Homework 1

Aldo Tapia

Note: All the code was executed in my local machine instead of using codespaces for this task (just to run Quarto).

Problem 1: Supercomputer architecture (25 points)

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:

1. The name and location of the supercomputer.
2. The architecture of the supercomputer (e.g., CPU type, number of cores, memory, interconnect).
3. The peak performance of the supercomputer in FLOPS (floating-point operations per second).
4. A brief discussion of the applications or research areas that benefit from this supercomputer's capabilities.

Report:

The supercomputer I have choosen is called Sunway TaihuLight, which is located in National Supercomputing Center in Wuxi, China. The reason I chose this supercomputer is because it caught my attention due to its fabrication year (2016). This supercomputer is top 24 in TOP500 list (updated November 2025) and it is ranked in a very high position consireding it is 10 years old (it was the World's fastest supercomputer from June 2016 to June 2018).

The CPU is called Sunway SW26010, and it's composed by 260 cores orzanized in a hierchical way. The supercomputer has 40,960 nodes (each node has 256 cores for processing plus 4 cores for system management and other tasks), which gives a total of 10,649,600 CPU cores. The clock speed of the cores is 1.45 GHz, and the clock speed is related to the performance of the supercomputer, but it is not the only factor that determines its performance. The memory of the supercomputer is 1.32 PB (petabytes) of RAM, and each node has a large DDR3 block of ~256 GB/node. The DDR3 memory is described as "high bandwidth memory" and it is

designed to provide high performance for data-intensive applications. The interconnect used in the Sunway TaihuLight is called Sunway Network (designed exclusively for this supercomputer), which is a high-speed interconnection that allows for fast communication between the nodes.

The network is centralized by 1 management core (MPE) which control 256 computing cores (CPE), each one plays a similar role to a GPU core (this supercomputers doesn't use GPUs, but the CPEs are designed to perform similar tasks). The acceleration is achieve by a massive on-chip parallelism.

A FLOPS (floating-point operations per second) is a measure of a computer's performance, especially in fields of scientific calculations that require a large number of floating-point calculations. From TOP500 website, we have two different metrics: Rmax and Rpeak. Rmax is the maximum performance achieved by the supercomputer on the LINPACK benchmark (a data-intensive benchmark that measures the performance of a system in solving a dense system of linear equations), while Rpeak is the theoretical peak performance of the supercomputer based on its hardware specifications. For Sunway TaihuLight, Rmax is 93.01 PFLOPS (petaFLOPS) and Rpeak is 125.44 PFLOPS. A common desktop computer can achieve around 100 GFLOPS (gigaFLOPS), which is 0.0001 PFLOPS, so the Sunway TaihuLight is about 930,000 times faster than a common desktop computer.

The main applications that are ran in this suppercomputer are related to scientific research. For echample, Earth Sysyem Modeling, like global and local climate modilng, ocean-atmosphere interaction, and weather forecasting. Another applications are related to Fluid dynamics, Seismology, Material science and physics, among others. A summary of a few studies can be found in: Fu, H., Liao, J., Yang, J. *et al.* The Sunway TaihuLight supercomputer: system and applications. *Sci. China Inf. Sci.* **59**, 072001 (2016). https://doi.org/10.1007/s11432-016-5588-7

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

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:

1. Load the data into a pandas DataFrame.
2. Use linear regression to model the relationship between the year and the number of transistors.
3. Plot the original data points and the fitted regression line.
4. Compute the doubling time of transistors based on your regression model, and compare it to the commonly cited value of two years. 5 (for fun). 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?

Answer:

Aproach:

Let X be the year and y the number of transistors. The relationship can be described as:

$$y = \beta_0 + \beta_1 X + \epsilon$$

Where β_0 is the intercept, β_1 is the slope, and ϵ is the error term. Since the relationship of the number of transistors is exponential, we can take the logarithm of y to linearize the relationship:

$$\log(y) = \beta_0 + \beta_1 X + \epsilon$$

Then, normalizing back to the original scale, we can express the model as:

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

For testing the regression, I used three different models: OLS, Ridge and Lasso. The RMSE of the three models are very similar, with the OLS model having the lowest RMSE. This indicates that the OLS model is the best fit for the data, but the Ridge and Lasso models also provide a good fit. The differences are shown in the plot below with an inset zoomed in to the last 5 years of data, where the differences between the models are more evident (which are not big):

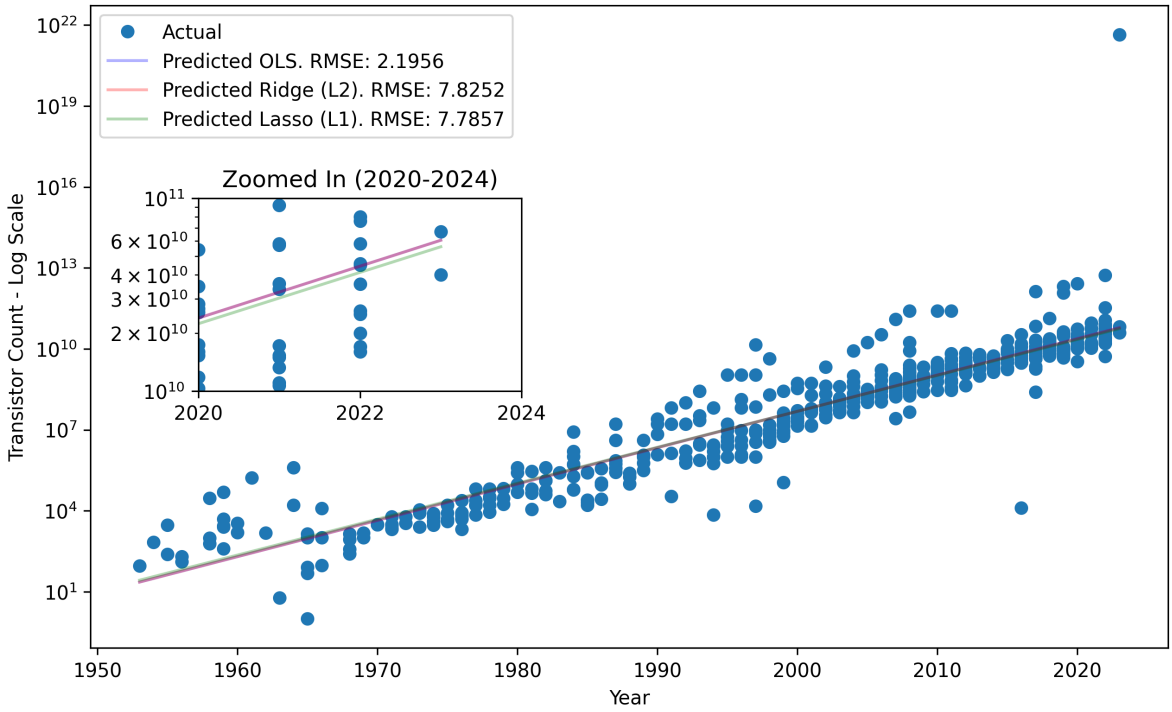


Figure 1: Actual vs Predicted Transistor Count by year

The figure is log-scaled because the number of transistors grows exponentially, and it is easier to visualize the relationship between the year and the number of transistors on a log scale. The plot shows that the number of transistors has been increasing exponentially over the years, and the fitted regression lines (OLS, Ridge, and Lasso) closely follow the actual data points.

To compute the doubling time of the number of transistors, we can use the formula:

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

Where T_d is the doubling time and β_1 is the slope of the linear regression model. This formula arises from the fact that the number of transistors doubles when X increases by T_d, which can be derived from the exponential growth model.

```
td = np.log(2) / model1.coef_[0][0] # OLS model

print(f"The doubling time is approximately {td:.2f} years.")
```

```
The doubling time is approximately 2.24 years.
```

To test if the doubling approach is working, here a small verification:

```
y_test = model1.predict([[1980],[1980 + td]])
multiplier_factor = np.exp(y_test[1][0]) / np.exp(y_test[0][0])

print(f"Empirically, the multiplier factor" +
      f" after {td:.2f} years is approximately " +
      f"{multiplier_factor:.2f}.")
```

```
Empirically, the multiplier factor after 2.24 years is approximately 2.00.
```

Then, we can compute the regression for the first 10 years of data and the last 10 years of data to see if the doubling time has changed over the history of computing:

```
first_year, last_year = X.min(), X.max()
Xfirst = X[(X >= first_year) & (X <= first_year + 9)].reshape(-1, 1)
yfirst = y[(X >= first_year) & (X <= first_year + 9)]
Xlast = X[(X >= last_year - 9) & (X <= last_year)].reshape(-1, 1)
ylast = y[(X >= last_year - 9) & (X <= last_year)]

model_first = LinearRegression(fit_intercept=True).fit(Xfirst, yfirst)
model_last = LinearRegression(fit_intercept=True).fit(Xlast, ylast)

td_first = np.log(2) / model_first.coef_[0]
td_last = np.log(2) / model_last.coef_[0]

print(f"Doubling time in the first 10 years: {td_first:.2f} years.")
print(f"Doubling time in the last 10 years: {td_last:.2f} years.")
```

```
Doubling time in the first 10 years: 1.44 years.
Doubling time in the last 10 years: 1.34 years.
```

This can be shown in the plot below, where the doubling time is represented by the slope of the regression line. The steeper the slope, the shorter the doubling time.

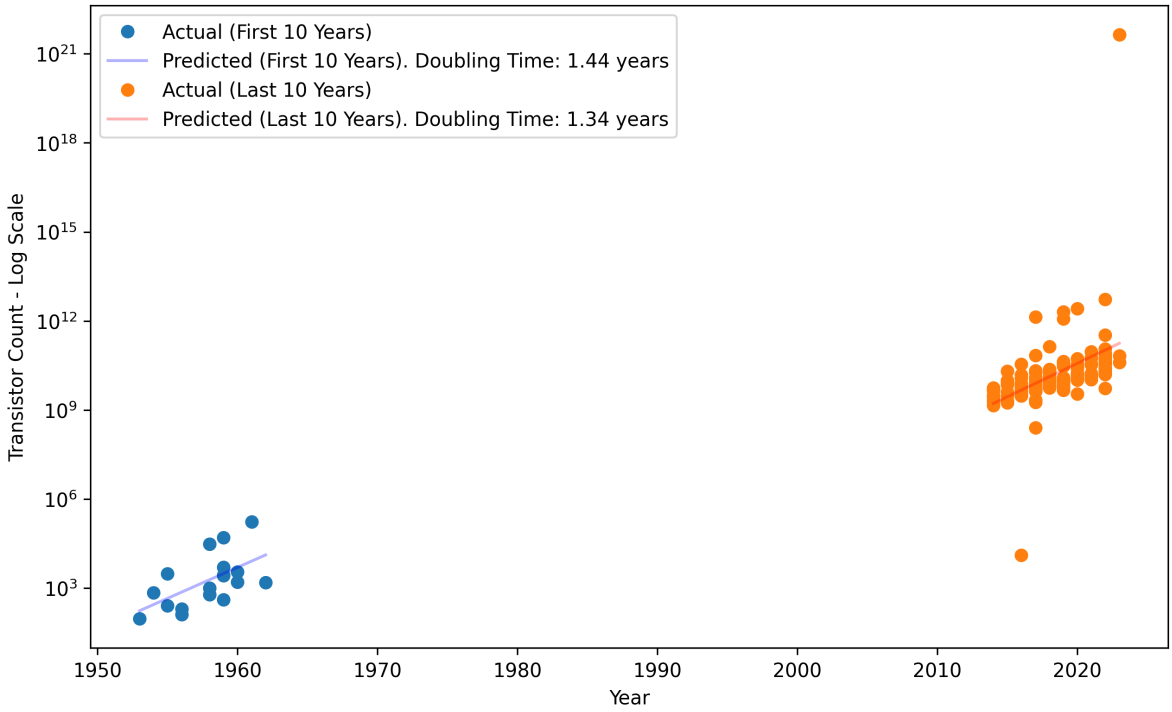


Figure 2: Actual vs Predicted Transistor Count by year for the first and last 10 years

In both cases, the doubling time is similar. I see there are some outliers. The same procedure but removing outliers (defined as points with a residual greater than 2 standard deviations):

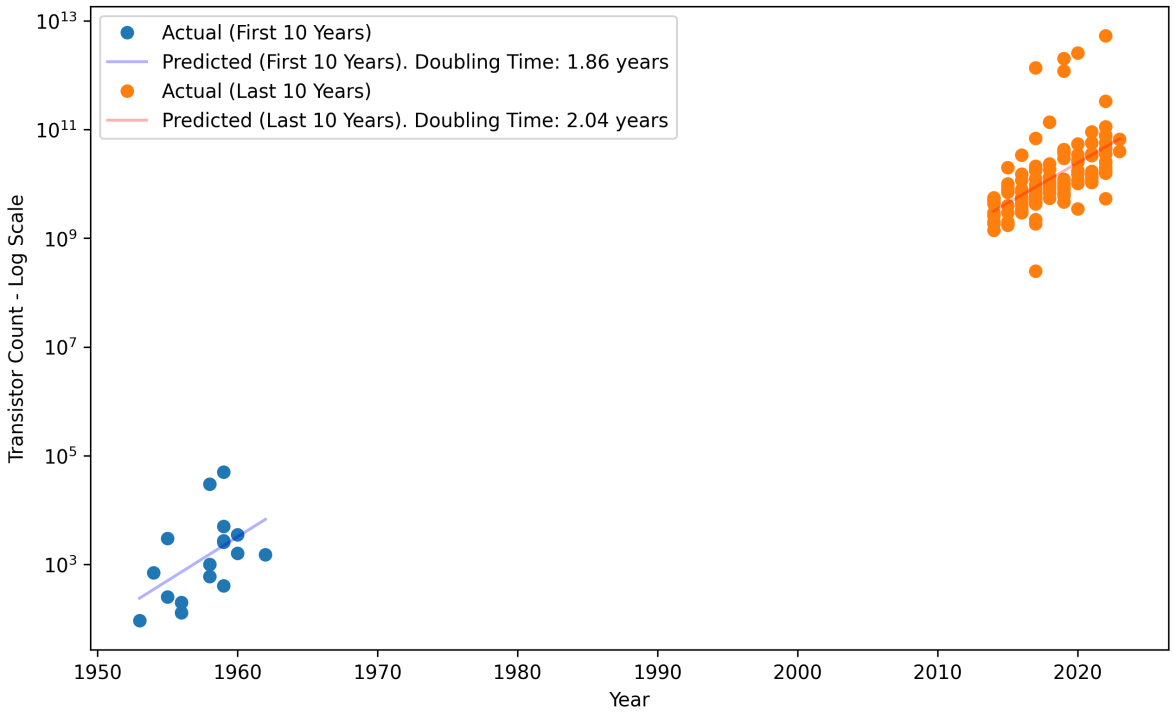


Figure 3: Actual vs Predicted Transistor Count by year for the first and last 10 years (Outliers Removed)

The new doubling times are similar to the previous ones, which indicates that the outliers do not have a significant impact on the estimation of the doubling time.

Problem 3: Row vs column order data access (25 points)

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:

1. Create a large 2D NumPy array (e.g., 10,000 x 10,000) filled with random numbers from a distribution of your choosing.
2. Implement two functions to compute the sum of all elements in the array using python loops:
    - One function that accesses the array in row-major order.
    - Another function that accesses the array in column-major order.
3. 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.

4. 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.

Histograms of the execution times for row-major and column-major access patterns:

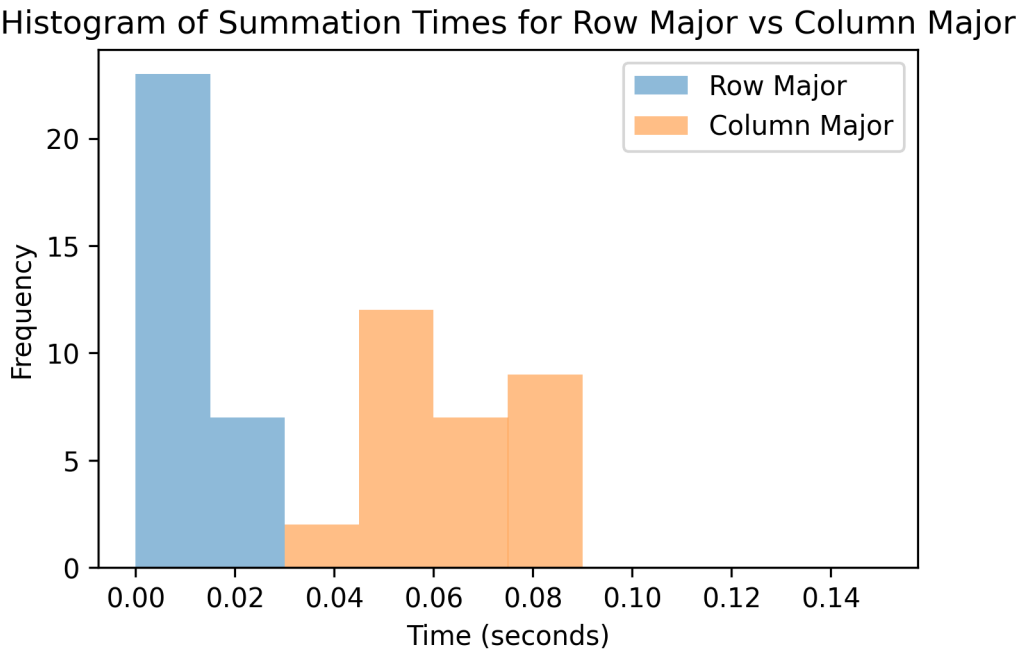


Figure 4: Execution Time Distribution for Row-Major and Column-Major Access Patterns

Finally, the statistics of the execution times for both access patterns:

	rm	cm
count	30	30
mean	0.014693	0.0632494
std	0.00107319	0.0136558
min	0.0133357	0.044203
25%	0.0140896	0.0499529
50%	0.0144511	0.0636624
75%	0.0149864	0.0753977
max	0.0183082	0.08409

Both the histogram and the statistics show that the execution times for row-major access are generally lower than those for column-major access. This performance difference can be

attributed to cache locality and memory access patterns. NumPy arrays are stored in row-major order by default, which means that elements in the same row are stored contiguously in memory. When we access the array in row-major order, we take advantage of spatial locality, which allows the CPU to efficiently load data into the cache.

Problem 4: Scaling and parallel computing (25 points)

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 -) / , where is the mean and is the standard deviation.

Perform the following tasks:

1. Create a function that generates a large Dask array filled with random numbers and computes the z-score normalized array.
2. **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.
3. **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.
4. 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.

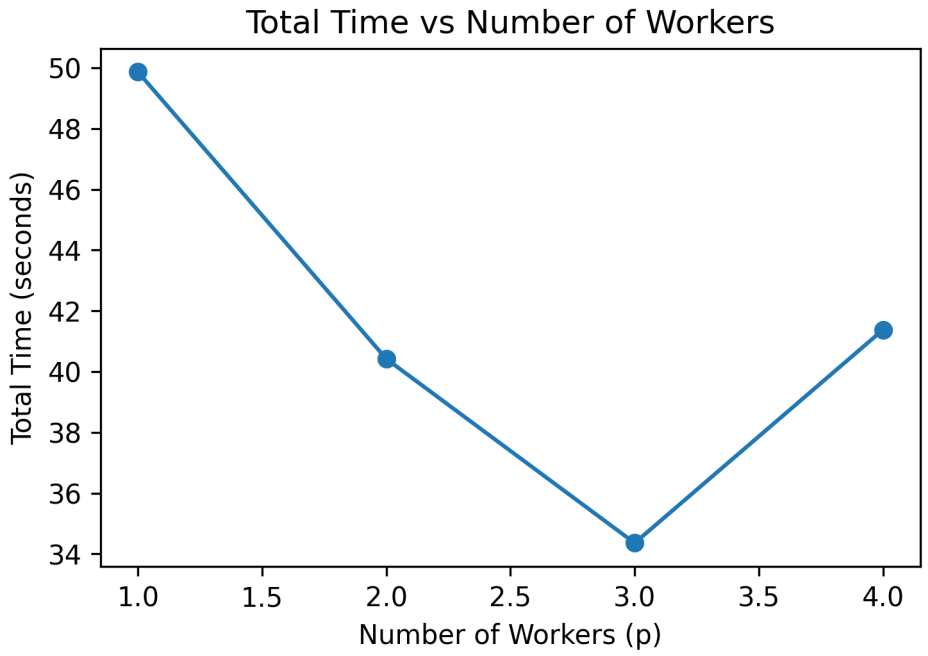
Code for the process:

```
SCHEDULER = "processes"

def strong_scaling(t1, tp):
    return t1 / tp

def efficiency(t1, tp, p):
    return strong_scaling(t1, tp) / p

def experiment(chunks=(2000, 2000)):
    x = da.random.random((20000, 20000), chunks=chunks)
    return zscore(x)

results = []
```

```
for p in range(1, 5):

    with dask.config.set(
        scheduler=SCHEDULER,
        num_workers=p
    ):
        t_ini = time.perf_counter()
        array = experiment()
        array.compute()
        t_total = time.perf_counter() - t_ini

    results.append({"p": p, "t_total": t_total})

df = pd.DataFrame(results)
```

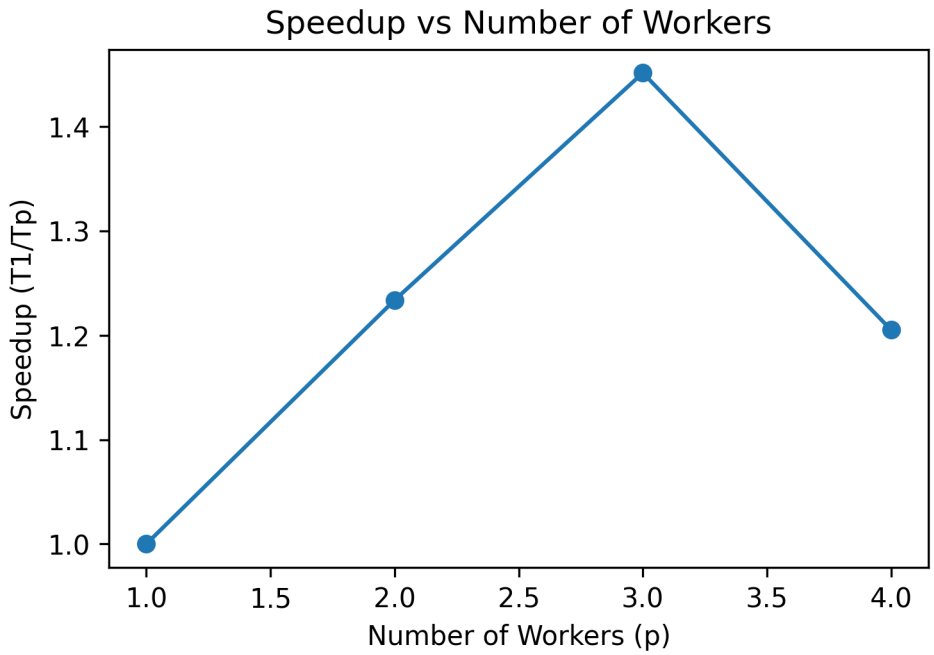
```
Text(0.5, 1.0, 'Total Time vs Number of Workers')
```

Total Time vs Number of Workers



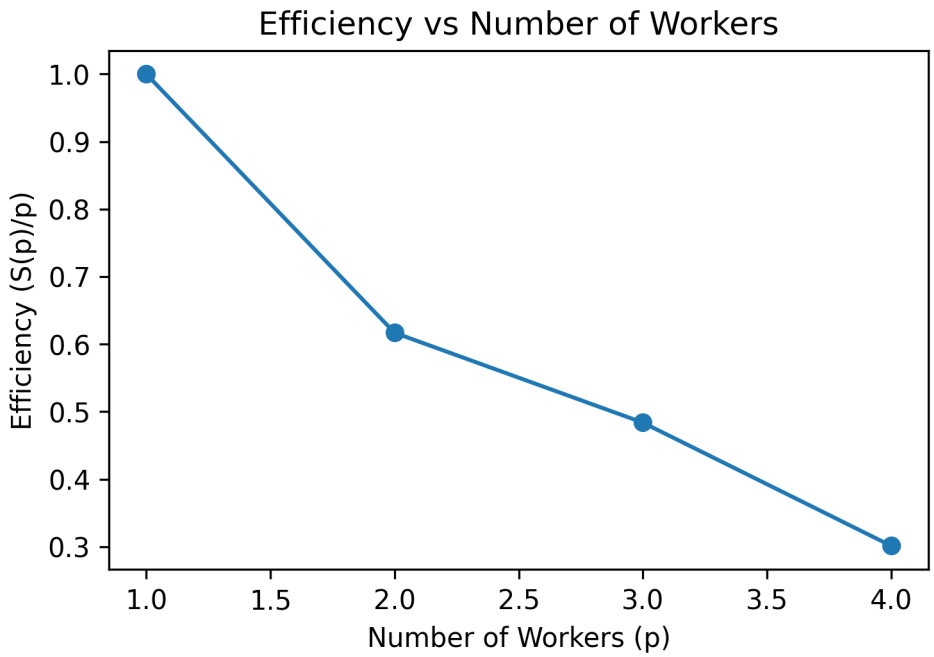

```
Text(0.5, 1.0, 'Speedup vs Number of Workers')
```

Speedup vs Number of Workers




```
Text(0.5, 1.0, 'Efficiency vs Number of Workers')
```

Efficiency vs Number of Workers



The results indicates the total processing time decreases (as expected) as the number of workers increases, demonstrating that the implementation achieves good scaling. However, the speedup is not linear, which can be attributed to several factors, like the communication between workers or the nature of the workload. This is also expressed in the efficiency plot, which shows a decrease in efficiency as the number of workers increases. If the workload is not perfectly parallelizable, many tasks could fail. In this case, since the chunk size is fixed, the workload may not be perfectly balanced across workers.

side note: I did many tests, and sometimes using 4 cores is slower than 3 cores. Since I'm rendering the document in Quarto, I cannot see the results until I render the document, but I can see the results in the notebook and I have gotten this behavior several times.