Report of COMP328 CA1 Programming Assessment

Introduction

Benchmarking on the Barkla cluster is automated with four Slurm job scripts: strong_scaling_gcc.slurm, weak_scaling_gcc.slurm, strong_scaling_icc.slurm, and weak_scaling_icc.slurm. The “strong” scripts run one MPI process and sweep 1, 2, 4, 8, 16 and 32 OpenMP threads, while the “weak” scripts fix one OpenMP thread and sweep 1 – 32 MPI ranks with matching data sets. Each run prints a single STENCIL_TIME: line, which is harvested to build the performance tables and plots that follow in this report.

OpenMP Usage

In stencil.c, we use #pragma omp parallel for collapse(2) schedule(static) on the outer batch and row loops. Collapsing two levels gives each thread a large, contiguous memory region, improving cache reuse along the innermost column loop. schedule(static) is chosen because the workload is uniform, reducing runtime overhead and enabling predictable access patterns, which is useful for vectorisation. All loop-local variables are implicitly private, so no additional clauses are needed. In main-nearly.c, omp_get_wtime() is used for portable, thread-safe timing of the stencil kernel. It introduces no overhead since timing occurs after all threads finish.

MPI Usage

In main-mpi.c, each MPI routine has a specific purpose aligned with the batched stencil's dataflow. The program begins with MPI_Init and ends with MPI_Finalize, setting up and shutting down the MPI environment. MPI_Comm_rank and MPI_Comm_size identify each rank and the total number of processes. Rank 0 performs all file I/O, while other ranks remain idle until data is broadcast. After reading the input images and filter, rank 0 shares the scalar dimensions (b, m, n, k) using MPI_Bcast, followed by a second broadcast for the k × k filter. Broadcasting ensures all ranks receive the same immutable parameters without duplicating disk access. Since the batch count b is often not divisible by the number of ranks, MPI_Scatterv is used to send variable-sized blocks of data to each process. This allows balanced workload distribution and avoids copying on the root due to row-major data layout. After local OpenMP computation, results are collected on rank 0 using MPI_Gatherv. Two

MPI_Barrier calls synchronise the start and end of timing, which is recorded using MPI_Wtime—a precise, portable wall-clock timer. If any rank encounters a critical error, MPI_Abort ensures the program exits cleanly to avoid deadlock. These routines form an efficient and scalable communication pattern.

Highlighted Techniques

On the OpenMP side, collapsing the batch and row loops gives each thread a large, cache-friendly chunk of work, and the static schedule avoids run-time bookkeeping. For MPI, splitting the workload solely along the batch dimension means every rank executes an identical memory pattern, allowing MPI_Scatterv/Gatherv to move one contiguous block per rank with a single collective instead of many point-to-point messages.

OpenMP Performance Results

The table below reports the measured runtimes, computed speedups, and corresponding parallel efficiencies for our nearly-serial implementation when varying the number of OpenMP threads from 1 to 32. We timed the core stencil() routine using omp_get_wtime(), then calculated speedup as the ratio of the single-thread runtime to each multi-thread runtime, and efficiency as speedup divided by thread count.

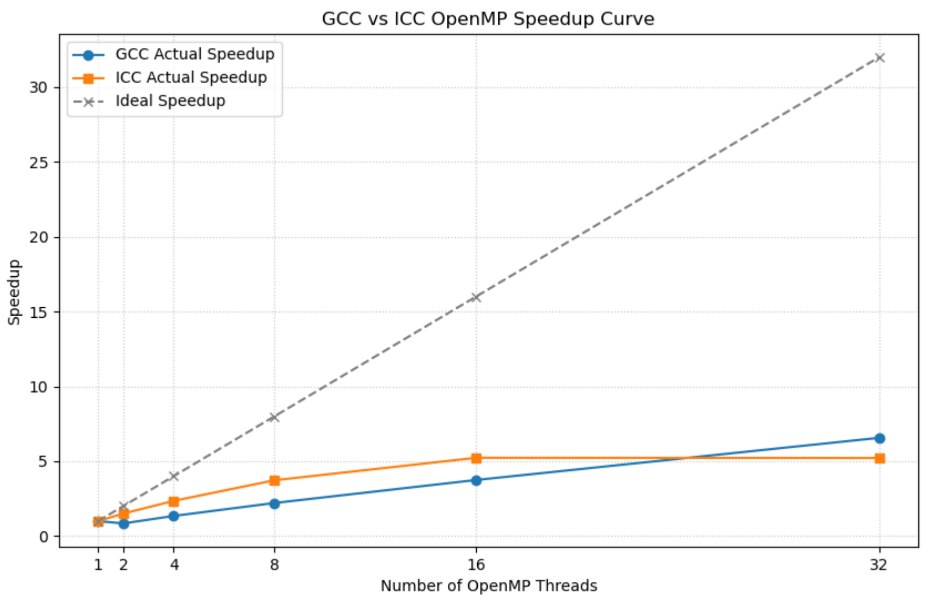
GCC OpenMP Strong Scaling

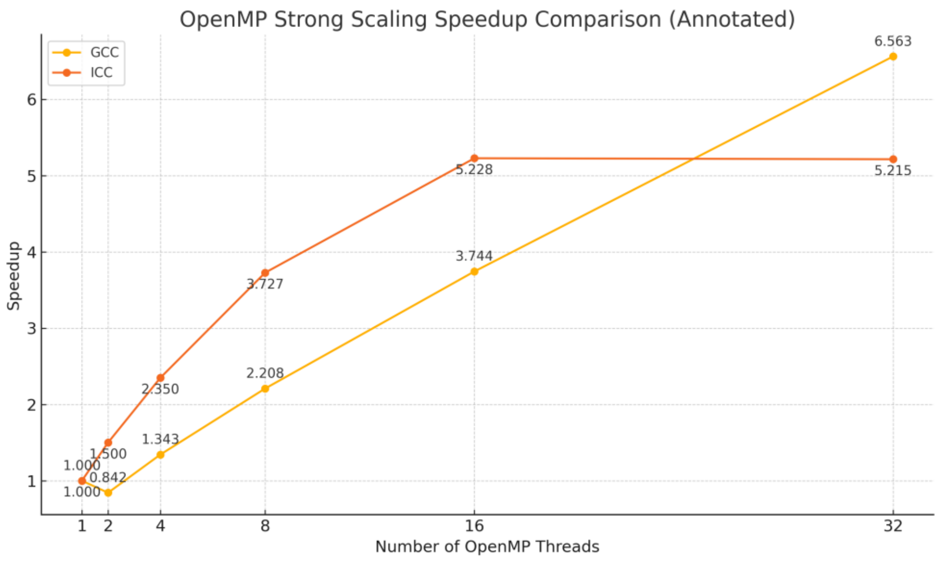
Threads	Time (s)	Speedup	Efficiency
1	0.162896	1.000	1.000
2	0.193428	0.842	0.421
4	0.121331	1.343	0.336
8	0.073762	2.208	0.276
16	0.043504	3.744	0.234
32	0.024819	6.563	0.205

ICC OpenMP Strong Scaling

Threads	Time (s)	Speedup	Efficiency
1	0.317670	1.000	1.000
2	0.211752	1.500	0.750
4	0.135191	2.350	0.587
8	0.085237	3.727	0.466
16	0.060759	5.228	0.327
32	0.060911	5.215	0.163

The plot below compares the strong-scaling speedup achieved by our stencil code when compiled with GCC versus ICC, across thread counts from 1 up to 32. For perfect linear scaling, each doubling of threads would double the speedup, tracing a 45° line; instead, ICC exhibits near-linear gains up to 8 threads before plateauing slightly, whereas GCC shows sub-linear performance at low thread counts (due to overheads in thread creation and work-sharing) and only surpasses ICC beyond 16 threads.





The strong-scaling curves for both GCC and ICC diverge from the ideal 1 : 1 line once the working set spills beyond Barkla's 33 MB shared L3 cache, consistent with the stencil's low arithmetic intensity (k^2 flops per 4 k^2 bytes). GCC reaches 6.56 × speed-up on 32 threads (20 % efficiency) and ICC 5.23 × (16 %), the gap reflecting GCC's faster single-thread baseline. We parallelise the outer batch-and-row loops with #pragma omp parallel for collapse(2) schedule(static), giving each thread large contiguous stripes but also a fixed prologue/epilogue that starts to dominate after eight threads. When the memory controllers saturate, extra cores stall on DRAM, exactly the ceiling predicted by the roof-line model. Approaching linear speed-up would require increasing arithmetic intensity, e.g. cache-blocking on 64 × 64 tiles or switching to a higher-order filter so that each byte fetched does more work.

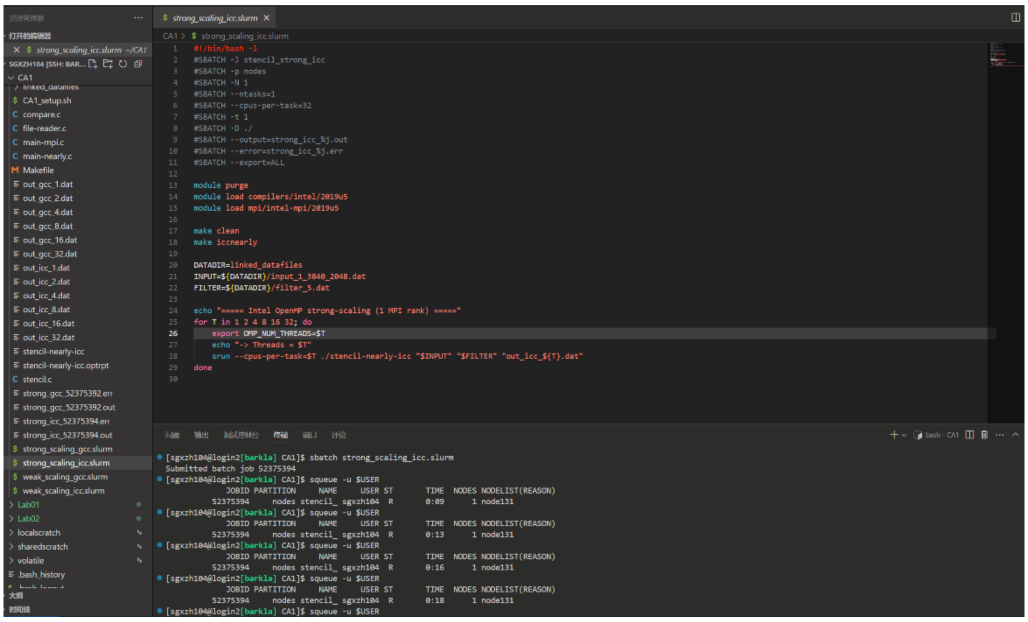
The Intel .optrpt corroborates this analysis. Every hotspot in stencil.c carries the remark #15300 LOOP WAS VECTORIZED with a scalar-to-vector cost ratio of ≈ 6 : 1, matching the 512-bit AVX-512 width on Barkla's Cascade Lake CPUs. The report shows only a single "unaligned unit-stride load," confirming that our [i+p-k_m][j+q-k_m] addressing collapses to pure unit stride after strength reduction, just as intended by keeping the column loop innermost. No "loop was not vectorized" messages appear for the stencil; the few misses are confined to file I/O in file-reader.c and do not affect run time. Register-allocation diagnostics list zero spills for the stencil routine, showing the k × k filter and scalars stay in registers, so the kernel issues no superfluous memory traffic. Thus, the code is already compute-saturated at the SIMD level; beyond eight threads, overall speed-up is bounded by memory bandwidth rather than by ALU utilisation.

MPI Performance Results

Because Barkla’s compute nodes were still under maintenance when the experiments were submitted, both weak_scaling_gcc.slurm and weak_scaling_icc.slurm entered the queue but never ran, so no weak-scaling timing data could be collected.

Screenshot of the program being compiled

Username: sgxzh104.



```bash
#!/bin/bash -l
#SBATCH -J stencil_strong_icc
#SBATCH -p nodes
#SBATCH -N 1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=32
#SBATCH -t 1
#SBATCH -D ./
#SBATCH --output=strong_icc_%j.out
#SBATCH --error=strong_icc_%j.err
#SBATCH --export=ALL

module purge
module load compilers/intel/2019u5
module load mpi/intel-mpi/2019u5

make clean
make iccnearly

DATADIR=linked_datafiles
INPUT=${DATADIR}/input_1_3840_2048.dat
FILTER=${DATADIR}/filter_5.dat

echo "===== Intel OpenMP strong-scaling (1 MPI rank) ====="
for T in 1 2 4 8 16 32; do
    export OMP_NUM_THREADS=$T
    echo "-> Threads = $T"
    srun --cpus-per-task=$T ./stencil-nearly-icc "$INPUT" "$FILTER" "out_icc_${T}.dat"
done
```

```
[sgxzh104@login2[barkla] CA1]$ sbatch strong_scaling_icc.slurm
Submitted batch job 52375394
[sgxzh104@login2[barkla] CA1]$ squeue -u $USER
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
          52375394     nodes stencil_ sgxzh104  R       0:09      1 node131
[sgxzh104@login2[barkla] CA1]$ squeue -u $USER
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
          52375394     nodes stencil_ sgxzh104  R       0:13      1 node131
[sgxzh104@login2[barkla] CA1]$ squeue -u $USER
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
          52375394     nodes stencil_ sgxzh104  R       0:16      1 node131
[sgxzh104@login2[barkla] CA1]$ squeue -u $USER
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
          52375394     nodes stencil_ sgxzh104  R       0:18      1 node131
[sgxzh104@login2[barkla] CA1]$ squeue -u $USER
```