JP McLaughlin

CMPSC 472

Project 1

1. Project Description

This project deals with two MapReduce style tasks. The first part deals with parallel sorting by having an array sorted into chunks and having threads or processes sort them to then merge them back into one sorted array. The map portion of this splits the array into chunks to be sorted and the reduce part of the task is the merging of the sorted subarrays back into one sorted array. The second task is to have a shared global maximum integer which is accessed by several workers to change the maximum value if their local maximum value is higher than the global maximum. It is a project to see parallelism, inter-process communication, and synchronization more clearly.

Multithreading, multiprocessing, and synchronization are important tools for parallelism and inter-process communication and are used frequently with operating systems. They all have different advantages and disadvantages. Multithreading offers fast switching between threads, they are lightweight and have low overhead, have better responsiveness, and are better for I/O-bound tasks. Multiprocessing provides process isolation with separate memory, better fault tolerance, and is better for CPU-bound tasks. Synchronization is important to reduce race conditions and ensure the code functions the way you want it to. These were chosen for these two tasks in the assignment because they are both situations that show clear distinction between multithreading and multiprocessing and it also has clear instances of where synchronization is crucial to make sure everything functions correctly.

2. Instructions

The code all exists in a single main.c file so running that file in an IDE like CLion will cause the full program to run. The main function provides the multithread and multiprocess version of the parallel sorting as well as the max value task. So, when the program runs it will go through the entire tests for either the 32-integer array or the 131072-integer array. The default values are 32 for the array size and 100 for the max value of the numbers in the array in the generate_random_array function. The other values are commented out so to run with a 131072- integer array with a max value of 100000 the default values must be deleted (specifically the sequences of '32;' and '100);') and the higher values should have the '//' removed so they are no longer comments. Both lines of code that require editing to run each scenario are marked to alert what should be changed. The program should be recompiled after making any changes.

3. Structure of the Code

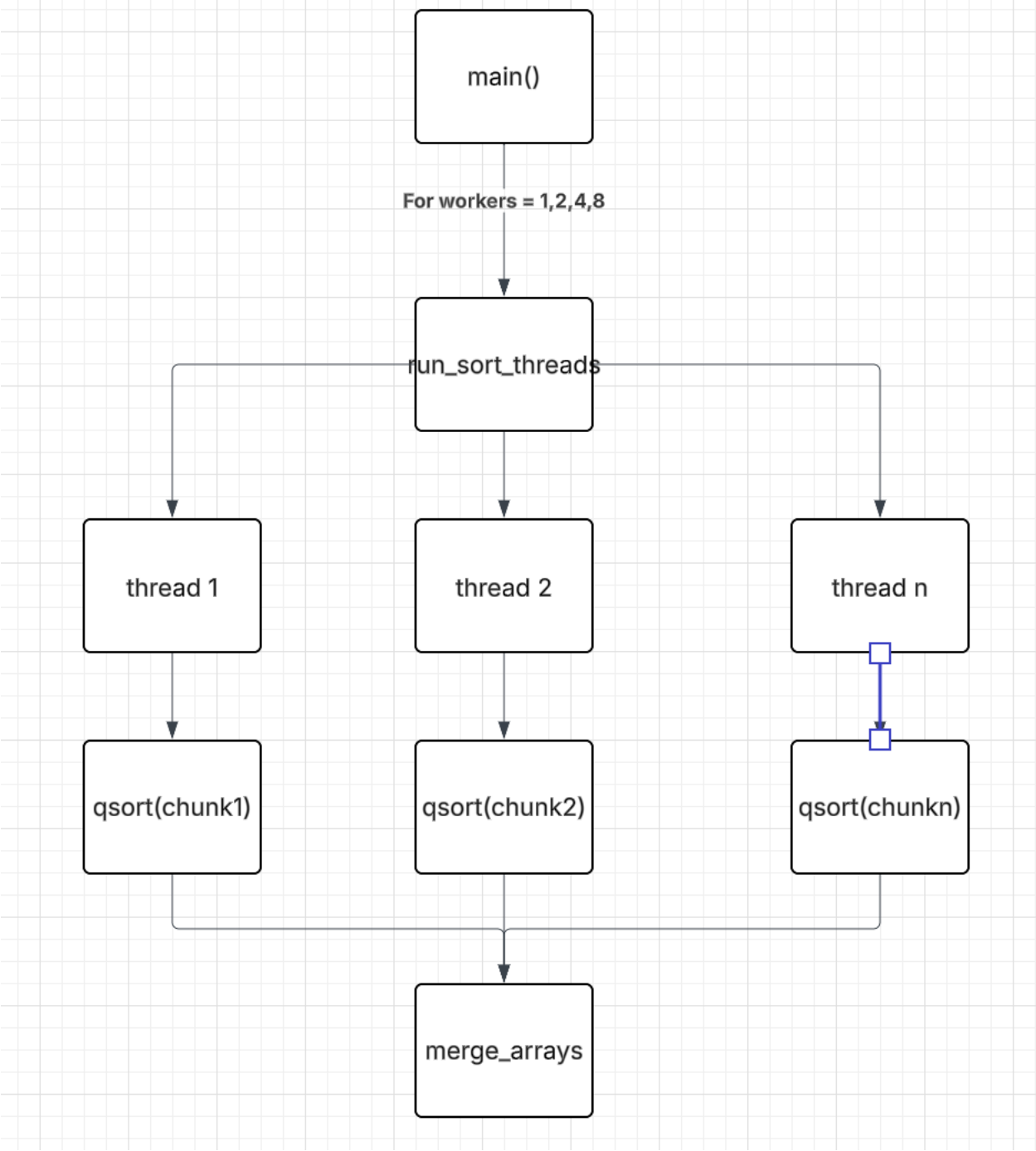
main()
For workers = 1,2,4,8
run_sort_threads
thread 1
thread 2
thread n
qsort(chunk1)
qsort(chunk2)
qsort(chunkn)
merge_arrays

Figure 1


main()
For workers = 1,2,4,8
run_sort_processes
mmap()
fork()
fork()
fork()
child 1
child 2
child n
qsort(chunk1)
qsort(chunk2)
qsort(chunkn)
wait()
merge_arrays

Figure 2

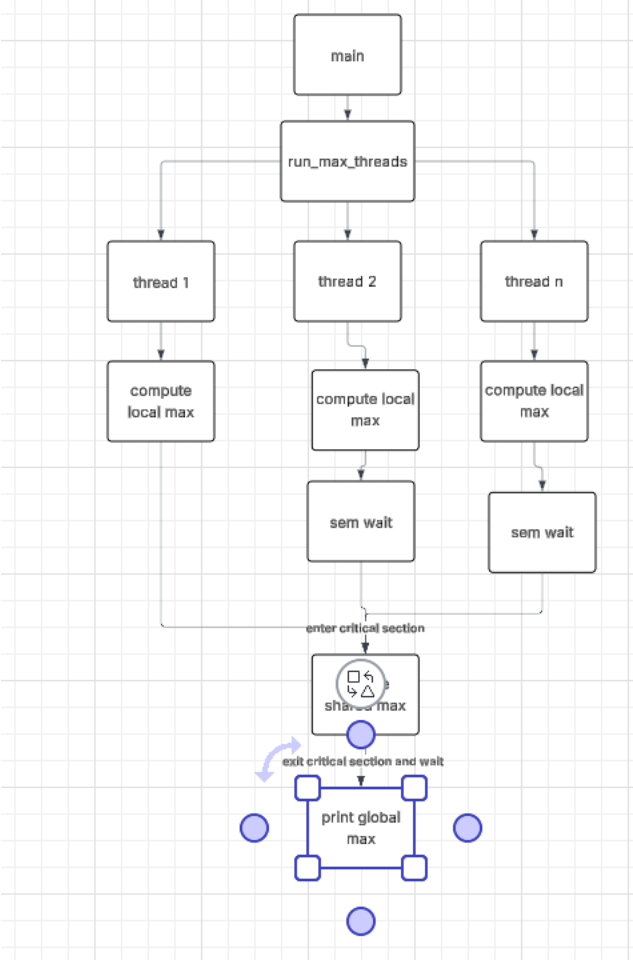
main
run_max_threads
thread 1
thread 2
thread n
compute local max
compute local max
compute local max
sem wait
sem wait
enter critical section
exit critical section and wait
print global max

Figure 3

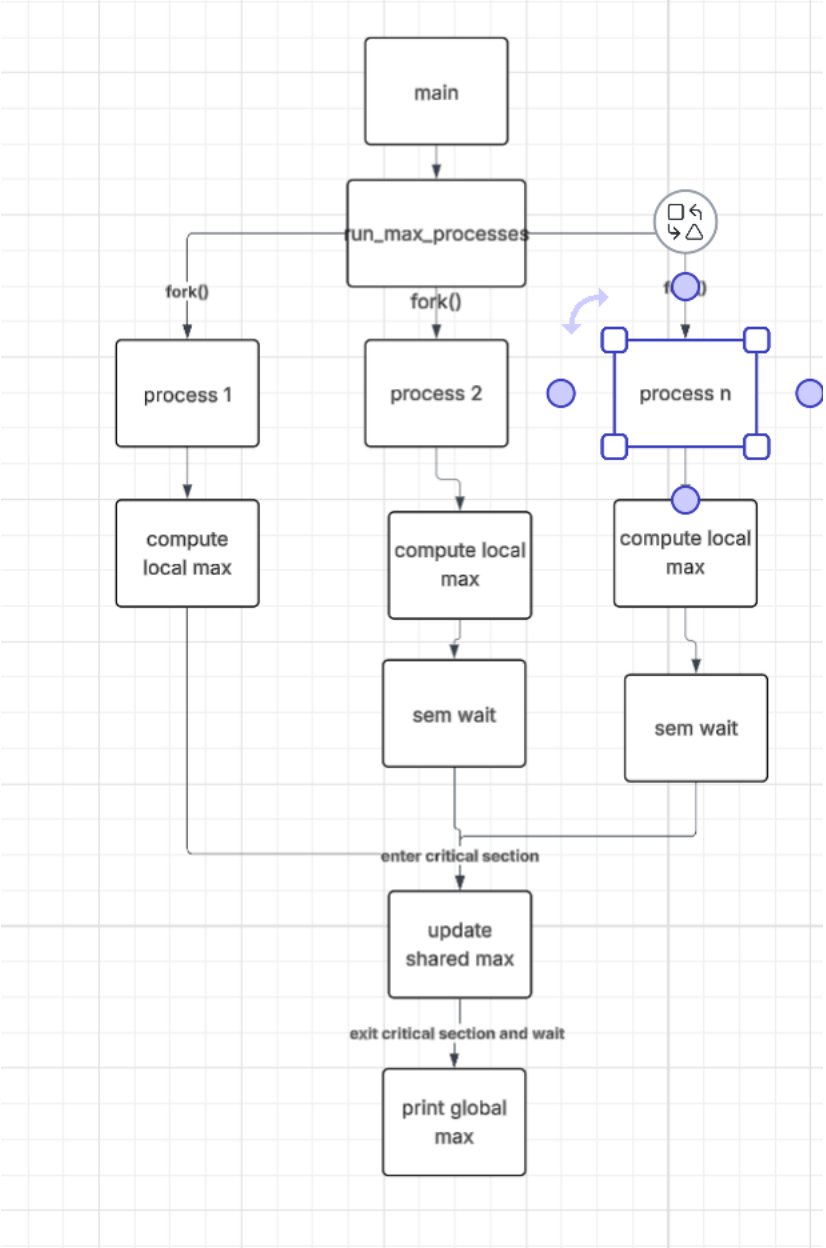

Figure 4

In Part 1 which is figures one and two, the sorting task follows the MapReduce structure by dividing the input array into chunks which would be the Map phase, where each thread or process independently sorts its assigned chunk using qsort. These sorted chunks are the intermediate results which are then

merged into a final sorted array using the merge_arrays() function and this step can be seen as the Reduce phase.

In Part 2 which is diagrammed in figures 3 and 4, each worker computes the local maximum of its assigned data chunk which can be seen as the Map phase, and then synchronizes access to a shared memory buffer to update a global maximum only if its local result is larger and that step can be the Reduce phase. Semaphores ensure mutual exclusion during the critical section, avoiding race conditions. Parallel computing with MapReduce tasks are seen in both tasks with parallelism through threads and processes, inter-process communication via shared memory like the mmap, and synchronization using semaphores.

4. Description of the implementation required by the project requirements

the project utilizes several C libraries. <pthread.h> is used for threads. <unistd.h>, <sys/wait.h>, and <sys/mman.h> are used for processes. <semaphore.h> is used to create semaphores that are used for synchronization. <time.h> is used for measuring time in seconds for each task.

Worker processes are created with the fork system call. A new child is created for every chunk of the array that requires sorting. The parent process waits for them to finish with the wait call. This helps with control but requires memory sharing and synchronization.

The main IPC used for multiprocessing is shared memory using mmap that contains arguments that allow the forked processes to read and write from a shared memory buffer. In the first task subarrays get written into a shared memory space and then the parent merges the chunks into a full sorted array. In task 2 there is a shared memory space the size of a single integer for the global maximum number and there are semaphores to protect access.

Threading is done by manual thread creation. for each chunk of data a thread is created with its own arguments that specify the segment of the array to work on. The threads are joined with pthread_join after they have executed.

To avoid race conditions semaphores were used. For threads the semaphore was initialized locally with sem_init. The processes used shared semaphores in shared memory with mmap and sem_init. Sem_wait is used before entering the critical section and sem_post is used when exiting the shared section.

I measured performance using clock time measured in seconds. It is measured for each amount of workers either 1, 2, 4, or 8. This measurement is what we used in classes like CMPSC 462 so I figured it would be OK to use here.

5. Performance evaluation

```
Part 1: Parallel Sorting (MapReduce Style)
[Threads=1] Sorting done in 0.00003 seconds
2 2 3 20 26 26 34 36 43 47 52 54 57 58 58 65 67 68 70 73 73 74 74 78 81 82 86 87 92 95 96 98
[Threads=2] Sorting done in 0.00006 seconds
2 2 3 20 26 26 34 36 43 47 52 54 57 58 58 65 67 68 70 73 73 74 74 78 81 82 86 87 92 95 96 98
[Threads=4] Sorting done in 0.00010 seconds
2 2 3 20 26 26 34 36 43 47 52 54 57 58 58 65 67 68 70 73 73 74 74 78 81 82 86 87 92 95 96 98
[Threads=8] Sorting done in 0.00024 seconds
2 2 3 20 26 26 34 36 43 47 52 54 57 58 58 65 67 68 70 73 73 74 74 78 81 82 86 87 92 95 96 98
[Processes=1] Sorting done in 0.00015 seconds
2 2 3 20 26 26 34 36 43 47 52 54 57 58 58 65 67 68 70 73 73 74 74 78 81 82 86 87 92 95 96 98
[Processes=2] Sorting done in 0.00018 seconds
2 2 3 20 26 26 34 36 43 47 52 54 57 58 58 65 67 68 70 73 73 74 74 78 81 82 86 87 92 95 96 98
[Processes=4] Sorting done in 0.00035 seconds
2 2 3 20 26 26 34 36 43 47 52 54 57 58 58 65 67 68 70 73 73 74 74 78 81 82 86 87 92 95 96 98
[Processes=8] Sorting done in 0.00069 seconds
2 2 3 20 26 26 34 36 43 47 52 54 57 58 58 65 67 68 70 73 73 74 74 78 81 82 86 87 92 95 96 98
```

```
Part 2: Max-Value Aggregation with Co
[Threads=1] Max = 98 (0.00002 sec)
[Threads=2] Max = 98 (0.00003 sec)
[Threads=4] Max = 98 (0.00006 sec)
[Threads=8] Max = 98 (0.00012 sec)
[Processes=1] Max = 98 (0.00009 sec)
[Processes=2] Max = 98 (0.00017 sec)
[Processes=4] Max = 98 (0.00032 sec)
[Processes=8] Max = 98 (0.00069 sec)
```

```
 Part 1: Parallel Sorting (MapReduce Style)
[Threads=1] Sorting done in 0.01515 seconds
first 5: 12 27 53 56 58
last 5: 999987 999989 999989 999991 999997
[Threads=2] Sorting done in 0.01556 seconds
first 5: 12 27 53 56 58
last 5: 999987 999989 999989 999991 999997
[Threads=4] Sorting done in 0.01636 seconds
first 5: 12 27 53 56 58
last 5: 999987 999989 999989 999991 999997
[Threads=8] Sorting done in 0.01974 seconds
first 5: 12 27 53 56 58
last 5: 999987 999989 999989 999991 999997
[Processes=1] Sorting done in 0.00114 seconds
first 5: 12 27 53 56 58
last 5: 999987 999989 999989 999991 999997
[Processes=2] Sorting done in 0.00218 seconds
first 5: 12 27 53 56 58
last 5: 999987 999989 999989 999991 999997
[Processes=4] Sorting done in 0.00360 seconds
first 5: 12 27 53 56 58
last 5: 999987 999989 999989 999991 999997
[Processes=8] Sorting done in 0.00589 seconds
first 5: 12 27 53 56 58
last 5: 999987 999989 999989 999991 999997
```

```
 Part 2: Max-Value Aggregation with Constrai
[Threads=1] Max = 999997 (0.00020 sec)
[Threads=2] Max = 999997 (0.00022 sec)
[Threads=4] Max = 999997 (0.00030 sec)
[Threads=8] Max = 999997 (0.00050 sec)
[Processes=1] Max = 999997 (0.00023 sec)
[Processes=2] Max = 999997 (0.00022 sec)
[Processes=4] Max = 999997 (0.00039 sec)
[Processes=8] Max = 999997 (0.00079 sec)
```

Both tasks require synchronization to perform in the correct way. The first task needs to have synchronization during the reduce phase where the subarrays get merged. The results stay the same with the number of threads and processes increasing and that is visible both when the array is 32 integers and 131072 integers with the print outs matching for each run.

Synchronization is more important to the second task where there is a singular global max which if not checked properly can end up being the incorrect number. Semaphores are important to prevent race conditions here. And again, the results stay the same with each run, so the synchronization looks like it is working. So, both tasks show the use of synchronization and how it is crucial to get the correct results.

6. Conclusion

The main result that sticks out to me is the fact that the processes were quicker at the first task with the 131072 integer array than the threads. At first, I was surprised, but then I remembered that even though threads are lightweight they are better for I/O bound tasks. So, this can be evidence that the processes are better for running CPU bound tasks.

The second thing that seems particularly important is that the synchronization was especially important for the second task. Without the semaphores syncing everything together race conditions could affect the final result.

Another thing I noticed is that the increase in the number of workers also caused the time for each task to increase. When there is more synchronization there is more overhead so everything slows down especially with the singular shared source of the second task.

The most immediate challenge I faced during implementation is that I am not the most comfortable using C. this led to many simple mistakes like forgetting ; at the end of lines or incorrectly using pointers. I also was having trouble with github pushing and pulling kept giving me errors, but it was my own fault for not properly saving before I was trying to do it. I also started out using pipes, but that was not working that great so then I changed to the mmap. The final really big challenge I had was with the diagrams. I have no idea if they are anywhere close to what they should be. They don't look like the diagrams from the powerpoints, but I wasn't sure how to do it properly. So, I am hoping those diagrams are at least OK.

One limitation could be that this is a simple simulation of MapReduce tasks so it isn't necessarily a full look at them. Everything is pretty scaled down. I also realized I didn't do any error handling. That would be an immediate improvement that I would make to the project since error handling would make debugging easier. Another improvement that could be made would be to have more varied data to test like having I/O bound tasks to see if the threads perform better with those than the processes.