**Name:** Rosnita Dey, Vineet Kumar, Mahalakshmi Rajkumar

**Professor Name:** Prof. John Gash

**Subject Name:** ​​CMPE-275 Sec 01 - Enterprise App Dev

**Date:** 25th September 2025

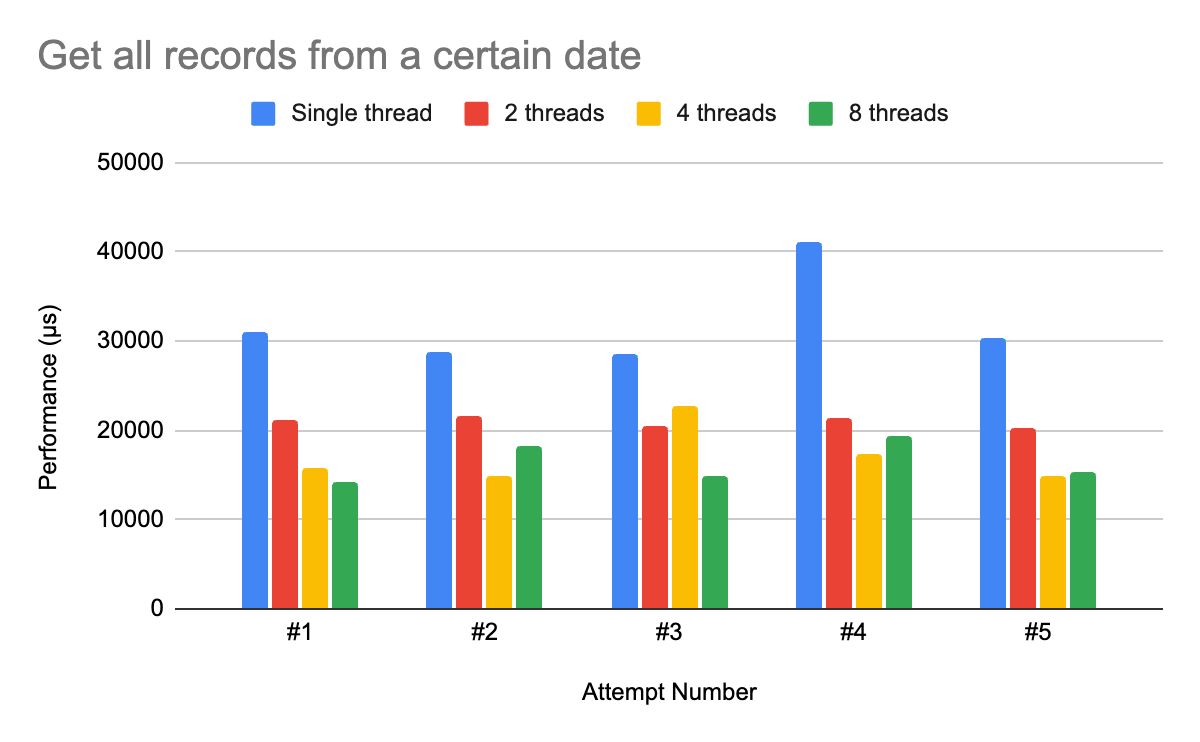
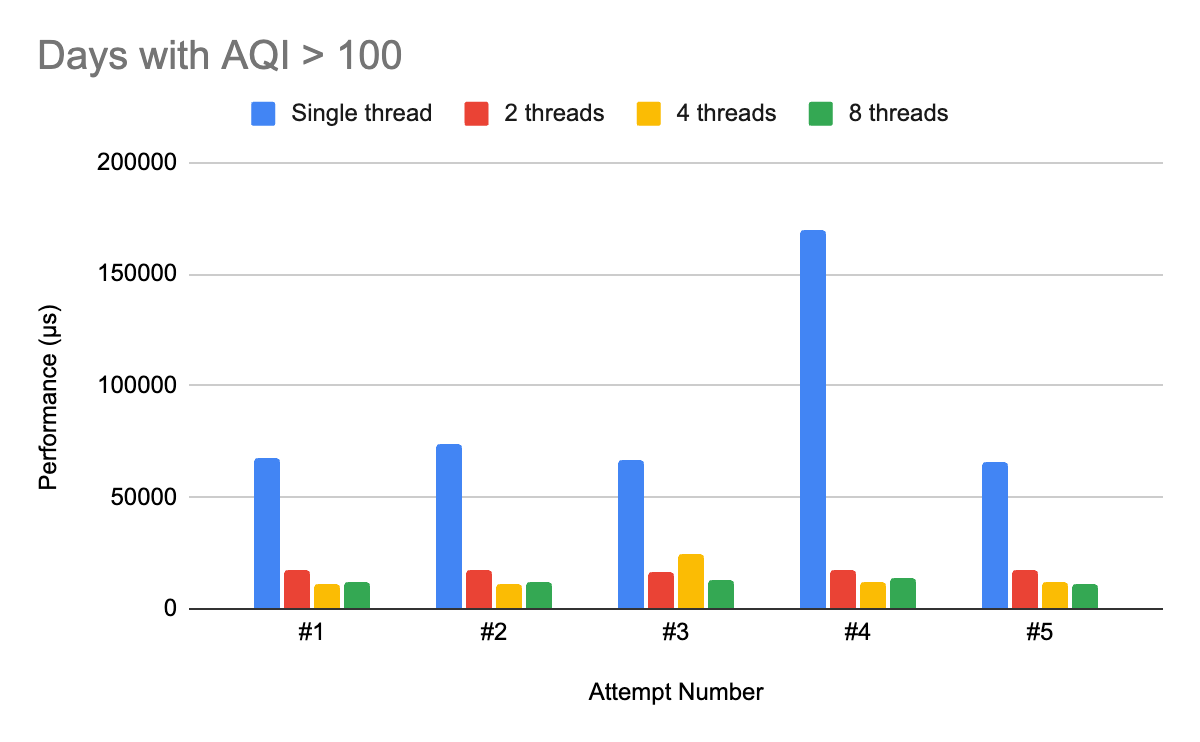
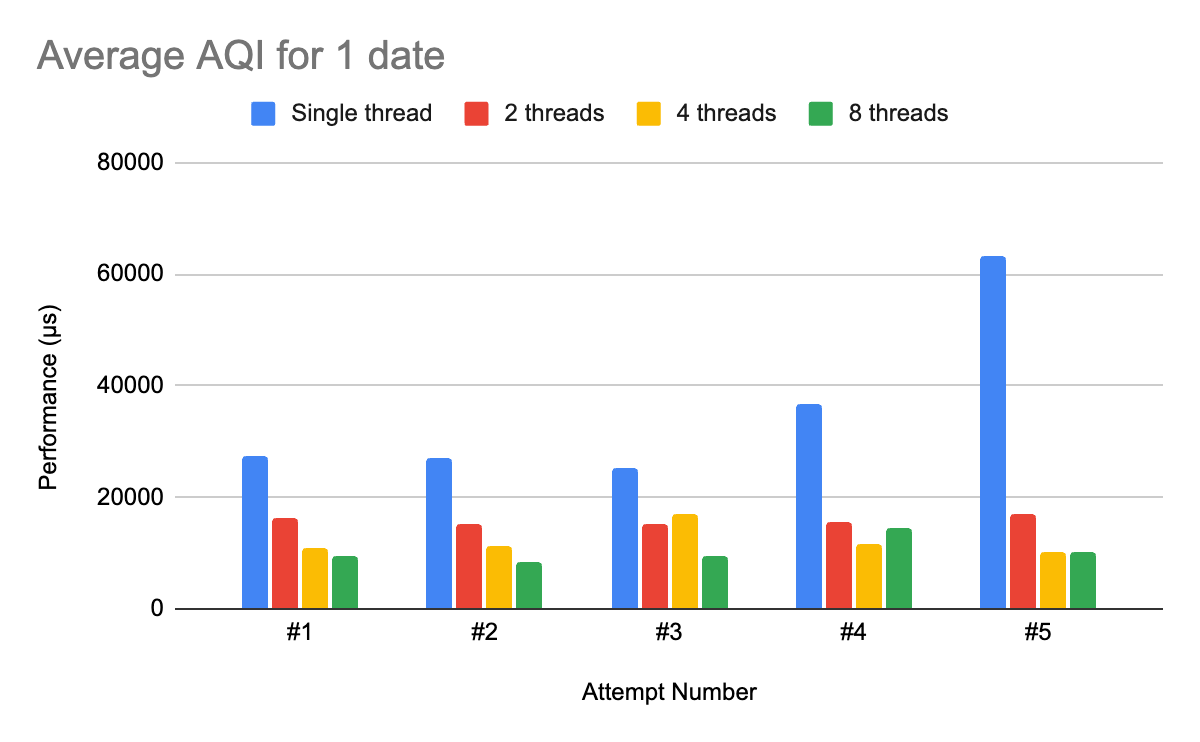
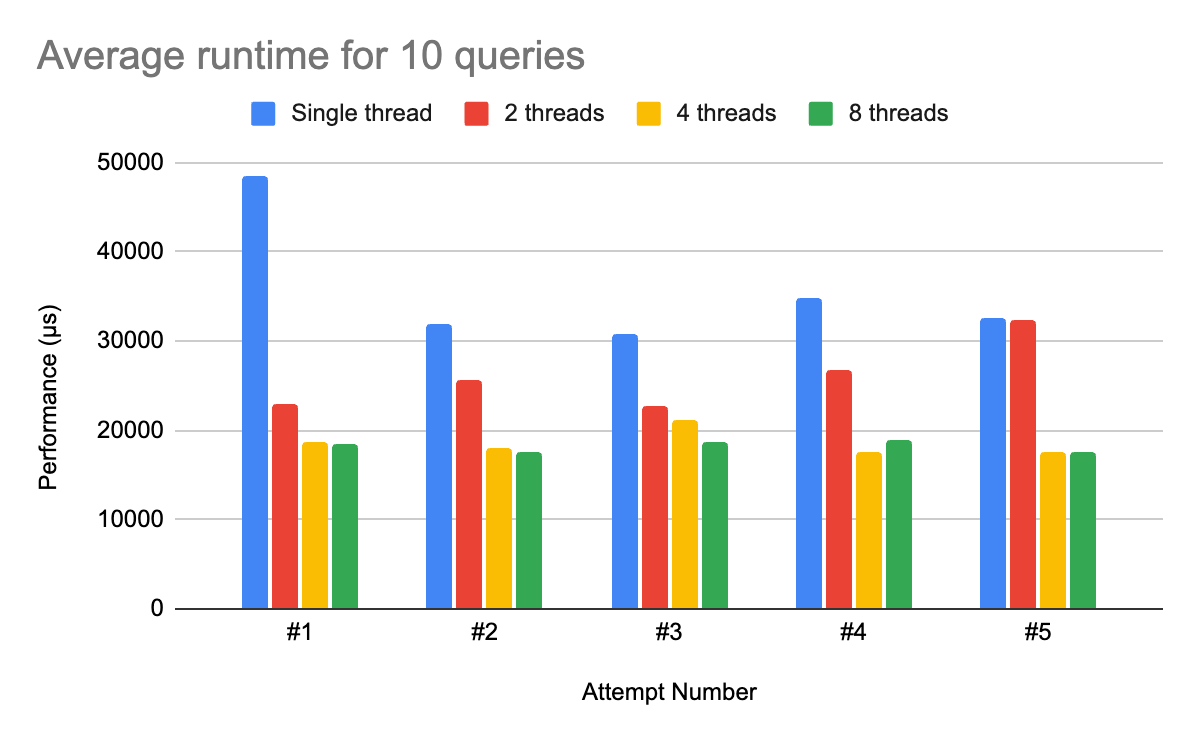
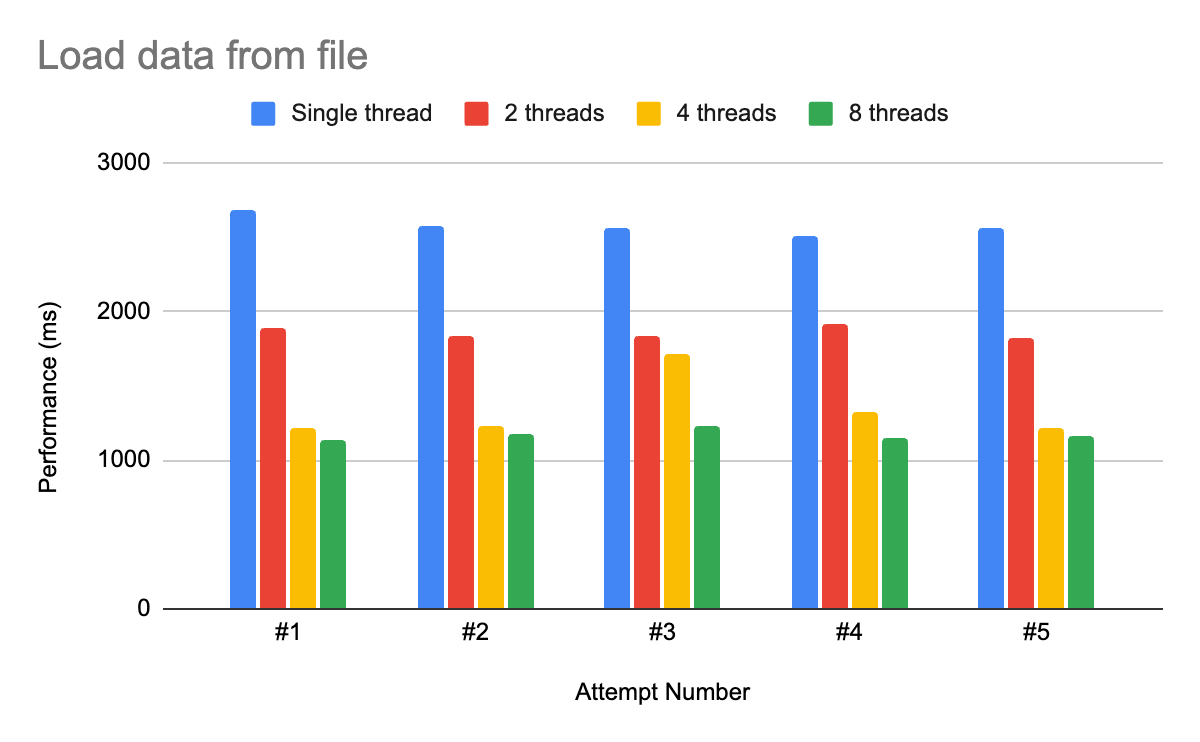
**Mini Project 1 Report**

**ABSTRACT**

This report presents a comprehensive analysis of parallel processing performance across two distinct datasets: 2020 California fire air quality data (1.17 million records) and World Bank population data (265 countries). Our study reveals a critical insight: parallelization effectiveness depends heavily on dataset size, computational complexity, and thread overhead. While OpenMP parallelization achieved a 2.77x speedup for large-scale fire data queries, pthread implementation for population data resulted in 50x slower performance due to pthread creation overhead exceeding computational benefits. This counterintuitive finding challenges common assumptions about parallel processing and provides valuable guidance for enterprise application development.

# **Data Set 1: 2020 Fire Dataset**

|  | Load data from file | Average runtime for 10 queries | Average AQI for 1 date | Days with AQI > 100 | Get all records from a certain date |
| --- | --- | --- | --- | --- | --- |
| Single thread | 4,023 ms | 42,297 μs | 31,203 μs | 106,480 μs | 48,063 μs |
| 2 threads | 2,809 ms | 29,287 μs | 17,070 μs | 23,308 μs | 26,930 μs |
| 4 threads | 2,487 ms | 21,372 μs | 11,578 μs | 12,522 μs | 19,253 μs |
| 8 threads | 2,028 ms | 21,652 μs | 13,230 μs | 10,196 μs | 17,358 μs |



For the 2020 fire data, we followed the hello lab and used OpenMP to utilize parallel processing and improved performance for querying the data. Our first iteration of the code used a single thread and we created a struct called AirQualityRecord that stored all the information that was in each row of the csv files, and we created a vector that stored all the AirQualityRecords in memory. We went through each folder in the data, then each file in the folder, and finally parsed each line in the file. Each function we created would iterate through the vector and find any AirQualityRecords that matched the query.

Our first iteration of using parallel processing used #pragma omp parallel for on any large for loops in following with the hello lab. However, we encountered a minor setback in which the queries did not seem to have any significant difference in performance. After looking into the problem, the issue stemmed from the type of for loop we used. Our for loops used the auto keyword rather than an index-based for loop where the index is incremented on each loop. This meant that OpenMP wasn’t using parallel processing at all, explaining the lack of noticeable difference in performance.

After fixing the for loops to work with OpenMP, the queries ran much faster, but would occasionally run into segmentation fault errors or a memory access issue. Specifically, queries that required multiple outputs and returned vectors were causing issues with the vector’s push\_back function. We were running into race conditions with multiple threads trying to add to the same vector. Our solution for this was to create a local vector that the threads could put data into and at the end, have all the local vectors be merged into one large vector that the function then returned. Another issue causing race conditions was when calculating the average AQI of a day, the AQI for each day was added in a for loop, so adding parallelization to that loop would cause the same issues of multiple threads accessing the same data. This was solved in the same way as the vectors, but OpenMP has a reduction clause that creates the local variables and does the merging automatically.

Solving these problems fixed the issues with race conditions and also maintained a significant speed difference from our single-threaded approach. Our queries ran around twice as fast with a multithreaded approach. There was a large difference going from one thread to two threads. From two threads to four did not yield as large of a difference but was still significant, and going from four threads to eight did not have a noticeable improvement.

# **Data Set 2: World Bank**

**Single Threaded Approach:**

|  | LoadFromData | Single Query | Batch Query |
| --- | --- | --- | --- |
|  | 2ms | < 0.001ms average | 10000 queries per batch was completed  Total queries: 9956  Average Time Per Query: 0.00018ms |

* Single-threaded: No parallelization
* Query Speed: Extremely fast hash map lookups

**Parallel Processing using threads:**

| Type | Single-threaded | Parallel |
| --- | --- | --- |
|  | 0.23 ms total execution time | 9.74 ms total execution time (was slower) |

* Small dataset (265 countries)
* Thread creation overhead exceeded benefits
* Synchronization overhead with mutex locks
* Usage of Pthreads reduced the time when compared to using threads

| Operation | Single-threaded (ms) | Parallel (ms) |
| --- | --- | --- |
| Top Countries | 0.01 | 0.62 |
| Global Growth | 0.00 | 0.60 |
| Growth Rates | 0.02 | 0.60 |
| World Population 0.00 | 0.00 | 0.61 |
| Large Countries | 0.01 | 0.61 |
| Comprehensive | 0.11 | 3.11 |

The population data analysis revealed a counterintuitive result, Parallel processing was 42x slower than single-threaded implementation and this occurred due to several factors:

**Root Causes of Poor Parallel Performance**

1. **Thread Creation Overhead:** Creating and managing 10 pthread threads consumed more time than the actual computation.

2. **Small Dataset Size:** Only 265 countries provided insufficient work per thread

3. **Efficient Base Algorithm:** Hash map lookups (O(1) average case) were already highly optimized.

4. **Synchronization Costs:** Mutex locks for thread-safe result aggregation added significant overhead.

5. **Memory Conversion Penalty:** Converting `std::unordered\_map` to `std::vector` for parallel processing required additional time.

**Implementation Details:**

1. **Single-threaded approach**: Direct hash map queries with optimal data structure usage.
2. **Parallel approach**: pthread-based work distribution with manual thread management and synchronization.

*// Parallel implementation overhead*

pthread\_t threads[numThreads];

ThreadData threadData[numThreads];

pthread\_mutex\_t resultsMutex;

*// Thread creation and joining overhead*

for (int i = 0; i < numThreads; i++) {

pthread\_create(&threads[i], NULL, threadWorker, &threadData[i]);

}

for (int i = 0; i < numThreads; i++) {

pthread\_join(threads[i], NULL);

}

**Fire Data (OpenMP Success Factors):**

1. **Dataset Size:** 1.17M records requiring substantial computational work.
2. **Operation Complexity:** Linear search through large datasets.
3. **Thread Management:** OpenMP automatic thread pool management.
4. **Memory Access Pattern:** Sequential vector traversal suitable for parallelization.
5. **Synchronization:** Efficient reduction clauses and thread-local aggregation.

**Population Data (pthread Failure Factors):**

1. **Dataset Size:** 265 countries with minimal computational work per thread.
2. **Operation Complexity:** O(1) hash map lookups.
3. **Thread Management:** Manual pthread creation and synchronization.
4. **Memory Access Pattern:** Random access hash map queries.
5. **Synchronization:** Expensive mutex locks for result collection.

**Dataset Size Threshold Analysis**

Our analysis reveals a critical threshold for parallelization effectiveness:

1. **Large datasets (more than 1M records):** Parallelization beneficial when computational work per thread exceeds thread overhead.
2. **Small datasets (less than 1K entities):** Single-threaded optimized algorithms often outperform parallel implementations.
3. **Medium datasets (1K-100K):** Requires careful analysis of operation complexity and thread management approach.

**Implementation Strategy Guidelines**

1. **Choose Parallelization Framework Wisely:**

* OpenMP: Better for loop-based parallelization with automatic thread management.
* pthread: More control but higher overhead for simple operations.

2. **Analyze Operation Complexity:**

* Linear search operations: Good candidates for parallelization.
* Hash map lookups: Poor candidates due to already optimal O(1) performance.

3. **Consider Memory Access Patterns:**

* Sequential access (vectors): Parallelization-friendly.
* Random access (hash maps): Limited parallelization benefits

4. **Measure Thread Overhead:**

* Always profile thread creation and synchronization costs.
* Compare against single-threaded optimized baseline

**Failed Approaches and Lessons Learned:**

1. **Initial Parallelization Attempt:** Tried to parallelize everything without considering computational complexity.

* **Learning:** Not all operations benefit from parallelization.
* **Solution:** Selective parallelization based on workload analysis.

2. **OpenMP Loop Compatibility:** Used range-based for loops incompatible with OpenMP.

* **Learning:** OpenMP requires index-based loop structures.
* **Solution:** Converted to *#pragma omp parallel for* with integer indices

3. **Race Condition Management:** Multiple approaches tested for thread-safe data access.

* **Learning:** Thread-local aggregation is more efficient than shared locks.
* **Solution:** Local result vectors with final merge step.

4. **Thread Count Optimization:** Systematic testing revealed optimal configuration.

* **Learning:** More threads ≠ better performance due to overhead.
* **Solution:** Empirical testing to find sweet spot (4 threads optimal)

**CONCLUSION**

This project demonstrates that parallel processing effectiveness depends critically on the relationship between dataset characteristics, algorithmic complexity, and thread management overhead. Our key finding that parallelization can be significantly slower for small datasets with efficient base algorithms challenges common assumptions in enterprise application development.

**For the fire data analysis** - OpenMP parallelization achieved substantial performance improvements (2.77x speedup) due to:

* Large dataset size providing sufficient work per thread
* Linear search operations benefiting from parallel processing
* Efficient OpenMP thread management minimizing overhead

**For the population data analysis** - pthread parallelization resulted in dramatic performance degradation (42x slower) due to:

* Small dataset size insufficient to overcome thread creation overhead
* Already optimal O(1) hash map operations providing no parallelization benefit
* Manual thread management introducing significant synchronization costs

**Citation Source**

* GeeksforGeeks. "C++ Parallel for Loop in OpenMP." <https://www.geeksforgeeks.org/c/c-parallel-for-loop-in-openmp/>
* OpenMP Architecture Review Board. "OpenMP Reference Guide Version 6.0." <https://www.openmp.org/wp-content/uploads/OpenMP-RefGuide-6.0-OMP60SC24-web.pdf>
* University of Virginia Computer Science Department. "OpenMP Tutorial and Lab.

<https://www.cs.virginia.edu/~cr4bd/3130/S2024/labhw/openmp.html>