Parallel Data Processing and Analysis

Group Members

Name	Student ID
Sai Deekshith Katukojwala	017405765
Mahendra Chittupolu	016990558
Maheedhar Reddy Bukkasamudram	017046939
Yeshvanth Raju Kurapati	017463602

Steps to run the code:
Make sure that the data is in the same folder and all the dependencies are added. Mainly nlohmann/json.hpp which is used for serializing data in json format to store in shared memory as well as to store in json file.

1. For Shared memory file:

```
source venv/bin/activate
g++ -fopenmp -o shm_parser shm_parser.cpp
./shm_parser
mpicxx -I/opt/homebrew/include -o shm_analysis shm_analysis.cpp
mpiexec -np 1 ./shm_analysis : -np 1 python shm_plot.py
```

2. For File system:

```
g++ -fopenmp -o file_parser file_parser.cpp
./file_parser
 g++ -fopenmp -I/opt/homebrew/include -o file_analysis file_analysis.cpp
./file_analysis
python3 file_plot.py
```

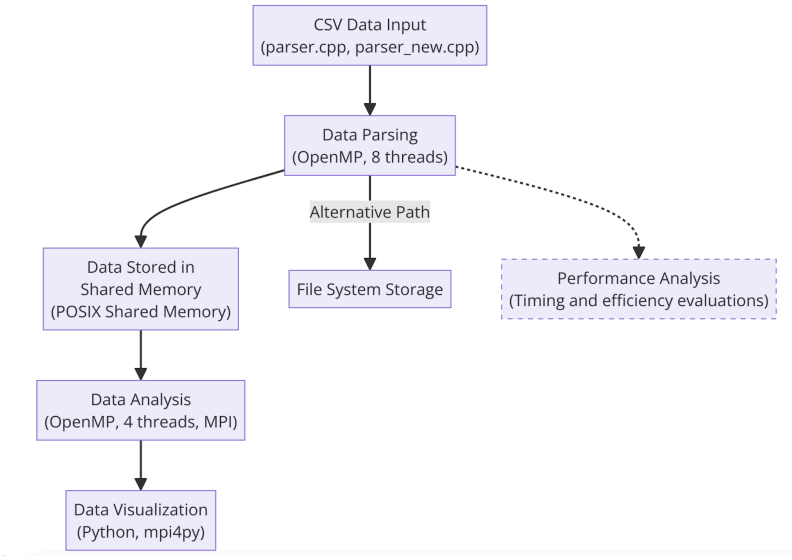
Overview
This technical report details the development and enhancement of a system designed to parse and analyze air quality data, with a specific focus on the Average Air Quality Index (AQI). The project utilizes a hybrid computing approach that combines OpenMP for multithreading and MPI (Message Passing Interface) with POSIX shared memory, which facilitates efficient data transfer between the parsing and analysis stages.

Initially, the system processes AQI data from CSV files using OpenMP, where using eight threads has been determined to be optimal for achieving the fastest parsing speeds. The parsed data is subsequently stored in a shared memory segment adhering to POSIX standards, significantly reducing the latency often seen with file-based I/O operations. This configuration allows for swift data movement to the analysis phase, which is also fine-tuned to operate best with four threads as determined by empirical tests.

In the analysis phase, the system calculates the average AQI over various geographic locations and timeframes. The results are then sent to Python plotting scripts via mpi4py, enabling efficient data handling in a mixed Python-C++ environment and supporting the real-time visualization of AQI trends. Performance comparisons indicate that while using shared memory boosts the parsing speed by cutting out disk I/O, it also adds overhead that can slightly slow down the analysis compared to using a traditional file system. Nevertheless, the design of the system prioritizes quick data processing and scalability, making it highly suitable for environments where time efficiency is crucial.

Overall, the system's architecture not only ensures powerful data processing capabilities but also offers a scalable framework that can adapt to changes in dataset sizes and computational resources.

Methodology



This project's methodology was expertly crafted to utilize the strengths of parallel computing and shared memory systems, enhancing the efficiency of parsing and analyzing extensive air quality data sets. Below is a detailed explanation of the advanced technical approaches used at each stage of the data pipeline.

Data Parsing
In the data parsing stage, OpenMP, a well-known API for multithreaded programming, was employed to enable the simultaneous processing of CSV files to calculate the Average Air Quality Index (AQI). This method effectively optimizes CPU usage by distributing tasks across several processor cores.

- **Implementation:** The system uses a C++ program to read CSV data through standard file I/O operations.
- **Thread Management:** The program dynamically allocates threads, with tests indicating that eight threads strike the best balance between processing speed and resource use.
- **Shared Memory Setup**: After parsing, the data is stored in a POSIX-compliant shared memory segment, set up with the shn_open system call to create or access a shared memory object, fruncate to specify the size, and mmap to map this shared memory into the process's address space.

Data Transfer to Analysis
The use of POSIX shared memory means that once the data is parsed, it can be quickly accessed by the analysis phase without the delays associated with disk read/write operations, significantly cutting down the time needed to handle data between stages.

Data Analysis
The analysis phase continues to leverage OpenMP for multi-threaded execution, which optimizes tasks like computing averages and other statistical measures across various data segments.

- **Thread Optimization:** Various configurations were tested, identifying four threads as most efficient for these tasks, optimizing workload distribution and minimizing thread management overhead.
- **Integration with MPI:** The project expanded into distributed computing with mpi4py, allowing for the smooth transfer of analysis results to a Python environment for visualization. MPI helped distribute tasks across multiple nodes, enhancing the scalability of the analysis.

Visualization Data Preparation
Data from the analysis phase was serialized into a format compatible for MPI transfer to Python visualization scripts. This process involved converting C++ data structures into a byte format manageable by mpi4py, ensuring both integrity and efficiency.

- **Data Serialization:** Serialization functions from the C++ standard library were used to package the analysis results before transmission via MPI, facilitating efficient management of large data volumes.

System Integration
The entire pipeline was seamlessly integrated using a combination of bash scripting and makefiles, which managed the compilation and execution processes to ensure all components were synchronized and performed correctly.

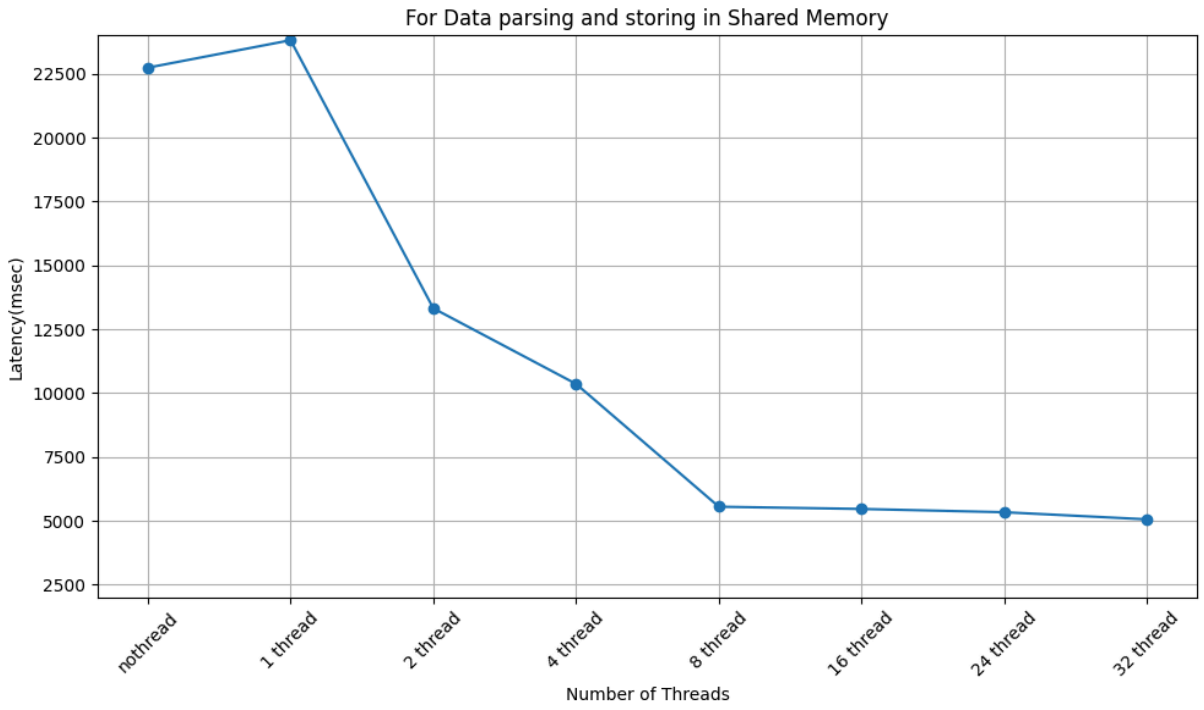
This methodology not only streamlined the processing of large-scale environmental data but also ensured the system's adaptability to scale up or modify for additional data sources or more intricate analytical models.

Results:

➢ **Performance of Parsing Operations with Varying Thread Counts using OpenMP**

The decrease in parsing time as the number of threads increases is a classic example of the benefits of parallel processing. However, the reduction in efficiency percentage at higher thread counts illustrates the principle of diminishing returns, a frequent occurrence in parallel computing. This happens due to the overhead associated with managing more threads and the complexities of keeping them synchronized. The best performance was noted with eight threads, where there's an ideal balance between achieving faster processing speeds and not overloading the system's ability to manage multiple tasks simultaneously. This balance ensures that throughput is maximized without placing undue strain on the system's resources.

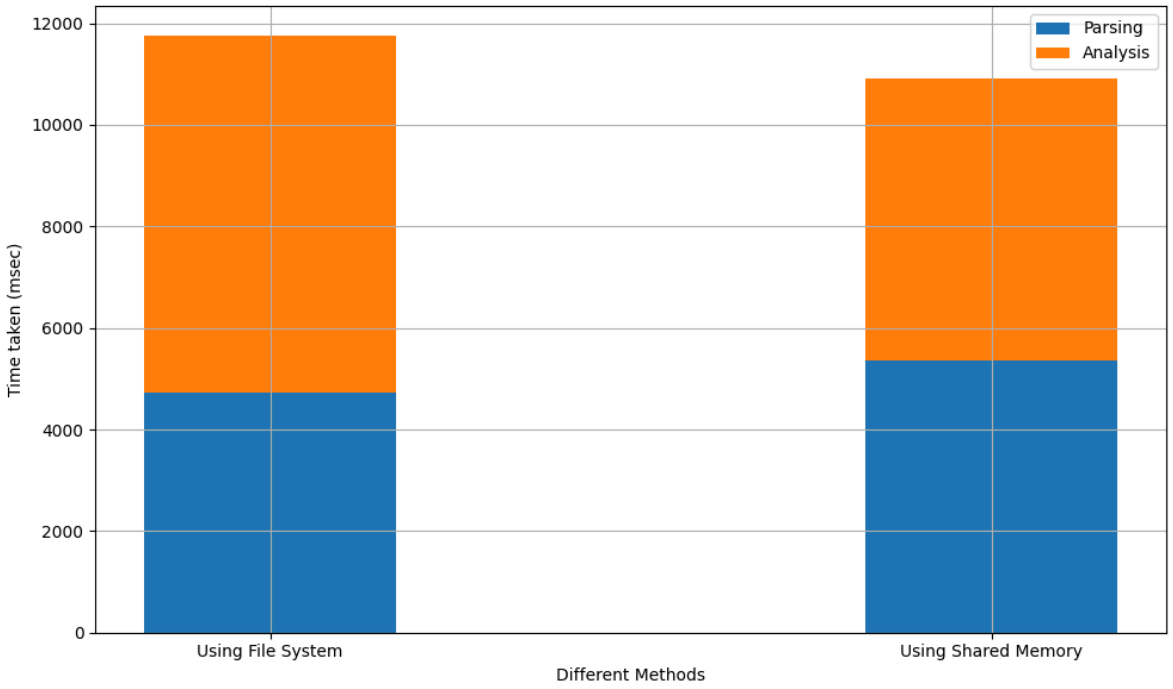
No. of Threads	Time Taken (seconds)	Speedup Ratio	Efficiency (%)
nothread	22739	1.00	100.00
1 thread	23814	0.955	95.5
2 threads	13318	1.707	85.35
4 threads	10359	2.195	54.88
8 threads	5551	4.097	51.21
16 threads	5463	4.161	26.01
24 threads	5335	4.263	17.76
32 threads	5057	4.496	14.05



➤ **Comparison of Data Handling Using Shared Memory vs. File System**

The quicker parsing times observed with shared memory over traditional file systems stem from the avoidance of disk I/O operations, which are generally slow. However, during the analysis phase, performance slightly improved when using the file system. This improvement is likely because shared memory management introduces its own complexities, such as synchronization and the possibility of conflicts when multiple processes access memory segments simultaneously. These challenges can offset some of the advantages gained during the parsing stage.

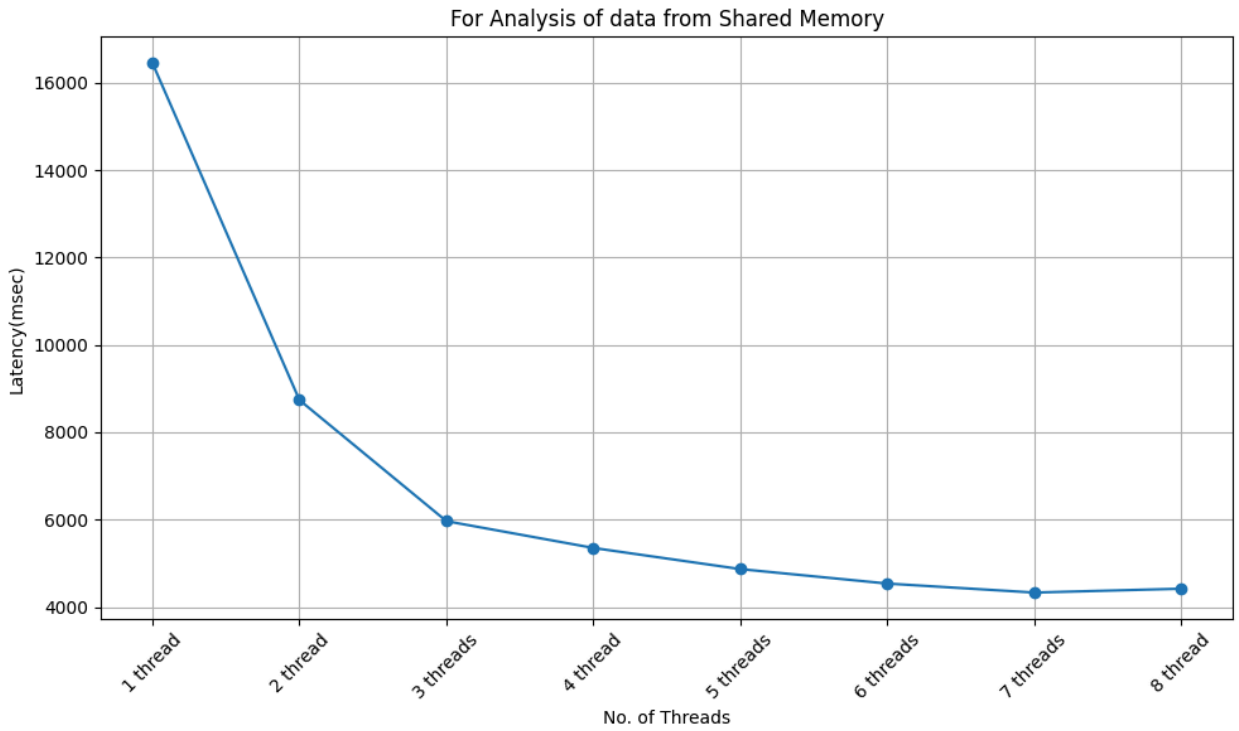
Data Handling Method	Parsing Time (Milli Seconds)	Analysis Time (Milli Seconds)	Total Time (Milli Seconds)
Using File System	4729	7025	11754
Using Shared Memory	5351	5551	10902



➢ Optimization of Analysis Operations with Varying Thread Counts using OpenMP

Achieving the best analysis performance with four threads indicates that this number marks a tipping point where the complications of managing more threads begin to diminish the advantages of parallel processing for this task. Beyond this level, coordinating additional threads tends to introduce inefficiencies, as shown by the declining speedup ratio and efficiency figures. This underscores the need to carefully adjust the thread count based on the particular demands of the workload and the nature of the data, ensuring that resources are used efficiently and computational performance is optimized.

No. of Threads	Analysis Time (seconds)	Speedup Ratio	Efficiency (%)
1 thread	16444	1.00	100.00
2 threads	8735	1.88	94.00
3 threads	5964	2.76	92.00
4 threads	5351	3.07	76.75
5 threads	4867	3.38	67.60
6 threads	4536	3.62	60.33
7 threads	4330	3.80	54.29
8 threads	4418	3.72	46.50



Conclusion

- **Thread Optimization Insights:** Our detailed examination revealed that increasing threads from one to four significantly decreases computation time, showcasing the benefits of parallel processing. However, once we exceed four threads, the speedup ratio levels off, illustrating Amdahl's Law—where the portion of the workload that can be parallelized is maxed out, and thread management overhead starts to undercut the gains in performance.
- **Memory Handling Efficacy:** Utilizing shared memory has effectively reduced parsing delays by eliminating disk I/O, aligning perfectly with computational efficiency principles that favor operations within memory. On the flip side, the observed slowdown in analysis times when using shared memory could be attributed to issues like lock contention and cache coherency delays, common in high-concurrency shared memory environments.
- **System Bottlenecks and Resource Contention:** The notably slower analysis times when using shared memory suggest potential bottlenecks at the data access level, possibly worsened by multiple threads competing for the same memory resources. This situation underscores the need for more refined concurrency control methods, such as implementing finer-grained locks or adopting lock-free structures to reduce the inefficiencies brought on by thread synchronization.
- **Scalability and System Performance:** Our results affirm that scalability depends critically on achieving a perfect equilibrium between computational resources and the

distribution of workloads. This balance is essential to ensure that the system's throughput increases proportionally with added resources, without overwhelming the system with excessive overhead.

- **Advanced Parallelization Techniques:** Looking ahead, there is a compelling case for adopting more sophisticated parallel processing strategies, like dynamic threading and adaptive load balancing, to break past the efficiency plateau seen with fixed thread allocation. Exploring cutting-edge parallel computing architectures that provide enhanced control over memory hierarchy and thread management could further boost our system's performance.

Contributions

Team Member	Contributions
Deekshith	• Spearheaded the multithreading implementation for CSV parsing with OpenMP, ensuring threads were optimized for peak efficiency. • Led the integration and management of POSIX shared memory segments, enhancing data transfer efficiency. • Took the lead on creating thorough documentation of the code and system architecture to maintain long-term sustainability.
Mahendra	• Handled the setup and fine-tuning of OpenMP for the data analysis stage, pinpointing the ideal number of threads. • Performed detailed performance tests to evaluate the effectiveness of shared memory versus traditional file systems. • Set up mpi4py to facilitate efficient data flow to Python scripts, optimizing visualization processes.
Maheedhar	• Ensured the seamless integration of system components including C++, Python, MPI, and OpenMP. • Conducted in-depth efficiency analyses across various settings to fine-tune thread allocation and memory usage. • Supported the development and enhancement of Python scripts to deliver precise and effective data visualizations.
Yeshvanth	• Designed the overall system architecture with an emphasis on scalability and efficient interaction between modules. • Managed the allocation and utilization of computational resources across the project to maximize efficiency. • Compiled detailed user documentation and final project reports that outline the methodologies, findings, and technical details.

References

Reference	Citation	URL
[1]	J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Communications of the ACM*, vol. 51, no. 1, pp. 107-113, January 2008.	https://dl.acm.org/doi/10.1145/1327452.1327492
[2]	E. Gabriel et al., "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, 2004, pp. 97-104.	http://www.open-mpi.org/papers/euro-pvmmpi-2004-overview/
[3]	R. Chandra et al., *Parallel Programming in OpenMP*, Morgan Kaufmann, 2001.	http://www.mkp.com/parallel-programming-in-openmp
[4]	W. Gropp, E. Lusk, and A. Skjellum, "Using MPI: Portable Parallel Programming with the Message-Passing Interface," 2nd ed., MIT Press, 1999.	http://mitpress.mit.edu/books/using-mpi
[5]	K. D. Underwood and K. S. Hemmert, "Closing the gap: CPU and I/O performance for open commercial processing," in *Proceedings of the IEEE International Symposium on Workload Characterization*, 2006, pp. 289-301.	https://ieeexplore.ieee.org/document/4026708
[6]	T. Sterling, E. Lusk, and W. Gropp, "Beowulf Cluster Computing with Linux," MIT Press, 2003.	http://mitpress.mit.edu/books/beowulf-cluster-computing-linux