Aasish Kumar Sharma

Performance Engineering & Benchmarking

2026-04-09

Where We Are – Recap Of Part I

What we learned in the last hour

- Every cluster has two ceilings: peak FLOP/s and peak memory BW
- For SCC `medium`: $\sim$7.07 TFLOP/s and $\sim$563 GB/s per node
- Most real HPC code is memory-bound
- Amdahl bounds strong scaling; Gustafson describes weak scaling
- $S(P) = T_1/T_P$, $E(P) = S(P)/P$

The natural next question

How close to those ceilings does real code actually get – and how do we measure it without lying to ourselves?

Part II – Benchmarking

How do we measure correctly?

- The eight rules of correct benchmarking
- Three live demos (`trivial`, `dce_demo`, `stream_lite`)
- Exercise 2: STREAM on SCC

What Is A Benchmark?

Definition

- A benchmark is a controlled measurement of performance,
- Designed to be reproducible and interpretable.
  - It is not “running my code once and writing down the time”
  - It is a measurement with rules, like a chemistry experiment
  - Done badly, it lies with confidence
  - Done correctly, it is the most powerful tool you have

Three Types Of Benchmarks

1. Microbenchmark

- Tiny isolated operation – memory copy, FMA, MPI call
  - Examples: **STREAM**, **lmbench**, **OSU MPI benchmarks**

2. Kernel benchmark

- One important inner loop of an application class
  - Examples: **HPL** (Top500), **HPCG** (sparse linear algebra)

3. Application benchmark

- Real, full application on a representative input
  - Examples: **SPEC HPC**, **NPB**, your own production code

→ Today: microbenchmarks. They isolate one thing and tell you one truth.

The Eight Rules Of Correct Benchmarking

1. Run on a compute node, never the login node
2. Warm up – discard the first iteration
3. Repeat – report best or median, never one run
4. Pin threads / processes to cores
5. Use the result, or the compiler will delete your loop
6. Use a real timer (`clock_gettime`, `MPI_Wtime`)
7. Compile with `-O2` or `-O3`
8. Compare to a ceiling (peak FLOP/s or peak BW)

→ The next eight slides explain why each rule exists.

Rule 1 – Choose Correct Node (Actual System)

Compute Node, Not Login Node

- Login node is shared by hundreds of users
- Their compilations and edits add unpredictable noise
- Numbers depend on who else is logged in, not on your code
- Many cluster admins kill long jobs on login nodes

Always start with

→ `srun -p medium -N 1 -n 1 -time=00:10:00 -pty bash`

Rule 2 – Prepare

Warm Up

- First iteration is always slower
- Cold caches – data must be loaded from RAM
- Lazy memory allocation – pages committed on first touch
- MPI sets up connections lazily on first collective
- CPU may still be in low-power frequency state

→ Without warmup, your first number can be 2–10× too slow.

Rule 3 – Calibrate

Repeat And Take A Statistic

- Even on a quiet node, wall time fluctuates 2–10%
- Sources – OS interrupts, frequency drift, cache state, neighbours
- One run is a coin flip; five runs is a measurement
- Report best (STREAM convention) or median (more honest)
- Never report a single run

Rule 4 – Trace The Flow

Pin Threads And Processes To Cores

- OS scheduler can move threads between cores
- Each move loses cache state and adds NUMA-remote traffic
- On a 96-core 4-NUMA node, this collapses bandwidth 3–4×

OpenMP:

```
OMP_PROC_BIND=close OMP_PLACES=cores ./my_app
```

MPI (OpenMPI):

```
mpirun --bind-to core --map-by core -np 16 ./my_app
```

Rule 5 – Measure As You Go

Use The Result, or The Compiler Deletes Your Code

```
double sum = 0.0;
for (long i = 0; i < n; ++i) sum += a[i];
/* sum is never used -> at -O2, the loop is deleted */
```

- Compiler proves `sum` is unused
- It removes the entire loop
- Wall time goes to 0; you publish; you are wrong

The fix: use the result.

```
    printf("%f\n", sum);   /* observable side effect = loop survives */
```

Rule 6 – Use A Real Timer

Wall Clock Time

- **Bad** – 1-second resolution, jumps with NTP:

```
time_t t0 = time(NULL);
```

- **Good** – nanosecond resolution, monotonic, portable POSIX:

```
struct timespec ts;
clock_gettime(CLOCK_MONOTONIC, &ts);
double now = ts.tv_sec + ts.tv_nsec * 1e-9;
```

For MPI codes, `MPI_Wtime()` works the same way.

Rule 7 – Compile With Optimisation

Use Correct Optimization Compiler Flag

- `-O0` keeps everything in memory
- No vectorisation, no inlining
- You measure the compiler, not the algorithm
- Use `-O2` as default; try `-O3` for floating point
- Always state the flags – numbers without flags are useless

Rule 8 – Compare To A Ceiling

Know The Peak Performance

- “350 GB/s” alone tells you nothing
- “350 GB/s out of 563 GB/s peak = 62%” is a result
- The ceiling came from Part I
- This ratio tells you whether to optimise – and what

The performance budget question

- Memory-bound at 65% of peak BW – very hard to improve, near the wall
- Memory-bound at 10% of peak BW – lots of room, check access patterns

Demo plan

Try Provided Code

1. **Optimisation matters** – the same source at -O0, -O2, -O3
2. **Dead-code elimination is real** – two compiles, one prints, one does not
3. **STREAM** – 1 thread vs all cores, % of peak

All demo code in code/

- `trivial.c`
- `dce_demo.c`
- `stream_lite.c`
- `Makefile`

→ Build with one command: `cd ~/pchpc-perf/code && make`

Demo 1 – Optimization Level Changes Everything

- **One source, three compiles – Makefile builds all of them with one `make`:**

```
cd ~/pchpc-perf/code
make   # builds trivial_O0, trivial_O2, trivial_O3, dce_*, stream_lite

# Note: the suffix is capital "O" + digit (O0 = "Oh-zero" = no optimisation)
```

- **Measured (laptop, Intel i5-1145G7, Tiger Lake, 4C/8T):**

```
./trivial_O0 100000000     # ~0.222 s
./trivial_O2 100000000     # ~0.100 s   (~2.2x faster)
./trivial_O3 100000000     # ~0.100 s
```

→ On a fast Xeon the gap is 10–20×. On the laptop it is 2× – the loop is memory-bound.

Demo 2 – Dead-Code Elimination, The Silent Trap

- **Two builds of the SAME source – one uses `sum`, one does not:**

```
# Makefile recipes:
# dce_use: gcc -O3 -DUSE_RESULT -o dce_use dce_demo.c   # printf(sum)
# dce_dce: gcc -O3              -o dce_dce dce_demo.c   # no printf
```

- **Measured (laptop, n = 100M doubles):**

```
./dce_use 100000000     # ~0.100 s    <- real measurement
./dce_dce 100000000     # 0.000000 s  <- compiler deleted the loop
```

→ The compiler proved `sum` is never observed and removed the loop.

Demo 3 – STREAM-Lite On The Laptop

- **Theoretical peak BW (dual-channel DDR4-3200): ∼51.2 GB/s**

```
OMP_NUM_THREADS=1 OMP_PROC_BIND=close OMP_PLACES=cores ./stream_lite
# TRIAD: ~24.0 GB/s   ( 47% of peak)

OMP_NUM_THREADS=4 OMP_PROC_BIND=close OMP_PLACES=cores ./stream_lite
# TRIAD: ~39.2 GB/s   ( 77% of peak)

OMP_NUM_THREADS=8 OMP_PROC_BIND=close OMP_PLACES=cores ./stream_lite
# TRIAD: ~36.3 GB/s   ( 71% of peak)  (!) slower than 4 threads
```

Wait – 8 Threads Slower Than 4 Threads. Why?

Knowing Your Edge Device Before The HPC System

- Laptop has 4 physical cores with 2 hyperthreads each = 8 logical CPUs
- STREAM is memory-bound – every iteration moves bytes
- Memory bandwidth comes from 2 DDR4 channels, not from cores
- Hyperthreads share channels and cache – they fight for bandwidth
- Result: more contention, lower measured bandwidth

The lesson nobody tells you in MPI tutorials

- More threads is not always faster.
  - On memory-bound code, optimal thread count is the number of *physical cores*.

Demo 3b – STREAM On The SCC "medium" Node (Real Cluster)

- **Theoretical peak BW:**
  - 12 channels × 2 sockets × DDR4-2933 ≈ 563 GB/s
- **Measured on amp095 (full node, 96 cores):**

```
OMP_NUM_THREADS=1  ./stream_lite     # TRIAD:  12.81 GB/s   ( 2.3% of peak)
OMP_NUM_THREADS=96 ./stream_lite     # TRIAD: 347.50 GB/s   (61.7% of peak)
# COPY 335   SCALE 313   ADD 351   TRIAD 347   GB/s
```

27× more bandwidth than the laptop

- The Xeon node has 24 memory channels vs the laptop's 2.
- Performance lives in the hardware, not the code.

Memory-Bound Vs Compute-Bound – The Practical Test

If you do not know what bounds your code

1. Run STREAM and get the measured peak BW
2. Estimate bytes per second of your hot loop
3. If $> 50\%$ of measured TRIAD $\Rightarrow$ memory-bound
4. If $> 50\%$ of peak FLOP/s $\Rightarrow$ compute-bound
5. If $< 10\%$ of both $\Rightarrow$ overhead, latency, or sync issue

$\rightarrow$ Most real HPC code is memory-bound – and most people optimise the wrong thing.

An Honest Aside

Queue Reality On A Shared Cluster

- `-exclusive` on a busy partition can mean unbounded waiting
- Slurm gives you 96 cores only when 96 are free on the same node

Practical strategy

1. For interactive demos – use a `:test` partition (1-hour limit)
2. For long exclusive runs – submit the night before, save the output
3. Always have a local fallback – your laptop is a real measurement

Exercise 2 – STREAM On Your Node (10 minutes)

Goal:

- Reproduce the STREAM measurement on an SCC `medium` node and compute % of peak.

```
cd ~/pchpc-perf/code
make stream_lite
sbatch stream.sbatch
squeue --me
cat stream_<jobid>.out
```

- **Fill in the worksheet:**
  - TRIAD GB/s with 1 thread
  - TRIAD GB/s with all 96 threads, pinned, NUMA-correct
  - % of theoretical peak BW from Exercise 1

Takeaway Of Part II

What you now know

- How to write a benchmark that does not lie (the 8 rules)
- How to measure your node’s real memory bandwidth with STREAM
- How to know if your code is memory-bound or compute-bound
- Real numbers from SCC – ∼347 GB/s = 62% of 563 GB/s peak

10-minute break – back at 11:05.