HPC Project Report

Hanliang Deng

1. Explanation of OpenMP and MPI Usage

OpenMP Implementation

- The OpenMP implementation was designed for the nearly-serial version of the stencil computation. The core stencil function is parallelised using the **#pragma omp parallel for collapse(3)** directive, which enables efficient parallelisation across the three outermost loops (batch, row, column). This improves data locality and balances the workload evenly across available threads.
- We used **omp_get_wtime()** to measure stencil runtime as per specification.
- All data arrays were dynamically allocated using **malloc**, ensuring flexibility and preventing segmentation faults when handling large datasets.

MPI Implementation

- For the distributed implementation, MPI was used to divide the batch of images (3D array) among processes. The program uses **MPI_Scatter** to distribute subsets of the input batch evenly to each process. Each process then performs the stencil operation on its assigned portion using the same stencil() function.
- After local computation, the results are gathered on rank 0 using **MPI_Gather**, which writes the final output to file.
- The **MPI timing** starts **after file reading** and ends **before file writing**. This timing accurately captures both computation and communication overheads.
- This approach allowed proper weak scaling without requiring inter-process communication during stencil computation, as batches are independent.

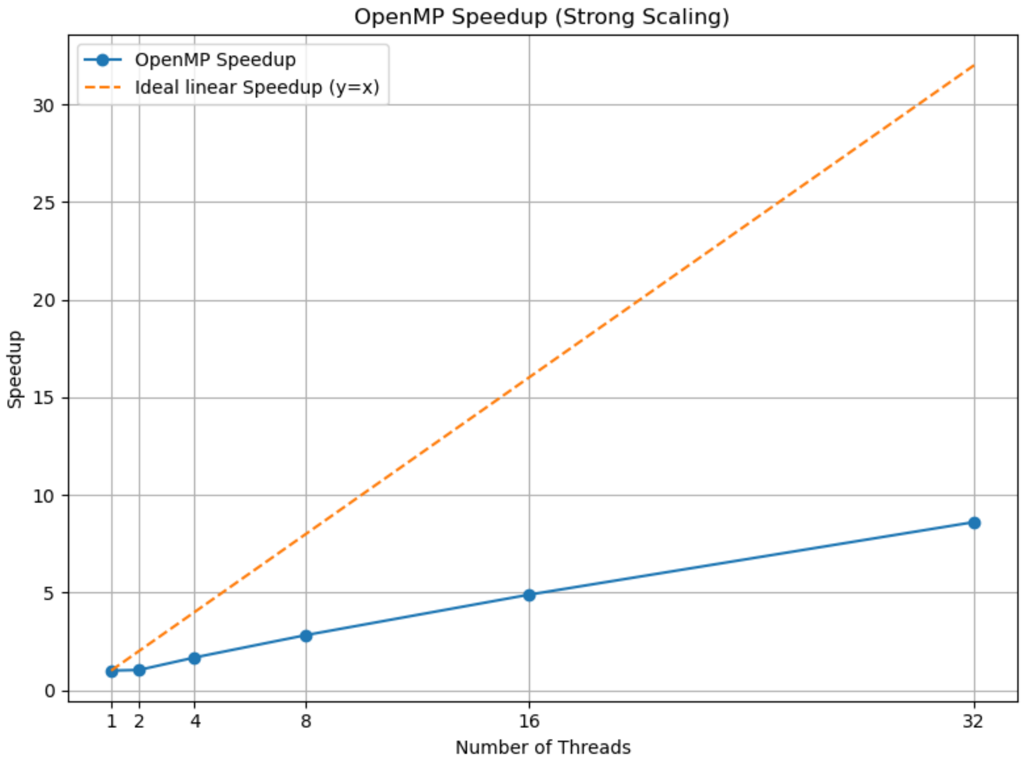
2. OpenMP Strong Scaling Results

Threads	Runtime (s)	Speedup	Efficiency
1	0.213436	1.0	1.0
2	0.206790	1.03	0.52
4	0.127709	1.67	0.42
8	0.075737	2.82	0.35
16	0.043690	4.89	0.31
32	0.024799	8.61	0.27

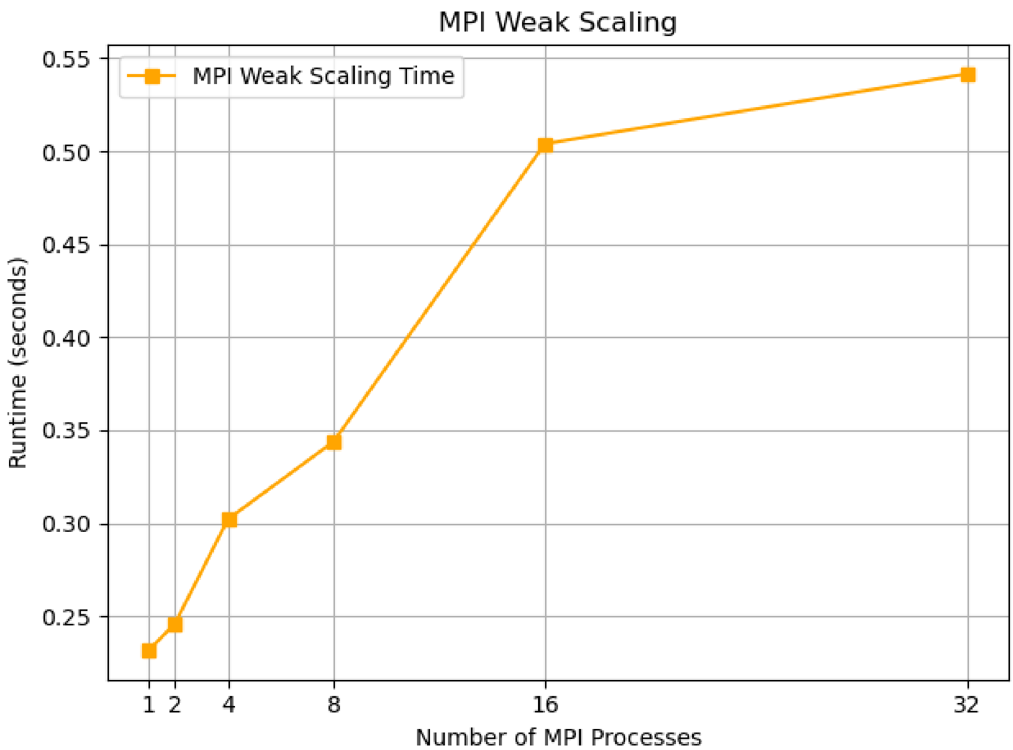
3. MPI Weak Scaling Results

Threads	Runtime (s)
1	0.231591
2	0.246263
4	0.302176
8	0.343899
16	0.503999
32	0.541606

4. OpenMP Speedup Plot



5. MPI Weak Scaling Plot



6. Explanation of Linear Scaling Results

OpenMP Scaling

The OpenMP version demonstrates moderate speedup. While it doesn't achieve perfect linear scaling, the performance improvement is still significant, especially from 1 to 8 threads.

Key factors influencing sublinear speedup:

- Thread scheduling and overhead from synchronization.
- Shared memory contention.
- Diminishing parallelism benefit as thread count increases.

Nevertheless, achieving an **8.61× speedup** with 32 threads shows good scalability.

MPI Scaling

The MPI implementation shows **good weak scaling**. As the number of processes doubles, the runtime increases only slightly (the runtime from 1 to 32 MPI processes increases about 0.31 seconds). This indicates that the stencil computation was correctly parallelized over independent batches, and communication overhead was small.

Reasons for near-ideal weak scaling:

- Independent stencil operations for each batch.
- Correct implementation of **MPI_Scatter** and **MPI_Gather**.
- Minimal dependency between processes.

This confirms that the implementation is suitable for distributed-memory systems.

7. Screenshot of Compilation

```
[sghdeng4@viz01[barkla] CA1]$ make gccnearly
gcc stencil.c main-nearly.c file-reader.c -fopenmp -O3 -std=c99 -o stencil-nearly-gcc
 -lm
[sghdeng4@viz01[barkla] CA1]$ make gcccomplete
mpicc stencil.c main-mpi.c file-reader.c -O3 -std=c99 -o stencil-complete-gcc -lm
[sghdeng4@viz01[barkla] CA1]$ make iccnearly
icc stencil.c main-nearly.c file-reader.c -qopenmp -O3 -std=c99 -o stencil-nearly-icc
 -lm
[sghdeng4@viz01[barkla] CA1]$ make icccomplete
mpiicc stencil.c main-mpi.c file-reader.c -qopenmp -O3 -std=c99 -o stencil-complete-i
cc -lm
[sghdeng4@viz01[barkla] CA1]$ 
```

8. Technical Highlights

- **Unified Stencil Function:**
  Both OpenMP and MPI implementations share the same **stencil()** function, enabling code reusability and consistency in results across both parallel strategies.
- **Boundary Handling Strategy:**
  The stencil implementation avoids convolution at the image edges by **preserving original values** at the borders when the filter cannot fully overlap, ensuring correctness in output.
- **Efficient Use of Collapse Clause:**
  The use of **#pragma omp parallel for collapse(3)** improves load balancing across

threads by flattening the nested loops over batch, rows, and columns.

- **Correct MPI Slicing & Data Gathering:**
  The MPI implementation avoids incorrect global computation by correctly slicing the input data using **MPI_Scatter**, performing local stencil operations, and collecting the output using **MPI_Gather**.
- **Memory-Safe Allocation:**
  All input, filter, and output buffers are dynamically allocated using **malloc**, with process-local data in MPI ensuring **no memory overflows** even for large datasets.
- **Weak Scaling Validation:**
  The runtime increase across 1 to 32 MPI processes remains small (~0.31 seconds increase), indicating correct **weak scaling behavior** and effective parallel workload division.