CMPSC 472 - Project 1

Contents

1 Project Description **1**

2 Instructions **2**

3 Structure of the Code **2**

3.1 Part 1 2

3.2 Part 2 4

4 Description of the implementation required by project requirements **5**

4.1 Part 1 5

4.2 Part 2 6

5 Performance Evaluation **7**

5.1 Part 1 7

5.2 Part 2 9

6 Conclusion **10**

6.1 Key Findings 10

6.2 Challenges Faced During Implementation 11

6.3 Limitations and Possible Improvements 11

6.4 Take Aways 11

7 References **13**

1 Project Description

This project implements two MapReduce-style systems, one for parallel sorting and the other for maximum value aggregation with constrained memory where both demonstrate the operating system concepts such as multithreading, multiprocessing, inter-process communication (IPC), and synchronization. The goal is to simulate parallel and distributed computation on a single computer while understanding the differences threads and processes have to offer.

The first part, Parallel Sorting, applies MapReduce by dividing a large input array into smaller chunks that are sorted in parallel using Merge Sort. Each worker, a thread or a process independently sorts its designated chunk during the Map phase, and the Reduce phase merges these sorted chunks into a single fully sorted array. This design models distributed sorting in a simplified environment, where communication between workers and the reducer is implemented through local IPC mechanisms such as shared memory or message passing. The performance of the sorting system is measured using 1, 2, 4, and 8 workers to analyze how concurrency affects execution time and memory consumption with either a 32 or 131,072 array of integers.

The second part, Max-Value Aggregation with Constrained Shared Memory, focuses on synchronization and race condition prevention. Each worker computes the local maximum of its data chunk and attempts to update a shared memory buffer that holds only one value, the current global maximum. Since multiple workers may try to write to the shared buffer simultaneously, synchronization mechanisms are essential to ensure correct behavior. The reducer then reads the final maximum value. Similar to the sorting task, this system measures performance under different worker counts to evaluate how synchronization impacts parallel efficiency.

2 Instructions

The program is designed to be understandable and simple to test for different scenarios. The main thing I manually changed for each run was the array size. This could have been automated with additional programming, but since we only tested two array sizes, I chose to set it manually each time. Each execution automatically runs the computations for different worker counts: 1, 2, 4, and 8. This setup makes the program simple to use and allows testing multiple scenarios without needing further modifications.

3 Structure of the Code

3.1 Part 1

Note: Array size is subject to change, this is just one instance of an example.

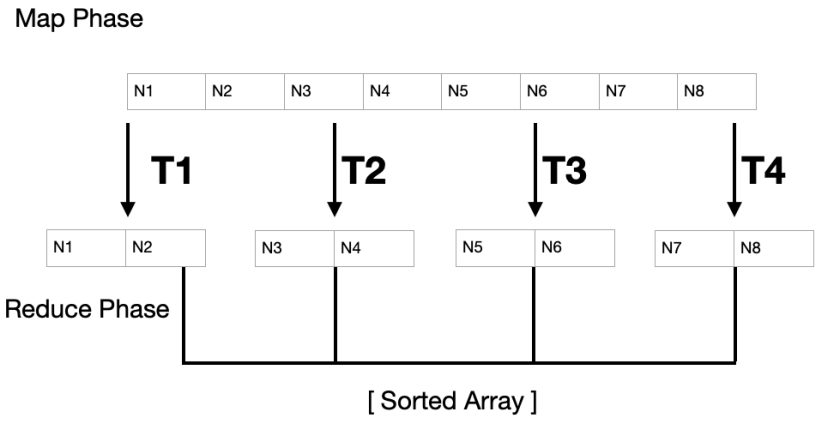

Figure 1: Thread-Based Implementation

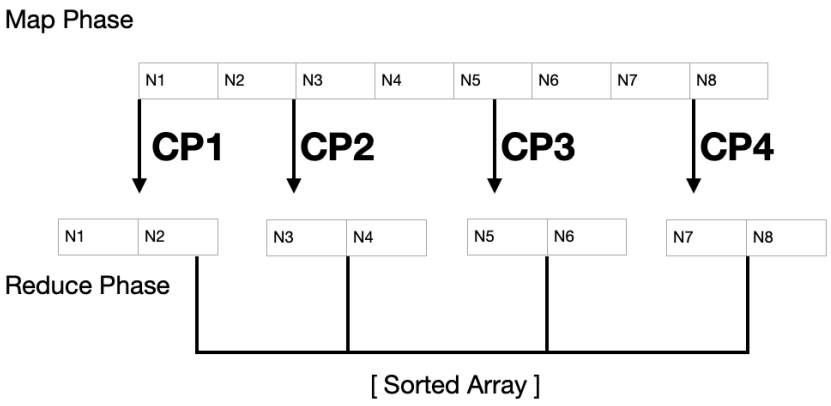

Figure 2: Process-Based Implementation

The parallel sorting implementation is designed with a MapReduce-style. In the map phase, the input array is divided into chunks, and multiple workers process these chunks in parallel, using an algorithm like merge sort. For the multithreaded version, as shown by Figure 1, each worker is implemented as a thread, and has distributed chunks to process. All threads share access to the same memory region so this shared memory enables efficient communication between threads, allowing them to directly read and write to the array. Additionally, synchronization mechanisms such as mutexes are used to prevent race conditions during concurrent access to shared data.

For the multiprocessing version, the diagram is similar, but the parent process forks multiple child processes, as shown by Figure 2. Each child operates on its own chunk of the array, which is allocated in a shared memory section to allow inter-process access. Communication from the children to the parent is performed with a POSIX pipe, where each child sends a small descriptor containing the starting index and length of its sorted chunk. The parent process collects all descriptors and performs a merge in the reduce phase, the final sorted array.

This structure resembles the MapReduce style as the map phase independently sorts chunks, and the reduce phase merges the results into the final sorted array. Both implementations utilize paral-

lelism to improve performance, but they are different in the method of communication and memory sharing, threads benefit from direct shared memory access, while processes require inter-process communication.

3.2 Part 2

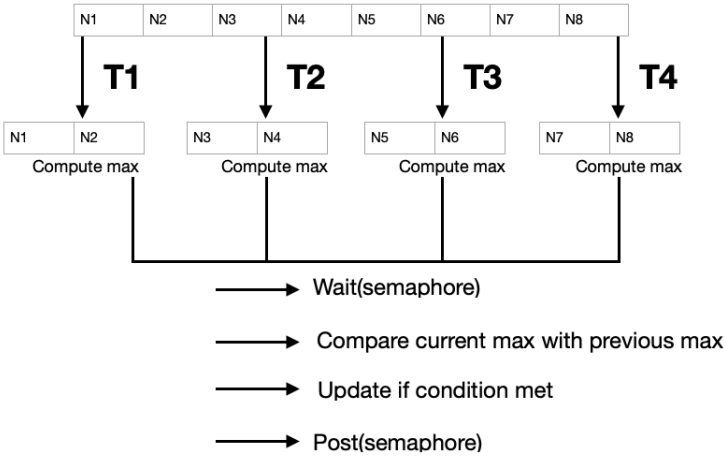

Figure 3: Thread-Based Implementation

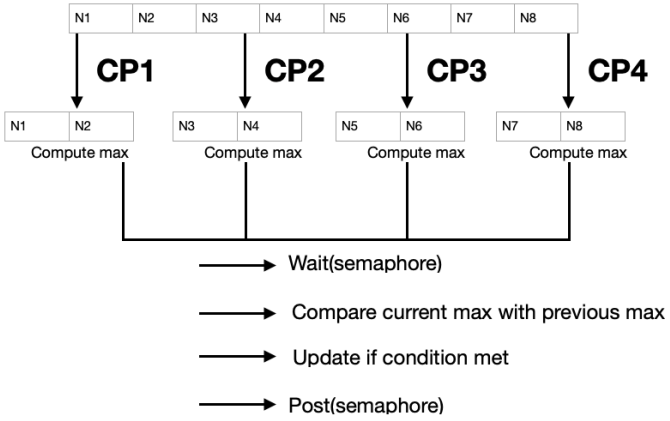

Figure 4: Process-Based Implementation

Part 2 implementation of max-value aggregation is once again a MapReduce Style. In the thread-based version, the array is divided into chunks, and multiple threads compute the local maximum of each chunk in parallel. All threads share access to the global maximum variable in memory, and synchronization is enforced using either a mutex or atomic operations to ensure that updates to the shared integer are performed safely. Once all threads complete their computations, the main thread reads the final value, completing the reduce phase. In the process-based version, the parent forks multiple child processes, each of which computes the local maximum of its assigned chunk. The global maximum is stored in a shared variable, and access is synchronized with a semaphore. Each child waits on the semaphore before comparing and potentially updating the global maximum. After all children terminate, the parent reads the final value from shared memory, producing the reduced result. Both implementations incorporate to the MapReduce structure, the map phase performs independent local computations, and the reduce phase merges results into a single global maximum. The difference is in memory and communication, threads operate in shared memory directly, while processes require explicit shared memory and inter-process synchronization through semaphores.

4 Description of the implementation required by project requirements

4.1 Part 1

Part 1 has the parallel sorting implementation that uses POSIX threads for multithreading. The array is first allocated in the main thread, and multiple worker threads are spawned manually to perform the map phase, where each thread independently sorts a chunk of the array using merge sort. The sorted chunk information is communicated to the reducer and protected by a mutex and condition variables,

ensuring safe access and proper synchronization between mapper threads and the reducer. A dedicated reducer thread collects all chunk descriptors from the queue and performs a merge to produce the final sorted array. Manual thread creation is used for both mapper and reducer threads rather than a thread pool, to allow for simpler synchronization and clear control of thread lifetimes. Performance is evaluated by measuring map and reduce execution times, as well as maximum resident set size to track memory usage. This design uses shared memory for efficient inter-thread communication while maintaining synchronized access to shared resources.

4.2 Part 2

Part 2 implements a global maximum computation over a large integer array using both threads and processes. The threaded version uses POSIX threads, where each worker thread calculates a local maximum for its assigned array chunk. All threads share access to a single global maximum variable, and synchronization is handled with a mutex to ensure race-free updates. The process-based version uses fork to spawn multiple child processes, each computing the local maximum of its assigned chunk. The global maximum is stored in a shared memory region, and updates are synchronized using a POSIX semaphore, which provides kernel-level mutual exclusion suitable for inter-process communication. Children write their local maximum to this shared variable inside a critical section protected by the semaphore, and the parent waits for all children to terminate before reading the final global maximum. Performance is measured like in Part 1, using high-resolution timers for the map and reduce phases to monitor CPU and memory usage for both parent and child processes. The code shows explicit worker management, proper IPC with shared memory and semaphores, and careful synchronization, demonstrating a clear MapReduce style, the map phase performs independent local computations, and the reduce phase merges the results into a single global value.

5 Performance Evaluation

5.1 Part 1

```
Multithreading, array size: 32
Workers=1 | Map=0.189 ms | Reduce=0.003 ms | Total=0.192 ms | MaxRSS=104112 KB
Workers=2 | Map=0.102 ms | Reduce=0.001 ms | Total=0.103 ms | MaxRSS=104112 KB
Workers=4 | Map=0.183 ms | Reduce=0.016 ms | Total=0.199 ms | MaxRSS=104112 KB
Workers=8 | Map=0.549 ms | Reduce=0.011 ms | Total=0.559 ms | MaxRSS=104112 KB
```

Figure 5

```
Multiprocessing sort demo, size=32
Workers=1 Map=0.319ms Reduce=0.009ms Total=0.328ms
Workers=2 Map=0.648ms Reduce=0.004ms Total=0.651ms
Workers=4 Map=0.906ms Reduce=0.004ms Total=0.910ms
Workers=8 Map=1.013ms Reduce=0.004ms Total=1.016ms
```

Figure 6

Comparing the total time for multithread and multiprocess, for an array of 32 elements, based on Figure 5 and Figure 6, it is evident that multithreading was faster, which aligns with my initial hypothesis. This makes sense because threads share the same memory space, allowing them to read and write to the array without the overhead of creating separate memory spaces or performing inter-process communication. The smaller array size also means that the overhead of thread management is minimal compared to the small computation workload, so threads can complete their tasks quickly.

```
Multithreading, array size: 131072
Workers=1 | Map=34.238 ms | Reduce=2.641 ms | Total=36.879 ms | MaxRSS=104112 KB
Workers=2 | Map=22.483 ms | Reduce=4.745 ms | Total=27.228 ms | MaxRSS=104112 KB
Workers=4 | Map=20.785 ms | Reduce=8.672 ms | Total=29.457 ms | MaxRSS=104112 KB
Workers=8 | Map=19.362 ms | Reduce=9.999 ms | Total=29.361 ms | MaxRSS=104112 KB
```

Figure 7

```
Multiprocessing sort demo, size=131072
Workers=1 Map=34.304ms Reduce=2.327ms Total=36.631ms
Workers=2 Map=7.974ms Reduce=2.185ms Total=10.159ms
Workers=4 Map=7.821ms Reduce=2.372ms Total=10.192ms
Workers=8 Map=7.665ms Reduce=2.686ms Total=10.351ms
```

Figure 8

However, comparing the total time for multithread and multiprocess for an array of 131,072 elements, based on Figure 7 and Figure 8, it is evident that multiprocessing was significantly faster, which initially might seem surprising. This behavior can be explained by memory and CPU utilization, for larger arrays, multiprocessing allows each process to run on a separate CPU core with its own memory space, reducing contention for shared resources. In contrast, multithreading, while still parallel, may suffer from contention due to synchronization mechanisms such as mutexes protecting the global maximum variable. As the workload grows, the overhead from threads frequently waiting on locks can increase, slowing down overall execution.

5.2 Part 2

```
Max value aggregation (mutex mode), array size=32
Workers=1 | GlobalMax=2138202520 | Map=0.246 ms | Reduce=0.000 ms | Total=0.246 ms | MaxRSS=117680 KB
Workers=2 | GlobalMax=2138202520 | Map=0.093 ms | Reduce=0.000 ms | Total=0.093 ms | MaxRSS=117680 KB
Workers=4 | GlobalMax=2138202520 | Map=0.129 ms | Reduce=0.000 ms | Total=0.129 ms | MaxRSS=117680 KB
Workers=8 | GlobalMax=2138202520 | Map=0.421 ms | Reduce=0.000 ms | Total=0.421 ms | MaxRSS=117680 KB
```

Figure 9

```
Max value aggregation (process + semaphore), array size=32
Workers=1 | GlobalMax=2138202520 | Map=0.292 ms | Reduce=0.000 ms | Total=0.292 ms | MaxRSS=117680 KB
Workers=2 | GlobalMax=2138202520 | Map=0.497 ms | Reduce=0.000 ms | Total=0.497 ms | MaxRSS=117680 KB
Workers=4 | GlobalMax=2138202520 | Map=0.672 ms | Reduce=0.000 ms | Total=0.672 ms | MaxRSS=117680 KB
Workers=8 | GlobalMax=2138202520 | Map=0.938 ms | Reduce=0.000 ms | Total=0.938 ms | MaxRSS=117680 KB
```

Figure 10

For max-value aggregation with an array of size 32, it is clear from figure 9 and figure 10 that multithreading is faster than multiprocessing. This makes sense because threads share memory, so they can directly access and update the global maximum without creating separate memory spaces or doing inter-process communication. Threads also have lower creation and context-switching overhead, which matters more when the array is small and tasks are lightweight.

```
Max value aggregation (mutex mode), array size=131072
Workers=1 | GlobalMax=2147467951 | Map=0.628 ms | Reduce=0.000 ms | Total=0.628 ms | MaxRSS=117680 KB
Workers=2 | GlobalMax=2147467951 | Map=0.453 ms | Reduce=0.000 ms | Total=0.453 ms | MaxRSS=117680 KB
Workers=4 | GlobalMax=2147467951 | Map=0.528 ms | Reduce=0.000 ms | Total=0.529 ms | MaxRSS=117680 KB
Workers=8 | GlobalMax=2147467951 | Map=0.623 ms | Reduce=0.000 ms | Total=0.623 ms | MaxRSS=117680 KB
```

Figure 11

```
Max value aggregation (process + semaphore), array size=131072
Workers=1 | GlobalMax=2147467951 | Map=1.227 ms | Reduce=0.000 ms | Total=1.227 ms | MaxRSS=117680 KB
Workers=2 | GlobalMax=2147467951 | Map=1.321 ms | Reduce=0.000 ms | Total=1.321 ms | MaxRSS=117680 KB
Workers=4 | GlobalMax=2147467951 | Map=2.106 ms | Reduce=0.000 ms | Total=2.106 ms | MaxRSS=117680 KB
Workers=8 | GlobalMax=2147467951 | Map=2.345 ms | Reduce=0.000 ms | Total=2.346 ms | MaxRSS=117680 KB
```

Figure 12

Even for the larger array size of 131,072, multithreading still appears faster in our results. This can be explained by the fact that although the larger workload benefits from parallelism, threads still avoid the extra overhead of forked processes, which must create separate memory spaces and coordinate updates to the shared global maximum through semaphores. Additionally, because the update to the global maximum is protected by a mutex, contention can slow threads slightly, but this overhead is still less than the cost of creating and managing multiple processes. In other words, shared memory and synchronization in multithreading allow it to perform better in both small and moderately large workloads, while processes only show advantages in very large or CPU-bound tasks where separate memory and parallel CPU utilization outweigh the overhead.

6 Conclusion

6.1 Key Findings

For small arrays, multithreading is faster than multiprocessing because threads share memory and have lower overhead for creation and context switching, making them more efficient for lightweight tasks. For large arrays, multiprocessing can outperform multithreading since each process works independently, reducing contention on shared variables and better utilizing multiple CPU cores. Synchronization mechanisms like mutexes and semaphores have a significant impact on performance. Threads often spend time waiting for locks when updating shared variables, which slows execution as dataset size grows. The MapReduce-style divides work into chunks in the map phase and combining results in the reduce phaseâproved effective for both threads and processes, especially when the workload increases, demonstrating the benefits of parallelization.

6.2 Challenges Faced During Implementation

I encountered multiple obstacles during this project. Managing synchronization correctly was difficult, especially when making sure updates to the global maximum were atomic and preventing race conditions. Debugging the process-based version was harder because each process has its own memory space, and using POSIX semaphores for inter-process communication added more challenges. Choosing between atomic operations, mutexes, and semaphores also involved understanding the differences between performance and correctness, which sometimes required experimenting.

6.3 Limitations and Possible Improvements

Thread and process creation are currently handled manually, however, implementing thread or process pools could reduce overhead and improve performance, especially for repeated or large-scale tasks. Synchronization in the multithreaded implementation could also be optimized by minimizing updates to the shared maximum or using more fine-grained locking mechanisms. Memory usage tracking is simple, and incorporating more detailed implementation could offer deeper insights into system resource consumption. Lastly, maybe to distribute work more evenly among workers could enhance both efficiency and overall responsiveness.

6.4 Take Aways

This project demonstrates the differences between multithreading and multiprocessing for parallel computing. Threads excel in low-overhead, shared-memory tasks, while processes are better for large workloads with potential contention. The experiments demonstrate the importance of synchronization and the trade-offs between speed and correctness. Overall, the project provides a practical understanding of operating system concepts like threads, processes, shared memory, and semaphores, and shows

how careful design is important for performance and safety. These insights can be applied to real-world parallel computing problems where workload size, memory contention, and synchronization overhead must be considered.

7 References

- https://www.geeksforgeeks.org/c/multithreading-in-c/
- https://www.geeksforgeeks.org/python/multiprocessing-python-set-1/
- https://www.geeksforgeeks.org/dsa/merge-sort/
- https://docs.oracle.com/cd/E19455-01/806-5257/6je9h032r/index.html
- https://www.kernel.org/doc/html/latest/locking/locktypes.html
- OpenAI. (n.d.). *ChatGPT*.
- https://www.youtube.com/watch?v=mc4BtkJ86Ww
- https://www.youtube.com/watch?v=q9YBJM3IaMQ
- https://www.youtube.com/watch?v=nlHIuG3RQ0g