Part 1: Distributed Sorting System Performance

Implementation Analysis:

In this distributed sorting system, we explored two approaches: Distributed Merge Sort and Distributed Count Sort.

For Distributed Merge Sort, we opted to create a thread for each merge operation. This approach allows each merge task to be handled concurrently, which benefits performance by distributing workloads across threads, enabling quicker completion of individual merge tasks. However, it comes with certain trade-offs:

- **Pros**: Increased efficiency due to concurrent task processing, especially on multi-core systems. It also scales relatively well with larger datasets.
- **Cons**: Higher memory overhead due to the creation of multiple threads, and potential thread management complexity. This approach may not be optimal for very large data sizes or systems with limited memory.

For Distributed Count Sort, we chose a simpler approach with fewer concurrent operations. Since Count Sort is highly efficient for data with a limited range of values, this approach minimizes memory usage, as each counting operation does not require complex recursive splitting or merging.

- **Pros**: Lower memory usage and reduced execution time for datasets with smaller or moderately sized data ranges.
- **Cons**: Does not scale as well as Merge Sort for large datasets with a broad range of values, as the efficiency of Count Sort diminishes with larger, varied data.

Execution Time Analysis:

The execution times for Distributed Merge Sort and Distributed Count Sort were measured across small, medium, and large file counts, as follows:

- **Distributed Merge Sort**:
  - 31 files: 0.0102 seconds
  - 101 files: 0.0136 seconds
  - 1929 files: 0.0277 seconds
- **Distributed Count Sort**:
  - 31 files: 0.0025 seconds
  - 101 files: 0.0027 seconds
  - 1929 files: 0.0119 seconds

From these measurements, Distributed Count Sort consistently showed lower execution times than Distributed Merge Sort across all tested file counts. However, the performance gap narrows as the file count increases, indicating that Distributed Merge Sort scales better with larger datasets.

Memory Usage Overview:

The memory usage for each algorithm was assessed with small and large datasets:

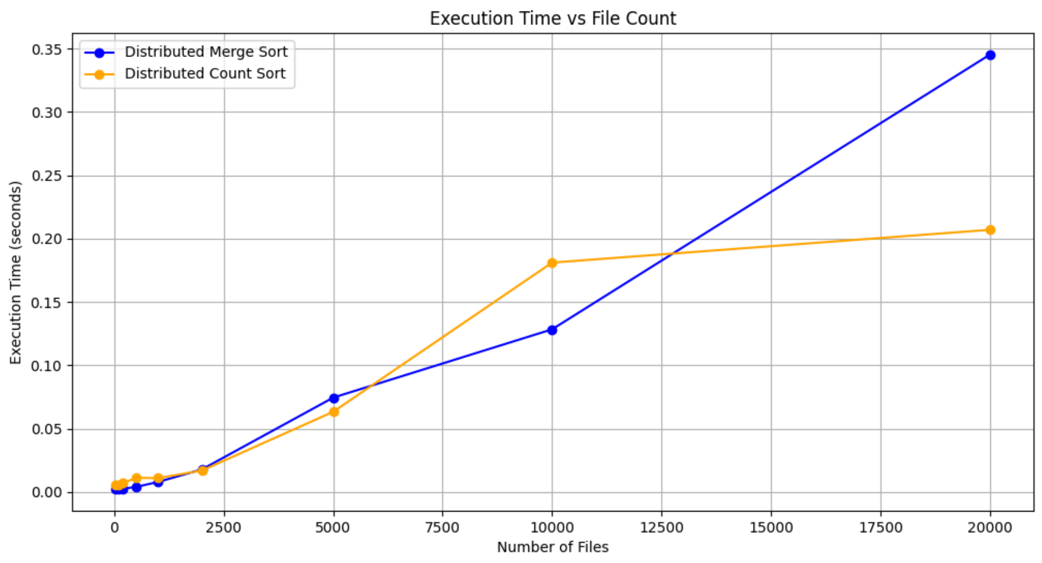
- **Distributed Merge Sort**:
  - 31 files: 14,155,776 bytes
  - 101 files: 14,155,776 bytes
  - 1929 files: 14,417,920 bytes
- **Distributed Count Sort**:
  - 31 files: 14,024,704 bytes
  - 101 files: 14,024,704 bytes
  - 1929 files: 14,286,848 bytes

Distributed Count Sort consistently used less memory compared to Distributed Merge Sort. The difference is modest but can be impactful when scaling to extremely large datasets or operating in memory-constrained environments.

Graphs:

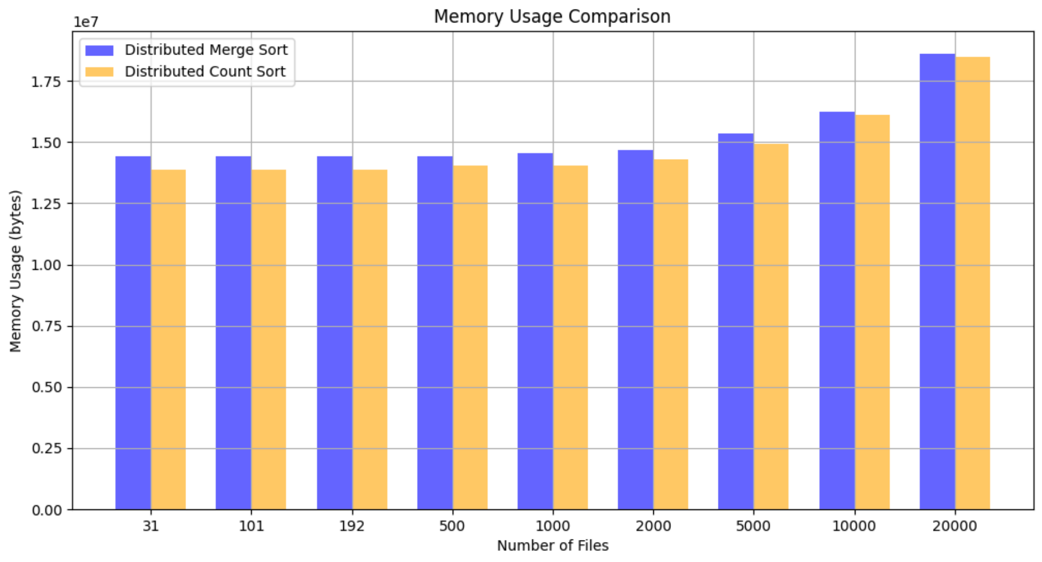
Execution Time:

A line graph should illustrate the execution times across different file counts, with each algorithm represented by a distinct line to visualise scaling performance differences.



Memory Usage:

A bar chart should compare memory usage between Distributed Count Sort and Distributed Merge Sort across small and large datasets, emphasising the difference in resource requirements.



Summary:

Our findings indicate that Distributed Count Sort is more efficient in terms of both execution time and memory usage for smaller datasets and lower file counts. However, Distributed Merge Sort scales better for larger datasets, with a slower increase in execution time as file counts grow.

Potential Optimizations: To further enhance performance for larger datasets, we could implement a hybrid approach that selectively uses Count Sort for smaller subproblems within Merge Sort. Additionally, implementing dynamic memory allocation could help manage resource usage, particularly in systems with limited memory.

Part-2: Copy-on-Write (CoW) Fork Implementation

Implementation Details

Spinlock.h
In the `spinlock.h` file, the structure of the extern `struct ref_spinlock` is defined. This structure is used to hold a reference counter for each physical page through the integer array `reference_count`.

Kalloc.c
In the `kalloc.c` file, this global struct is created, named `ref_c`.

Function `freerange()`
In the existing for loop, the reference counter for each page in the current iteration is initialised to 1. This setup is crucial because `kfree()` will be called later, and it will decrement this counter. Therefore, initialising the counter to 1 ensures that it starts correctly before `kfree()` performs its decrement.

Function `kfree()`
When `kfree()` is invoked, it begins by decrementing the reference counter of the specified page. A check follows to see if this counter has reached 0. If it has, this indicates that no processes are currently using the page, so it can be returned to the list of available pages, following the original logic in `kfree()`. If the counter isn't zero, no additional action is required.

Function `kalloc()`
The reference counter for the specific page is initialised to 1.

Trap.c

Function `usertrap()`
An `else if (r_scause() == 15)` condition was added to identify page fault 15, specifically a Store/AMO page fault. When a page fault with code 15 occurs, it is recognized as a valid Copy-on-Write (CoW) fault, indicating the page is valid, user-level, etc. The function first checks if the virtual address (obtained from `r_stval()`) is allowed. Next, it retrieves the page table entry (PTE) for the specific page by calling the `walk()` function. It verifies that the PTE is both valid and user-level. The physical address linked to this PTE is then stored in the variable `pa0`. According to the assignment, a new page is allocated using `kalloc()`, and the contents of the old page are copied to this new page via `memmove()`. Finally, the function retrieves the flags of the old page's PTE, updates the PTE to point to the physical address of the new page, and sets the `PTE_W` bit to 1. The original (read-only) page is then freed by calling `kfree()`.
The global variable `pagefault_counter`, initialised to zero, is incremented in the `else if (r_scause() == 15)` condition block.

Vm.c

Function `uvmcopy()`
The `uvmcopy()` function is updated so that instead of creating new pages, it reuses existing physical memory (`pa`) pages when calling `mappages()` to map the parent's physical memory pages to the child. Additionally, the `PTE_W` flag in both the parent's and child's page table entries (PTEs) is cleared by using a bitwise AND operation. During each iteration over each page, the reference counter is incremented.

Function `copyout()`

The `walkaddr()` function is used to directly retrieve the physical address associated with a virtual address without triggering the hardware MMU, meaning no page fault has occurred. In `copyout()`, when encountering a Copy-on-Write (CoW) page, the process now uses the same checks and procedures as a page fault. Specifically, if the virtual addresses are within permissible bounds and do not exceed `MAXVA`, the function retrieves the PTE, verifies that it is valid and user-level, and stores the physical address in `pa0`. If the PTE is read-only (i.e., the `PTE_W` bit is 0), it follows the same steps as `usertrap()` to handle CoW pages. Then, in the subsequent `memmove()`, the old page is replaced with the new one if necessary. If the page is not read-only, no new page is created, and the old page remains unchanged.
A system call `pagefault_count` is implemented to track how many times a page fault has occurred.

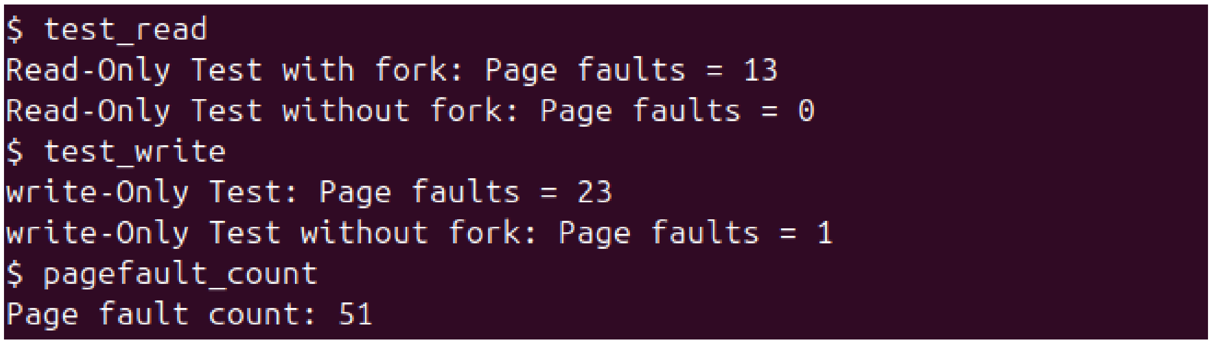
Additionally, `test_read` and `test_write` are implemented to give the number of page faults occurring in read-only and write-only operations, respectively.

Performance Analysis

Page Fault Frequency

- `pagefault_count`: This system call provides the total number of page faults that have occurred. This count includes all the page faults from all the tests conducted.
- `test_read`: This test measures the number of page faults in a read-only operation.
  Scenario:
  Suppose we have 10 pages to read from; then 13 page faults will occur if we read after forking.
  Page Faults = 13
  Why 13 Page Faults?
  The OS triggers a page fault for each shared page as part of its internal management during the `fork()` process (even if no explicit writes are done by the user-level code). The page faults likely occur when the operating system checks the shared pages or performs internal operations that mark them as modified, invoking the CoW process. Each page fault corresponds to a situation where the system needs to copy a page due to an access attempt after `fork()` (even if it's a read access). Thus, the number of page faults may correspond to the total number of pages involved in the CoW process.
  Page Faults = 0
  In this case, there is no `fork()` involved, so there are no shared pages between processes. Each process operates independently, and since no writing occurs, no page faults are triggered.
- `test_write`: This test gives the total number of page faults in a write operation (modifies memory).
  Scenario:

Suppose we have 10 pages that can be modified; then 23 page faults will occur if we write after forking.
Page Faults = 23
In the write-only test, there are more page faults because every write operation results in a page fault. In a typical scenario, when writing to a page that was previously shared, CoW causes a page fault to copy the page before the write can occur. Therefore, the number of page faults is high since every write attempt to a shared page will trigger the CoW mechanism.
Page Faults = 1
Without `fork()`, the process operates independently, and only one page fault occurs initially when the process writes to a page. This fault is likely due to the allocation of new memory for writing since the page is no longer shared.



```
$ test_read
Read-Only Test with fork: Page faults = 13
Read-Only Test without fork: Page faults = 0
$ test_write
write-Only Test: Page faults = 23
write-Only Test without fork: Page faults = 1
$ pagefault_count
Page fault count: 51
```

Conclusion

Thus, there are more page faults in a write operation as compared to a read operation. In a write operation, page faults occur more frequently than in a read operation due to the Copy-on-Write (CoW) mechanism, which triggers a page fault whenever a process attempts to modify a shared page.

Brief Analysis:

Benefits
Copy-on-Write (CoW) fork provides significant benefits for efficiency and memory conservation. When a process forks, CoW allows the parent and child processes to share the same physical memory pages initially, rather than creating identical copies. This sharing reduces memory usage substantially, as separate pages are only allocated when one of the processes modifies a page, making CoW particularly beneficial in scenarios where forked processes perform more reads than writes. This deferred copying reduces the overhead involved in process creation, allowing the system to allocate memory more judiciously and efficiently handle memory-intensive applications.

Optimization

There are areas where CoW could be further optimised. For example, CoW can be enhanced by tracking and preemptively freeing pages that are no longer shared, improving memory reclaiming when pages become exclusive to a single process. Additionally, finer-grained memory tracking and intelligent prefetching for anticipated write operations could further reduce latency and overhead in high-demand applications where writes to shared memory are more frequent.