Performance Engineering & Benchmarking on the SCC cluster

Today we answer one question: **"How do I know if my job is using the cluster well?"** Three short exercises build on each other. Each one starts with a goal, then the steps, then the questions you must answer in your worksheet (`code/worksheet.md`).

Contents

Prerequisites

- SSH access to the GWDG SCC cluster (`login-mdc.hpc.gwdg.de`).
- Modules: `gcc/14.2.0` and `openmpi/5.0.6`.
- The `code/` directory from this exercise (copy to `$HOME/pchpc-perf/code`).
- Familiarity with `srun`, `sbatch`, and basic MPI – covered earlier this week.

The eight rules for every measurement today

1. Run on a compute node, never the login node.
2. Warm up; throw the first iteration away.
3. Repeat at least 5 times; report best or median.
4. Pin threads/processes to cores.
5. Use the result, or the compiler will delete the loop.
6. Use a real timer (`clock_gettime` / `MPI_Wtime`).
7. Compile with `-O2` or `-O3`.
8. Compare every number to a theoretical ceiling.

Task 1: Exercise 1 – Know your node (20 min)

Goal For the SCC compute node you are allocated, find the hardware spec (cores, NUMA, RAM, ISA), then *compute* its theoretical peak floating-point rate and peak memory bandwidth.

Steps

1. Log in and load the modules: `$ ssh u<id>@login-mdc.hpc.gwdg.de`
   `$ module load gcc/14.2.0 openmpi/5.0.6`
2. Get an interactive shell on a compute node: `$ srun -p medium -N 1 -n 1 --time=00:10:00 --pty bash`
3. Inspect the silicon and the memory layout: `$ lscpu`
   `$ numactl -H`
   `$ free -h`
4. Fill in the table in `worksheet.md`, section "Exercise 1" (CPU model, sockets, physical cores per node, base clock, NUMA domains, RAM per node).
5. Compute the theoretical peak DP FLOP/s per core, then per node:

   $$\text{peak/core} = \text{clock} \times \text{FMA units} \times \text{SIMD width} \times 2$$

   For Intel Cascade Lake with AVX-512: FMA units = 2, SIMD width = 8.
6. Compute the theoretical peak memory bandwidth per node from the CPU spec sheet: channels $\times$ sockets $\times$ 8 B $\times$ MT/s.

Reference (SCC `medium`, Xeon Platinum 9242)

- Per core: $2.30 \times 2 \times 8 \times 2 = 73.6$ GFLOP/s.
- Per node: $73.6 \times 96 \approx 7.07$ TFLOP/s.
- Memory: $12 \times 2 \times 8 \times 2933 \approx 563$ GB/s.

Questions to answer in your worksheet

- What is the peak DP FLOP/s of your node?
- What is the peak memory bandwidth of your node?
- How many NUMA domains does it have? Why does that matter for memory-bound code?

Task 2: Exercise 2 – Benchmarking basics: timing, DCE, and STREAM (20 min)

Goal Practise correct measurement (Rules 2–7), then run the STREAM-lite microbenchmark to find out how close your node gets to its peak memory bandwidth.

Step 1 – Build everything `$ cd $HOME/pchpc-perf/code`
`$ module load gcc/14.2.0 openmpi/5.0.6`
`$ make`

Step 2 – The trivial loop and optimisation Run the same source code at three optimisation levels and record the wall time: `$ ./trivial_O0 100000000`
`$ ./trivial_O2 100000000`
`$ ./trivial_O3 100000000`

- Why is `-O0` so much slower than `-O2`?
- If you remove the `printf` of `sum` from `trivial.c` and rebuild with `-O3`, the wall time becomes near zero. *Why?*

Step 3 – STREAM-lite (memory bandwidth) Submit the batch job that runs STREAM-lite with 1 thread and with all 96 threads: `$ sbatch stream.sbatch`

`$ squeue --me`
Once the job has finished, read the output: `$ cat stream_<jobid>.out`

Record the TRIAD bandwidth in your worksheet:

- TRIAD with 1 thread (GB/s).
- TRIAD with 96 threads, pinned, NUMA-correct (GB/s).
- % of theoretical peak BW from Exercise 1.

Reference (measured on `amp095`, full node)

- TRIAD 1 thread: 12.81 GB/s.
- TRIAD 96 threads: 347.50 GB/s ($\approx 62\%$ of peak).
- Achieving 60–70% of peak BW with STREAM is normal – the rest is unrecoverable real-world overhead (refresh, row activates, prefetcher waste).

Questions to answer

- Why is the achieved bandwidth always less than theoretical peak? Give two reasons.
- If your application's hot loop runs at 0.4× measured TRIAD, is it memory-bound or compute-bound?

Task 3: Exercise 3 – Strong and weak scaling of MPI pi (20 min)

Goal Run a real parallel MPI program (numerical integration of π), measure its wall time on $1, 2, 4, 8, 16, 32, 64, 96$ ranks on a single node, plot the speedup curve, and *interpret* it. Strong scaling will show Amdahl's law in real numbers; weak scaling will show Gustafson's.

Step 1 – Submit the scaling sweep `$ sbatch scaling.sbatch`
`$ squeue --me`

The script runs both a strong-scaling and a weak-scaling sweep and writes the results to `results.csv`.

Step 2 – Plot `$ cat results.csv`
`$ python3 plot_scaling.py results.csv`
`$ ls scaling.png`

Step 3 – Fill in the tables For each rank count, compute speedup $S(P) = T_1/T_P$ and parallel efficiency $E(P) = S(P)/P$. Do this for both the strong and weak sweeps. Tables are in `worksheet.md` section "Exercise 3".

Reference (measured on `amp095`, $N = 4 \times 10^8$) **Strong scaling**: $T_1 = 0.606$ s, $T_{96} = 0.0122$ s, speedup $\approx 49.7\times$, efficiency $\approx 52\%$.
Weak scaling: $T_1 \approx T_{96} \approx 0.012$ s, weak efficiency $\approx 92\%$.

Discussion questions

- At what rank count does strong-scaling efficiency start to drop noticeably? Why?
- Why is weak-scaling efficiency typically closer to 1.0 than strong-scaling efficiency at the same rank count?
- Is 52% efficiency at 96 cores "good"? Justify your answer.
- Use Amdahl's formula to estimate the serial fraction s that would explain your measured $S(96)$.
- If you wanted to know *which line* of code is causing the inefficiency, which kind of tool would you reach

for next? *(This is what the afternoon session covers.)*

Hints

- **Order matters.** Do Exercise 1 before 2 before 3 – each one uses results from the previous.
- **Always work on a compute node.** Submit a batch job, or use `srun --pty bash`. Never measure anything on the login node.
- **If your batch job is queued for a long time**, the `medium` partition is busy. Try `medium96s:test` or `standard96:test` (1-hour limit, often empty). The CPU model is the same Intel Xeon Platinum class – the formulas in Exercise 1 still apply.
- **Use the cheatsheet** (`code/SCC-cheatsheet.md`) when you forget a command.
- **Sample outputs** are in `code/sample_outputs/` so you can compare your numbers against ours.

Optional Task 4: Optional bonus – IO benchmarking with IOR & mdtest (0 min)

This is a difficult **additional** task which will strengthen your understanding in the topic.

If you finish early, you can explore I/O benchmarking with the IO500 suite. This is bonus material – not assessed and not required.

Goal Run a small-scale **IOR** (sequential I/O bandwidth) and **mdtest** (metadata operations) benchmark on the SCC scratch filesystem and interpret the results.

- **IOR** measures sequential read/write bandwidth of the file system.
- **mdtest** measures metadata operations: file creation, stat, deletion.

Quick steps

1. Build a working directory under your scratch space: `$ mkdir -p $WORK/io500-small && cd $WORK/io500-smal`
2. Clone and build IO500: `$ git clone https://github.com/IO500/io500.git`
   `$ cd io500 && make`
3. Run a small IOR test (block size 64 MB, transfer size 1 MB, 4 ranks): `$ mpirun -np 4 ./ior -t 1m -b 64m -o ./scratch/ior-out`
4. Run a small mdtest (1000 files, 4 ranks): `$ mpirun -np 4 ./mdtest -d ./scratch/mdtest -n 1000 -u -L`
5. Record write/read bandwidth (IOR) and create/stat/delete rates (mdtest).

Discussion

- Why is read bandwidth often different from write bandwidth on a parallel file system?
- How do metadata operations limit performance for workloads with many small files?

Further reading

- Williams, Waterman, Patterson, "Roofline: An Insightful Visual Performance Model", CACM 2009.
- Hager & Wellein, *Introduction to High Performance Computing for Scientists and Engineers*, CRC Press 2010 – the standard textbook on the topics covered today.

- McCalpin, STREAM benchmark: `https://www.cs.virginia.edu/stream/`
- IOR Benchmark User Guide: `https://media.readthedocs.org/pdf/ior/latest/ior.pdf`
- IO500: `https://io500.org/`