OSN MP3 REPORT

Part 1: Distributed Sorting System Performance

Implementation Analysis:

The initial dataset is divided into smaller subarrays. This division continues recursively until each subarray contains specified no.of elements as mentioned like 19.Each pair of subarrays is merged in parallel by creating a new thread for each merge operation.Threads are created dynamically as merge operations are required. Each thread performs a merge and then terminates upon completion.Proper synchronization mechanisms (condition variables) are used to ensure that the main thread waits for all merge operations to complete before proceeding to the next level of merging.

Pros:

- **Maximized CPU Utilization:** By running multiple merge operations in parallel, we can make better use of multi-core systems, reducing idle CPU time and increasing the throughput of the sorting process.
- **Reduced Latency**: Parallel execution of merge operations can significantly reduce the time required to sort large datasets, as multiple parts of the data are being merged simultaneously.
- **Dynamic Load Balancing**: By creating threads dynamically, the system can adapt to the current load, potentially leading to more efficient CPU usage.

Cons:

- **Thread Overhead:** Creating and managing a large number of threads can introduce significant overhead. This includes the cost of thread creation, context switching, and synchronization.
- **Complexity**: Managing a large number of threads, ensuring proper synchronization, and avoiding race conditions increases the complexity of the implementation.

Assumptions:

1. string length must be less than 5 for countsort.

2. string names with extensions are not recommended.

Execution Time Analysis:

Mergesort:

Size of data	1e1	1e2	1e3	1e4	1e5
Name	0	0	0.01	0.15	1.61
ID	0	0	0.01	0.15	1.67
Timestamp	0	0	0.01	0.15	1.63

Small Data Sizes (1e1, 1e2): For very small data sizes, the execution time is negligible (0 seconds). This indicates that the overhead of initiating the distributed sort dominates the actual sorting time, making it effectively zero for such small inputs.

Medium Data Size (1e3): For a data size of 1e3, the execution time is recorded as 0.01 seconds for all categories (Name, ID, Timestamp). This shows the start of noticeable sorting time, but it is still quite minimal.

Large Data Size (1e4): When the data size increases to 1e4, the execution time increases to 0.15 seconds for all categories. This suggests a linear increase in execution time with respect to the data size.

Very Large Data Size (1e5): For the largest data size of 1e5, the execution time is around 1.61 to 1.67 seconds. This further confirms the scaling behavior, showing that as the data size increases by a factor of 10, the execution time increases significantly but not exponentially.

Countsort:

Size of data	10	20	30	40
Name	7.47	11.4	13.61	14.64
ID	0	0	0	0
Timestamp	1.27	4.7	8.5	9.31

Name Sorting: Countsort shows high execution times for sorting names, even for relatively small datasets. The time increases significantly with the data size, indicating that Countsort may not be the best choice for sorting strings.

ID Sorting: Countsort is extremely efficient for sorting numerical IDs, with execution times remaining negligible (0 seconds) across all tested data sizes.

Timestamp Sorting: The execution time for sorting timestamps increases with the dataset size but remains lower than sorting names. This indicates that Countsort handles numerical and timestamp data more efficiently than string data.

Memory Usage Overview:

Mergesort:

Size of data	1e1	1e2	1e3	1e4	1e5

Name	1536	1664	2688	12892	90528
ID	1664	1792	2688	12768	90316
Timestamp	5248	1792	2560	12884	90476

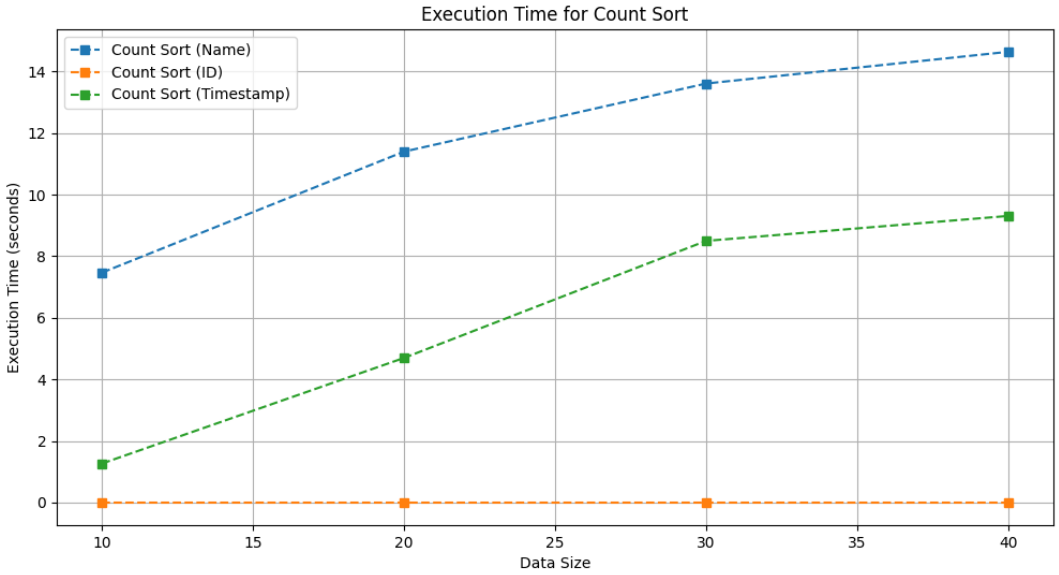
Countsort:

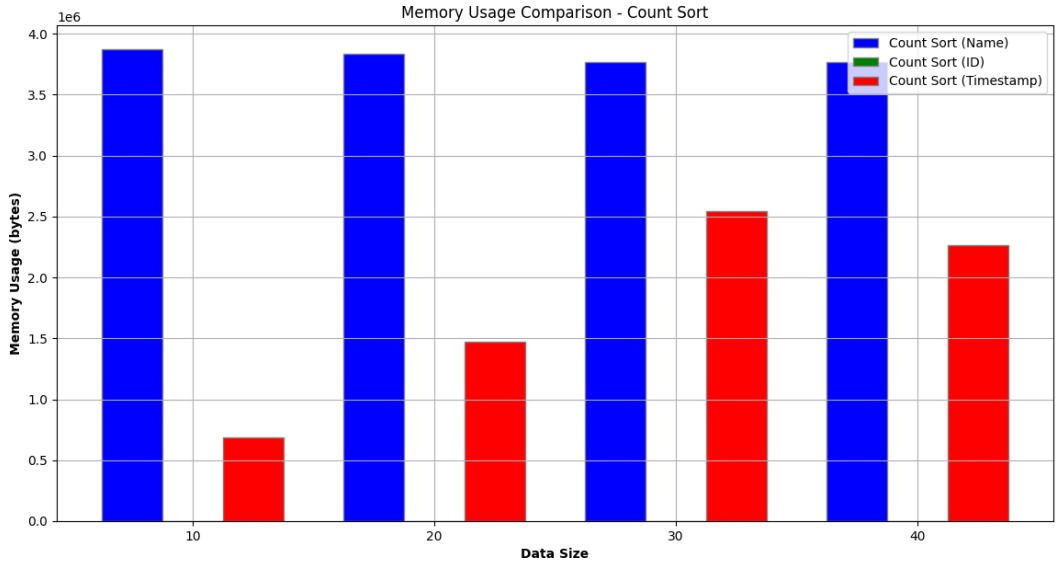
Size of data	10	20	30	40
Name	3874816	3836160	3767808	3768192
ID	1664	1536	1664	1664
Timestamp	686976	1474176	2540928	2265600

Mergesort: Memory usage increases linearly with the size of the dataset. It remains efficient for small datasets and moderately efficient for large datasets across all data types (Name, ID, Timestamp).

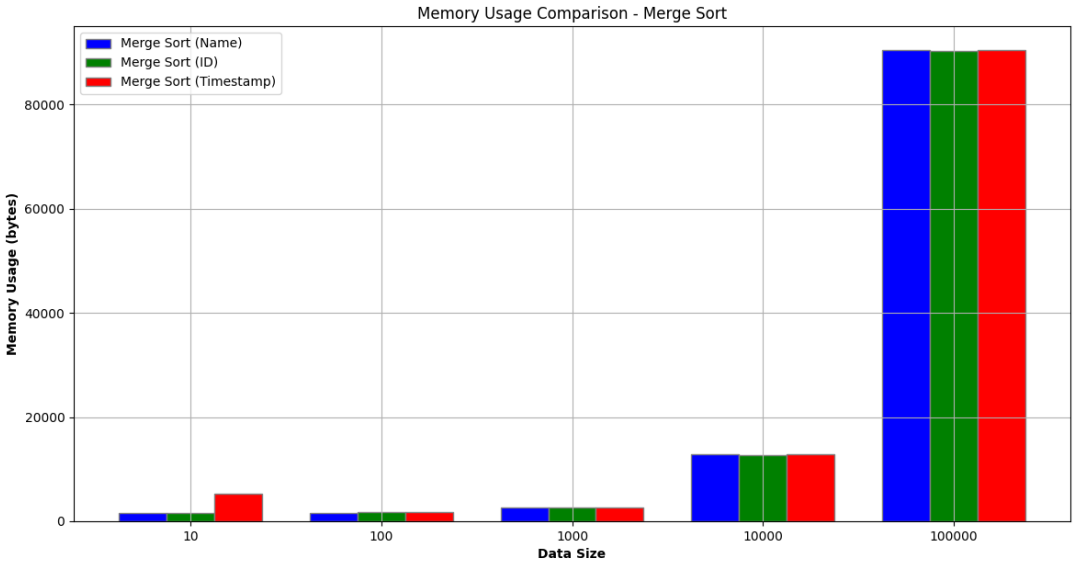
Countsort: Shows extremely high memory usage for sorting strings (names), making it less suitable for such data types. However, it is very efficient for numerical data (IDs) with minimal memory usage. Memory usage for timestamps is moderate, increasing with larger datasets but still more efficient than sorting names.

Graphs:




Memory Usage Comparison - Count Sort
1e6
Count Sort (Name)
Count Sort (ID)
Count Sort (Timestamp)
Memory Usage (bytes)
Data Size


Execution Time for Merge Sort
Merge Sort (Name)
Merge Sort (ID)
Merge Sort (Timestamp)
Execution Time (seconds)
Data Size



Summary:

Execution Time:

- **Mergesort**: As the file count increases, the execution time for Mergesort scales significantly. For very small datasets (1e1, 1e2), the execution time is negligible, but as the dataset size grows to 1e4 and 1e5, the time increases substantially. This behavior is expected due to the O(n log n) time complexity of the algorithm.
- **Countsort**: Countsort, on the other hand, handles smaller datasets efficiently, with execution time increasing gradually from 7.47 seconds for 10 files to 14.64 seconds for 40 files. Countsort is generally more efficient than Mergesort for datasets with smaller or fixed range values, but its performance does not scale well with very large datasets, especially those that require large counts or have high value ranges.

Memory Usage:

- **Mergesort**: Memory usage increases as the file count grows, which is typical for Mergesort, as it requires additional memory for temporary arrays to hold intermediate results during the merge process. Memory usage for Merge Sort grows significantly with larger datasets, reaching up to 90,528 bytes for larger sizes (1e5 files).
- **Countsort**: Countsort exhibits significant memory usage even for smaller datasets, particularly for the `Name` category. This is due to the creation of large counting arrays, which are proportional to the range of the input values. As the file count increases, Countsort's memory usage continues to grow, but it does not require as much extra space as Merge Sort for larger datasets.

COPY ON WRITE FORK in xv6

Modified Files:

1. vm.c
    - uvmcopy: used mappages to map the PTE with the same pa
    - cow_fault: copies the content of the shared page to newly allocated page
    - copyout: This checks for cow_fault and then copies out
2. kalloc.c
    - added a refcount arary
    - inc_ref & dec_ref: function that increase and decrease the ref count array for particular index based on physical address pa
    - kfree: every time kfree is called it will call dec_ref and if refcount becomes zero then it frees the page
    - kalloc: intialization of refcount of particular index based on physical address pa to 1
3. trap.c
    - checks for pagefault if it happens then it calls the cow_fault to check pagefault happend because of cow if it is then it kills the process

ANALYSIS:

To reord the frequency of page faults i have taken a variable no_of_pagefaults in proc struct and intilazed them to 0 then every time a page fault happens in the user trap i have incremented the variable
I have used the forks given in lazytest
READ ONLY : Simple Fork

Number of Page Faults = 0

WRITE ONLY: Three Fork

Number of Page Faults =6554

In a **Copy-On-Write (COW)** fork, page faults occur when a process modifies shared memory, triggering the creation of a new memory page. The results show that with **read-only** processes, there are no page faults, as pages are shared without modification. Whereas, **write-only** processes result in 6554 page faults, as each modification requires a new page allocation. This demonstrates COW's efficiency for read-only operations and the expected increase in page faults for write operations to maintain memory isolation.

Number of Times COW Mechanism Called:

READ ONLY : Simple Fork

Number of times cow mechanism called is 17

WRITE ONLY: Three Fork

Number of times cow mechanism called =16197

For read only process cow mechanism is called 17 times (all in copyout) and for write only process cow mechanism is 16197 (6554 times in usertrap 9643 times in coyout)

Memory Conservation:

Reduced Initial Copies: In a typical fork, each of the n processes with m pages requires all n×m pages to be copied. With Copy-On-Write (COW), these pages remain shared initially, so no copies are made until a modification is necessary.

Selective Copying: COW only duplicates pages when a process modifies them. For many processes that do little or no writing (such as those that immediately call exec), only a small portion of the n×m pages need to be copied.

Efficient for Lightweight Forks: COW is particularly efficient for processes that don't modify much memory after forking, as only modified pages are duplicated, conserving memory.

COW allows the parent and child to share memory pages rather than creating separate copies immediately. This reduces the initial memory overhead because no additional memory is consumed for the duplicated pages as long as they remain unchanged.

Efficiency:

Reduced Memory Usage: COW fork allows the parent and child processes to share memory pages until a modification is made, conserving memory by avoiding unnecessary duplication.

Faster Forking Process: Since memory pages aren't copied immediately, the fork operation completes faster, reducing the overhead associated with starting new processes.

On-Demand Duplication: COW only copies pages when a process writes to them, so for processes that perform minimal or no modifications, very few pages are actually duplicated.

Ideal for Short-Lived Processes: Processes that quickly call `exec` or perform limited writes benefit greatly from COW, as little to no extra memory is allocated, enhancing system efficiency.

Improved Scalability: With COW, the system can handle more concurrent processes with lower memory usage, supporting better scalability in multi-process environments.

FURTHER OPTIMIZATIONS:

Further optimizations for the Copy-On-Write (COW) fork can boost memory efficiency and system performance

Delayed Page Allocation: Instead of copying pages on the first write, COW could delay allocations by batching writes or grouping them. This approach would save both memory and time by reducing the number of page copies, especially when multiple writes happen close together.

Efficient Page Tracking: By using finer-grained tracking for modified sections of pages, the system could avoid copying entire pages when only small parts are modified. This would be particularly useful for processes that make localized changes, improving memory efficiency.

Page Sharing Across Forks: For processes that fork and read the same static data or shared libraries, the system could maintain shared pages across multiple forks. This would further conserve memory, especially for data that changes rarely, such as read-only resources like libraries or configuration files.