LAZY Sort - Report

1. Distributed Merge Sort

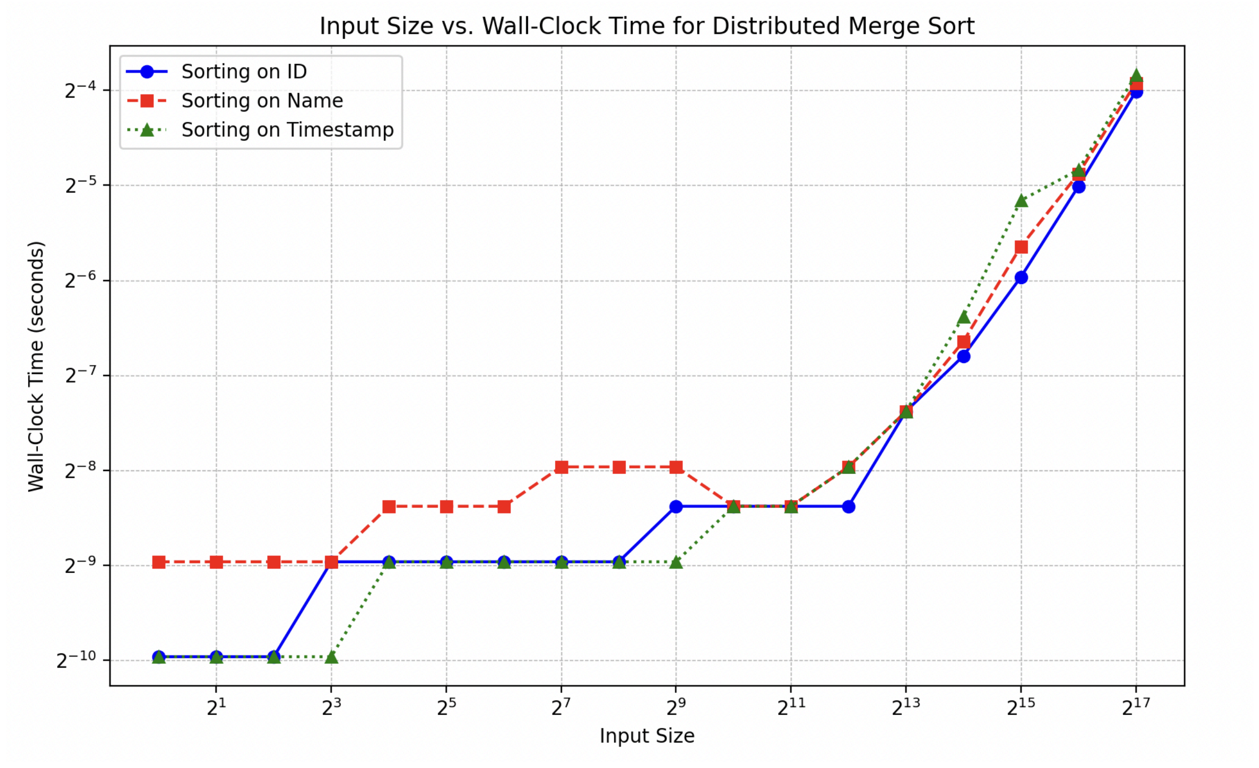
Implementation Analysis

Distributed merge sort is performed on inputs of size greater than 42 files. At the start of the program, we find the total number of threads available.

While we have threads available, every time we partition the array, we assign a new thread for performing merge sort on the left half and let the right half be sorted by the current thread itself. The left and right halves of each partitioning modify different areas of memory, making threading possible.

The runtime of this approach is limited by the number of threads available on the system. Once we have run out of threads, each subsection is sorted sequentially rather than parallelly.

Execution Time Analysis



Size of Input	Wall-Clock Time (in seconds)
4	0.001
16	0.002
256	0.002
1024	0.003
4096	0.003
8192	0.006
16384	0.009
32768	0.016
65536	0.031
131072	0.062

2. Distributed Count Sort

Implementation Analysis

Distributed merge sort is performed on inputs of size greater than 42 files. At the start of the program, we find the total number of threads available.

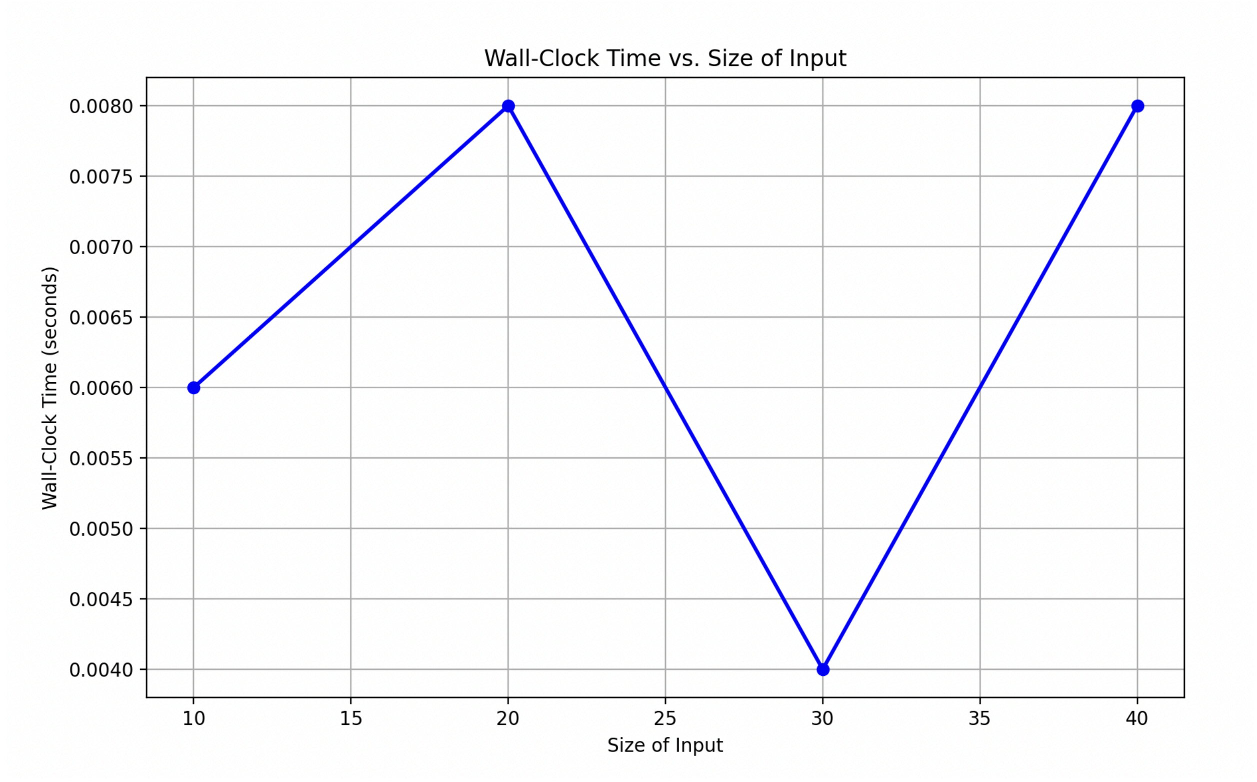
Distributed count sort is performed on inputs of not more than 42 files. Again, at the start of the program, we find the total number of threads.

We have an algorithm to map each file to a key according to what we're sorting based on:

- We convert each string to a base 26 number if we're sorting based on Name (assuming there can be at most 5 characters and the names are unique, with the first character from the left having a weight of $26^{4)}$.
- When sorting on ID, we assume a constraint of 1e5 on the ID and assume that ID is unique, and treat the ID directly as the key.
- Each time stamp is converted into the number of minutes that have passed since the start of the year (we assume that all files are made in the same year and two files can't be made in the same minute, to avoid collisions).

We pass through the array with the input once to find the highest possible value of the key. We then make another file array big enough to store every value from 1 to the highest possible key. We make n / (t + 1) partitions of the input array, where n is the total number of files and t is the total number of threads, and each of the partitions are traversed parallelly and assign each file in them to the map array's respective value of the key. The keys are unique, hence we won't be facing issues with any race conditions.

Execution Time Analysis



Size of Input	Wall-Clock Time (in seconds)
10	0.006
20	0.008
30	0.004
40	0.008

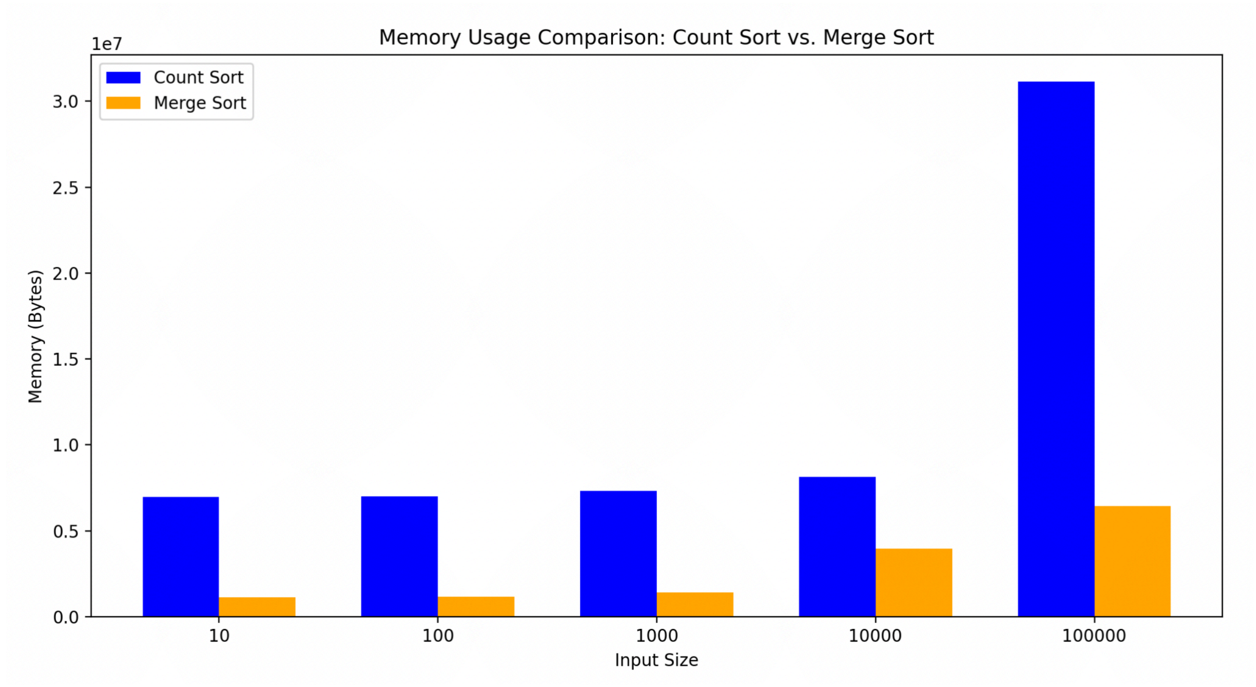
The values of the run-time don't seem to follow any sequence for wall-clock time, as count sort is dependent on the range of values of the keys rather than the size of input. They appear random as the input generated is also random, running this on more values would give a more consistent graph.

3. Memory Usage Analysis

Threads utilize the same memory as the respective processes.

In distributed merge sort, each thread directly makes modifications on the input array without any additional memory. Hence the memory stays mostly the same as regular sequential merge sort (O(n)), where there's no threading.

For distributed count sort, each thread passes through a certain portion of the input array and maps files to their respective keys in the map. The significant portion of memory for count sort lies in the array holding all the keys, as they must be able to store very high values.



4. Summary

Both merge sort and count sort (both traditional and distributed) are performed in constant time.

Distributed merge sort can be optimized further if we have access to more threads, but otherwise, it's dependent only on the size of the input array.

Distributed count sort is performed in constant time and is proportional to the value of the range of keys. Having values restricted to a smaller range of keys would greatly improve its speed and memory.