Mini Project 2 Report

Team

Harshil Patel 017423588
Dheer Manish Jain 016857022
Mahek Virani 017446936
Sri Charan Reddy Mallu 017419779

PPT: **High-Performance Computing in Air Quality Data Simulation**

Github: https://github.com/dheer-jain-2000/cmpe-275-mini2

Introduction -

This project utilizes advanced High-Performance Computing (HPC) technologies to implement a state-of-the-art parallel data extraction system. It features a complex distributed processing architecture that uses the Message Passing Interface (MPI) to conduct parallel computations across various processors. Additionally, it integrates OpenMP to optimize parallel processing within single processors, significantly boosting the system's efficiency and scalability. This design is particularly effective for handling extensive and intricate datasets, enabling detailed simulations of time series and geographic analyses.

Achievements -

- **Distributed File Reading** - The system effectively distributes portions of the AirQuality Dataset among various processes using calculated file offsets to balance the workload evenly. This strategy minimizes data transfers and enhances workload balance, thereby improving the scalability and efficiency of managing extensive datasets.

- **Parallel Record Processing** - Each process within the system employs OpenMP for parallel processing of air quality records, focusing on the cleaning and transformation of data for time series and geographic analysis. The use of multi-threading significantly reduces the processing time.

- **Logging and Debugging** - The system includes a logging mechanism to oversee its operations. This feature is crucial for debugging and tracking performance, particularly when operating across multiple nodes in a distributed computing environment.

- **Geospatial and Temporal Analysis of Air Quality Violations** - This analysis delves into the air quality records, capturing both geographic and temporal aspects. It identifies significant patterns in pollutant levels across various locations and explores changes over time, providing insights into weekly and monthly fluctuations.

- **Performance Metrics Analysis** - The application of High-Performance Computing (HPC) strategies has markedly improved data processing efficiency, as demonstrated by thorough performance metrics. Notable achievements include optimizing execution times across different numbers of processes and maximizing the rate of records processed per second with varied processor allocations.

Working of code

The code utilizes MPI for distributing data across processes to take advantage of distributed memory systems and employs OpenMP for threading within each process to leverage shared memory systems, thus maximizing data processing capabilities.
Below is the detail explanation of MPI and OpenMP usage within the code:

MPI Initialization and Work Distribution: The main function starts by initializing MPI, identifying the rank and size of each process. It uses MPI to distribute the workload across multiple processes by assigning different chunks of the dataset to each process.

File Handling with MPI: The distributeRecords function calculates file offsets for each process, enabling parallel, non-overlapping file reads. This means each MPI process reads only its portion of the data, which increases the efficiency of data handling.

OpenMP for Fine-Grained Parallelism: Within the processRecords function, OpenMP directives (#pragma omp) are used to create a team of threads within each MPI process. This allows simultaneous data processing across multiple cores of a single node.

Dynamic Load Balancing: The code calculates the number of records to process per MPI process, taking into account the remainder to ensure even distribution. If there are leftover records after an equal division, these are also distributed, ensuring that all processes have a balanced load.

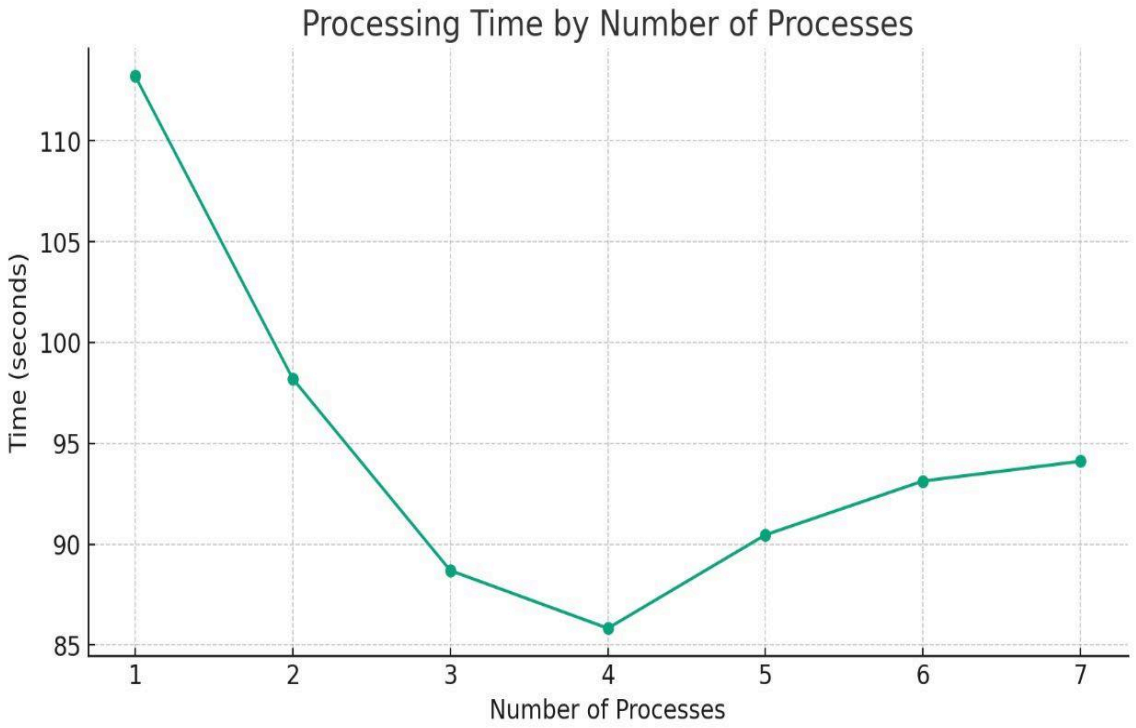
Performance Metrics: Each process collects metrics on the number of records processed per unit of time, highlighting the performance benefits of using multi-threading in conjunction with distributed processing.

Performance Metrics -

The effectiveness of our project is supported by detailed performance metrics that demonstrate how High-Performance Computing (HPC) strategies enhance the efficiency of data processing tasks.

Performance Metric 1: Execution Time Across Various Process Counts

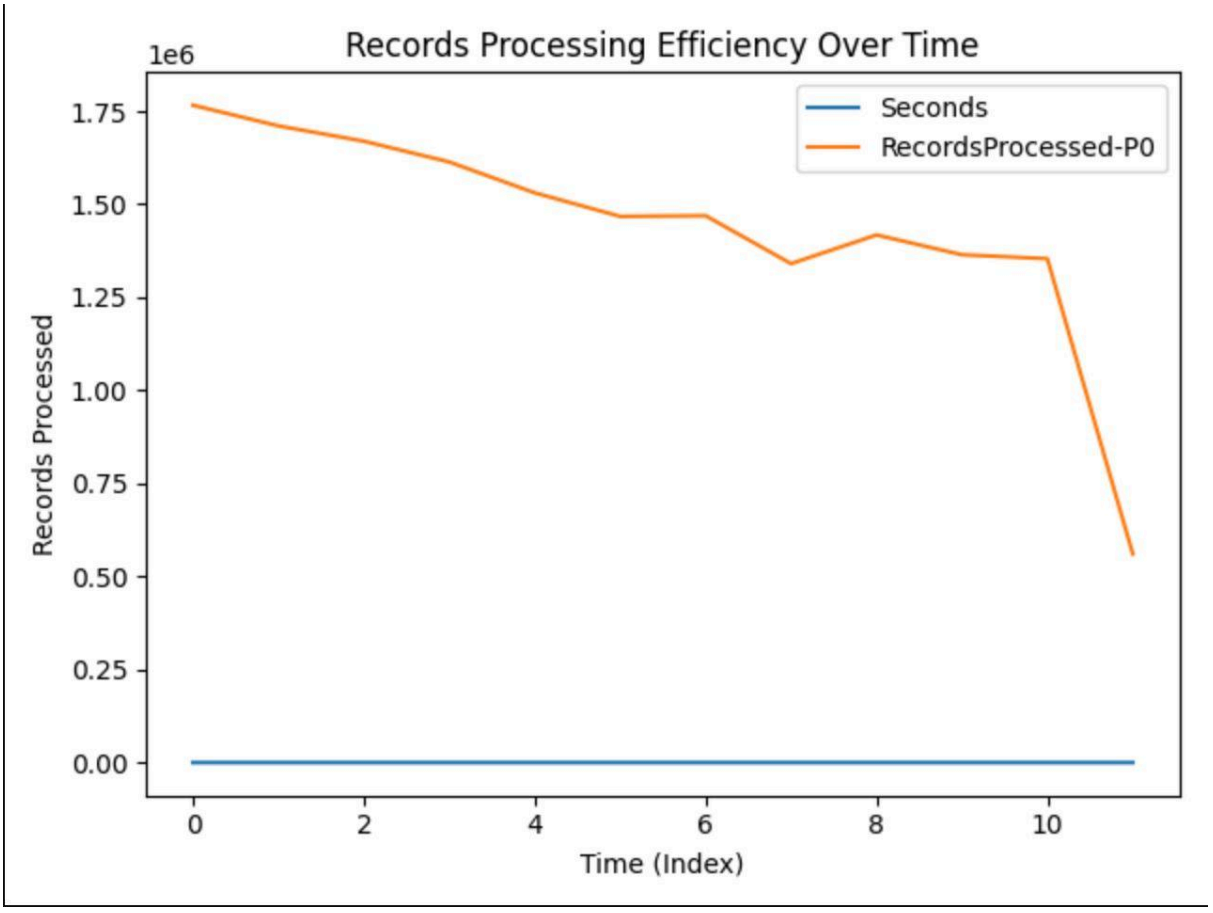
The graph depicting average execution time clearly shows the optimal number of processes for our system. As the number of processes increased to four, there was a noticeable reduction in execution time. However, the increase in execution time for the fifth and sixth processes indicates that there is a diminishing return on further parallelism.



Performance Metric 2: Records Processed Per Second Across Varying Processor Counts

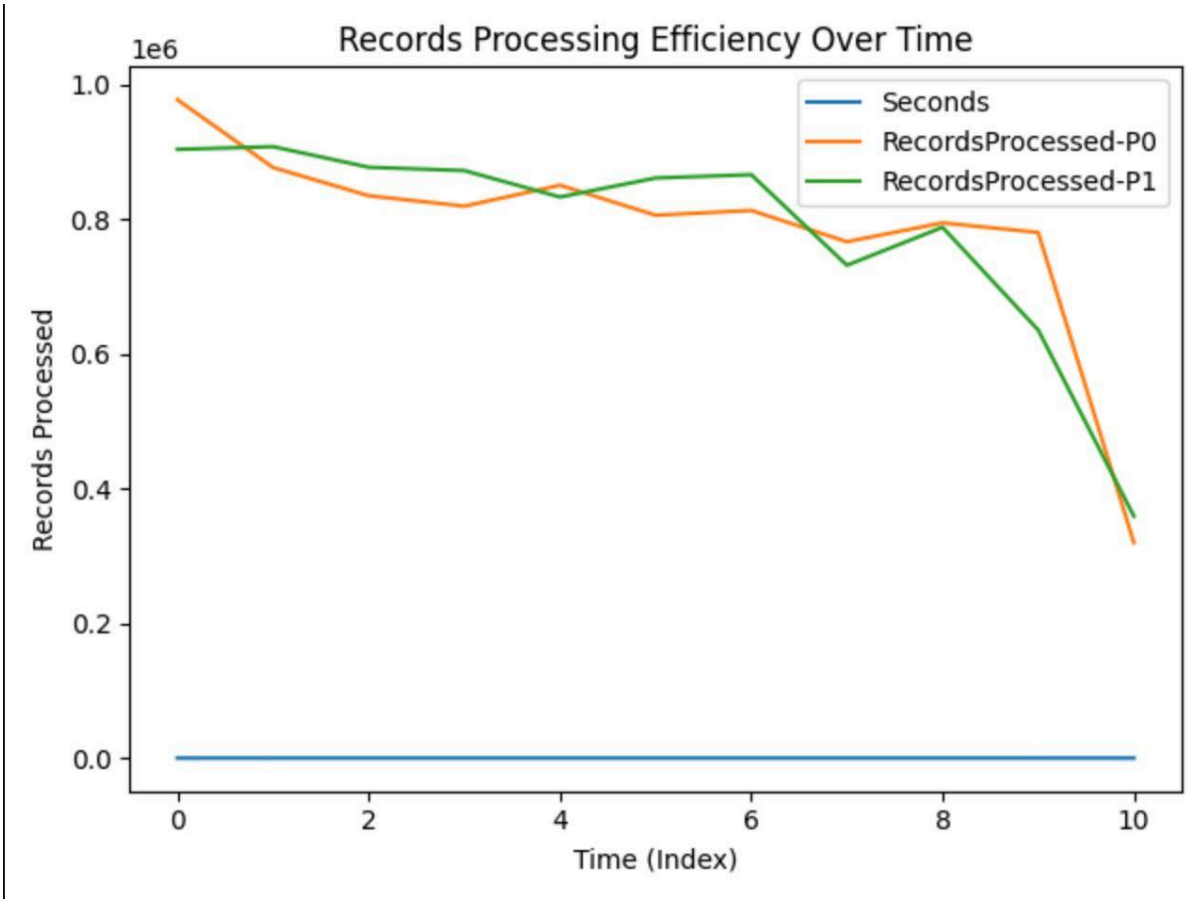
We assessed system performance by measuring the rate at which records were processed per second across varying numbers of processors.

Number of Processes = 1
P0's thorough analysis over time highlights the system's stability and speed, contributing to improved overall efficiency.



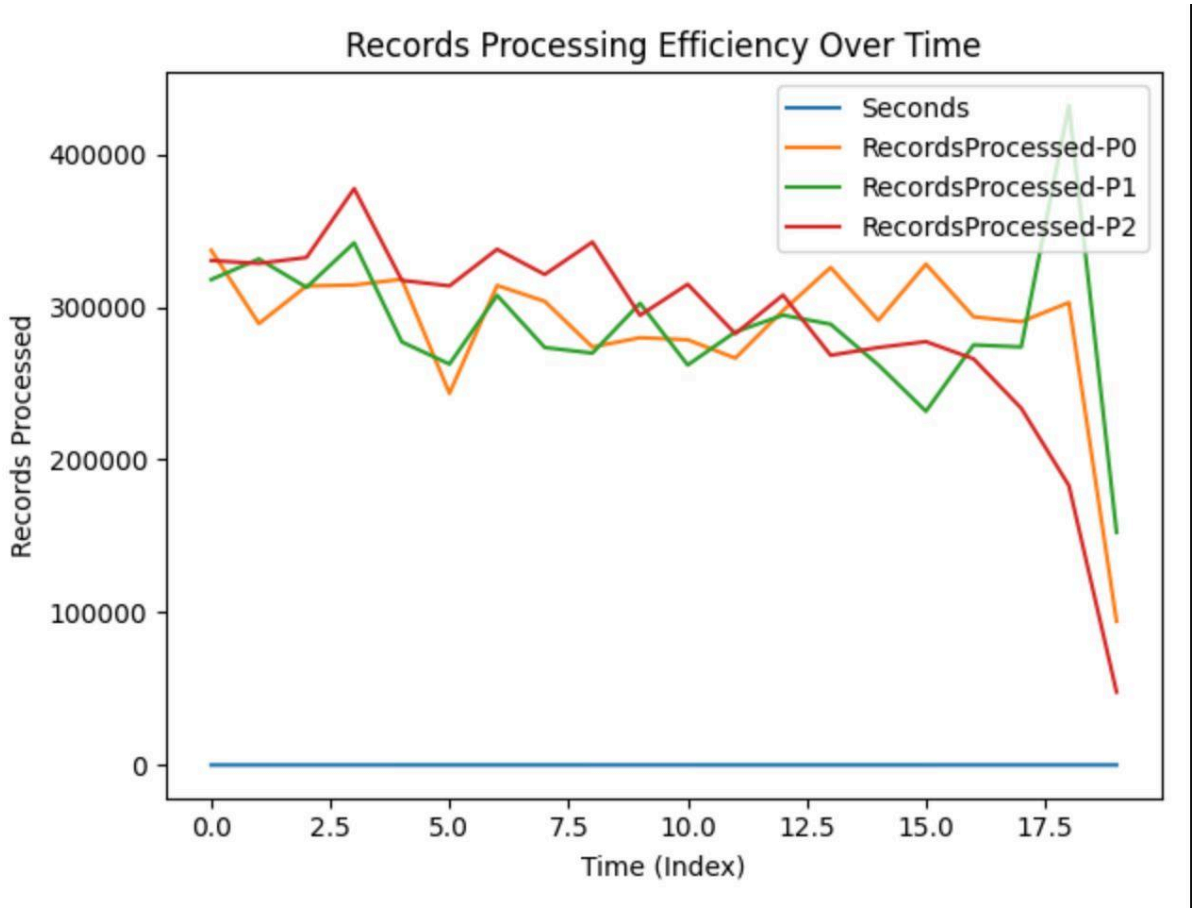
Number of Processes = 2

An examination of individual processes during the initiation phase reveals a swift processing rate that quickly stabilizes, indicating effective load balancing and optimal parallel processing without bottlenecks.



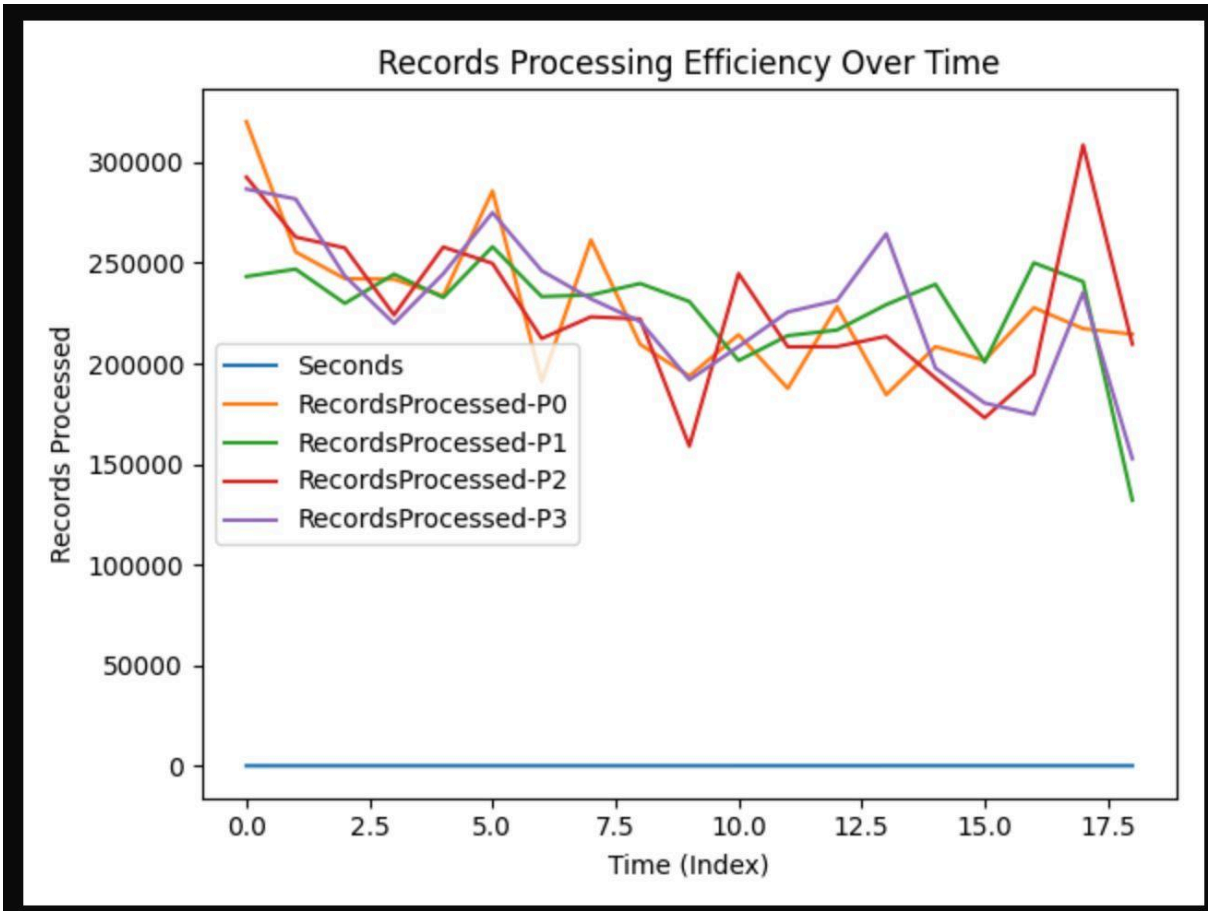
Number of Processes = 3

The enhanced processing power observed with up to four processors underscores the system's scalability. However, a performance plateau becomes evident upon adding the fifth and sixth processors.



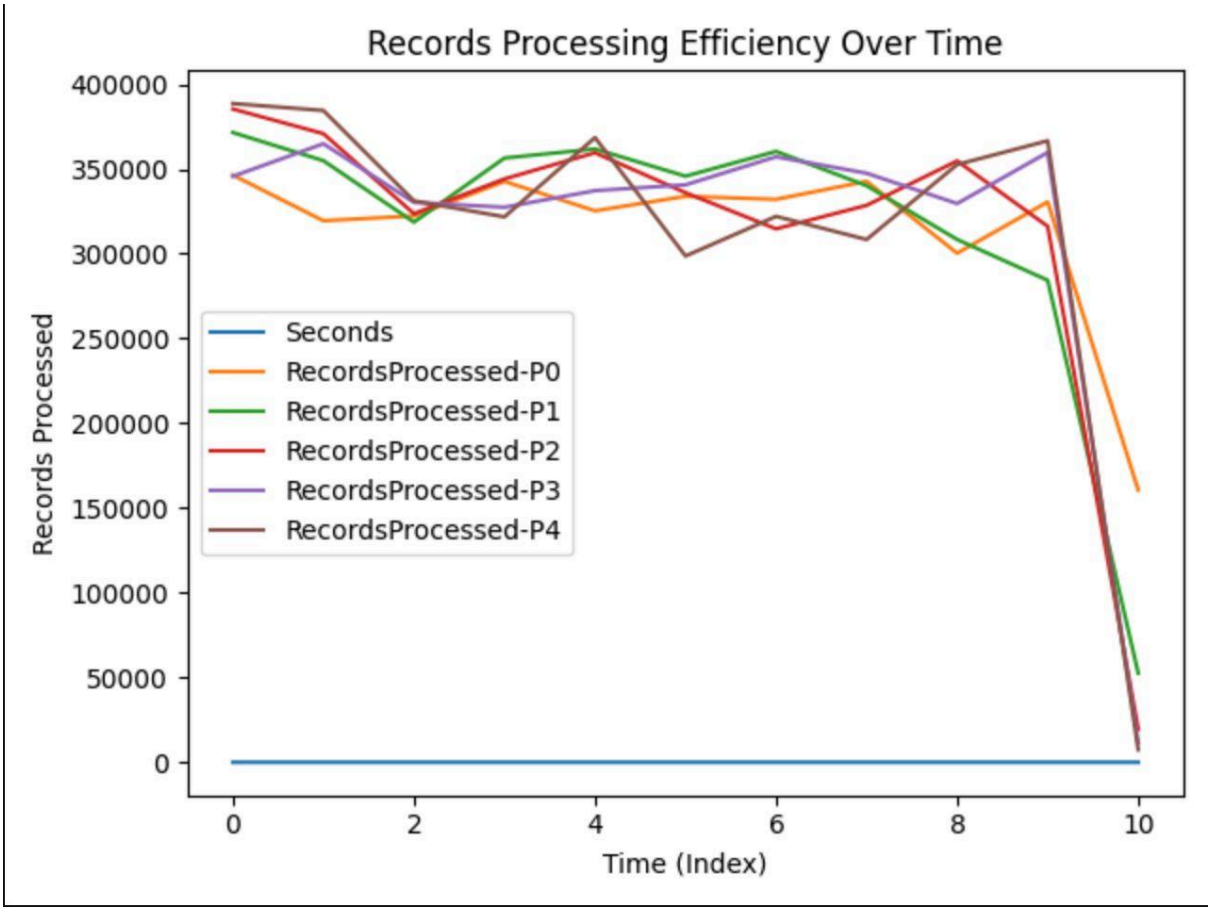
Number of Processes = 4

Uniform performance from P0 to P3 confirms the system's effective load-balancing capabilities.



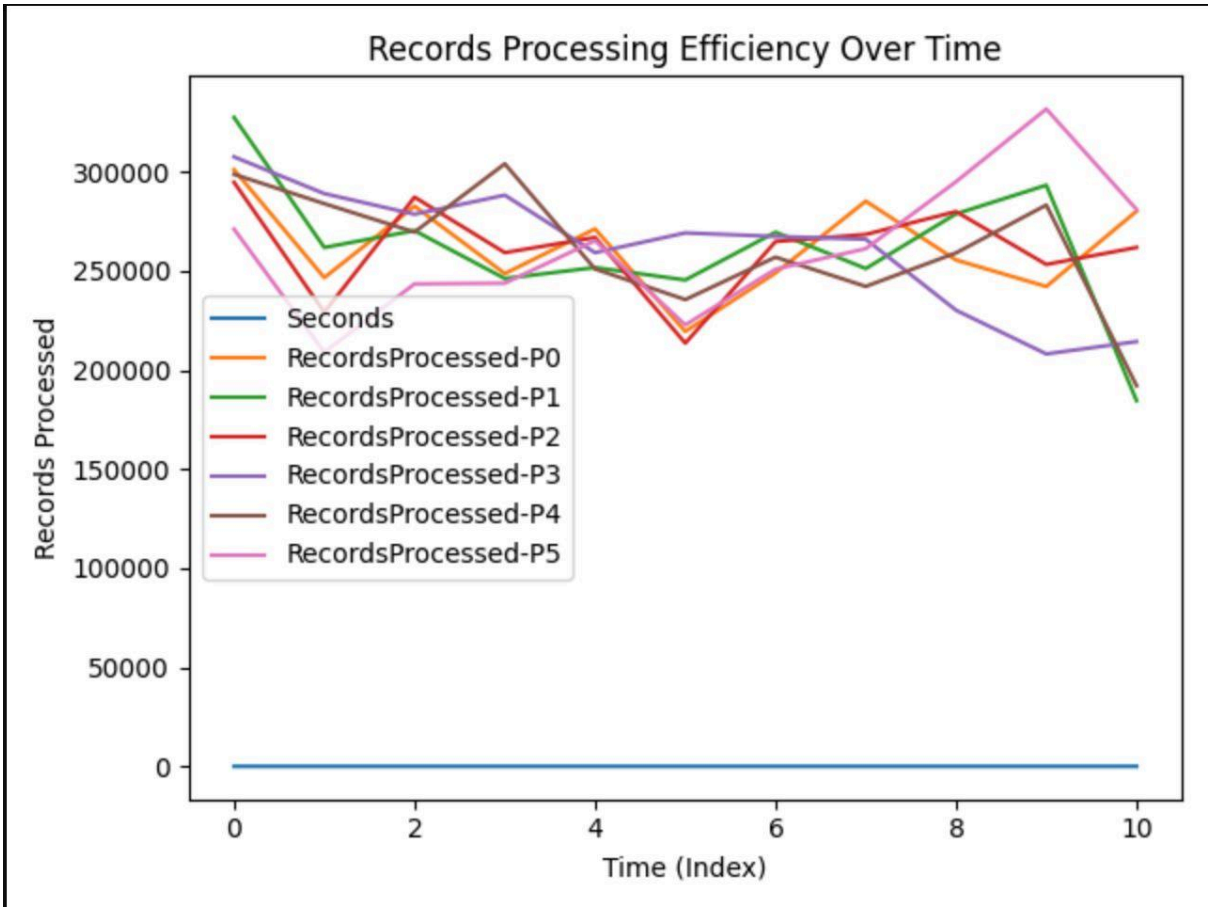
Number of Processes = 5

Extending the analysis to include P4 indicates an optimal balance within the system, hinting at diminishing returns when more processors are added.



Number of Processes = 6

A comprehensive output analysis from P0 to P5 shows a slight decline in performance when additional processors are used, suggesting the inherent challenges of managing extra parallel processes.



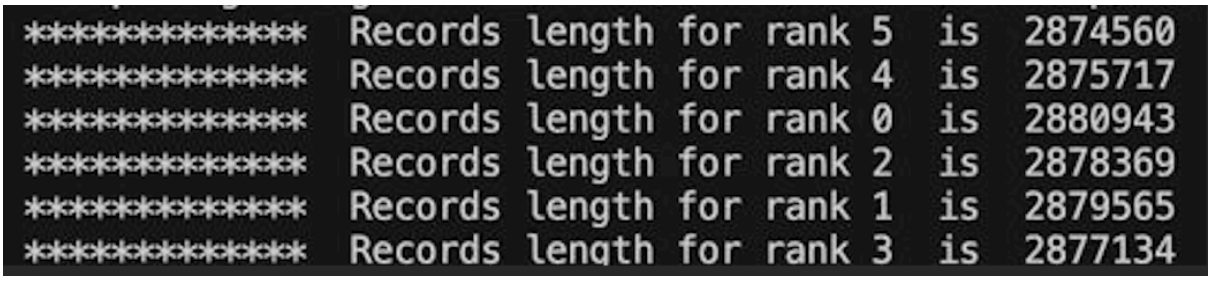
Overall Observation

Line graphs representing individual processor performance (P0 through P6) showed rapid processing onset across all processors, achieving a steady state quickly. This indicates successful load balancing, ensuring no single processor hinders overall processing. The data reveals significant performance gains as the number of processors increases to four. Nevertheless, with the introduction of the fifth and sixth processors, a plateau and slight performance dip were observed. This trend suggests a threshold beyond which additional processors do not equate to proportional performance gains, likely due to increased management overhead or synchronization challenges.

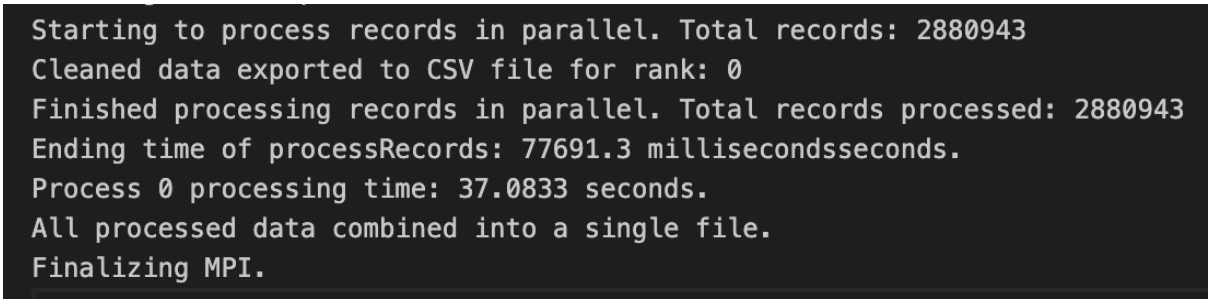
Screenshots -

Logging and Debugging -

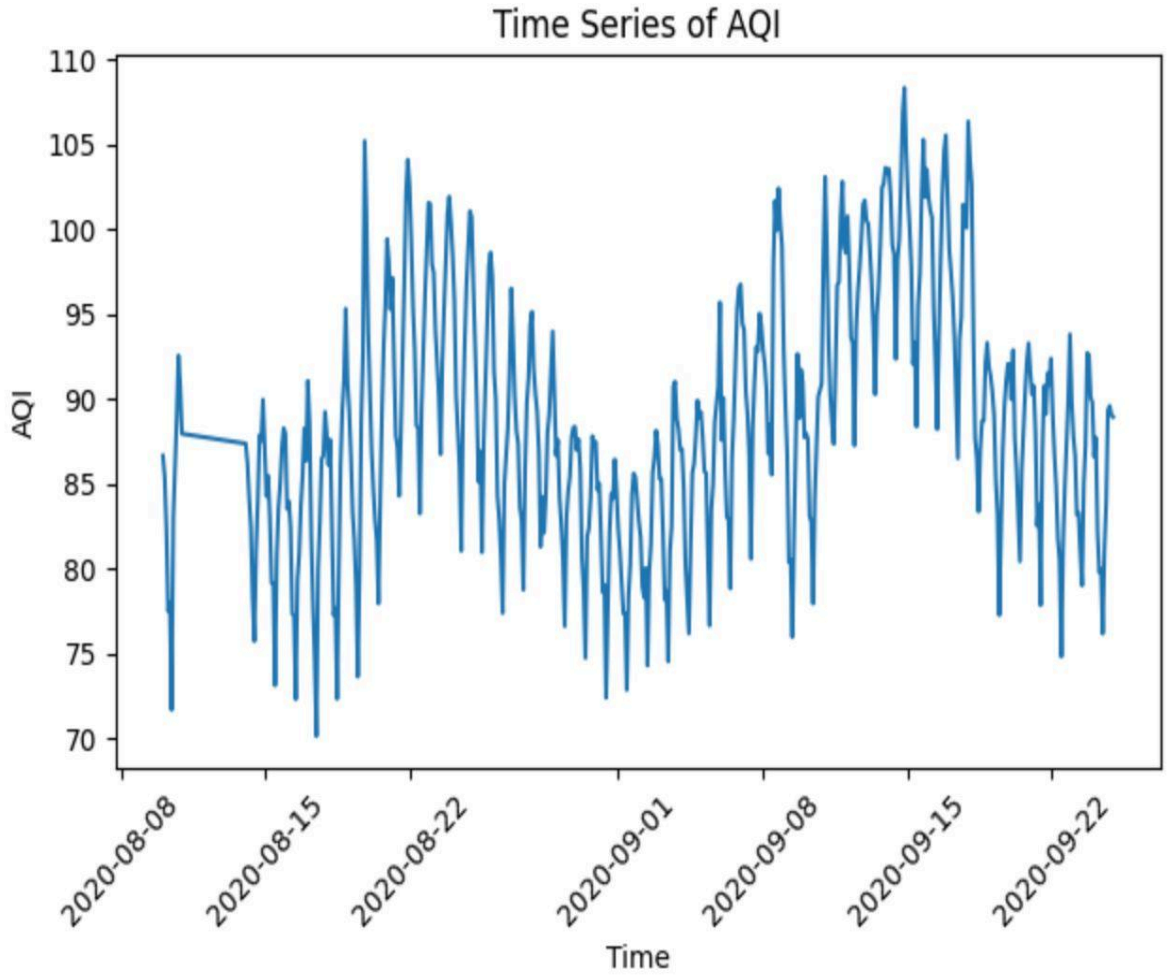
The terminal output displays the record distribution among six MPI processes (ranks 0 to 5) after data parallelization. Each process has been allocated a similar number of records to process, varying slightly due to load balancing, with counts in the 287,000 range. This indicates an even workload distribution essential for efficient parallel processing.



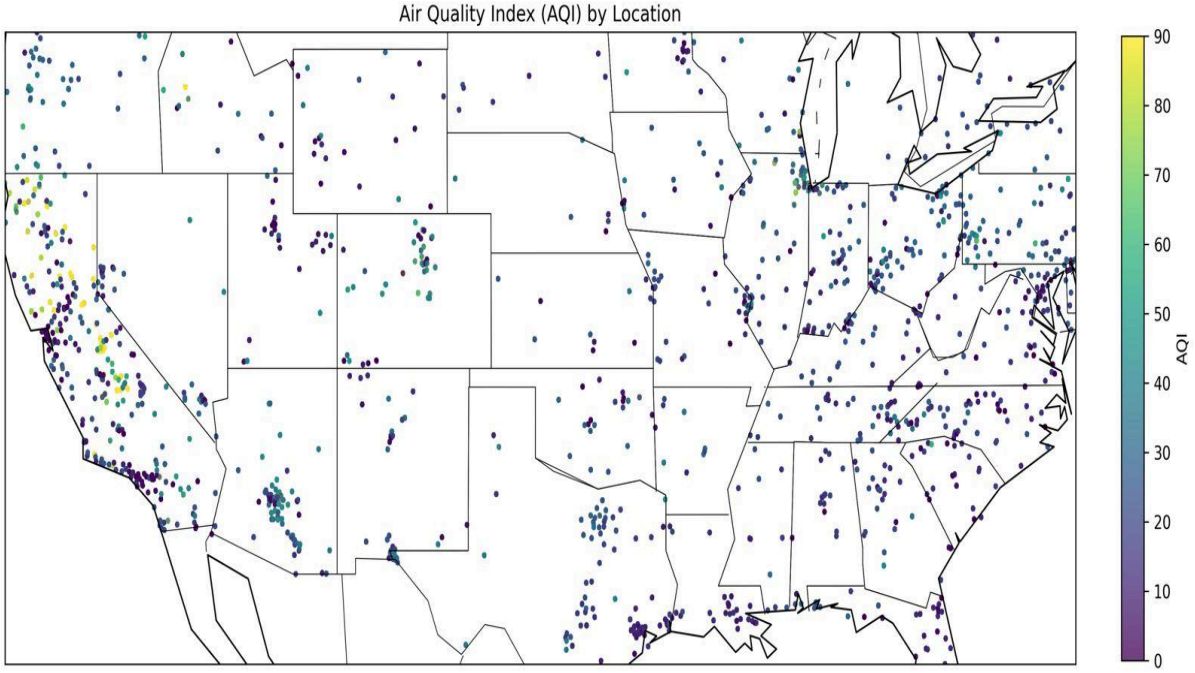
The below screenshot displays the terminal output of a single process in a parallelized data processing task. Process 1 reads 2,879,565 records, starts processing them in parallel, and upon completion, exports the cleaned data to a CSV file. The process begins at 40.6 seconds and ends at 77.8 seconds, taking a total of 37.1823 seconds to process the records. The output concludes with a message indicating the completion of the process and the finalization of MPI.



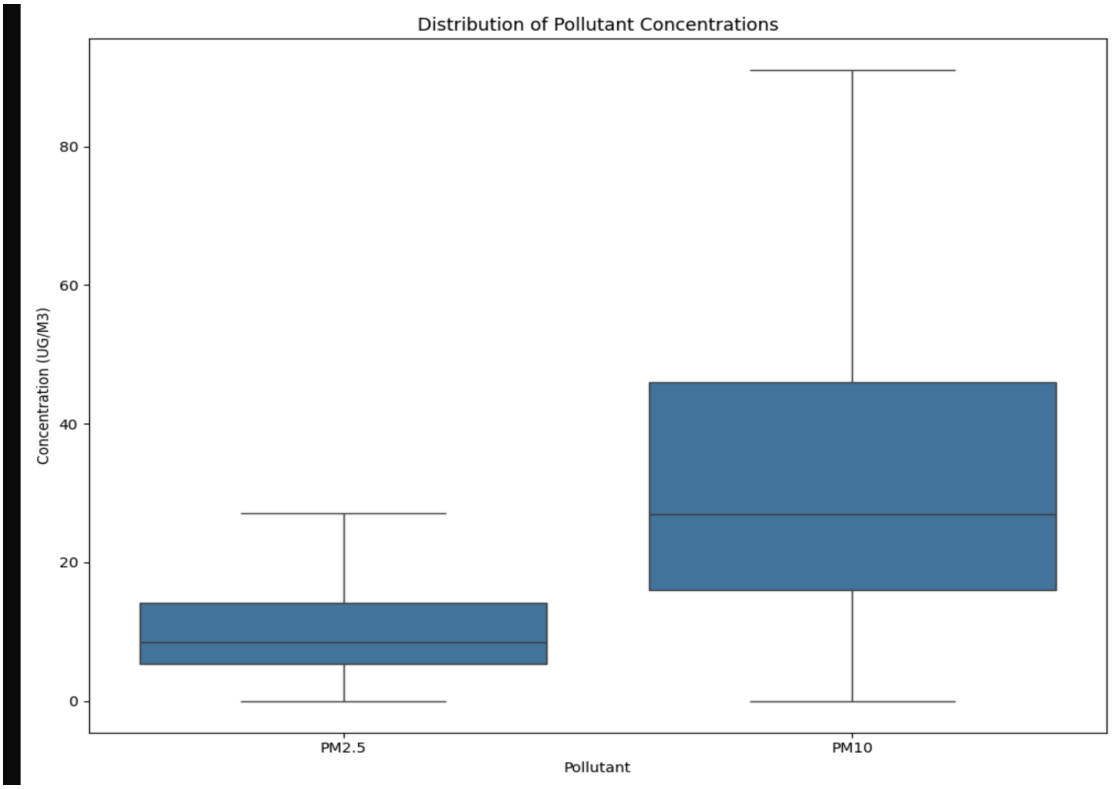
Visualizations of AQI data:



The above graph depicts a time series of Air Quality Index (AQI) readings over a period from early August to late September 2020. It shows fluctuations in AQI, with values peaking above 100 and dipping near 75. The overall trend appears to be slightly decreasing over time, with frequent short-term variations. This visualization helps in identifying patterns in air quality over the given time frame.



The above map displays the Air Quality Index (AQI) by location across various states, represented by color-coded dots indicating the AQI levels. The gradient color scale ranges from 0, signifying low pollution levels, to 90, indicating higher pollution levels. The spread of dots across the map illustrates the geographic distribution of air quality, with some areas exhibiting denser concentrations of higher AQI values. This visual representation aids in identifying regional air quality patterns and potential pollution hotspots.



The above graph is a box-and-whisker plot that compares the distributions of two types of air pollutant concentrations: PM2.5 and PM10. The y-axis represents the concentration in micrograms per cubic meter (μg/m³), and the x-axis categorizes the pollutants. PM2.5 has a much lower range of concentrations and a smaller interquartile range (IQR), showing less variability in the measured values compared to PM10, which has a wider range and IQR, indicating more variability. The longer whiskers on the PM10 box suggest that PM10 concentrations have more outliers or more widespread data points than PM2.5. The median concentration of PM10 seems higher than PM2.5, as indicated by the line within each box.

Member Contributions -

- Harshil Patel -
  Manage the Python script delete_files.py to ensure that unwanted files are efficiently deleted.
  Oversee the generation and validation of CSV files containing records of processing times (records_per_second_process_*.csv).
  Develop and refine data visualization scripts for the results of parallel computations, ensuring accurate representation of the processing metrics.

- Dheer Jain
  Handle the compilation of the C++ program, ensuring all dependencies are met and the application is built correctly.
  Configure and execute the MPI-based C++ program with the user-specified number of processes, monitoring for correct parallel execution.
  Establish scripts for logging performance metrics that will be collected by the parallel C++ program.

- Mahek Virani
  Lead the Python script csv_merger.py to merge CSV files post-processing, facilitating effective data aggregation.
  Set up the Flask application environment variable and ensure the server runs correctly by managing the background process.
  Coordinate with team members to collect data from parallel processes for server-side use.

- Sri Charan Reddy Mallu
  Guide the parallel processing within the C++ code, ensuring efficient utilization of resources and correct implementation of parallel algorithms.
  Supervise data export functionalities to ensure that processed data is saved correctly and is accessible for further analysis.
  Collaborate with Harshil and Dheer to analyze the parallel computing metrics for performance improvements.

Conclusion -

In conclusion, the project adeptly demonstrates the utilization of advanced airflow data management practices, underpinned by high-performance computing strategies to expedite the processing of extensive airflow datasets. Leveraging MPI facilitated the effective distribution of computations across a multitude of processors, while OpenMP was instrumental in fine-tuning parallel processing within each processor. This sophisticated orchestration has culminated in a significant diminution of data processing intervals, concurrently enhancing the scalability and adaptability of our system. These improvements are crucial for conducting intricate simulations related to time series and spatial analysis within the realm of airflow studies, paving the way for groundbreaking insights in environmental research and applications.