HWRS640 – Assignment 1: Computer Architecture and Parallel Computing

Nabin Kalauni

February 6, 2026

1 Problem 1: Supercomputer Architecture (25 pts)

1.1 Name and Location

The Frontier supercomputer (also designated OLCF-5) is located at the Oak Ridge Leadership Computing Facility (OLCF) at Oak Ridge National Laboratory (ORNL) in Oak Ridge, Tennessee, USA. It is operated by the U.S. Department of Energy's Office of Science and managed by UT-Battelle. As of the November 2025 TOP500 list, Frontier is ranked **#2** in the world, behind El Capitan at Lawrence Livermore National Laboratory.

1.2 Architecture

Frontier is built on the **HPE Cray EX235a** platform. Table 1 summarizes its key specifications.

Table 1: Frontier supercomputer specifications.

Component	**Details**
Manufacturer	Hewlett Packard Enterprise (HPE)
CPU	AMD Optimized 3rd Gen EPYC 7713 "Trento" 64-core, 2 GHz (9,472 CPUs × 64 cores = 606,208 CPU cores)
GPU / Accelerator	AMD Instinct MI250X (37,888 GPUs × 220 CUs = 8,335,360 GPU stream processors)
Total Cores	9,066,176 (combined CPU + GPU)
Interconnect	HPE Slingshot-11
Operating System	HPE Cray OS
Installation Year	2021 (open for users in 2022)
Power Consumption	24,607 kW

Each Frontier node consists of one AMD EPYC CPU and four AMD MI250X GPUs, connected via Infinity Fabric. The Slingshot-11 interconnect provides high-bandwidth, low-latency communication across the full system using a dragonfly topology.

1.3 Peak Performance

- **LINPACK (Rmax):** 1,353 PFlop/s (1.353 EFlop/s). This is the performance achieved on the LINPACK benchmark, which measures the system's ability to solve dense linear equations.

- **Theoretical Peak (Rpeak):** 2,055.72 PFlop/s. This is the maximum theoretical performance based on the hardware specifications (number of cores, clock speed, and FLOPs per cycle).
- **HPCG:** 14,054 TFlop/s (#3 on the HPCG ranking). HPCG (High Performance Conjugate Gradients) is a benchmark that measures performance on a more realistic workload involving sparse linear algebra, which is more representative of many scientific applications.

1.4 Applications and Research

Frontier supports a broad portfolio of scientific and engineering applications through the OLCF's Frontier Center for Accelerated Application Readiness (CAAR) program. Key research areas include:

- **Astrophysics:** Galaxy-scale simulations using the Cholla code, modeling Milky Way-like systems with detailed hydrodynamics.
- **Genomics and Biology:** Large-scale genome-wide epistasis studies (GWES) using CoMet for applications in bioenergy, clinical genomics, and disease research (e.g., Alzheimer's, opioid addiction).
- **Fluid Dynamics:** Direct numerical simulation of turbulence at unprecedented resolution ($\sim$35 trillion grid points) using the GESTS code.
- **Molecular Dynamics:** Virus entry mechanism studies (e.g., Zika virus) using NAMD, enabling large-scale biomolecular simulations.
- **Nuclear Physics:** Coupled-cluster calculations for nuclear structure and reactions using NuCCOR.
- **Materials Science and Condensed Matter:** First-principles alloy and magnetic system simulations using LSMS.
- **Plasma Physics:** Particle-in-cell simulations for advanced plasma accelerators using PIConGPU.
- **Nuclear Energy:** Modeling the entire lifespan of nuclear reactors, and integration of AI with traditional HPC modeling and simulation.

2 Problem 2: Moore's Law and Linear Regression (25 pts)

2.1 Data Loading

The dataset was loaded from `moores.csv`. The dataset contained **508 data points** spanning **1953–2023**.

2.2 Linear Regression Model

Since Moore's Law predicts exponential growth, we model $\log_2$(transistor count) as a linear function of year:

$$\log_2(N) = m \cdot \text{year} + b \tag{1}$$

Applying ordinary least-squares linear regression via `scipy.stats.linregress`:

$$m = 0.4472 \text{ (doublings per year)} \tag{2}$$

$$b = -868.94 \tag{3}$$

$$R^2 = 0.854 \tag{4}$$

2.3 Visualization

Figure 1 shows the original data (semi-log scale) with the fitted regression line.

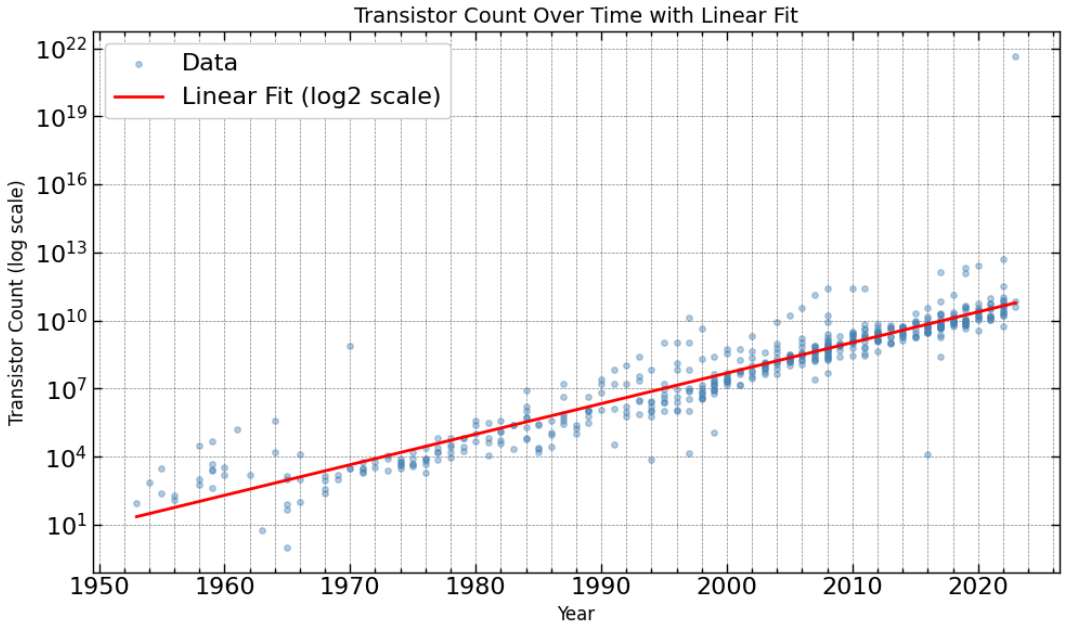

Figure 1: Transistor count versus year with exponential fit. The regression line corresponds to a doubling time of 2.24 years.

2.4 Doubling Time

The doubling time is the reciprocal of the slope:

$$T_{\text{double}} = \frac{1}{m} = \frac{1}{0.4472} \approx \mathbf{2.24} \quad \textbf{years} \tag{5}$$

This is close to but slightly longer than the commonly cited value of **2 years**.

2.5 Early vs. Late Era Comparison (for fun)

We repeated the regression for the first 10 years (1953–1963, $n = 19$) and last 10 years (2013–2023, $n = 143$) of the dataset.

Table 2: Regression results for early and late eras.

Era	Slope	Doubling Time (yr)	R^2
Full dataset (1953–2023)	0.4472	2.24	0.854
First 10 years (1953–1963)	0.2499	4.00	0.038
Last 10 years (2013–2023)	0.6957	1.44	0.202

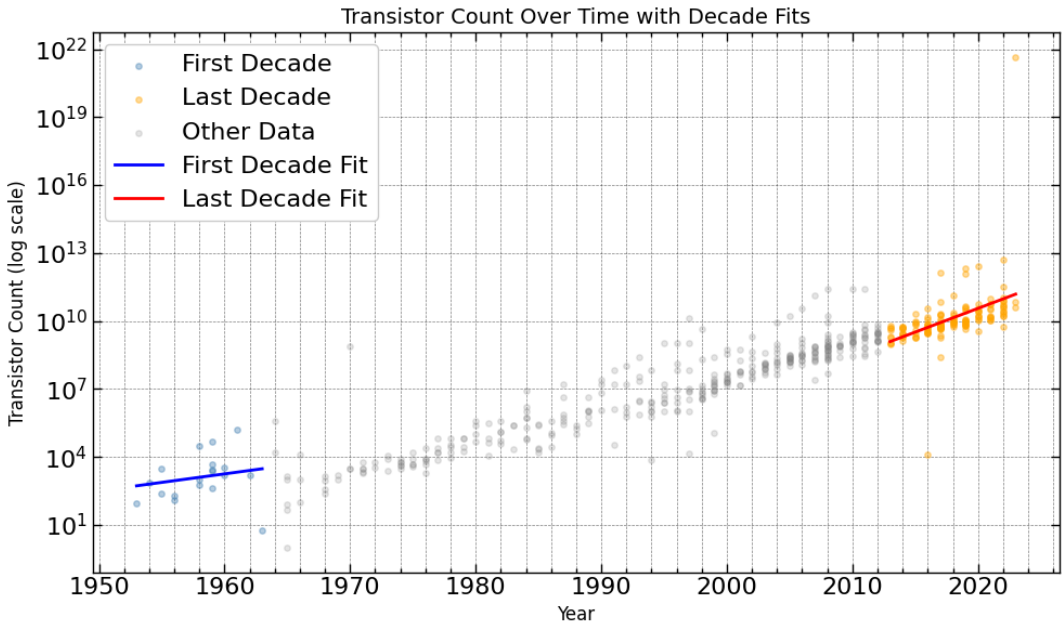

Figure 2: Comparison of transistor growth in the early and late eras.

The early era (1953–1963) shows a much slower doubling time of ∼4 years with very low R^2, since it was early days of computing hardware. The late era (2013–2023) shows a faster apparent doubling time of ∼1.44 years. The overall trend suggests that Moore's Law has, if anything, *accelerated* in terms of raw transistor counts.

3 Problem 3: Row vs. Column Order Data Access (25 pts)

3.1 Array Creation

A 10,000 × 10,000 NumPy array was created, filled with random numbers drawn from a standard normal distribution (`np.random.standard_normal`).

3.2 Row-Major and Column-Major Summation

Two Python functions were implemented using explicit nested `for` loops:

- **Row-major:** outer loop over rows, inner loop over columns (accesses `arr[i,j]`).
- **Column-major:** outer loop over columns, inner loop over rows (accesses `arr[i,j]` with swapped loop order).

3.3 Performance Measurements

Each method was timed over 30 repetitions using `time.perf_counter()`.

Table 3: Execution times for array summation (10,000×10,000).

Method	Mean Time (s)	Std Dev (s)
Row-major (Python loop)	7.2538	0.2287
Column-major (Python loop)	7.4301	0.2888
NumPy `np.sum()`	0.0148	0.0053

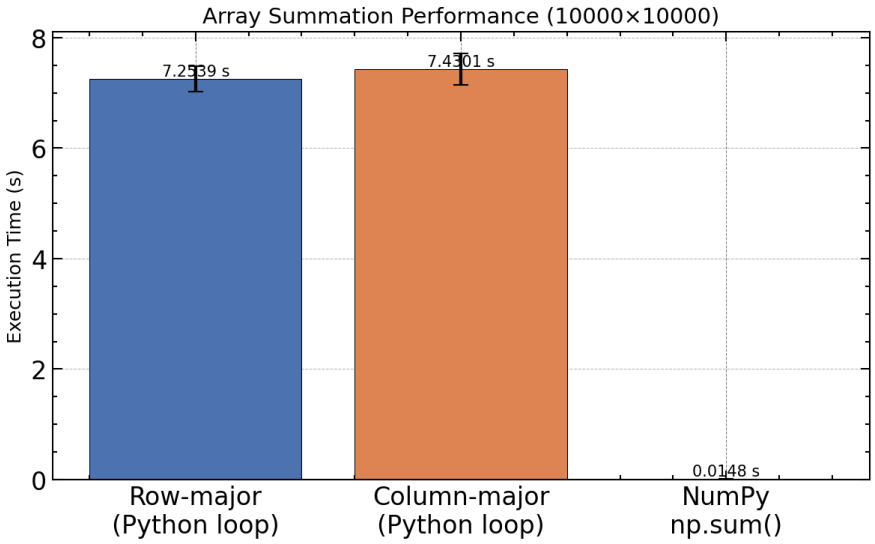

Figure 3: Comparison of summation methods.

3.4 Discussion

The performance differences arise from two main factors:

Cache locality and memory access patterns. NumPy arrays use row-major (C-order) memory layout by default. In row-major order, elements within the same row are stored contiguously in memory. When the row-major function iterates over columns in the inner loop, it accesses consecutive memory addresses, exploiting *spatial locality*: each cache line fetched from RAM contains multiple useful elements. Conversely, the column-major function accesses elements separated by N elements (an entire row) in memory, causing frequent *cache misses* as the CPU must fetch new cache lines for each element. This results in the column-major approach being slower than the row-major approach.

Python interpreter overhead. Both loop-based methods are dramatically slower than `np.sum()` because Python's interpreter incurs significant per-iteration overhead (type checking, reference counting, etc.) for each of the 10^8 iterations. NumPy's `np.sum()` delegates the summation to optimized C code that operates on contiguous memory blocks with vectorized SIMD instructions, eliminating Python overhead entirely.

4 Problem 4: Scaling and Parallel Computing (25 pts)

4.1 Z-Score Function

A function was implemented that generates a Dask array of shape (n, n) filled with random normal values and computes the element-wise z-score:

$$z = \frac{x - \mu}{\sigma} \tag{6}$$

where μ = `x.mean()` and σ = `x.std()` are computed lazily via Dask and materialized with `.compute()`. The number of Dask workers is configured via `dask.config.set(num_workers=n)`.

4.2 Strong Scaling

The array size was fixed at 20,000 × 20,000 and execution was timed using 1, 2, 3, and 4 cores.

Table 4: Strong scaling results (20,000×20,000 array).

Cores p	**Time** $T(p)$ **(s)**	**Speedup** $S(p)$	**Efficiency** $E(p)$
1	4.55	1.00	1.00
2	2.70	1.69	0.84
3	2.51	1.81	0.60
4	2.46	1.85	0.46

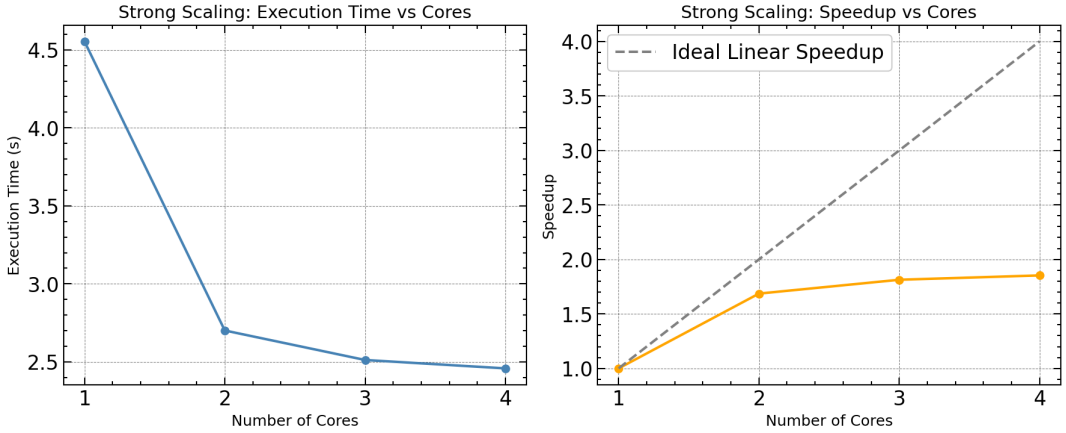

Figure 4: Strong scaling: execution time and speedup vs. number of cores.

4.3 Weak Scaling

For weak scaling, the work per core was held constant by scaling the array size proportionally with the number of cores. With a base size of 20,000 × 20,000 per core, the total array size for p cores is $(20{,}000\sqrt{p}) \times (20{,}000\sqrt{p})$.

Table 5: Weak scaling results.

Cores p	Array Size	Time (s)
1	$20{,}000 \times 20{,}000$	4.34
2	$28{,}284 \times 28{,}284$	11.81
3	$34{,}641 \times 34{,}641$	30.07
4	$40{,}000 \times 40{,}000$	64.55

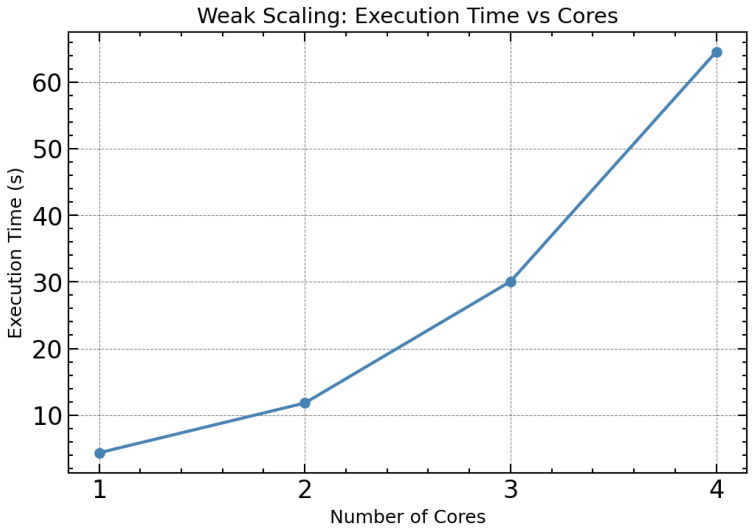

Figure 5: Weak scaling: execution time vs. number of cores.

4.4 Discussion

Several factors typically limit scalability in this setting:

1. **Amdahl's Law:** The z-score computation requires global reductions (mean and standard deviation) that introduce serial bottlenecks. These aggregation steps require inter-worker communication and cannot be perfectly parallelized.

2. **Task scheduling overhead:** Dask's task graph scheduler introduces overhead for creating, scheduling, and coordinating tasks across workers.

3. **Memory bandwidth:** The z-score computation is memory-bound (reading and writing large arrays with simple arithmetic). On shared-memory systems, cores compete for memory bandwidth, which limits the achievable speedup.

4. **Weak scaling and superlinear growth:** In the weak scaling experiment, the total array size grows with p (from $20{,}000 \times 20{,}000$ to $40{,}000 \times 40{,}000$). The $\sim$15$\times$ increase in execution time for a 4$\times$ increase in total work indicates that overhead grows superlinearly. This is likely because larger arrays exceed cache capacity, and the increased number of Dask task graph nodes (more chunks) compounds scheduling and communication costs.

In practice, speedup on 4 cores for this problem is only about 1.85$\times$ rather than the ideal 4$\times$, reflecting the combined effects of the factors above.