Part 1: Distributed Sorting System Performance

Implementation Analysis:

Distributed Merge Sort:

In our implementation, we decided to create a new thread for one half per merge operation as long as the depth of recursion stays less than the limit MAX_THREADS. New thread creation is limited by a max-thread ceiling to prevent overloading the processor. This limit can be modified for different processors. The algorithm divides the array into two halves, each is sorted recursively, and then the two sorted halves are merged back together.

The pros of this approach are that threads allow for parallelism by allowing multiple sorting operations to be performed concurrently, which can reduce sorting time. This allows us to distribute workload as well. The cons of this approach, however, are that it can add additional overhead if thread creation isn’t properly managed. Too many threads can lead to contention for the CPU which may lead to the system spending more time in context-switching, rather than the sorting task itself. To prevent this, we create threads based on depth of recursion, which makes sure we balance this distributed workload and don’t overwhelm the system with too many threads. We also have the MAX_THREADS limit to facilitate this.

Distributed Count Sort:

Execution Time Analysis:

Distributed Merge Sort:

```
(base) dishapant@pop-os:~/Desktop/mini-project-3-25/concurrency$ time ./a.out < 10.txt > output.txt

real    0m0.003s
user    0m0.001s
sys     0m0.003s
(base) dishapant@pop-os:~/Desktop/mini-project-3-25/concurrency$ time ./a.out < 100.txt > output.txt

real    0m0.021s
user    0m0.007s
sys     0m0.001s
(base) dishapant@pop-os:~/Desktop/mini-project-3-25/concurrency$ time ./a.out < 1000.txt > output.txt

real    0m0.004s
user    0m0.005s
sys     0m0.000s
(base) dishapant@pop-os:~/Desktop/mini-project-3-25/concurrency$ time ./a.out < 10000.txt > output.txt

real    0m0.011s
user    0m0.017s
sys     0m0.001s
(base) dishapant@pop-os:~/Desktop/mini-project-3-25/concurrency$ time ./a.out < 100000.txt > output.txt

real    0m0.056s
user    0m0.102s
sys     0m0.053s
```

As we can see from the above, for small file counts, like 10 files, the run time is 0.003s. For slightly bigger counts ranging 100, the run time sees a sudden increase to 0.21s as the overhead of creating threads may outweigh the benefits of parallelism. For even bigger counts like 1000 files, the run drops down again to 0.004s as the benefits of parallelism outweigh the overhead of threads, around 0.004s. For a medium file count of 10000 files, the run time gradually increases to 0.011s. For a large file count of 100000 files, the run time increases to 0.056s.

Distributed Count Sort:

Memory Usage Overview:

Distributed Merge Sort:

```
(base) dishapant@pop-os:~/Desktop/mini-project-3-25/concurrency$ bas
h measure.sh ./a.out < 10.txt
ID
uwhllzi 3209807 1998-09-22T09:07:35
ergfkk 6082578 1982-10-14T22:57:46
vebffx 8355576 1999-10-07T04:56:39
dbygmk 28734317 1980-07-23T09:24:18
zuogwhyl 34626606 1976-08-13T19:03:58
srnvha 38162405 1970-06-28T13:29:02
xjncjzfq 48520995 1994-04-01T19:11:57
ljgpssbv 53348688 1975-10-07T22:38:07
ysdinzy 80649143 2000-09-28T17:46:07
ufefxk 99605483 2000-05-31T23:14:58
Total Runtime (seconds): 0
Max Memory Usage (KB): 1536
```

For file count of 10.

```
Total Runtime (seconds): 0
Max Memory Usage (KB): 1664
```

For file count of 100.

```
Total Runtime (seconds): 0
Max Memory Usage (KB): 1920
```

For file count of 1000.

```
Total Runtime (seconds): 0.01
Max Memory Usage (KB): 5708
```

For file count of 10000.

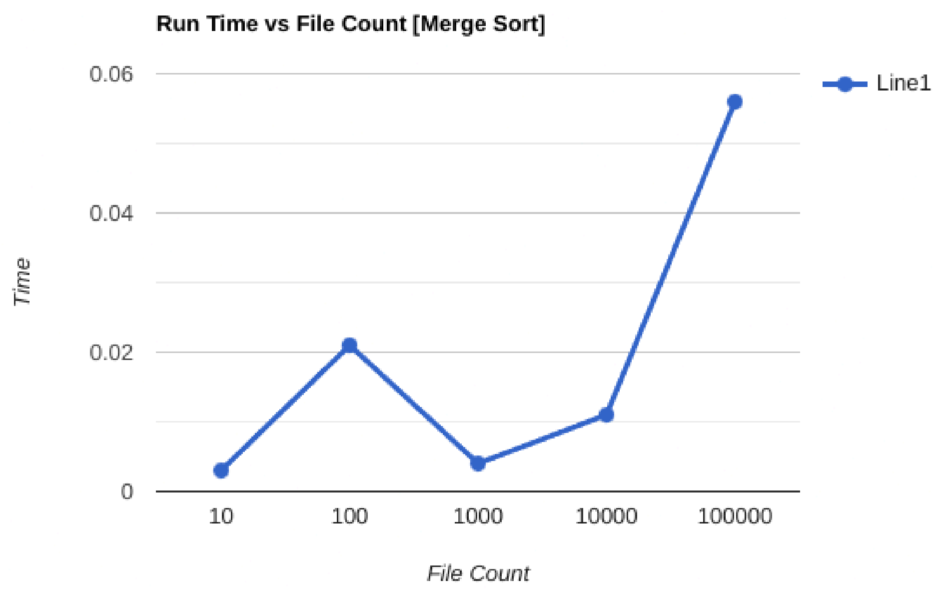
```
Total Runtime (seconds): 0.24
Max Memory Usage (KB): 39432
```

For file count of 100000.

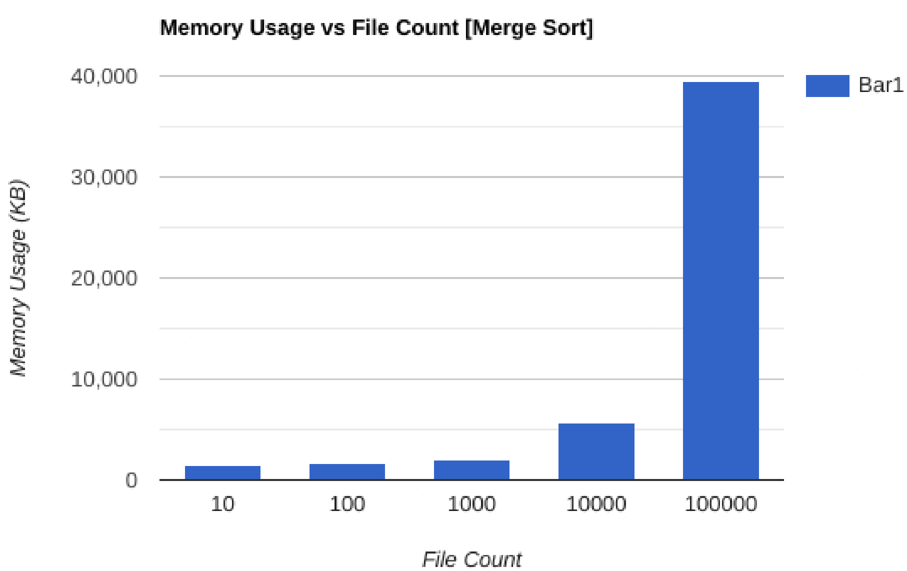
For smaller file counts, the number of threads is limited and so additional memory used is low. Memory usage increases as we have larger file counts, due to increased space taken by temporary arrays and each thread's stack space.

Distributed Count Sort:

Graphs:



Run Time vs File Count [Merge Sort]
0.06
0.04
0.02
0
Time
10
100
1000
10000
100000
File Count
Line1



Memory Usage vs File Count [Merge Sort]
40,000
30,000
20,000
10,000
0
Memory Usage (KB)
10
100
1000
10000
100000
File Count
Bar1

Summary:

Distributed Merge Sort:

So, as seen above, merge sort, for smaller file counts, merge sort is quite fast. It scales with file counts to yield higher run times, but still efficiently provides the required sorting. However, due to the use of multiple threads and temporary arrays, it has high memory consumption.

It could be further optimized for larger datasets, by maintaining a pool of threads to reduce the overhead of thread creation and destruction. We could also try to reuse temporary arrays to reduce the overhead of allocating this repeatedly. However both these measures bring with them added implementation complexity.

Distributed Count Sort:

Part 2: Copy-On-Write (COW) Fork Performance Analysis

Page Fault Frequency:

To conduct the page fault frequency analysis, I added a global variable to track page faults across program execution. The global variable is updated in the `cow_pagefault` function every time a cowfault is triggered. I also added a system call to return the value of this variable when requested. I then wrote a user program that ran various kinds of tests.

The following is the output of the respective tests

```
COW page faults for different tests:
Reading only: 0
Writing only: 10000
Writing to half the pages and reading from the other half: 5000
```

I wrote one test that read from the 10 pages and repeated this over 1000 iterations. I used the same number of pages and iterations but conducted two other tests. In one, I wrote to all the pages. In one, I wrote to half and read from the other half. As we can see, COW page faults only occur when a write operation is triggered, thus validating the correctness of our implementation.

Brief Analysis:

The advantage of using a Copy-on-Write fork is the reduced memory footprint of each process as a result of using it. It ensures that when a process is forked, a child process shares the same memory pages as the parent process. Thus, we don't duplicate these reusable pages if they contain read only data, and we can reduce the amount of extra physical memory we consume. Only pages that have been modified by a process are allocated. This also will reduce overhead while forking, thus speeding up the forking process.

We could try to add some mechanism to share further subparts of pages if only a small section has been modified. We could also look into analyzing memory access patterns in order to predict workload and define custom COW strategies to minimize unnecessary copying accordingly. Concurrency concepts could be implemented to distribute page fault handling or copying load across multiple cores. One research paper has looked into further optimizing the COW fork by exploiting spatial locality [1],

[1] https://www.mdpi.com/2079-9292/11/3/461

ChatGPT Commands

i need to implement cow fork in xv6.

this is what i found on mit's website
Here's a reasonable plan of attack.

Modify uvmcopy() to map the parent's physical pages into the child, instead of allocating new pages. Clear PTE_W in the PTEs of both child and parent for pages that have PTE_W set.
Modify usertrap() to recognize page faults. When a write page-fault occurs on a COW page that was originally writeable, allocate a new page with kalloc(), copy the old page to the new page, and install the new page in the PTE with PTE_W set. Pages that were originally read-only (not mapped PTE_W, like pages in the text segment) should remain read-only and shared between parent and child; a process that tries to write such a page should be killed.
Ensure that each physical page is freed when the last PTE reference to it goes away -- but not before. A good way to do this is to keep, for each physical page, a "reference count" of the number of user page tables that refer to that page. Set a page's reference count to one when kalloc() allocates it. Increment a page's reference count when fork causes a child to share the page, and decrement a page's count each time any process drops the page from its page table. kfree() should only place a page back on the free list if its reference count is zero. It's OK to to keep these counts in a fixed-size array of integers. You'll have to work out a scheme for how to index the array and how to choose its size. For example, you could index the array with the page's physical address divided by 4096, and give the array a number of elements equal to highest physical address of any page placed on the free list by kinit() in kalloc.c. Feel free to modify kalloc.c (e.g., kalloc() and kfree()) to maintain the reference counts.
Modify copyout() to use the same scheme as page faults when it encounters a COW page.

can you give me a more elaborate roadmap to doing this. don't give code, just give me an idea of the steps

Certainly! Implementing Copy-On-Write (COW) fork in xv6 involves several modifications to the existing memory management and process management code. Here's a detailed roadmap to guide you through the process:

Step-by-Step Roadmap

1. **Modify `uvmcopy` Function:**
   - **Current Behavior:** `uvmcopy` allocates new physical pages for the child process and copies the contents from the parent.
   - **New Behavior:** Modify `uvmcopy` to map the parent's physical pages into the child's address space without allocating new pages.
   - **PTE Adjustments:** Clear the `PTE_W` (write) flag in the PTEs of both the parent and the child for pages that were originally writable. This marks the pages as read-only.
2. **Implement Reference Counting:**
   - **Purpose:** To keep track of how many processes are sharing each physical page.
   - **Data Structure:** Use a fixed-size array to maintain the reference count of each physical page. You can index this array by dividing the page's physical address by 4096 (the page size).
   - **Initialization:** When a page is allocated with `kalloc`, set its reference count to one.
   - **Reference Management:**
     - Increment the count when `fork` shares a page.
     - Decrement the count when a process drops a page from its page table.
   - **Freeing Pages:** Modify `kfree` to only free a page if its reference count is zero.
3. **Handle Page Faults in `usertrap`:**
   - **Identify Page Faults**: Modify `usertrap` to recognize page faults, particularly write faults on COW pages.

3. **Handle Page Faults in `usertrap`:**
   - **Identify Page Faults:** Modify `usertrap` to recognize page faults, particularly write faults on COW pages.
   - **Write Fault Handling:**
     - Check if the faulting page is a COW page.
     - Allocate a new page with `kalloc`.
     - Copy the contents from the old page to the new page.
     - Update the PTE to point to the new page and set the `PTE_W` flag to make it writable.
   - **Read-Only Pages:** Ensure that pages that were originally read-only remain read-only and are shared between processes. If a process tries to write to such a page, it should be killed.
4. **Modify `kalloc` and `kfree` Functions:**
   - `kalloc`: When allocating a page, initialize its reference count to one.
   - `kfree`: Decrement the reference count of the page. Only free the page if its reference count is zero.
5. **Modify `copyout` Function:**
   - **Handle COW Pages:** When `copyout` encounters a COW page, it should use the same mechanism as in `usertrap` to handle write faults:
     - Allocate a new page.
     - Copy the old page's contents to the new page.
     - Update the PTE to point to the new page with the `PTE_W` flag set.
6. **Testing and Debugging:**
   - **Test Fork and Page Faults:** Thoroughly test the modified `fork`, page fault handling, and `copyout` functionalities.
   - **Edge Cases:** Test edge cases like multiple forks, deep copy scenarios, and processes

- **Edge Cases:** Test edge cases like multiple forks, deep copy scenarios, and processes terminating while holding shared pages.
- **Debugging:** Use debugging tools and print statements to verify that reference counts are managed correctly and that pages are copied only when necessary.

Additional Tips:

- **Page Table Entry (PTE) Management:** Ensure that you correctly manage the PTE flags and entries to prevent illegal memory accesses and segmentation faults.
- **Concurrency Considerations:** Be mindful of concurrency issues when managing reference counts, especially in a multiprocessor environment. Consider using locks or atomic operations as needed.
- **Code Organization:** Keep your changes well-organized and commented. This will make it easier to debug and maintain the code.

By following this roadmap, you should be able to implement the COW fork functionality in xv6 systematically and effectively.

Message ChatGPT

i added cow fork to xv6.
i now need to do the following:

Record the frequency of page faults during the operation of the COW fork. Test it with processes that read only, as well as those that modify memory.
The boss is particularly interested in knowing how many times the COW mechanism is triggered under different scenarios.

how can i record this

To record the frequency of page faults during the operation of the Copy-on-Write (COW) fork in xv6, you need to make modifications to the kernel to track and log these events. Here's a step-by-step guide on how you can implement this:

Step 1: Modify the Page Fault Handler

1. **Locate the page fault handler:** In xv6, the page fault handler is typically implemented in `trap.c` in the function that handles page faults. Look for a function like `trap()` and the case for page faults.
2. **Add a counter for COW page faults:** Add a global variable to keep track of the number of COW page faults.

c Copy code

```c
// In a suitable header file, e.g., defs.h or a new header file
extern int cow_page_faults;
```

c Copy code

```c
// In trap.c or an appropriate source file
int cow_page_faults = 0;
```

3. **Increment the counter:** In the page fault handler, increment this counter whenever a COW page fault occurs.

```c
void
trap(struct trapframe *tf)
{
    if(tf->trapno == T_PGFLT){
        uint va = rcr2();
        pte_t *pte;

        pte = walkpgdir(myproc()->pgdir, (void *)va, 0);
        if(pte && (*pte & PTE_COW)){
            // This is a COW page fault
            cow_page_faults++;

            // Handle COW by making a private copy of the page
            char *mem = kalloc();
            if(mem == 0){
                cprintf("trap: kalloc failed\n");
                myproc()->killed = 1;
            } else {
                memmove(mem, (char*)P2V(PTE_ADDR(*pte)), PGSIZE);
                *pte = V2P(mem) | PTE_P | PTE_W | PTE_U;
                lcr3(V2P(myproc()->pgdir)); // Refresh TLB
            }
            return;
        }

        // Existing page fault handling code
    }
    // Other trap handling code
}
```

Step 2: Expose the Counter to User Space

1. **Create a system call:** Add a new system call to fetch the value of `cow_page_faults`.
   - **Define the system call number:** Add an entry in `syscall.h`.

     c Copy code
     ```c
     #define SYS_get_cow_page_faults 22 // Or an appropriate number
     ```
   - **Implement the system call:** Add an implementation in `sysproc.c`.

     c Copy code
     ```c
     int
     sys_get_cow_page_faults(void)
     {
         return cow_page_faults;
     }
     ```
   - **Declare the system call:** Add a prototype in `syscall.h` or `defs.h`.

     c Copy code
     ```c
     int sys_get_cow_page_faults(void);
     ```
   - **Add the system call to the syscall table:** Update `syscall.c`.

     c Copy code
     ```c
     extern int sys_get_cow_page_faults(void);
     ```

     c Copy code
     ```c
     static int (*syscalls[])(void) = {
         // ... other syscalls ...
         [SYS_get_cow_page_faults] sys_get_cow_page_faults,
     };
     ```

```
    [SYS_get_cow_page_faults] sys_get_cow_page_faults,
};
```

Step 3: Test the Implementation

1. **Create a user-space program:** Write a user-space program to test the COW mechanism and record the page faults.

```c
#include "types.h"
#include "stat.h"
#include "user.h"
#include "syscall.h"

int
main(void)
{
    int initial_faults = get_cow_page_faults();

    int pid = fork();
    if(pid < 0){
        printf(1, "fork failed\n");
        exit();
    }

    if(pid == 0){
        // Child: Perform some memory reads and writes
        char *ptr = sbrk(4096);
        for(int i = 0; i < 4096; i += 1024){
            ptr[i] = i; // Write to trigger COW
        }
        exit();
    } else {
        // Parent: Wait for child to complete
        wait();
```

```c
            ptr[i] = 1; // Write to trigger COW
        }
        exit();
    } else {
        // Parent: Wait for child to complete
        wait();
        int final_faults = get_cow_page_faults();
        printf(1, "COW page faults: %d\n", final_faults - initial_faults);
    }

    exit();
}
```

2. **Compile and run the program:** This program forks a child process, performs some memory operations, and then reports the number of COW page faults that occurred during the execution.

By following these steps, you will be able to record and report the frequency of page faults triggered by the COW mechanism in xv6. Make sure to test both read-only and write scenarios to fully understand how often COW is being triggered under different conditions.

how can i implement merge sort parallely using threads in C

Implementing parallel merge sort using threads in C involves dividing the array into sub-arrays, sorting them in parallel, and then merging the sorted sub-arrays. Here is a step-by-step guide to achieve this:

1. **Include Necessary Headers:**

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
```

2. **Define a Structure to Pass Parameters to Threads:**

```c
typedef struct {
    int *array;
    int left;
    int right;
} ThreadData;
```

3. **Merge Function:**

```c
void merge(int arr[], int l, int m, int r) {
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;

    int *L = (int *)malloc(n1 * sizeof(int));
    int *R = (int *)malloc(n2 * sizeof(int));
```

Copy code

```c
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    i = 0;
    j = 0;
    k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }

    free(L);
    free(R);
}
```

4. **Serial Merge Sort Function:**

```c
void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;

        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);
    }
}
```

5. **Thread Function for Merge Sort:**

```c
void *threadedMergeSort(void *arg) {
    ThreadData *data = (ThreadData *)arg;
    int left = data->left;
    int right = data->right;
    int *arr = data->array;

    if (left < right) {
        int mid = left + (right - left) / 2;

        pthread_t tid1, tid2;
        ThreadData leftData = {arr, left, mid};
        ThreadData rightData = {arr, mid + 1, right};

        pthread_create(&tid1, NULL, threadedMergeSort, &leftData);
        pthread_create(&tid2, NULL, threadedMergeSort, &rightData);

        pthread_join(tid1, NULL);
        pthread_join(tid2, NULL);
```

```
        pthread_join(tid1, NULL);
        pthread_join(tid2, NULL);

        merge(arr, left, mid, right);
    }

    return NULL;
}
```

6. **Main Function:**

```c
int main() {
    int arr[] = {12, 11, 13, 5, 6, 7};
    int arr_size = sizeof(arr) / sizeof(arr[0]);

    printf("Given array is \n");
    for (int i = 0; i < arr_size; i++)
        printf("%d ", arr[i]);
    printf("\n");

    ThreadData data = {arr, 0, arr_size - 1};

    pthread_t tid;
    pthread_create(&tid, NULL, threadedMergeSort, &data);
    pthread_join(tid, NULL);

    printf("\nSorted array is \n");
    for (int i = 0; i < arr_size; i++)
        printf("%d ", arr[i]);
    printf("\n");

    return 0;
}
```

```
    return 0;
}
```

Copy code

Explanation:

1. **ThreadData Structure:**
   - This structure holds the array and the left and right indices for the portion of the array that needs to be sorted.
2. **Merge Function:**
   - This function merges two sorted halves of the array.
3. **Serial Merge Sort Function:**
   - This is a traditional recursive merge sort function, provided for reference.
4. **Threaded Merge Sort Function:**
   - This function is the core of the parallel merge sort. It splits the array and creates new threads to sort each half in parallel.
5. **Main Function:**
   - This function initializes the array, creates the initial thread to start the merge sort, and waits for the thread to complete before printing the sorted array.

Notes:

- The `threadedMergeSort` function uses pthreads to create new threads for sorting the left and right halves of the array.
- The main thread waits for the sorting threads to finish using `pthread_join`.
- Proper memory management is done by freeing dynamically allocated memory after merging.

By following these steps, you can implement a parallel version of merge sort using threads in C.