

**1. Explain your implementation, especially in the following aspects:**

**★How do you partition the task? /**

My implementation utilizes a hybrid parallel programming model combining MPI for inter-process communication across nodes and OpenMP for multi-threading within each process. This two-level parallelization strategy is designed to efficiently utilize the resources of a modern computing cluster.

- **MPI-level Partitioning (Process-based):** The primary task partitioning is handled by MPI. After the root process (rank 0) generates the full Gaussian and Difference-of-Gaussians (DoG) pyramids, these data structures are broadcast to all other MPI processes. The most computationally intensive part of the algorithm, the search for keypoint extrema in the DoG pyramid, is then partitioned among the processes. Specifically, the columns of each image within the scale-space octaves are divided. Each MPI process is responsible for scanning a distinct vertical slice (a range of x-coordinates) of the images to detect, refine, and discard keypoints. This is a form of spatial decomposition.
- **OpenMP-level Partitioning (Thread-based):** Within each MPI process, OpenMP is used to further parallelize computationally expensive loops. This applies to several stages:
  - **Keypoint Detection:** The main loop that iterates over the assigned columns (x coordinates) and all rows (y coordinates) to find extrema is parallelized using an OpenMP `parallel for` pragma.
  - **Descriptor Calculation:** After the initial keypoints are found, the subsequent step of computing orientations and descriptors for each keypoint is also parallelized. The list of candidate keypoints is divided among threads, with each thread handling a subset of keypoints independently.

- **Image Processing Functions:** Many of the helper and image processing functions (e.g., `rgb_to_grayscale`, `gaussian_blur`, image copying, and saving) are parallelized with OpenMP to accelerate these stages as well.

Finally, after each process has computed its local set of keypoints, the results are gathered back to the root process using `MPI_Gatherv` to form the complete list of keypoints for the entire image.

### **★ What scheduling algorithm did you use: static, dynamic, guided...?**

I employed a mix of OpenMP scheduling strategies tailored to the nature of the workload in different parts of the program:

- `schedule(dynamic, 4)`: This is used for the main keypoint detection loop (`find_keypoints`) and the descriptor computation loop (`compute_keypoints_from_pyramids`). In these loops, the amount of work per iteration is unpredictable. For example, some regions of the image may contain many potential keypoints requiring expensive refinement, while others have none. Dynamic scheduling is ideal here because it assigns a small chunk of iterations (4 in this case) to a thread and, once a thread is finished, it requests a new chunk. This helps to naturally balance the load between threads, preventing situations where some threads finish early and sit idle while others are still working on computationally heavy regions.
- `schedule(static)`: This is used for more regular and predictable loops, primarily in the `image.cpp` file for tasks like image format conversion, blurring, and copying. In these cases, each iteration of the loop (e.g., processing a single pixel) takes roughly the same amount of time. Static scheduling, which divides the iterations evenly among threads at the beginning of the loop, is more efficient here as it has lower scheduling overhead than dynamic scheduling.

### **★ What techniques do you use to reduce execution time?**

A key technique used to optimize performance was improving **data locality**. The `Image` class stores pixel data in a channel-major format:

`data[c*width*height + y*width + x]`. This memory layout means that pixels that are adjacent horizontally (i.e., with consecutive `x` values) are also contiguous in memory.

To take advantage of this, I ensured that in all nested loops that iterate over pixels, the inner loop iterates over the `x` coordinate and the outer loop iterates over the `y` coordinate.

This access pattern results in a linear scan through memory (a stride of 1), which allows the CPU's prefetcher to work effectively and maximizes cache hits. Swapping the order of these loops would lead to large strides in memory access, causing frequent cache misses and significantly degrading performance. This simple change yielded a substantial reduction in execution time.

### **★ Other efforts you make in your program?**

Beyond parallelizing the core SIFT algorithm, I also applied OpenMP parallelization to many of the auxiliary image processing and data manipulation routines. This includes:

- Loading the image from a file and converting it to a floating-point representation.
- Converting RGB images to grayscale.
- Applying the separable Gaussian blur.
- Copying image data in the copy constructor and assignment operator.
- Saving the final image with keypoints drawn on it.

While these operations are not as computationally dominant as the keypoint detection and descriptor generation, parallelizing them contributes to reducing the overall execution time, especially for large images where these preparatory and finalization steps can become non-trivial. This ensures that the entire pipeline, not just the main algorithm, is optimized.

### **★ What difficulties did you encounter in this assignment? How did you solve them?**

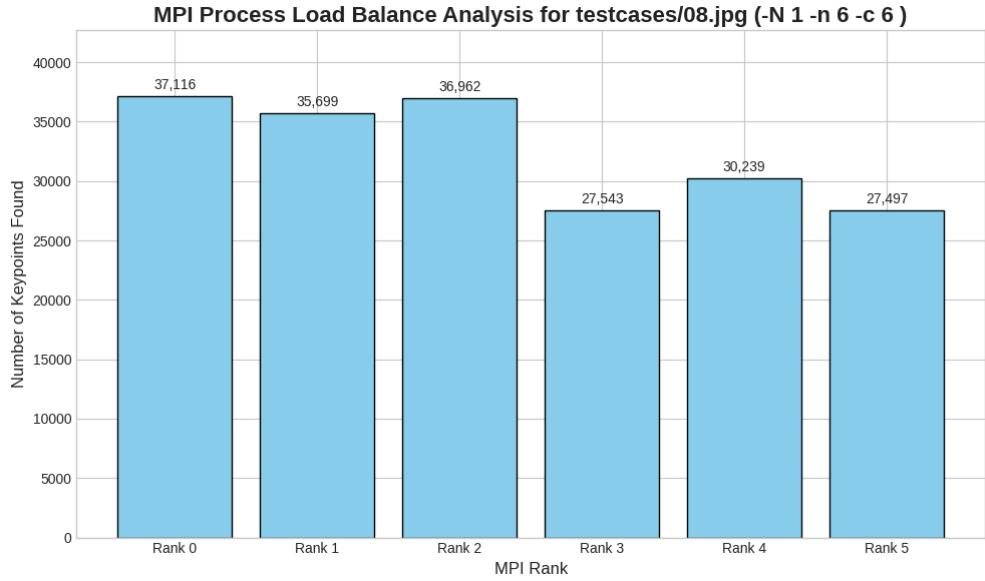
- **Incorrect Initial Partitioning Strategy:** My first thought for parallelization was to physically slice the input image into tiles and have each MPI process run the entire SIFT algorithm on its own sub-image. This approach was fundamentally flawed because the SIFT algorithm relies on building scale-space pyramids. Operations like Gaussian blurring mean that the value of a pixel depends on its neighbors. If the image is split beforehand, this introduces severe incorrect boundary artifacts, as pixels near the edge of a tile cannot access their true neighbors from the adjacent tile. I solved this by realizing the image data must be kept whole during the pyramid generation phase. The correct strategy was to build the full pyramids on the root process, broadcast them to all processes, and only then partition the search space (the image coordinates) for keypoint detection among the processes.
- **Unexplained Poor Performance:** Initially, my parallel code was not scaling as well as expected, and even the sequential version was slower than I anticipated. After significant debugging, I discovered the issue was poor cache utilization due to incorrect loop ordering. As mentioned in the previous section, my loops were iterating over  $y$  on the inside and  $x$  on the outside. This led to non-sequential memory access patterns and a high rate of cache misses. The solution was to analyze the memory layout of the image data and re-order the nested loops to ensure the innermost loop iterates over the contiguous memory dimension ( $x$ ). This was a valuable lesson learned (and later reinforced in class on 10/17) about how crucial memory access patterns are for performance in computational tasks.

## 2. Analysis:

**★Design your own plots to show the load balance of your algorithm between threads/processes.**

- **MPI Process Load Balance:** This experiment aims to visualize how evenly the workload (number of keypoints found) is distributed

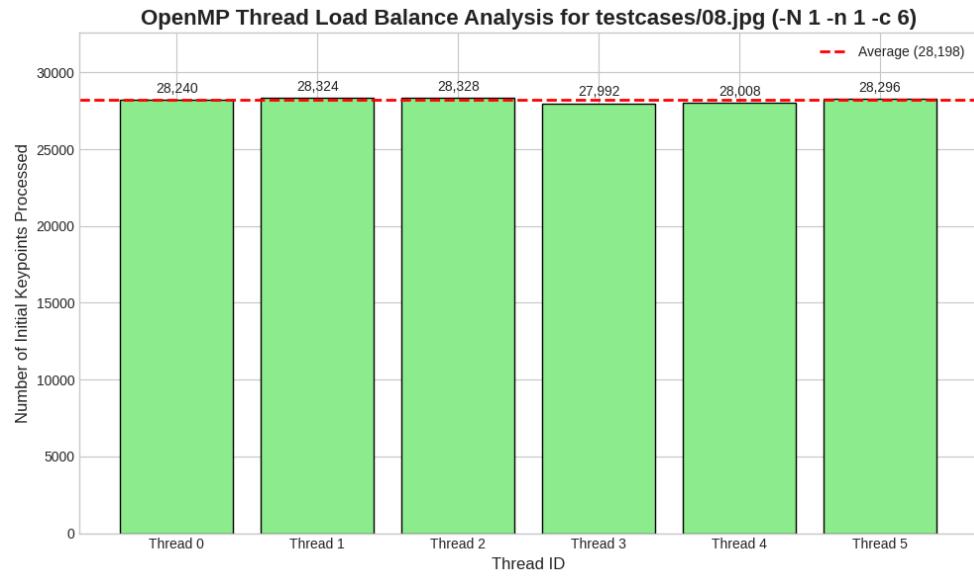
among MPI processes.



The results reveal a significant load imbalance among the MPI processes. The number of keypoints detected per process varies widely, from a low of 27,497 (Rank 5) to a high of 37,116 (Rank 0), a difference of over 35%.

This imbalance is a direct result of the static partitioning strategy. Since image features are not distributed uniformly, assigning fixed vertical slices of the image to each process means that processes responsible for feature-rich areas (like Ranks 0, 1, and 2) perform substantially more work. This static approach, while simple to implement, limits the overall performance, as faster processes must wait for the most heavily loaded ones to complete.

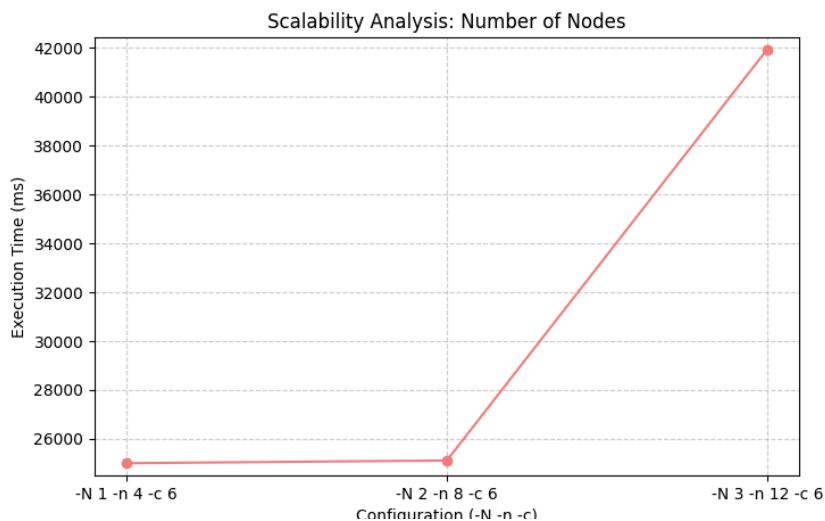
- **OpenMP Thread Load Balance:** This experiment aims to visualize how evenly the `schedule(dynamic)` policy distributes work among threads within a single process (Measure the number of initial keypoint candidates processed by each thread as a proxy for workload).



The workload distribution among OpenMP threads is exceptionally well-balanced. The number of initial keypoints processed by each of the 6 threads is very close to the average of ~28,200. The standard deviation is only 135 keypoints, which represents a variation of less than 0.5% from the mean. This result strongly validates the choice of schedule (dynamic, 4). It demonstrates that for the irregular task of processing keypoint candidates, the dynamic scheduling policy effectively distributes the work, ensuring all threads remain busy and minimizing idle time.

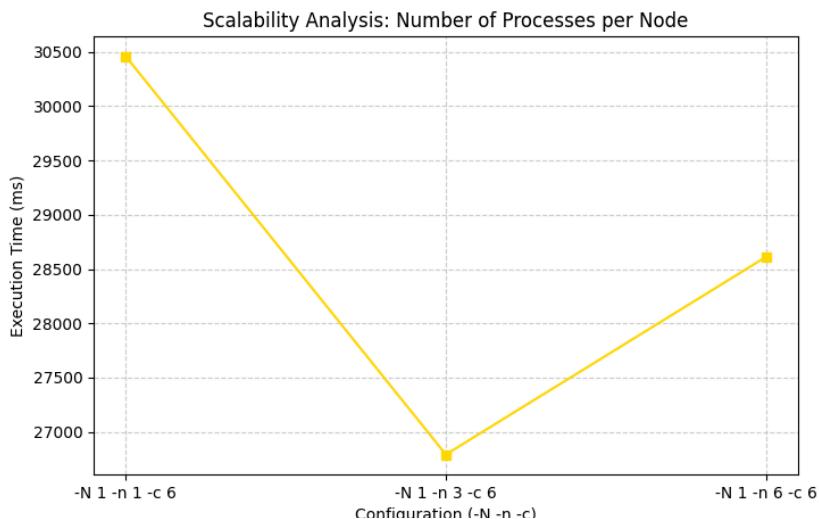
### **★Analyze your program's scalability in the following aspects:**

- **number of nodes**



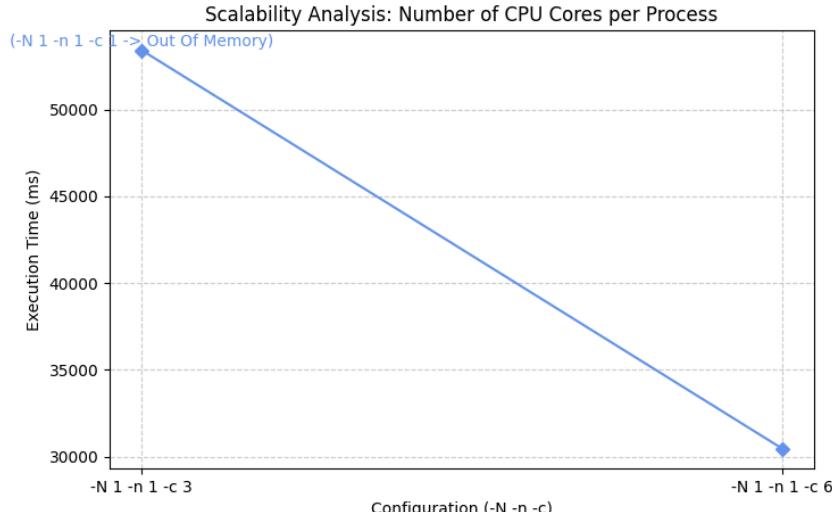
The application scales well from one to two nodes, with the execution time remaining nearly constant. This indicates that the performance gain from doubling the computational resources is almost perfectly offset by the added inter-node communication overhead. However, there is a severe performance degradation when scaling to three nodes. This sharp increase in execution time suggests that inter-node communication, particularly the broadcasting of large scale-space pyramids and gathering results, has become the dominant bottleneck, overwhelming the benefits of additional parallel computation.

- **number of processes per node**



Performance improves when scaling from one to three processes on a single node, indicating that the task partitioning is beneficial. However, the execution time increases when moving from three to six processes. This suggests that for this problem, the overhead associated with MPI communication and process management begins to outweigh the computational gains beyond three processes. Resource contention on a single node likely contributes to this performance degradation.

- **number of CPU cores per process**



The results show good strong scaling when increasing the number of CPU cores per process. Doubling the cores from 3 to 6 results in a significant reduction in execution time, demonstrating that the OpenMP `parallel for` directives are effectively distributing the workload among threads. The "Out of Memory" error with a single core suggests the dataset is too large to be processed sequentially within the available memory, highlighting the necessity of parallel processing for this problem size.

### 3. Conclusion:

#### **★What have you learned from this assignment?**

This assignment provided a comprehensive, hands-on experience in parallel programming, teaching me how to effectively implement a hybrid MPI+OpenMP model for different granularities of parallelism. A critical lesson was matching the parallelization strategy to the algorithm's characteristics; for SIFT's irregular workload, this meant first handling the data-dependent pyramid generation before parallelizing the keypoint search, and then using OpenMP's dynamic scheduling to balance the uneven computational load. Furthermore, I discovered the immense impact of hardware-aware optimization, as improving data locality by reordering nested loops to match memory layout yielded significant performance gains through better cache efficiency. Finally, this project made the trade-offs of distributed computing tangible, highlighting how communication overhead from MPI operations can become a limiting

factor to scalability, creating a necessary balance between parallel computation and communication costs.