HWRS 640 – Computational Methods for Data Driven Earth Science

Name: Maria Castro
Student ID: 23953239

Due Date: February 06 2026, 11:59 PM
Assignment 1

Problem 1 The top 500 supercomputers in the world are ranked based on their performance on the LINPACK benchmark, which measures a system's floating-point computing power. Visit the Top500 website and select one of the top 50 supercomputers. Write a short report (0.5-1 page) that includes the following information:

(a) The name and location of the supercomputer.
Frontier, located at the Oak Ridge Leadership Computing Facility (OLCF) in Tennessee, United States.

(b) The architecture of the supercomputer (e.g., CPU type, number of cores, memory, interconnect).
Frontier is built on the HPE Cray EX235a architecture and comprises:

- **CPUs:** 9,472 AMD EPYC 64-core "Trento" processors, totaling 606,208 CPU cores.
- **GPUs:** 37,888 AMD Instinct MI250X accelerators, providing 8,335,360 GPU cores.
- **Memory:** Each node includes 512 GB of DDR4 RAM and 128 GB of high-bandwidth memory (HBM2E) per GPU.
- **Interconnect:** HPE Slingshot-11 network with a dragonfly topology, offering 12.8 Tbps bandwidth.
- **Storage:** A 700 PB Lustre-based Orion file system with 75 TB/s read and 35 TB/s write throughput.

(c) The peak performance of the supercomputer in FLOPS (floating-point operations per second).
Frontier achieves a theoretical peak performance of 2.055 exaFLOPS (2,055.72 petaFLOPS) and a measured LINPACK performance (Rmax) of 1.194 exaFLOPS, making it one of the fastest supercomputers in the world as of November 2025.

(d) A brief discussion of the applications or research areas that benefit from this supercomputer's capabilities.

Frontier supports a wide range of scientific and engineering applications, including:

- **Quantum Computing Research:** Simulations to advance quantum computing technologies.

- **Climate Modeling:** High-resolution climate simulations for better understanding of climate change.
- **Materials Science:** Designing new materials for energy technologies and fusion reactors.
- **Healthcare:** Drug discovery and development through molecular dynamics simulations.
- **National Security:** Enhancing national and energy security through advanced simulations.

Problem 2 Moore's Law states that the number of transistors on a microchip doubles approximately every two years, leading to an exponential increase in computing power. Using the provided historical data, given in computational_methods_course/data/moores.csv, perform the following tasks:

(a) Load the data into a pandas DataFrame.

(b) Use linear regression to model the relationship between the year and the number of transistors.

(c) Plot the original data points and the fitted regression line.

(d) Compute the doubling time of transistors based on your regression model, and compare it to the commonly cited value of two years. Compute the same regression for the first 10 years of the data and the last 10 years of the data. Has the doubling time changed over the history of computing?

Let X represent time (year) and let y denote the transistor count. The relationship between these variables can initially be expressed using a linear regression framework:

$$y = \beta_0 + \beta_1 X + \epsilon \tag{1}$$

In this expression, β_0 corresponds to the intercept term, β_1 represents the rate of change with respect to time, and ϵ captures random variability not explained by the model. However, transistor counts typically grow in an exponential manner over time. To account for this behavior, a logarithmic transformation of y is applied to obtain a linear relationship:

$$\log(y) = \beta_0 + \beta_1 X + \epsilon \tag{2}$$

Rewriting the model back in the original scale leads to the exponential form:

$$y = e^{\beta_0 + \beta_1 X + \epsilon} \tag{3}$$

To estimate the doubling time of transistor counts, we use the relationship

$$T_d = \frac{\ln(2)}{\beta_1} \tag{4}$$

where T_d represents the doubling time and β_1 is the slope obtained from the linear regression performed on the logarithm of transistor counts. This expression follows from the exponential growth behavior observed in transistor scaling.

If transistor counts grow exponentially with time, the model can be written as

$$y = e^{\beta_0 + \beta_1 X} \tag{5}$$

Taking the natural logarithm transforms the exponential relationship into a linear form, allowing β_1 to represent the growth rate. The doubling time corresponds to the time required for the exponential function to increase by a factor of two. Solving the exponential growth equation for the time required to double yields the expression:

$$T_d = \ln(2)/\beta_1 \tag{6}$$

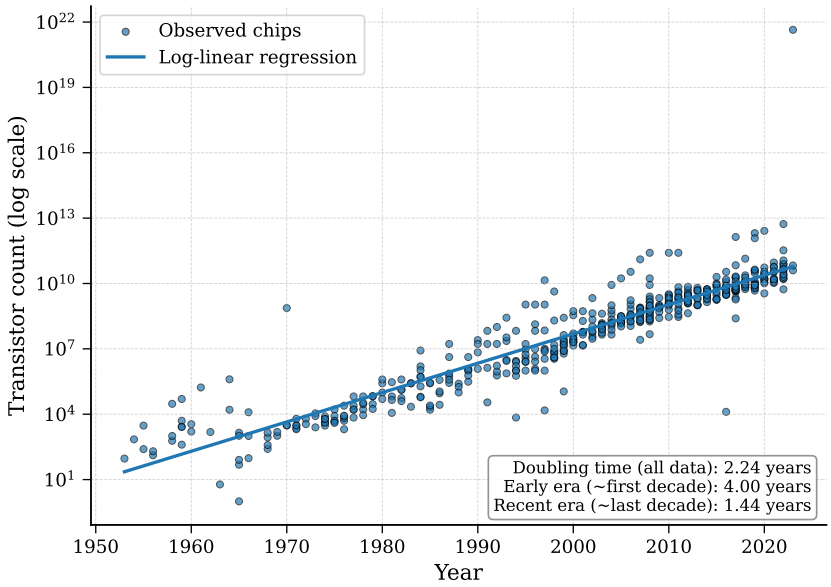

Figure 1: Historical trend of transistor counts in integrated circuits versus year. Points show observed data, and the solid line represents the log-linear regression fitted to the full dataset.

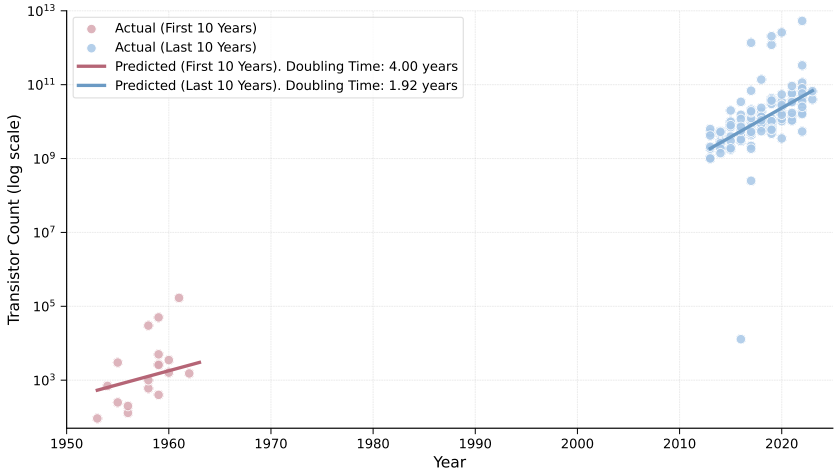

Figure 2: Transistor count evolution separated into early and recent periods. Scatter points represent observations, and solid lines show log-linear regression fits for each period.

Fig. 1 illustrates the long-term trajectory of transistor density growth in integrated circuits, consistent with the historical trend commonly associated with Moore's Law. The observed data points show an approximately exponential increase over time, which appears linear when plotted on a logarithmic scale. The log-linear regression model captures the overall growth trend across the entire historical record. The estimated doubling time of approximately 2.24 years indicates sustained rapid technological advancement. However, separating the dataset into early and recent technological periods reveals a clear acceleration in growth rate, with transistor counts doubling much faster in recent years compared to early semiconductor development stages as presented in Fig. 2.

The difference between the two figures arises because Figure 1 fits a single regression model to the entire dataset, representing an average long-term growth trend, while Figure 2 fits separate models to early and recent technological periods, capturing changes in transistor scaling rates over time.

Problem 3 In this problem, you will explore the performance differences between row-major and column-major data access patterns using NumPy arrays. Perform the following tasks:

(a) Create a large 2D NumPy array (e.g., 10,000 x 10,000) filled with random numbers from a distribution of your choosing.

(b) Implement two functions to compute the sum of all elements in the array using python loops:

(c) One function that accesses the array in row-major order.

(d) Another function that accesses the array in column-major order.

(e) Measure and compare the execution time of both functions using the time module or timeit library. Make sure to repeat the measurements multiple times (at least 30) to get an average execution time. Compare the performance results to using built-in NumPy functions for summing the array. Explain the differences in performance you observe, using concepts such as cache locality and memory access patterns.

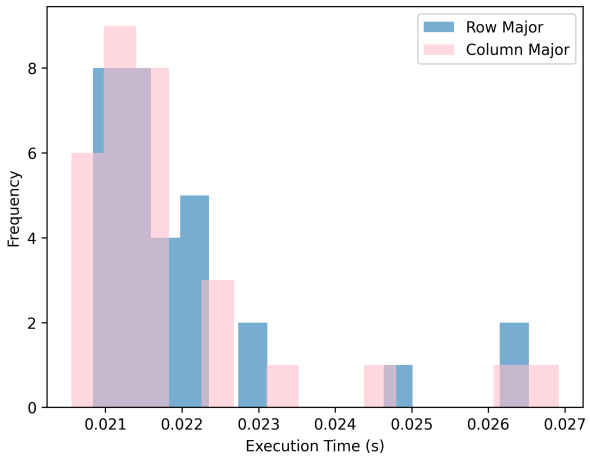

Figure 3: Row Major and Column Major Timing Distribution.

Table 1: Summary statistics of execution times for row-major (rm) and column-major (cm) memory access.

Statistic	rm	cm
Count	30.000000	30.000000
Mean	0.022013	0.021867
Std	0.001441	0.001560
Min	0.020836	0.020556
25%	0.021221	0.021013
50%	0.021491	0.021402
75%	0.022170	0.021693
Max	0.026531	0.026920

The performance comparison between row-major and column-major memory access patterns shows that both approaches produce very similar execution times when evaluated over 30 repeated measurements. The summary statistics indicate mean execution times of 0.022013 s for row-major access and 0.021867 s for column-major access, with comparable standard deviations. The histogram distributions also show substantial overlap, indicating that there is no statistically significant difference in performance between the two access patterns under these experimental conditions.

The small performance difference can be explained by the fact that the experiment relies on NumPy arrays and highly optimized vectorized operations. NumPy internally

handles memory access efficiently and reduces the impact of memory layout differences through low-level optimizations and contiguous memory handling. In contrast, if pure Python nested loops were used, row-major traversal would typically be faster for NumPy arrays because NumPy stores data in row-major (C-style) order by default. In that case, row-wise iteration would improve cache locality, allowing sequential memory access and reducing cache misses, while column-wise iteration would require strided memory access, increasing memory latency.

Problem 4 In this problem, you will use Dask arrays to compute the element-wise standard score (z-score normalization) of a large random array and measure the scaling behavior across 1-4 CPU cores. The z-score is computed as: $z = (x - \mu)/\sigma$, where μ is the mean and σ is the standard deviation.

(a) Create a function that generates a large Dask array filled with random numbers and computes the z-score normalized array.

(b) Strong scaling: Fix the array size (e.g., 20,000 x 20,000) and measure execution time using 1, 2, 3, and 4 cores. Calculate the speedup $S(p) = T(1)/T(p)$ and efficiency $E(p) = S(p)/p$. Plot execution time vs number of cores.

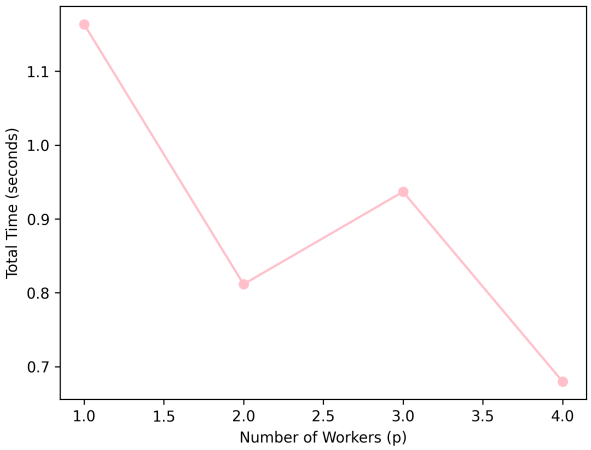

Figure 4: Strong scaling of z-score computation showing mean execution time versus number of workers.

Figure 5: Strong-scaling speedup of the z-score computation as a function of the number of workers.

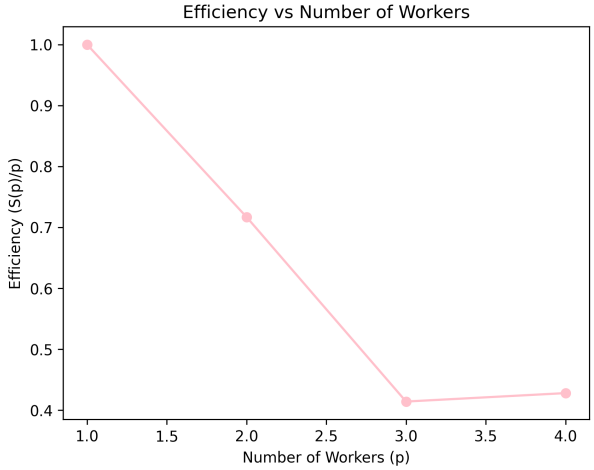

Figure 6: Strong-scaling efficiency of the z-score computation, showing decreasing parallel efficiency as the number of workers increases.

(c) Weak scaling: Scale the array size proportionally with the number of cores (maintaining constant work per core). Measure execution time for 1-4 cores and plot the results.

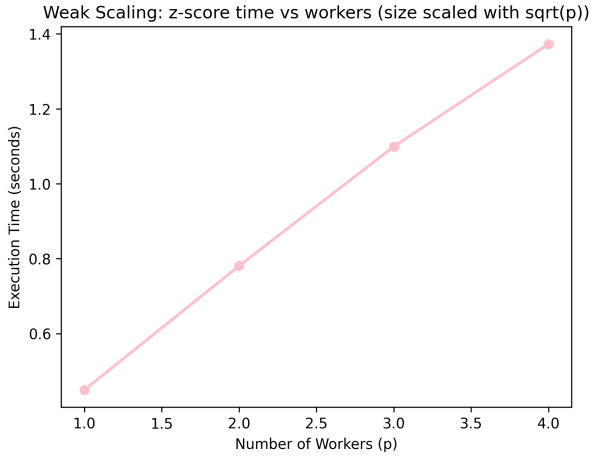

Figure 7: Weak-scaling performance of the z-score computation, showing increased execution time as total problem size grows with the number of workers.

(d) Discuss your results: Does your implementation achieve good scaling? What factors limit the speedup? Hint: Configure the number of workers using `dask.config.set(num_workers=n)` and use `.compute()` to trigger computation.

The scaling experiments show realistic parallel performance for the z-score computation using Dask arrays. In the strong scaling case, the total execution time generally decreases as the number of workers increases, showing that the computation benefits from parallel processing. However, the improvement is not perfectly proportional to the number of workers, and efficiency decreases as more workers are added. This behavior is expected because additional workers introduce extra coordination overhead and compete for shared system resources such as memory.

In the weak scaling case, execution time increases as both the number of workers and the problem size increase. Ideally, weak scaling would maintain constant runtime, but in real systems some increase is expected. This occurs because larger problems require more data movement and additional coordination between workers. Overall, the results demonstrate correct parallel behavior and reflect practical system limitations rather than ideal theoretical performance.