OSN MP3 Report

Karthik Venkat Malavathula and Ansh Acharya

November 2024

1 Distributed Sorting System Performance

1.1 Implementation Analysis:

1.1.1 Distributed Merge Sort

We have implemented Merge sort that each recursive call of the function is computed by a separate thread. We chose this method since it was pretty easy to understand of along with being efficient.

Pros:

- Utilizing multiple threads can significantly speed up the merge sort process, especially on multi-core processors. Each merge operation can be handled concurrently, leading to faster overall sorting times.
- The approach can scale well with the number of available CPU cores. More threads can lead to better utilization of CPU resources.
- In a multi-threaded environment, other processes or operations can remain responsive as the sorting can be performed in the background.

Cons:

- Creating and managing threads incurs overhead. The cost of thread creation, context switching, and synchronization can sometimes outweigh the benefits, especially for small data sets.
- Multiple threads can lead to contention for shared resources (e.g., memory bandwidth), which can reduce the performance gains from parallelism.

1.1.2 Distributed count Sort

Let index of the file be the nth location of the file when given input. We have implemented countsort by creating an array of structs that contains two entities. One is the value, and the other entity is an integer array of size max frequency that we have assumed to be the maximum number of common elements in the table. We also create a large array X that we can use for countsort. Then, depending on Name, Id, or Timestamp, we populate X based on values that

map each of these entities (Name, Id, Timestamp) to an integer. Then, we split the array and call threads to count the frequency of each element in array X. Then we can populate the array of structs with the values and corresponding index of the file. Then, we loop through this array of structs and print the details of the corresponding files.

Pros:

- Distributing the Count Sort algorithm across multiple nodes can lead to substantial speedup for very large data sets, especially when the count arrays and data need to be split and processed independently.
- This method scales effectively with the number of available nodes, as each node can handle part of the counting or accumulation phase independently, leading to efficient parallelization.
- Distributed Count Sort can handle data sizes that would otherwise be too large for a single machine, as the distribution reduces memory limitations.

Cons:

- Network overhead can become a bottleneck, as data and count arrays must be communicated between nodes. This overhead can reduce the efficiency of the algorithm, especially with frequent communication or large data exchanges.
- Synchronization between nodes can incur delays, particularly when nodes need to aggregate or redistribute counts. This can lead to latency that diminishes the performance gains from distribution.
- Distributed Count Sort is best suited for data with a limited range of values. For datasets with a very large range, the memory and time costs may increase, making distributed count sorting less efficient than alternative methods.

1.2 Execution Time Analysis:

1.2.1 Distributed Merge Sort

For each problem of size n we are solving two subproblem of size n/2 with with merging which requires O(n) time that is

$$T(n) = T(\frac{n}{2}) + O(n) \tag{1}$$

On solving this we get

$$T(n) = O(n) \tag{2}$$

1.2.2 Distributed Count Sort

We are dividing the problem of size n in p parts, where p is number of threads, and counting frequencies of this parts concurrently in O(n/p) time and then traversing the frequency array after which is of size k, where k is range of values. Assuming range of values is relatively small to dataset size we have

$$T(n) = O(\frac{n}{p}) \tag{3}$$

1.3 Memory Usage Overview:

1.3.1 Distributed Merge Sort

Our implementation for Distributed Merge Sort requires additional memory proportional to the size of the dataset for temporary arrays. For small datasets, this overhead (usually O(n)) is not a significant concern but still exists. For larger data sets it becomes much more significant

1.3.2 Distributed Count Sort

Our implementation for Distributed Count Sort requires additional memory proportional to the range of the input values.For small datasets or large dataset the memory usage is almost same if both of them have similar range of values.

1.4 Graphs:

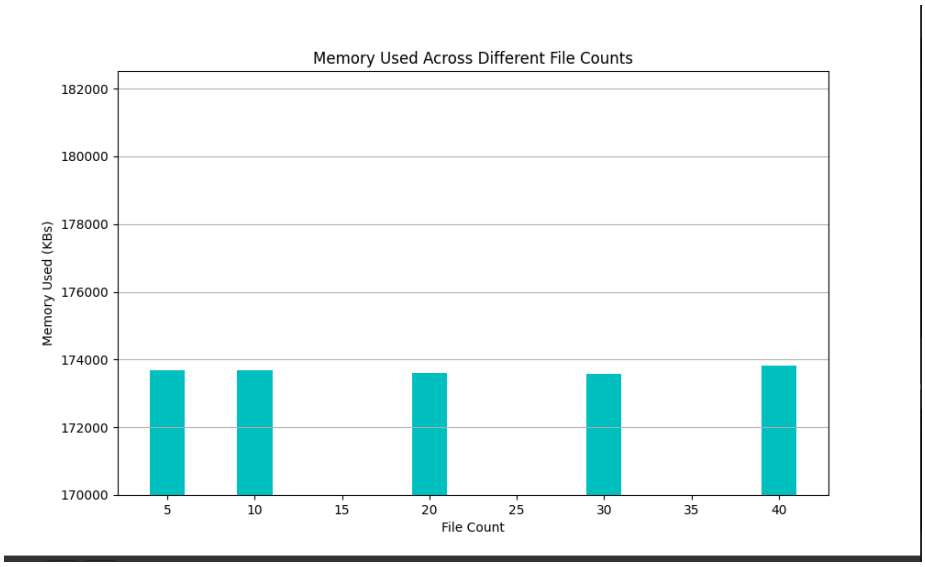

Figure 1: Memory Usage for Count sort

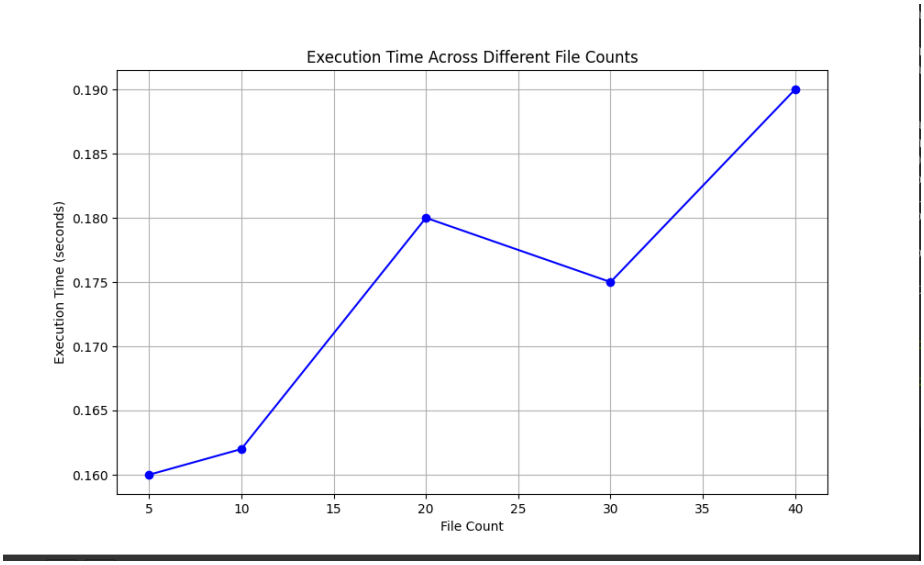

Figure 2: Execution Time for Count sort

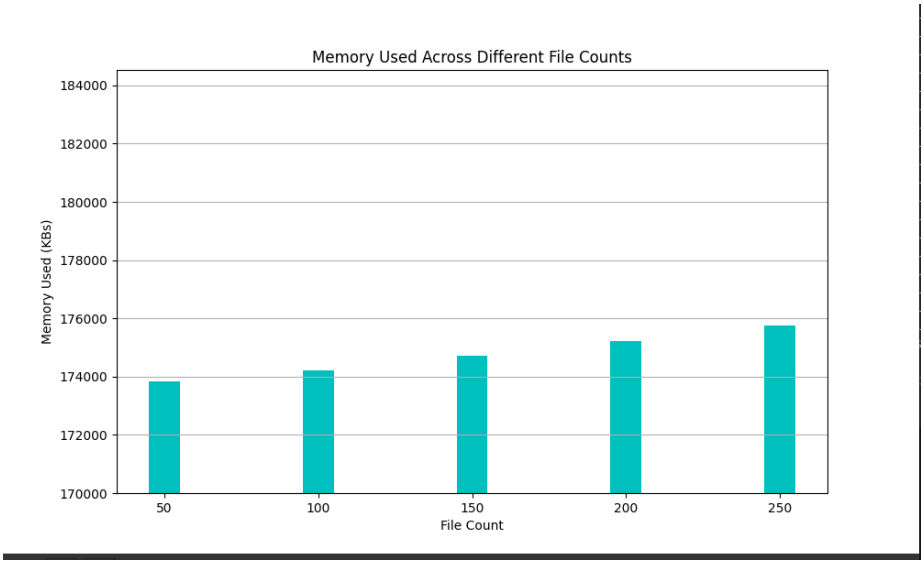

Figure 3: Memory Usage for Merge sort

1.5 Summary:

This report compares distributed Merge Sort and Count Sort implementations. Distributed Merge Sort works better from medium to larger number of files.

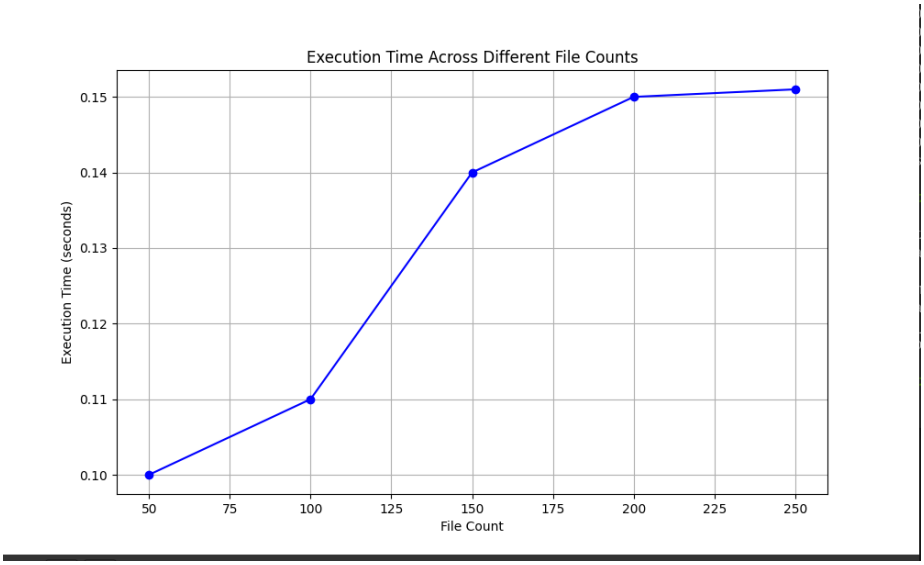

Figure 4: Execution Time for Merge sort

However, there might be an increased overhead due to larger amounts of threads. In contrast, Distributed Count Sort works best when there is a limited range (max - min) of data points to consider.

To improve performance with larger datasets, optimizations like increasing array size for countsort will allow it to work with larger datasets. For mergersort, we can perform the merge operation in more efficiently in O(log n) by merging concurrently such that at each iteration, we create an merged array double in size to the previous array using threads. This brings the overall time complexity to O(log n).

2 Copy-On-Write Fork Performance Analysis

2.1 Page Fault Frequency

We have implemented an additional system call which returns the number of page faults triggered up to that point. We added it to lazytest.c file and obtained following results.

For both simple test we got 1 Page fault. This allocates more than half of physical memory,then fork. This will fail in the default kernel, which does not support copy-on-write

For all three tests, we got around 18840 page faults. three processes all write COW memory which causes more than half of physical memory to be allocated, so it also checks whether copied pages are freed.

For file test we got 5 page faults. It is a test to check whether copyout simulates

COW faults or not.

2.2 Brief Analysis

2.2.1 Memory Conservation

- **Shared Pages**: When a process is forked, both the parent and child processes share the same physical memory pages, conserving memory.
- **Delayed Copying**: Memory pages are copied only when modified, ensuring that only the modified pages are duplicated.

2.2.2 Efficiency

- **Faster Forks**: The fork operation is faster as it involves duplicating page tables and marking pages as read-only.
- **Reduced Overhead**: Physical memory pages are not immediately copied, reducing the overhead associated with the fork system call.

2.2.3 Areas for Further Optimization in COW

- **Page Table Management**: Optimizing the duplication of page tables can reduce the time taken for the fork operation.
- **Memory Management**: Techniques that identify and copy only the modified portions of a page can help in conserving memory.
- **Synchronization Overhead**: Reducing the synchronization overhead involved in marking pages as read-only or handling page faults can improve performance.