.

What scheduling algorithm did you use?

- **MPI (Static Scheduling):** The partitioning of the image into 2D blocks is done once at the beginning of the `generate_gaussian_pyramid_parallel` function. Each process's workload is fixed throughout this stage. This is a form of static scheduling, which is highly efficient for uniform workloads like Gaussian blurring.
- **OpenMP (`schedule(dynamic)`):** For the descriptor calculation loop, we use the `dynamic` scheduling clause. The work required for each keypoint candidate can vary significantly depending on the number of orientations found. Dynamic scheduling is ideal here because it assigns loop iterations to threads as they become free, naturally balancing the load and preventing threads with "easy" tasks from sitting idle.

What techniques do you use to reduce execution time?

Several techniques were employed to optimize performance:

1. **Hybrid Parallelism:** Combining MPI and OpenMP allows the program to leverage both inter-node (distributed memory) and intra-node (shared memory) resources, enabling scaling on multi-core, multi-node clusters.
2. **Optimized Data Decomposition:** Using a 2D block decomposition for the image processing stage minimizes the halo exchange overhead compared to a simpler 1D (slab) decomposition, thus improving arithmetic intensity.
3. **Computation/Communication Overlap:** During the halo exchange, non-blocking MPI calls (`MPI_Irecv`, `MPI_Isend`) are used. The program initiates communication and then immediately begins computing the inner part of the image block that does not depend on the halo data. The program only waits for the communication to complete when it needs to process the boundary regions, effectively hiding communication latency behind computation.
4. **Adaptive Work Scheduling:** Using OpenMP's `dynamic` schedule for the descriptor calculation loop ensures high thread utilization even with an irregular workload.

Other efforts you make in your program.

A significant effort was made to **verify performance and correctness**. We instrumented the code to explicitly measure load balancing at both the MPI and OpenMP levels.

- For MPI, we timed the pyramid generation stage on each process and aggregated the min, max, and average times to quantify the balance.
- For OpenMP, we had each thread count the number of descriptors it computed to verify that the dynamic scheduler was distributing the irregular workload effectively.

What difficulty did you encounter in this assignment?

One of the primary challenges in this assignment was methodological, specifically in collaborating effectively with an AI assistant for code generation and modification. Initially, it was difficult to track the scope of code changes and to debug issues when the modifications were too broad.

To solve this, I adopted a more structured approach by creating a checklist to manage each change proposed by the AI assistant. This allowed me to break down the problem into smaller, verifiable steps. By focusing on incremental modifications, I could quickly validate that each small change was correct and contributed positively to the program's performance, leading to a more efficient and manageable development process.

Analysis

Load Balance Plots

The following tables are generated from the program's output on the **sixth test case**. They quantify the excellent load balance achieved in both parallel sections.

MPI Load Balance for Gaussian Pyramid Generation

Metric	Value
Min Time	10.501783 s
Max Time	10.501790 s
Avg Time	10.501786 s
Imbalance (Max-Min)	0.000007 s
Imbalance (Max/Avg)	**0.00%**

OpenMP Load Balance for Descriptor Calculation

Metric	Value
Total Threads	12
Total Descriptors	164445
Min work per thread	13475
Max work per thread	13755
Avg work per thread	13703.75
Imbalance (Max/Avg)	**0.37%**

(You can add your screenshot here and provide a brief description, for example: "The following bar chart visualizes the data above, showing the near-identical workload for each thread.")

Scalability Analysis

- **Number of Nodes:** The overall application scalability across nodes is limited by Amdahl's Law. While the initial `generate_gaussian_pyramid` step is fully distributed and should scale well with more nodes, the subsequent steps (DoG pyramid, keypoint finding, descriptor calculation) are confined to the single root process. Therefore, the speedup gained by adding more nodes will diminish as the single-node portion becomes the bottleneck.
- **Number of Processes per Node:** Increasing the number of MPI processes per node will continue to speed up the pyramid generation step, as long as the image blocks remain large enough to have a good computation-to-communication ratio. If too many processes are used, the blocks become very small, and the constant overhead of halo exchange will start to dominate, leading to diminishing returns.
- **Number of CPU Cores per Process (OpenMP Threads):** The descriptor calculation stage, which runs on the root process, scales very well with the number of cores. As shown by our load balance analysis, the work is distributed very evenly. The speedup in this section should be nearly linear with the number of cores, limited only by the small serial overhead of setting up the parallel region and merging results.

Conclusion

What have you learned from this assignment?

(This section is for your personal reflection. Here is a sample answer you can adapt.)

This assignment provided critical hands-on experience in developing a high-performance, hybrid parallel application. The key takeaways were:

1. **Problem Decomposition:** The importance of analyzing an algorithm to identify different types of parallelism (data-parallel vs. task-parallel) and applying the appropriate model (MPI vs. OpenMP) to each.
2. **Communication vs. Computation:** Understanding the trade-offs in data partitioning. The 2D block decomposition highlighted how minimizing the "surface-to-volume" ratio is critical for reducing communication overhead in stencil-based computations.
3. **Load Balancing Strategies:** Learning that different workloads require different scheduling policies. A static approach was perfect for the uniform pyramid generation, while a dynamic approach was essential for the irregular descriptor calculation.
4. **Performance Measurement:** Realizing that you cannot optimize what you cannot measure. Instrumenting the code to verify load balance was crucial for confirming that our design choices were effective.