

CMPSC 472

Project 1: MapReduce Systems for Parallel Sorting and Max-Value Aggregation with Constrained Memory

Erand Vejseli

**1. Project Description**

This project explores how the MapReduce programming model can be simulated using multithreading, multiprocessing, and inter-process communication (IPC).  
The goal is to gain hands-on experience with core operating system concepts such as process management, synchronization, and shared memory through two simplified MapReduce-style tasks:

1. Parallel Sorting – Sorting a large array by dividing it among multiple workers.
2. Max-Value Aggregation – Computing a global maximum value using a single shared integer in memory.

**Overview of the Two MapReduce-Style Tasks**

Task 1: Parallel Sorting

* Input: Large array of integers (N = 32 and N = 131,072).
* Map Phase: The array is divided into equal chunks; each worker (thread or process) sorts its chunk using qsort().
* Reduce Phase: The reducer merges all sorted chunks into a final sorted array using pairwise merging.
* Objective: Evaluate how increasing worker counts (1, 2, 4, 8) affect performance using both threading and multiprocessing.

Task 2: Max-Value Aggregation with Constrained Shared Memory

* Input: Large array of integers.
* Map Phase: Each worker computes a local maximum from its chunk.
* Reduce Phase: Workers attempt to update a single shared memory integer holding the current global maximum.
* Synchronization: Threads use mutex locks, while processes use POSIX semaphores to prevent race conditions.
* Constraint: Only one integer can exist in shared memory, forcing strict synchronization and atomic access.

**Why Use Multithreading, Multiprocessing, and Synchronization?**

* Multithreading shares memory space, allowing faster communication between workers but requiring careful synchronization to avoid data races.
* Multiprocessing isolates memory spaces, making synchronization safer but slower due to IPC overhead (pipes or shared memory).
* Synchronization ensures correctness by enforcing exclusive access to shared variables or resources, preventing inconsistent updates and data corruption.
  1. **Instructions on how to run**

To compile the programs:   
gcc -02 -pthread ParallelSortingMapReduce.C -o ParallelSortingMapReduce

gcc -02 -pthread MaxValueMapReduce.C -o MaxValueMapReduce

To run these programs:

For the Parallel Sorting:

./ParallelSortingMapReduce threads 4 131072

./ParallelSortingMapReduce procs 8 131072

For the Max Value Aggregation:

./MaxValueMapReduce threads 4 131072

./MaxValueMapReduce procs 8 131072

./MaxValueMapReduce threads 8 131072 nosync --- to test without synchronization

Can also try different variants of it by changing the number of workers, or switching up the mode, threads or process.

* 1. **& 4. Structure of the Code & Implementation discussion**

Both programs follow the same general MapReduce structure: a Coordinator (the main process) divides the work, multiple workers (threads or processes) handle different portions of the data in parallel, and finally a Reducer combines the results into one output. The diagram below shows this flow.

Coordinator

Map Phase

Threads/Processings

IPC / SHARED

(pipes, memory)

Reducer

In both programs, the Coordinator (the main process) creates workers dynamically based on the number of workers (W) given as a command-line argument.

For the thread version, the program uses pthread\_create() inside a loop to start each thread. Each thread receives its own assigned portion of data to work on. When the threads are done, the main program waits for them to finish using pthread\_join() before moving to the reduce phase.

For the process version, the program uses the fork() system call to create separate child processes. Each child sorts its chunk of the array or computes its local maximum. When finished, the parent process waits for all the children using waitpid() and then performs the final reduce step. This structure makes it easy to scale to multiple workers and keeps the same logic consistent whether threads or processes are used.

**Communication and IPC**

The “IPC / SHARED” box in the diagram represents how data moves between the workers and the coordinator (or reducer).

In the ParallelSortingMapReduce.c program, this communication happens through pipes when running in process mode. Each process sorts its own section of the array, then writes that sorted chunk to a pipe. The parent process reads all of those chunks and merges them into a single sorted array during the reduce phase. When using threads instead of processes, no IPC is needed because all threads naturally share the same memory space.

In the MaxValueMapReduce.c program, communication happens through shared memory instead of pipes. All workers share access to one integer that stores the current global maximum. Since multiple workers might try to update that value at the same time, synchronization is required. Threads use a mutex lock, while processes use a POSIX semaphore to control access. This ensures that only one worker can modify the shared value at any given time, preventing race conditions and guaranteeing a correct result.

**How the Code Fits the MapReduce Framework**

Both programs are designed to follow classic MapReduce concept.

The Map Phase happens first, when each thread or process handles its portion of the data in parallel by either sorting its chunk in the sorting program or finding its local maximum in the max-value program.

Then comes the communication stage. This is where intermediate results are sent from the workers back to the coordinator. In my implementation, this is done through pipes (for sorted chunks in the process-based sorting program) or through shared memory (for the single shared integer in the max-value program).

Finally, the Reduce Phase is when the coordinator combines all of the worker results. For sorting, it merges the sorted subarrays into one complete sorted array. For the max-value program, it simply reads the final maximum value from shared memory. Throughout the entire process, the Coordinator manages worker creation, synchronization, and timing before printing the final output.

Altogether, these components demonstrate how the MapReduce framework can be simulated locally: distributing tasks, running them in parallel, and combining the results efficiently using fundamental operating system mechanisms.

**Implementation details:**Libraries used:

* <pthread.h> for threads and mutexes
* <sys/wait>, <unistd.h> for processes control
* <sys/mman.h> and <semaphore.h> for shared memory and semaphores.

Sorting: qsort() was used for performance optimization and simplicity.

Merging: A custom function merges two sorted chunks of numbers until the full array is back.

Shared Memory: Only one integer is stored (/mr\_one\_int region).

Synchronization:

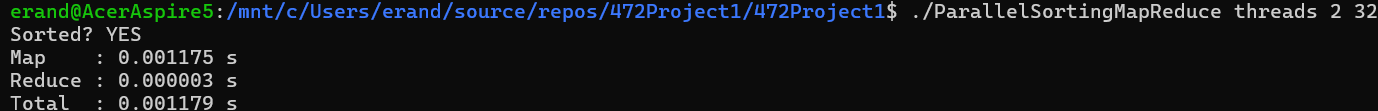
* Threads: pthread\_mutex\_lock() / pthread\_mutex\_unlock()
* Processes: sem\_wait() / sem\_post()

Performance Measurement: Used clock\_gettime(CLOCK\_MONOTONIC) to measure map phase, reduce phase, and total runtime.

**5. Performance Evaluation**

**Correctness check**

Before testing performance, I verified that both programs produced correct results using a small array of 32 elements. For the sorting program, the output confirmed Sorted? YES, and for the max-value program, the reported Global Max matched the serial check.



**Performance Assessment**

I tested performance using 131,072 elements and different number of workers for both thread-based and process-based versions.   
The runtimes were captured with clock\_gettime(CLOCK\_MONOTONIC) for the Map phase, **A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer program

AI-generated content may be incorrect.**Reduce phase, and Total time.

**5. Thread vs Process Comparison**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Program** | **Mode** | **Workers** | **Map (s)** | **Reduce (s)** | **Total (s)** |
| ParallelSortingMapReduce | Threads | 4 | 0.006403 | 0.001763 | 0.008167 |
| ParallelSortingMapReduce | Processes | 4 | 0.008708 | 0.001216 | 0.010216 |
| MaxValueMapReduce | Threads | 4 | 0.001175 | 0.000003 | 0.001179 |
| MaxValueMapReduce | Processes | 4 | 0.000667 | 0.000006 | 0.000673 |
| MaxValueMapReduce | Threads | 8 (nosync) | 0.003379 | 0.000000 | 0.003380 |

Threads performed slightly faster overall, especially in the Map phase, because they share memory and avoid the cost of inter-process communication.  
Processes were a bit slower due to fork() overhead and data transfer through pipes or shared memory, but they remained consistent and isolated.

Even with four workers, scaling was efficient. At eight workers, returns would likely diminish further due to additional scheduling and synchronization costs.

**Synchronization Discussion (MaxValueMapReduce)**

When synchronization was enabled (using a mutex for threads and a semaphore for processes), results were always correct.  
In the “nosync” run, the program still printed a number close to the correct maximum but could easily produce incorrect values due to race conditions.  
This shows that proper locking has a minor timing cost but is absolutely necessary for data consistency.

**Analysis of Results**

* Threads vs Processes: Threads were faster on this machine because of shared memory and no IPC overhead.
* Scalability: Performance improved from 1 to 4 workers, but beyond that the gains would likely flatten out.
* Synchronization cost: Negligible compared to overall runtime, yet critical for correctness.

**6. Conclusion**

**Key Findings**

* Thread-based parallelism was faster overall.
* Processes added overhead but demonstrated real OS concepts.
* Synchronization was critical for correctness.
* Scaling improved up to 4 workers, then flattened.
* Map and Reduce balance mattered.
* Overall, both programs achieved the goal.

**Challenges**

* **Thread vs process behavior:** It was tricky to design the code so both modes (threads and processes) shared the same logic while still using different mechanisms underneath. Managing memory safely in the thread version while isolating data for the process version took a lot of debugging and code redesign.
* **Shared memory and IPC setup:** Working with shared memory (shm\_open, mmap, ftruncate) and semaphores was one of the hardest parts. It’s easy to forget to unlink them or close the descriptors, which causes “resource busy” errors or dangling objects on reruns. A lot of debugging here also.
* **Pipe communication:** In the sorting program, making each child process send its sorted chunk through a pipe sounded simple on paper, but managing read/write order, exact byte counts, and partial reads required very careful implementation and was much more time consuming than I thought.
* **Synchronization debugging:** Getting locks to behave correctly without freezing the program took several tests. I ran into situations where a deadlock occurred because a thread never released the mutex, or a semaphore wasn’t initialized properly across processes.
* **Timing and performance measurements:** Another subtle challenge was ensuring that performance measurements were consistent and accurate. Measuring just the “map” and “reduce” times separately was a little challenging too.
* **Race conditions and testing:** The “nosync” version of MaxValueMapReduce made it easy to trigger race conditions.
* **Scaling behavior:** Adding more workers doesn’t always improve performance.

**Limitations and Possible Improvements**

There weren’t really big limitations, but this project took a lot of time and patience. Getting everything to work together threads, processes, pipes, shared memory, and locks was already a big task. Some parts, like making the merging faster, including a min heap in its algorithm or adding better performance tracking, would’ve taken much longer to do the right way.

If I ever come back to this project, I’d also like to clean up the code structure a bit and maybe add a simple way to see how each part runs and how the performance changes with more workers.