# Summary

We implemented a star trail generation program and accelerated it on the multicore CPU using SIMD and OpenMP/MPI.

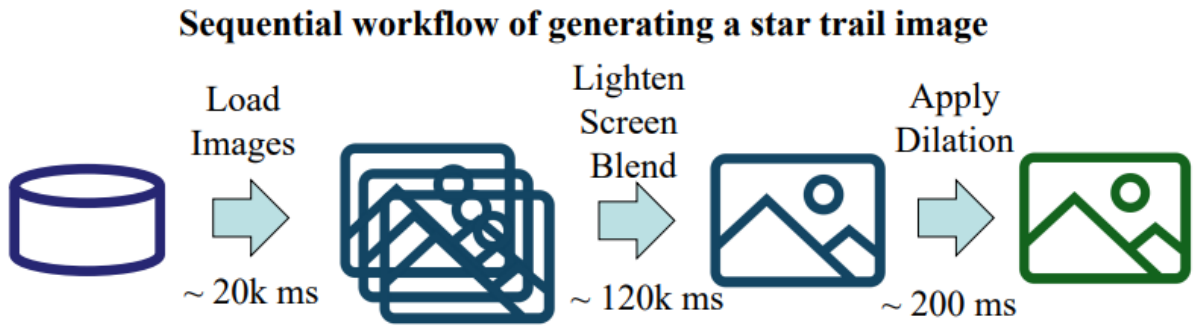
zhyueyan/15618Project

# 1. Background

## Introduction

Star trails are continuously moving orbits produced by stars which can be observed under hours of exposure. Due to the cost of long exposure, in actual photography, the photographer would typically choose to take more than 100 continuous photos with exposure times of around 30 seconds each, and later combine them in post-processing to create the star trails. The image below demonstrates the effect after image processing. 183 images (716 MB in total) were processed and combined to create the star trail effect shown in the image.

## Sequential Workflow




The algorithm involves subtraction and multiplication on 1D arrays storing pixel data in RGB format representing the images. It takes a large amount of high-resolution images as input and a single image as output.

We can observe from the workflow that:

1. The bottleneck of this algorithm is the first 2 steps. Image loading is I/O-intensive and light screen blending is compute-intensive, which could both benefit from parallelization.
2. The processing of every image is based on the result of all previous images, so the inter-image parallelism is infeasible. However, processing of pixels within an image is independent, and intra-image parallelism is viable.
3. The RGB information of pixels are stored sequentially in the 1D array, so SIMD executions processing adjacent pixels can exploit locality.

# 2. Method

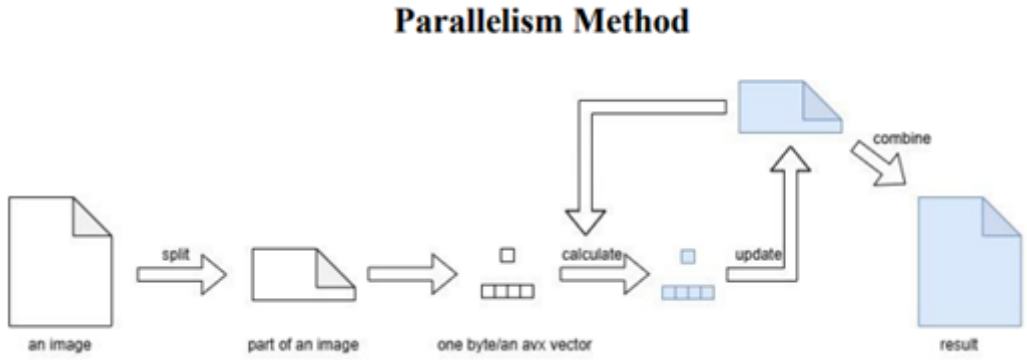
## Overview

Our optimizations focused on Step 1 (loading) and Step 2 (blending), which takes the majority of processing time. We adapted the following technics on multi-core CPU to accelerate:

- Multi-core CPU Acceleration with OpenMP and MPI.
- SIMD Acceleration with AVX instructions.

We chose C++ as the working language for its high performance and lower execution overhead.

## Parallelism Method



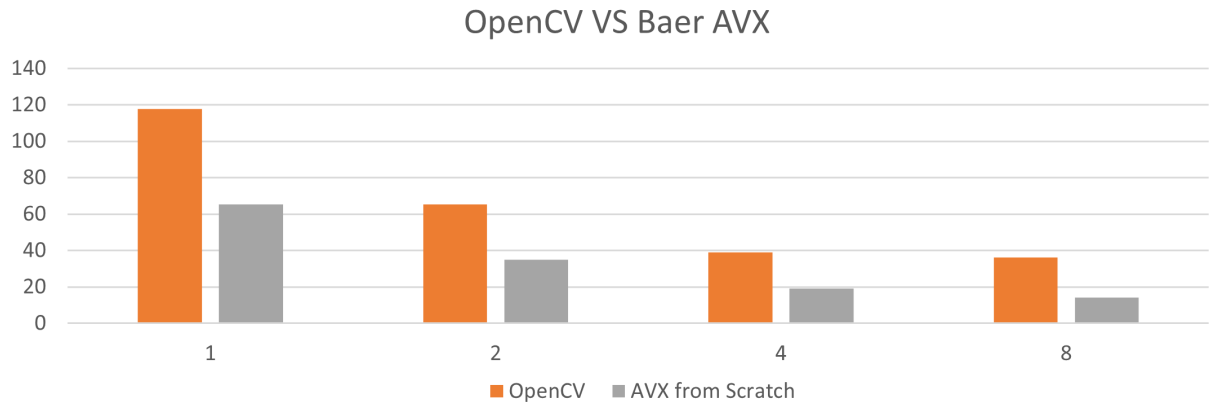
Generally, we do inter images parallelism when loading pictures and intra images parallelism when processing pictures.

**SIMD Parallelism.** We utilized the maximum 8 vector size to process 8 sequential pixels simultaneously. We implemented AVX operations with support of C++ lib to realize SIMD. For the sequential 1D array storage pattern, there is no need for padding or aligning.

**Multi-core Parallelism.** In the blending step, the image is divided into row-continuous chunks to ensure data locality for efficient memory access, and processing a chunk is a task to be assigned to a separate core, enabling parallel computation across multiple cores. In OpenMP, the task granularity is one single image in loading and one row in blending, and tasks are dynamically assigned to threads; in MPI, two steps of message passing are adapted: 1. Each round, the process owning the picture will use Scatterv to send chunks to corresponding processes. The total amount of message is n * image_size, where n is total image num. 2. At the end of iterations, the processes will use Gatherv to gather chunks to a whole picture.  The total amount of message is image_size.

### Roadmap

Our roadmap includes: 1.  first implementing 2 basic versions of the algorithm, one completely sequential, and the other using OpenCV library, with AVX vector operations but no multi-core optimization. 2. Then we tried with OpenMP and MPI on both versions, and discovered that multi-core acceleration was poor on OpenCV version at large core numbers. By digging into code, we observed that this issue arises because the OpenCV-provided interface does not use in-place algorithms, leading to increased data movement and memory overhead. 3. Therefore, we implemented an in-place image processing algorithm using AVX. The following comparison demonstrates that our custom implementation achieves significant performance improvement in the blend step and exhibits better scalability.




* Take work load with 150 photos, each with a resolution of 6240x4160 pixels as example

# 3. Result

### SIMD Speedup

We first evaluated the parallel performance of SIMD by comparing the single-core SIMD implementation against the single-core sequential version across varying problem sizes. The experiments were conducted on GHC machines with maximum 8 vector-size and the results are shown as below:

| Image Size | Image Num | Speedup |
|---|---|---|
| small | 50 | 3.808161193 |
| small | 100 | 4.219854432 |
| small | 150 | 4.463259346 |
| medium | 50 | 4.297167417 |
| medium | 100 | 4.639645407 |
| medium | 150 | 4.690935819 |
| large | 50 | 4.525072304 |
| large | 100 | 4.718855223 |
| large | 150 | 4.758935529 |

The SIMD technique provides approximately a **4x improvement in overall performance.** We also observed that the parallel performance improves as the computation workload increases. This is because we measured the speedup based on the total execution time of the entire process. As the computation workload grows, the proportion of the SIMD-parallelizable portion within the total execution flow becomes larger, leading to better overall parallel performance.

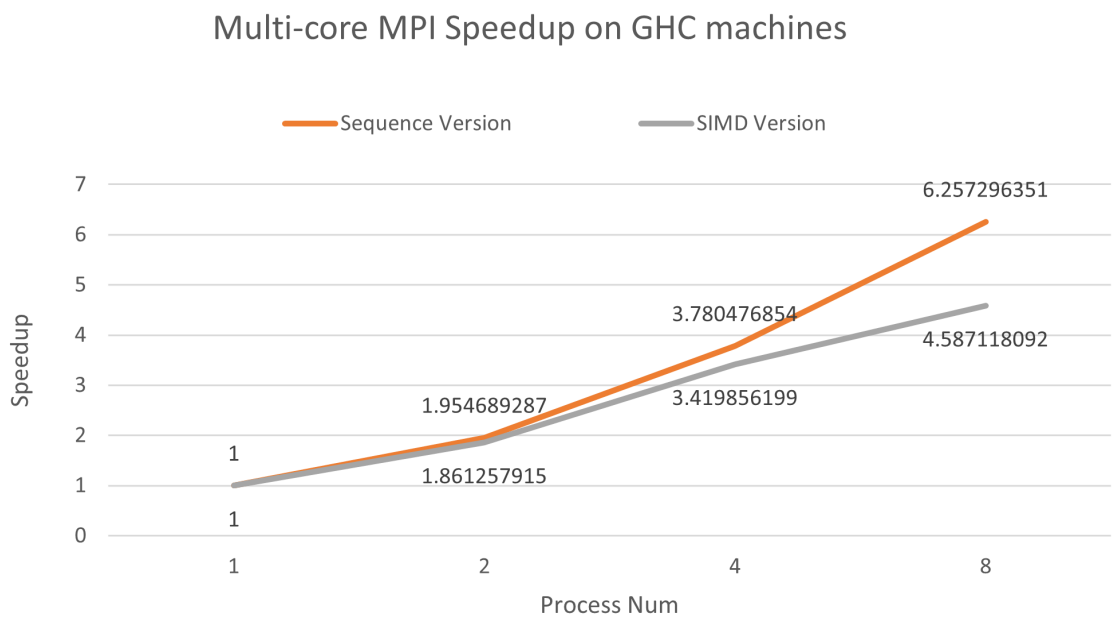
By separately measuring the execution time of the Lighten-Screen-Blend component, we found that it achieves **near-perfect parallelism and good scalability for different vector-size**. This is because the calculation workload for each pixel is nearly the same and no branches are needed, so there is little idle time for waiting.

| Vector width | Speedup |
|---|---|
| 2 | 1.91634236 |
| 4 | 3.53794305 |
| 8 | 7.57567105 |

## MPI Speedup Result

We extended multicore parallelism on top of SIMD and measured the performance of MPI-based multicore parallelism on both PSC and GHC machines. In this case, the speedup baseline is the single-core, non-SIMD sequential version. We evaluated the scalability of both SIMD-enabled and non-SIMD versions under multicore configurations. **The workload consists of processing 150 photos, each with a resolution of 6240x4160 pixels and a size of approximately 10 MB.**

**GHC Machine Result**

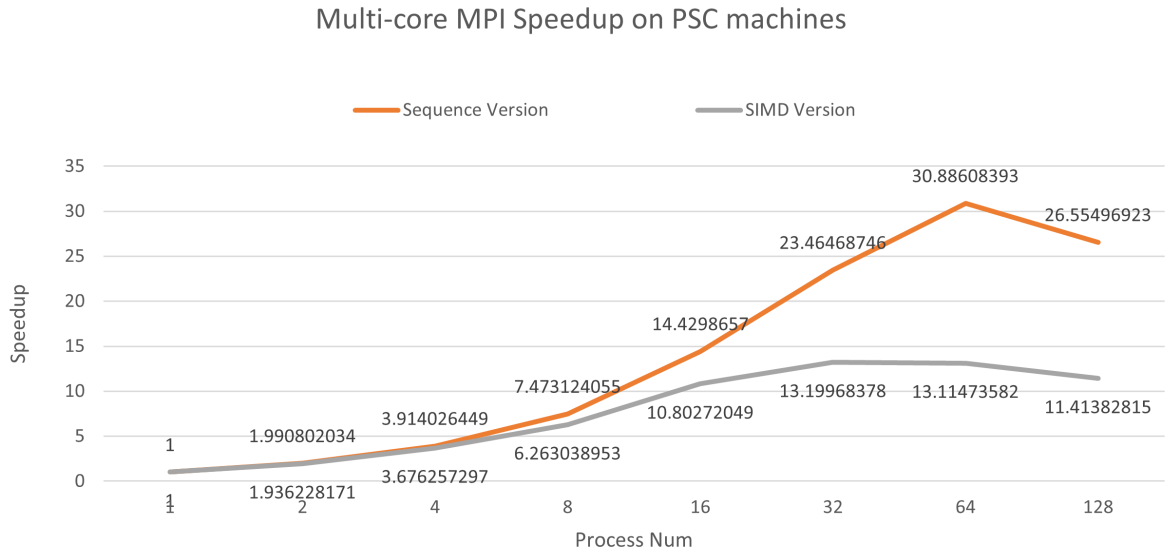



For the sequential version, the theoretical maximum speedup should be nx, where n is the number of processes. For 1, 2, and 4 processes, the program scales well and achieves nearly optimal speedup. However, the speedup at 8 processes is slightly lower than the ideal 8x.

For the SIMD version, we observed the same trend as the sequential version: it did not achieve perfect speedup and performed even worse at 8 threads.

**PSC Machine Result**

We tested the same workload on the PSC machine. The result is as below:

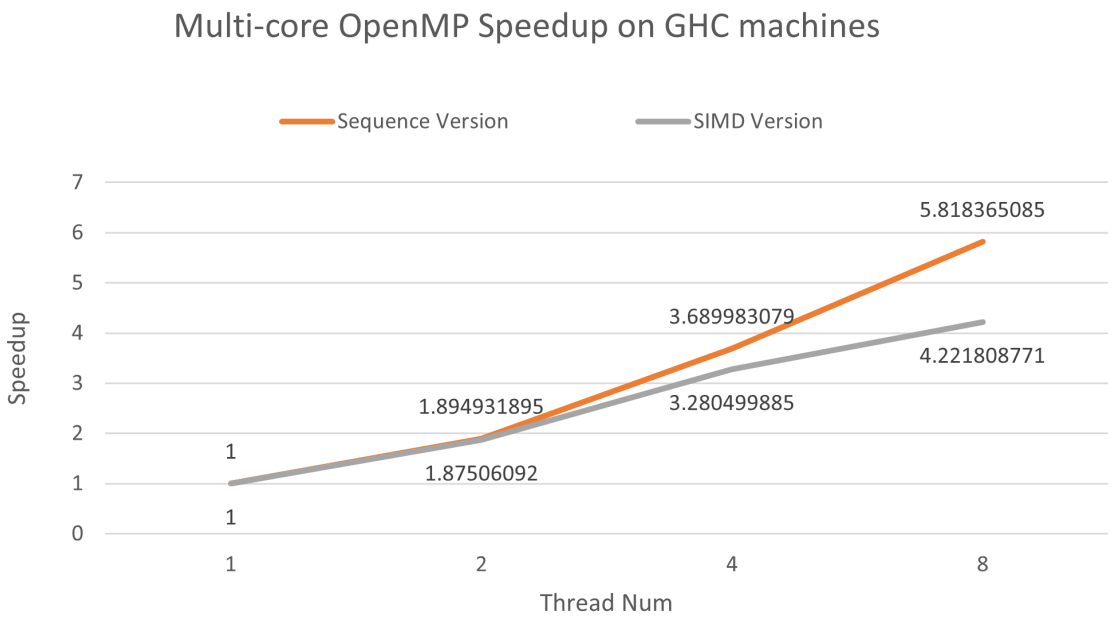



For the sequential version, the sequtial version scales good. We also observed better scalability on the PSC machine, with a higher speedup at eight cores compared to the GHC machine. The only dropoff here is at the 128-processes speedup.

For the SIMD version, the scaling performance was similarly worse; instead of exponential growth, the performance improvement exhibited a linear trend.

## OpenMP Speedup

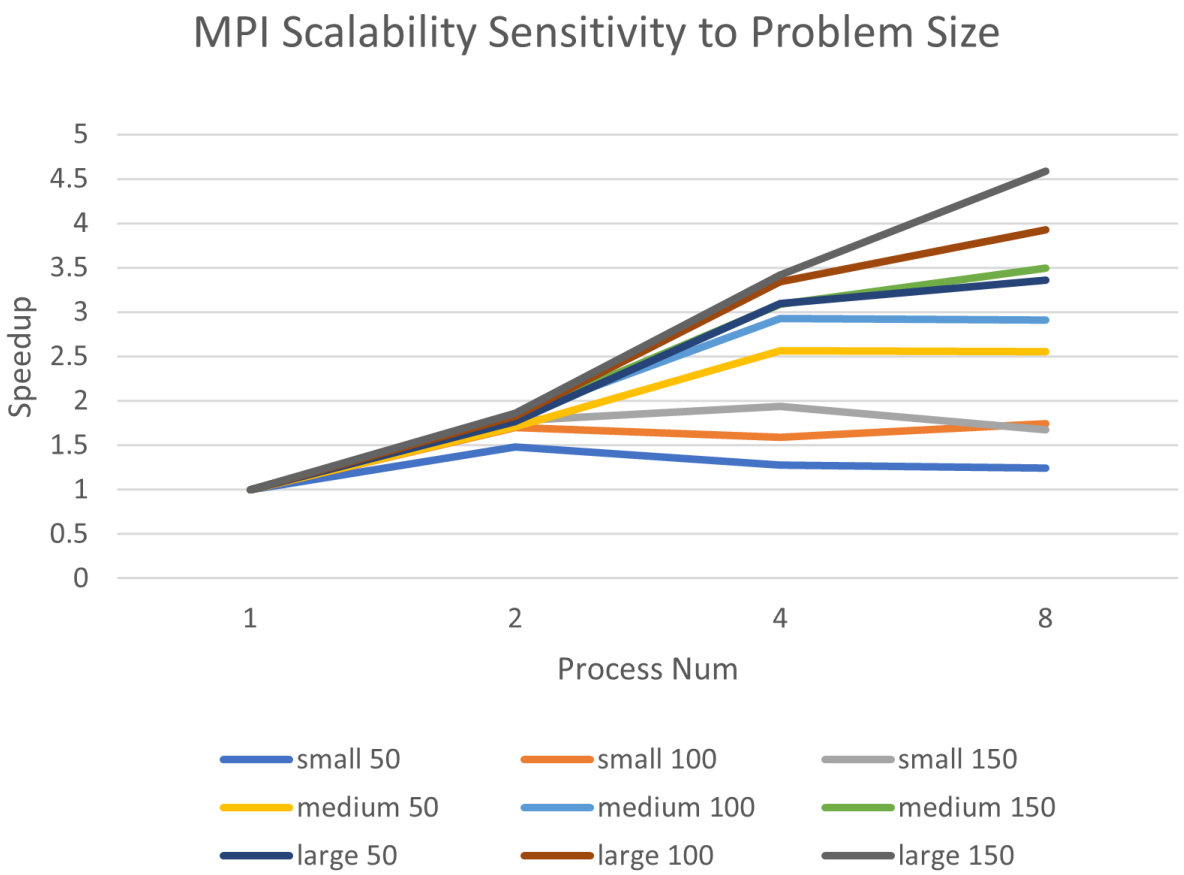
Like the MPI experiments, we used the same workload (150 photos, each with a resolution of 6240x4160 pixels and a size of approximately 10 MB) to measure the speedup effect of OpenMP model.




We observed a similar trend to MPI, where the sequential version scales well, but the SIMD version exhibits poorer scale-up performance at 8 threads.

## Sensitivity to Problem Size

To test our implementation's sensitivity to problem size, we measured the speedup rate for MPI and OpenMP method with SIMD on 1. different image sizes (large: 6240x4160, medium: 3120x2080, small: 1560x1040). 2. different image number (50, 100, 150). The experiments are conducted on GHC machines and the results are as below:

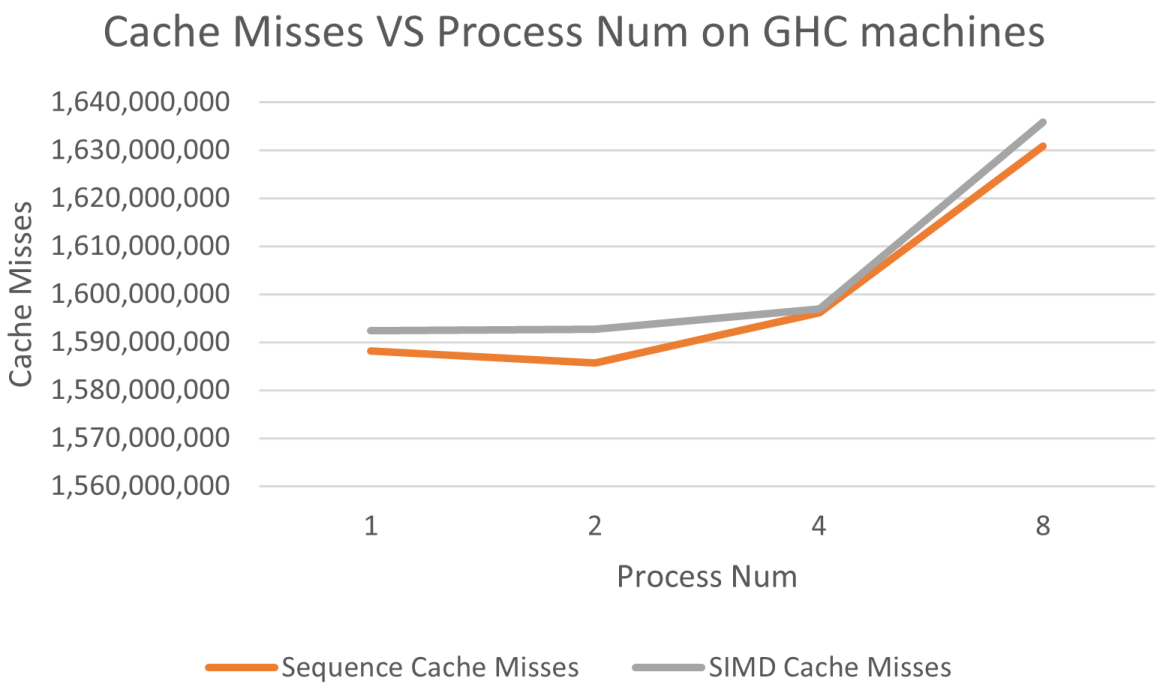



The smaller the total computation workload, the less significant the scale-up effect. This is because the proportion of computation in the total execution time decreases, while the share of non-parallelizable components increases.

## Bottleneck Analysis

### Cache Contention

By measuring cache misses on GHC machine, we observed that cache misses peaked when the number of processes was 8. Therefore, we infer that the performance degradation at 8 processes is due to **contention for the shared L3 cache**, which limits the ideal scalability.




Shared cache also accounts for the better scalability on PSC machines. By examining the L3 cache sizes of the two machines and comparing cache miss counts, we attribute this improvement to the **larger L3 cache** on the PSC machine, which alleviates cache contention between cores, resulting in better scalability. This further supports our explanation for the lack of perfect speedup on the GHC machine with 8 cores. The cache difference for 2 machines are as below:

PSC Cache:

```
L1d cache:           32K
L1i cache:           32K
L2 cache:            512K
L3 cache:            16384K
NUMA node0 CPU(s):   0-63,128-191
NUMA node1 CPU(s):   64-127,192-255
```

GHC Cache:

```
Caches (sum of all):
  L1d:                   256 KiB (8 instances)
  L1i:                   256 KiB (8 instances)
  L2:                    2 MiB (8 instances)
  L3:                    12 MiB (1 instance)
```

**Memory Bandwidth**

Memory bandwidth is not the main issue. Based on cache misses and total execution time, we calculated the memory throughput to be around 20 MB/s, while the actual memory bandwidth is in the GB/s range. Therefore, memory bandwidth is not the bottleneck.

```
 1,592,417,120      cache-misses              #   63.792 % of all cache refs
 2,496,278,988      cache-references

   64.054576384 seconds time elapsed

   58.132783000 seconds user
    5.492504000 seconds sys
```

**Message Passing & Memory Sharing**

In MPI, the time spent on scattering the image accounts for 25% of the total execution time, but it decreases proportionally as the number of processes increases. Therefore, it is not the primary reason for SIMD's inability to scale up. Similarly, the gather time accounts for only 0.03% of the total execution time, making it an insignificant factor.

In OpenMP, there is no need to synchronize among threads, so sharing memory can be executed parallelly.

Based on the breakdown of speedup, it is the SIMD computation part that fails to scale up as the number of processes increases. This suggests that cache contention is the primary bottleneck.

**NUMA Architecture**

We observe a dropoff at the 128-processes speedup on PSC for both MPI and OpenMP, which is mainly due to the NUMA architecture that increases message passing and memory sharing overhead.

```
NUMA node0 CPU(s):    0-63,128-191
NUMA node1 CPU(s):    64-127,192-255
```

# 4. Conclusion

Our parallelization achieved excellent results on single-core SIMD, and multicore sequential parallelization also demonstrated nearly ideal scale-up. However, for multicore SIMD, the scale-up results fell short of expectations. After analysis, we attribute this primarily to contention for the shared L3 cache and thus falling to slow memory load. Moving forward, we believe implementing acceleration on GPUs could yield better results, as the GPU architecture features more cores and higher memory bandwidth beneficial for high-resolution pictures parallelism and loading.

# 5. Contributions

Yueyan Zhang: SIMD, MPI: 50%

Tao Zhu: OpenMP, PSC & GHC tests: 50%