REPORT

Distributed Sorting System Performance:

Implementation Analysis:

Approach: Limited-Depth Multithreaded Merge Sort:

In this approach, each recursive call to the merge sort function creates a new thread until a specified depth limit is reached. Beyond that depth, the merge sort function continues sequentially without spawning additional threads. This setup leverages parallelism at higher levels of the merge sort tree, where there is more work to divide between threads, while keeping resource usage manageable.

Pros:

- Improved performance on multicore systems
- Reduced Thread Management Overhead
- Predictable Memory Usage

cons:

- Less Granular Parallelism
- Potential Underutilization
- Complex Implementation

Explanation of Count Sort with Threads:

Count Sort works by creating a frequency array to count the occurrences of each unique mapped value within the sorting range. Here, multithreading divides the array into segments, and each thread counts occurrences in its segment independently.

Pros:

- Parallel Processing
- Reduced Memory Usage

Cons:

- Synchronizing counts across multiple threads and managing thread arguments increases complexity.
- For small arrays, the overhead of thread creation and management might outweigh performance gains from parallelism.

Execution time analysis:

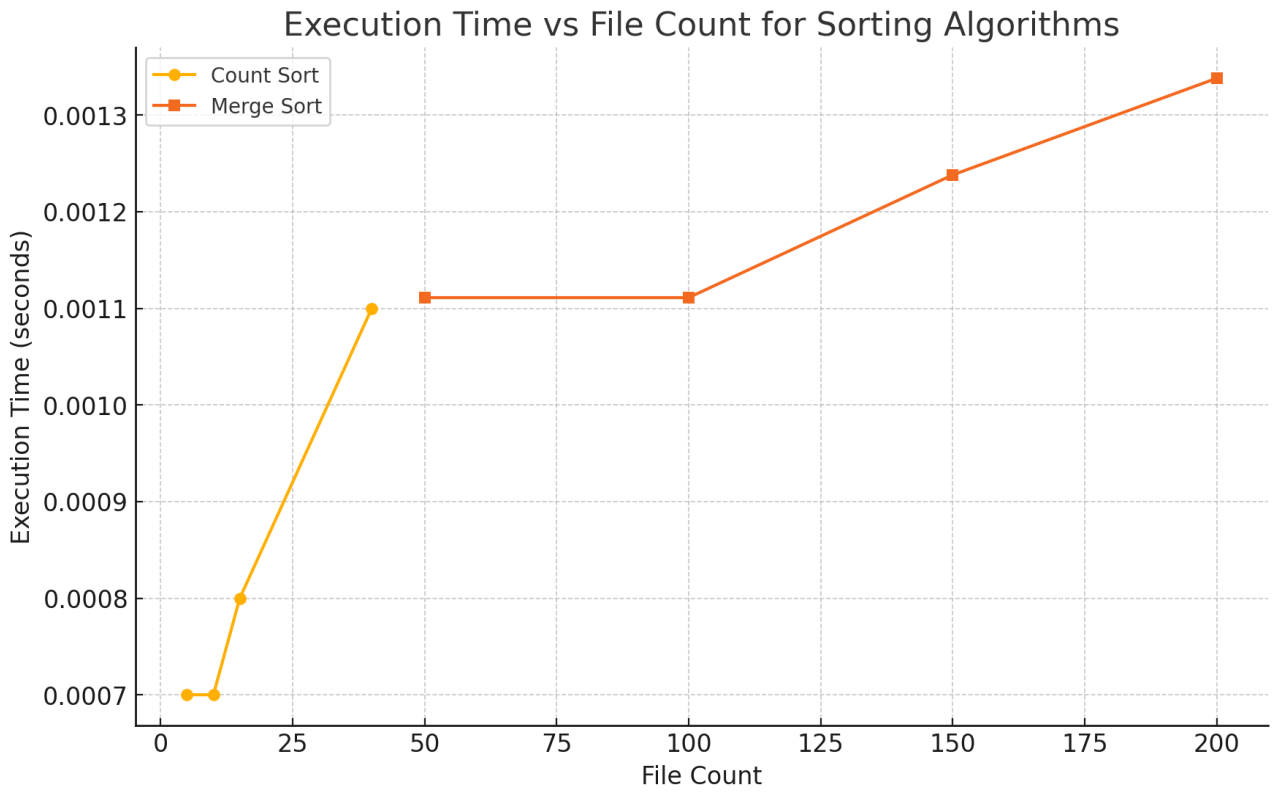
Count Sort:

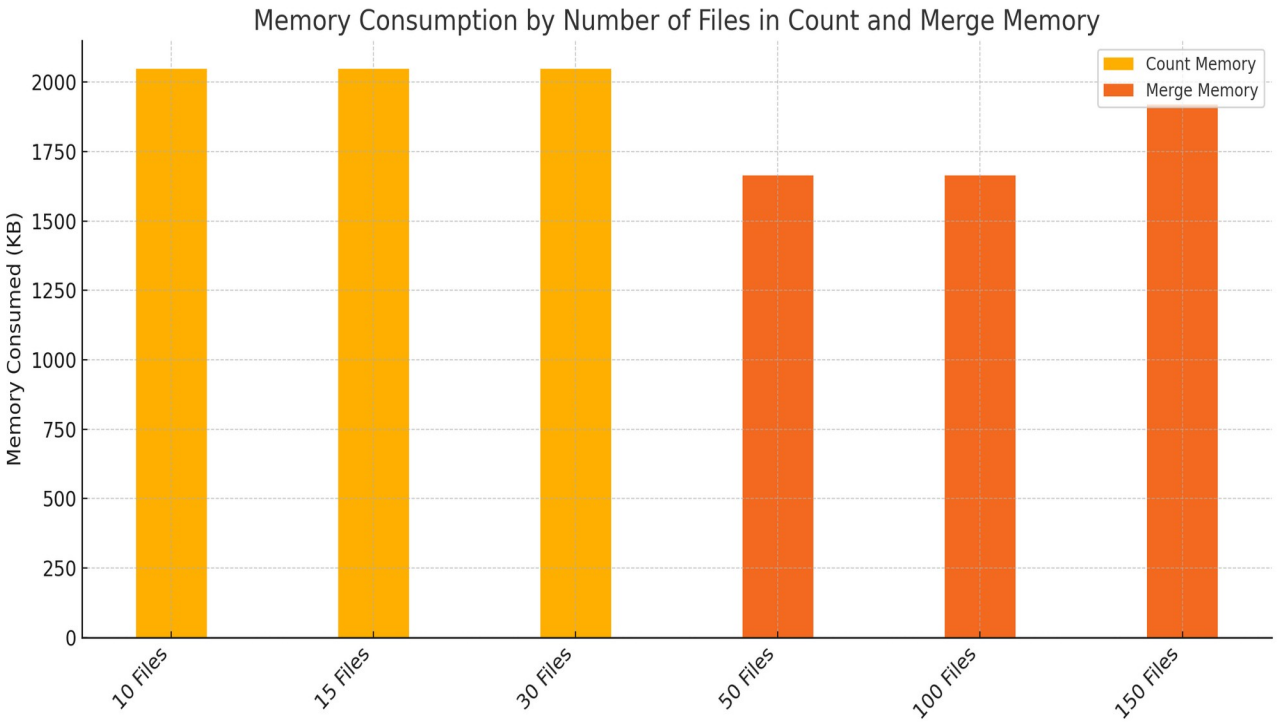
Number of files	Time in seconds
5	0.0007
10	0.0007
15	0.0008
40	0.0011

Merge Sort:

Number of files	Time in seconds
50	0.001111
100	0.001111
150	0.001238
200	0.001338

GRAPH:


Execution Time vs File Count for Sorting Algorithms
Count Sort
Merge Sort
Execution Time (seconds)
0.0013
0.0012
0.0011
0.0010
0.0009
0.0008
0.0007
0
25
50
75
100
125
150
175
200
File Count


Memory Consumption by Number of Files in Count and Merge Memory
Count Memory
Merge Memory
Memory Consumed (KB)
2000
1750
1500
1250
1000
750
500
250
0
10 Files
15 Files
30 Files
50 Files
100 Files
150 Files

Summary:

This project implements and analyzes both distributed merge sort and distributed count sort, focusing on how each method performs with different file counts and sorting attributes (Name, ID, Timestamp). Each method leverages multithreading to handle large data efficiently, distributing tasks across threads to reduce runtime.

Potential Optimizations for Large Datasets:

- Adjust count sort's value mapping or implement a hashing mechanism to constrain the count array size for larger value ranges.

Copy-On-Write (COW) Fork Performance Analysis:

Page Fault Frequency:

Read Only: The Page fault count is 0 since neither of the parent or child are modifying the page so no voilation of flags which doesnot cause any page fault

Modifying the Page:

Test	Number of Page Faults
Simple Test	3
Three Test	8193
File Test	5

Benefits of Copy-On-Write (COW) Fork:

- **Memory Conservation:** COW helps conserve memory by delaying the copying of pages until they are modified.

- **Performance Improvement**: Instead of copying entire memory spaces for child processes during a fork, which can be expensive in terms of both time and space, COW only requires the creation of a new page table entry that marks pages as read-only for both processes.
-

Areas for Further Optimization:

- **Lazy Page Copying**: Instead of copying the page right away, the system could **defer the copying** (do it later or in the background). This would allow the system to copy multiple pages at once when it has more resources, reducing the overhead of copying each page one by one.