CMPSC 472 – MapReduce Systems

Gabriel Silva

Environment: macOS, python 3.12

Date: October 2025

1: Project Description

This Project implements MapReduce style computing tasks on a single host to practive parallelism, inter-process communication, and synchronization using multithreading and multiprocessing.

Task 1: Parallel sorting

Map Phase: This is the input array of integers which is split into chunks; each mapper sorts its chunks using an in-memory merge sort.

Reduce Phase: The ruducer merges the sorted chunks into a globally sorted array.

This is implemented in both threaded and process-based versions.

Task 2: Max Value Aggregation with Constrained Memory

Each mapper computes the local maximum of its chunk

All Workers update a single shared integer that represents the current global maximum.

A lock that will guarantee atomic read -> compare -> write that will prevent race conditions.

Implemented with both threads and processes.

Why do multithreading, multiprocessing, and synchronization

Multithreading has a low overhead, shared memory, and has easier communication

Multiprocessing gives us true parallelism across all cpu cores and avoids Python GIL

Synchronization which is required in task 2 that will maintain correctness when multiple workers are attempting to update to a single shared thread

2: Instructions

Part 1 – Parallel Sorting

Mulithreading:

python mapreduce_os_project.py sort-thread --size 131072 --workers 4

Multiprocessing:

python mapreduce_os_project.py sort-proc --size 131072 --workers 4

Part 2 Max-Value Aggregation

Mulithreading:

python mapreduce_os_project.py max-thread --size 131072 --workers 8

Multiprocessing:

python mapreduce_os_project.py max-proc --size 131072 --workers 8

Benchmark Which is all modes sizes and worker counts

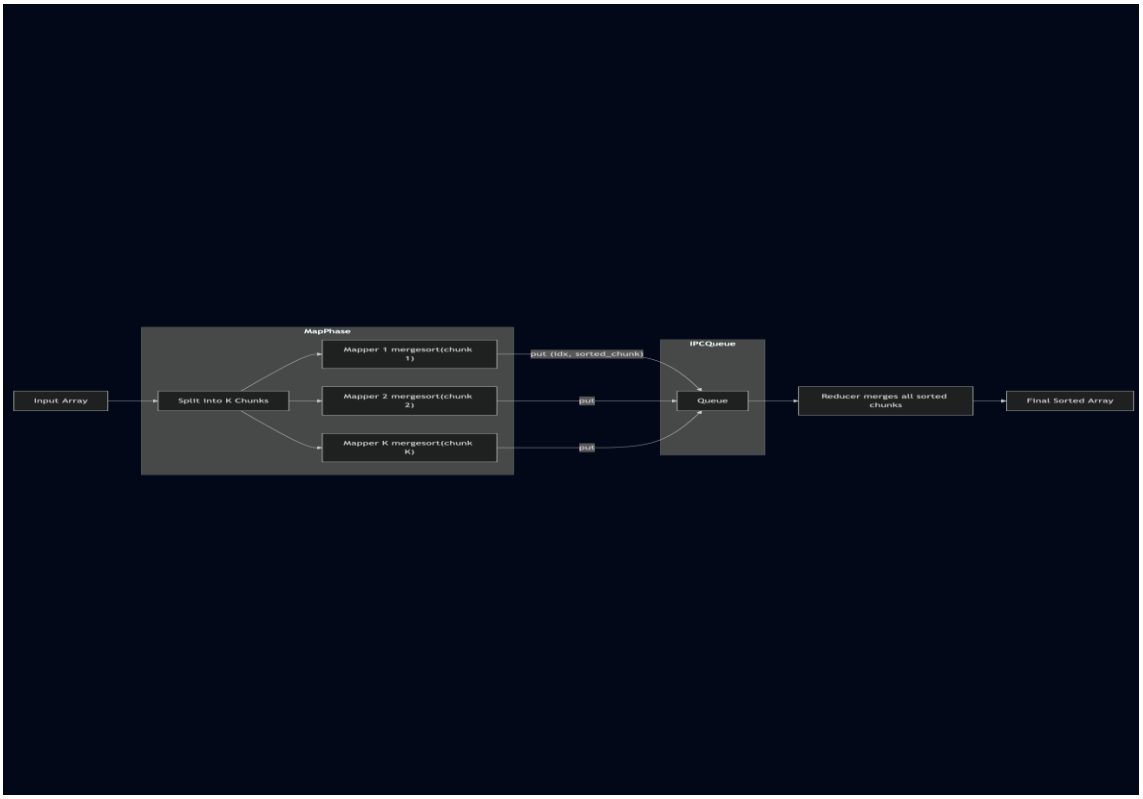
python mapreduce_os_project.py bench > results.csv

3: Structure of Code

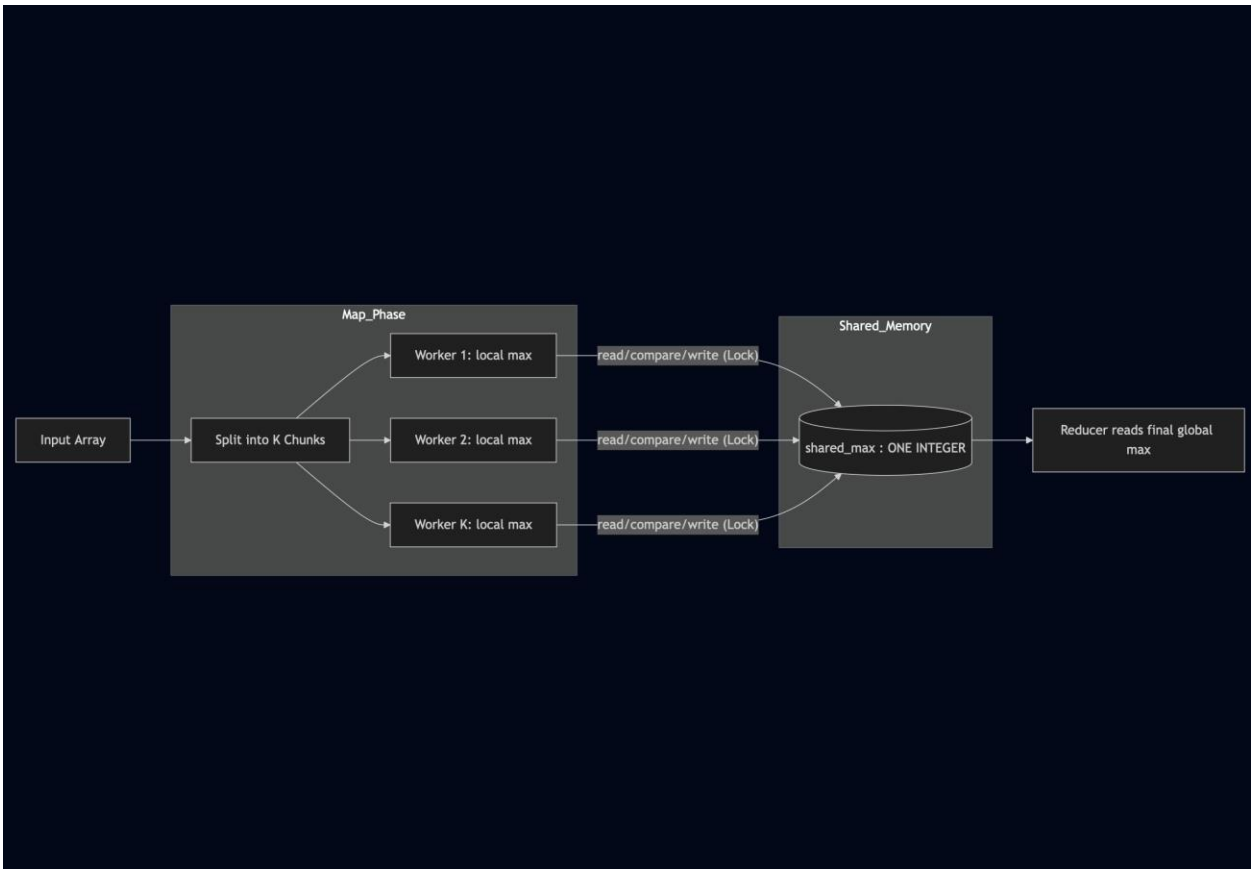
Function	Purpose
mergeSort()	Merge sort algo for mapper soring
reducer_merge	k-way merge of mapper outputs
sort_threaded()	Thread based parallel sorting
sort_process()	Process based soring wiht a simple queue IPC.
max_threaded()	Thread based single int global max
max_process()	Process based single in global max
run_bench()	Runs all tests

Diagrams:

Parallel Sorting:



Max Value Aggregation



4:Implementation Details

Tools/Libraries: threading, multiprocessing, queue, argparse, resource, time.

Process Management:

Each mapper process is started via the multiprocessing.get_context(“spawn”) process and the results are passed through a simple queue to avoid blocking.

IPC Mechanisms:

Sorting uses a queue threads or simple queue processes for transfering the sorted chunks.

The is drained before joining processes to prevent deadlocks

Max task uses a single shared Value('q') and a lock

Threading:

Manual thread creation with one thread per mapper, explicit joins.

Synchronization:

A lock ensures the atomic updates to the one integer shared buffer

5: Performance

Correctness:

All nodes produces identical results

Task	mode	Workers	times	Result
Sorting	Threaded	4	0.152 s	Ok_sorted = True
Sorting	Process	4	0.219s	Ok_sorted = True
Max Value	Threaded	8	0.00196s	Global_max = 999 990 429
Max Value	Process	8	0.223s	Global_max = 999 990 429

6: Results

Part 1:

Threaded sort:

```
(venv) gabrielsilva@Gabriels-MacBook-Air Project1_CMPSC472 % python map
ort-thread --size 131072 --workers 4

time=0.152168s, mem_delta_mb=6.406, ok_sorted=True
(venv) gabrielsilva@Gabriels-MacBook-Air Project1_CMPSC472 %
```

Process Sort:

```
(venv) gabrielsilva@Gabriels-MacBook-Air Project1_CMPSC472 % python mapreduce_os_pro
ort-proc --size 131072 --workers 4

time=0.211622s, mem_delta_mb=8.109, ok_sorted=True
(venv) gabrielsilva@Gabriels-MacBook-Air Project1_CMPSC472 %
```

Part 2:

Threaded Max Value:

```
(venv) gabrielsilva@Gabriels-MacBook-Air Project1_CMPSC472 % python mapreduce_os_project.py m
ax-thread --size 131072 --workers 8

time=0.001871s, global_max=999990429
(venv) gabrielsilva@Gabriels-MacBook-Air Project1_CMPSC472 %
```

Process Max Value:

```
(venv) gabrielsilva@Gabriels-MacBook-Air Project1_CMPSC472 % python mapreduce_os_project.py m
ax-proc --size 131072 --workers 8

time=0.226587s, global_max=999990429
(venv) gabrielsilva@Gabriels-MacBook-Air Project1_CMPSC472 %
```

Synchronization Test:

```
--- Synchronization Performance Comparison ---
(size=131072, workers=8 for max tests; threads/processes use same data)
max-thread SAFE        | time=0.002000s | global_max=999990429
max-thread UNSAFE      | time=0.001270s | global_max=999990429
max-proc SAFE          | time=0.231453s | global_max=999990429
max-proc UNSAFE        | time=0.228334s | global_max=999990429
(venv) gabrielsilva@Gabriels-MacBook-Air Project1_CMPSC472 %
```

Discussion/Conclusion:

The threads completed faster because they share memory and avoid serialization of large sorted chunks. The processes perform true parallel CPU work but have a

spawn + IPC overhead. Both versions returned the correct global maxima. Threaded execution was almost instantaneous because the lock content was minimal. The process version was a lot slower, because it was dominated by process startup and interprocess coordination rather than computation. Increasing worker count from 1 to 8 showed a diminishing return consistent with Amdahl's law and the GIL limiting parallel python bytecode execution.

Some key findings are that threads are more efficent for small to medium CPU bound tasks in Python due to lower IPC overhead. Processes enable true concurrency but are a lot more costly to start and communicate with. Lastly, the synchronization via a simple lock correctly protected shared memory and prevented race conditions.

Some of the challenges I faced were the fact that in macOS with the default spawn start method, the process targets must be a top-level function to be pickable. Also, a I accidentally created a join -> get() order that caused deadlocks when the queues filled, draining this before the join resolved this.

Some of the limitations i faced and some improvements I believe could help here are the Python GIL restricts CPU bound scaling, native extensions or C/NumPy function that releases GIL would scale better. Also, IPC could be optimized using multiprocessing.shared_memory or file back mmap for larger arrays. Finally, adding a automated chart generation from results.csv would help visualize the scaling trends.