**CMPSC 472 – Operating Systems--Section 001**

**Professor:** Janghoon Yang

**Author:** Jomiloju Odumosu

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

## Git Link for Code: https://github.com/jto5108/cmpsc472-project1.git

# 1. Project Description

## Overview of the Two MapReduce-Style Tasks

In this project, I implemented two MapReduce-style systems on a single machine: **Parallel Sorting** and **Max-Value Aggregation with Constrained Shared Memory**.
Both tasks follow the MapReduce model, where data is first processed in parallel by several workers (Map phase), and then combined by a single reducer (Reduce phase).

- **Parallel Sorting:**
  The input is a large array of integers. The array is divided into equal chunks, and each worker sorts one chunk in parallel using either threads or processes.
  After all chunks are sorted, a reducer merges the sorted chunks into a final sorted array.
- **Max-Value Aggregation:**
  Each worker computes the local maximum of its chunk. Workers share a single memory buffer that stores only one value — the current global maximum. Synchronization is required so that only one worker updates the shared value at a time if its local maximum is higher.

## Why I Considered Multithreading, Multiprocessing, and Synchronization

I used **multithreading** and **multiprocessing** to explore how parallelism affects performance and resource usage in different environments:

- **Multithreading** is lightweight and allows threads to share memory directly, making data transfer faster but still limited by the Global Interpreter Lock (GIL) in Python for CPU-bound tasks.
- **Multiprocessing** runs separate processes with independent memory spaces, allowing true parallel execution across CPU cores but requiring inter-process communication (IPC) through queues or shared values.

- **Synchronization** was essential in the Max-Value task to prevent race conditions when multiple workers attempted to update the same shared integer. Without locks, results would be unpredictable or incorrect.

Overall, I wanted to compare how threads and processes handle parallel sorting and aggregation under controlled synchronization and memory constraints.

## 2. Instructions

To run my system, I wrote two Python scripts:
`mapreduce_sort.py` (Parallel Sorting) and `max_aggregation.py` (Max-Value Aggregation).

Both scripts can be run from the terminal with adjustable parameters.

### Parallel Sorting

```
python mapreduce_sort.py --mode thread --workers 4 --size 131072
python mapreduce_sort.py --mode process --workers 8 --size 32
```

- `--mode` selects between `thread` and `process`
- `--workers` sets the number of parallel workers (1, 2, 4, or 8)
- `--size` defines the input array size

### Max-Value Aggregation

```
python max_aggregation.py --mode thread --workers 4 --size 131072
python max_aggregation.py --mode process --workers 8 --size 32
```

Each script prints:

- Mode, number of workers, and input size
- Timing results (map, reduce, total)
- Memory usage before and after
- Final output (sorted list or global max)

To test correctness, I used small arrays of size **32**, where I compared results with Python's built-in `sorted()` and `max()` functions.
For performance evaluation, I used size **131,072** to measure scalability with more workers.

# 3. Structure of the Code

## System Diagram – Parallel Sorting

```
Main Process
 ├── Worker 1 (Thread/Process): Sort Chunk 1
 ├── Worker 2 (Thread/Process): Sort Chunk 2
 ├── Worker 3 (Thread/Process): Sort Chunk 3
 └── Worker N (Thread/Process): Sort Chunk N

   Reducer (Main): Merge all sorted chunks via heapq.merge
```

- **Threads** share the array directly.
- **Processes** send sorted chunks through a `multiprocessing.Queue`.

## System Diagram – Max-Value Aggregation

```
Main Process
 ├── Worker 1 --> local_max --> try update shared buffer
 ├── Worker 2 --> local_max --> try update shared buffer
 ├── Worker 3 --> local_max --> try update shared buffer
 └── Worker N --> local_max --> try update shared buffer

   Reducer (Main): Reads final global_max from shared buffer
```

All workers attempt to update a single shared integer:

- Threads use a shared list `[value]` + `threading.Lock`
- Processes use `multiprocessing.Value` + `Lock`

## How My Code Supports the MapReduce Framework

My code abstracts the MapReduce model on a single host machine:

- **Map phase:** parallel workers independently process chunks of input data.
- **Intermediate results:** sorted chunks or local maxima.
- **Reduce phase:** the main thread/process merges or aggregates results.

This structure mimics the MapReduce paradigm , key-value computation and aggregation , without needing distributed infrastructure like Hadoop.

# 4. Description of the Implementation

## Tools or Libraries Used

- **Python Standard Library:** `threading`, `multiprocessing`, `heapq`, `time`, `argparse`, `random`, `sys, and psutil` (for memory measurement)
- **Typing support:** for type hints and readability

## Process Management

Workers are manually created:

- For threads: I used `threading.Thread` and `join()` to synchronize their completion.
- For processes: I used `multiprocessing.Process` and a shared `Queue` or `Value` to communicate results.

Each worker independently computes its result and terminates after sending data back to the reducer.

## IPC Mechanism

- For sorting: I used `multiprocessing.Queue` to pass sorted chunks to the reducer.
- For max aggregation: I used `multiprocessing.Value` to represent the single integer shared among all processes.
  These IPC choices were simple and reliable for passing small data amounts between processes.

### Threading Strategy

I used **manual thread creation** instead of a pool to have full control over chunk distribution, sorting, and joining.

### Synchronization Strategy and Implementation

- In multithreading, I used a `threading.Lock` to protect the single shared integer buffer.
- In multiprocessing, I used `multiprocessing.Lock` to ensure only one process updated the shared `Value` at a time.
  This synchronization guaranteed correctness but also introduced performance overhead, which I later analyzed.

## 5. Performance Evaluation

### Correctness Checks (Input Size = 32)

For both the **Parallel Sorting** and **Max-Value Aggregation** tasks, I began testing with a small input size of 32 to verify correctness.
Each run produced the correct sorted output and accurate global maximum value, matching Python's built-in `sorted()` and `max()` results. These small-scale tests confirmed that my implementation worked correctly for multiple worker counts, with negligible memory differences and very fast completion times (all under 0.002 seconds.
Example Output

```
@jto5108 ➜/workspaces/cmpsc472-project1 (main) $ python mapreduce_sort.py --mode thread --workers 1 --size 131072
map_time_s: 0.039111
reduce_time_s: 0.010033
total_time_s: 0.049146
mem_before_mb: 20.68 mem_after_mb: 23.57 delta_mb: 2.89
@jto5108 ➜/workspaces/cmpsc472-project1 (main) $ python max_aggregation.py --mode thread --workers 1 --size 32
MODE=thread workers=1 size=32
map_time_s (incl updates): 0.000236
total_time_s: 0.000236
mem_before_mb: 15.38 mem_after_mb: 15.50 delta_mb: 0.12
global_max: 990382744
@jto5108 ➜/workspaces/cmpsc472-project1 (main) $ python max_aggregation.py --mode thread --workers 1 --size 131072
MODE=thread workers=1 size=131072
map_time_s (incl updates): 0.004333
total_time_s: 0.004333
mem_before_mb: 20.33 mem_after_mb: 20.54 delta_mb: 0.20
global_max: 999985138
@jto5108 ➜/workspaces/cmpsc472-project1 (main) $ python max_aggregation.py --mode thread --workers 2 --size 32
MODE=thread workers=2 size=32
map_time_s (incl updates): 0.000375
total_time_s: 0.000376
mem_before_mb: 15.38 mem_after_mb: 15.50 delta_mb: 0.12
global_max: 968453386
@jto5108 ➜/workspaces/cmpsc472-project1 (main) $ python max_aggregation.py --mode thread --workers 2 --size 131072
MODE=thread workers=2 size=131072
map_time_s (incl updates): 0.004172
total_time_s: 0.004173
mem_before_mb: 20.62 mem_after_mb: 21.26 delta_mb: 0.64
global_max: 999984119
@jto5108 ➜/workspaces/cmpsc472-project1 (main) $ python max_aggregation.py --mode thread --workers 4 --size 32
MODE=thread workers=4 size=32
map_time_s (incl updates): 0.000598
total_time_s: 0.000599
mem_before_mb: 15.38 mem_after_mb: 15.50 delta_mb: 0.12
global_max: 812525487
@jto5108 ➜/workspaces/cmpsc472-project1 (main) $ python max_aggregation.py --mode thread --workers 4 --size 131072
MODE=thread workers=4 size=131072
map_time_s (incl updates): 0.004170
total_time_s: 0.004171
mem_before_mb: 20.50 mem_after_mb: 20.89 delta_mb: 0.39
global_max: 999991328
@jto5108 ➜/workspaces/cmpsc472-project1 (main) $
@jto5108 ➜/workspaces/cmpsc472-project1 (main) $ python max_aggregation.py --mode thread --workers 8 --size 32
MODE=thread workers=8 size=32
map_time_s (incl updates): 0.001561
total_time_s: 0.001562
mem_before_mb: 15.38 mem_after_mb: 15.50 delta_mb: 0.12
global_max: 999458747
```

```
@jto5108 ➜/workspaces/cmpsc472-project1 (main) $ python mapreduce_sort.py --mode thread --workers 2 --size 32
MODE=thread workers=2 size=32
map_time_s: 0.000413
reduce_time_s: 0.000025
total_time_s: 0.000439
mem_before_mb: 15.50 mem_after_mb: 15.75 delta_mb: 0.25
@jto5108 ➜/workspaces/cmpsc472-project1 (main) $ python mapreduce_sort.py --mode thread --workers 2 --size 131072
MODE=thread workers=2 size=131072
map_time_s: 0.019386
reduce_time_s: 0.017016
total_time_s: 0.036403
mem_before_mb: 20.50 mem_after_mb: 23.27 delta_mb: 2.77
@jto5108 ➜/workspaces/cmpsc472-project1 (main) $ python mapreduce_sort.py --mode thread --workers 4 --size 32
MODE=thread workers=4 size=32
map_time_s: 0.000598
reduce_time_s: 0.000022
total_time_s: 0.000621
mem_before_mb: 15.38 mem_after_mb: 15.38 delta_mb: 0.00
@jto5108 ➜/workspaces/cmpsc472-project1 (main) $ python mapreduce_sort.py --mode thread --workers 4 --size 131072
MODE=thread workers=4 size=131072
map_time_s: 0.017827
reduce_time_s: 0.021309
total_time_s: 0.039138
mem_before_mb: 20.62 mem_after_mb: 22.96 delta_mb: 2.33
@jto5108 ➜/workspaces/cmpsc472-project1 (main) $ python mapreduce_sort.py --mode thread --workers 8 --size 32
MODE=thread workers=8 size=32
map_time_s: 0.001072
reduce_time_s: 0.000026
total_time_s: 0.001099
mem_before_mb: 15.50 mem_after_mb: 15.75 delta_mb: 0.25
@jto5108 ➜/workspaces/cmpsc472-project1 (main) $ python mapreduce_sort.py --mode thread --workers 8 --size 131072
MODE=thread workers=8 size=131072
map_time_s: 0.018060
reduce_time_s: 0.026952
total_time_s: 0.045013
mem_before_mb: 20.50 mem_after_mb: 22.73 delta_mb: 2.23
@jto5108 ➜/workspaces/cmpsc472-project1 (main) $ python mapreduce_sort.py --mode thread --workers 1 --size 32
MODE=thread workers=1 size=32
map_time_s: 0.000224
reduce_time_s: 0.000011
total_time_s: 0.000237
mem_before_mb: 15.38 mem_after_mb: 15.50 delta_mb: 0.12
```

## Performance Assessment (Input Size = 131,072)

I next ran performance tests on a much larger dataset (131,072 integers).
Below are the summarized results for the **thread-based** version of both tasks.

Parallel Sorting Results

| Workers | Map (s) | Reduce (s) | Total (s) | Δ Mem (MB) |
|---|---|---|---|---|
| 1 | 0.039111 | 0.010033 | **0.049146** | +2.89 |
| 2 | 0.019386 | 0.017016 | **0.036403** | +2.77 |
| 4 | 0.017827 | 0.021309 | **0.039138** | +2.33 |
| 8 | 0.018060 | 0.026952 | **0.045013** | +2.23 |

Max-Value Aggregation Results

| Workers | Map + Update (s) | Total (s) | Δ Mem (MB) | Global Max |
|---|---|---|---|---|

| | | | | |
|---|---|---|---|---|
| 1 | 0.004333 | **0.004333** | +0.20 | 999985138 |
| 2 | 0.004172 | **0.004173** | +0.64 | 999984119 |
| 4 | 0.004170 | **0.004171** | +0.39 | 999991328 |
| 8 | 0.004563 | **0.004564** | +0.25 | 999981169 |

## Timing Comparisons and Discussion

### *Parallel Sorting*

For sorting, I noticed that going from 1 to 2 workers significantly reduced total time from 0.049 s to 0.036 s, showing a real benefit from parallelization.
However, increasing the number of threads beyond 2 did not continue to improve speed.
At 4 and 8 workers, the total time slightly increased again (0.039 s → 0.045 s), likely due to synchronization overhead and the Global Interpreter Lock (GIL) limiting true parallel execution in CPU-bound operations.

Memory usage grew modestly (around 2–3 MB increase), which is expected as each worker holds its own sorted subarray before merging.

### *Max-Value Aggregation*

The aggregation task showed very stable timing across all worker counts.
Even at 8 threads, execution time stayed around 0.0045 seconds.
Because each thread had to acquire a lock to update the shared global maximum, more threads introduced slight contention but no major slowdown.
This consistency shows that synchronization worked correctly and efficiently for the simple shared-integer update model.

### *Synchronization Discussion*

Synchronization played a critical role in the Max-Value task.
Without a lock, multiple threads would race to update the global maximum, producing inconsistent or incorrect results.
Using a `threading.Lock` ensures that only one thread updates the value at a time, maintaining accuracy.
However, it also revealed how synchronization limits scalability—beyond 4 threads, the performance gain flattened due to the sequential nature of the critical section.

For sorting, synchronization was minimal since each thread worked on independent chunks, but the GIL still prevented true multi-core parallelism, explaining why performance didn't continue improving with more threads.

# 6. Conclusion

## Key Findings

From my results, I found that **multithreading improved performance** compared to a single thread when using a small number of workers, especially for the sorting task.
However, increasing threads beyond 2 to 4 did not continue to improve performance because of Python's GIL and the overhead of managing multiple threads.
For aggregation, the results were highly consistent across all worker counts, proving that my synchronization design-maintained correctness while keeping the system lightweight.

I also observed that **memory growth remained small** (usually 2–3 MB for large inputs), showing that my approach was efficient in both time and space.

## Challenges Faced

Some challenges I faced included:

- Managing synchronization correctly to avoid race conditions when updating the shared global maximum.
- Balancing the number of threads to find an optimal point between speedup and overhead.
- Measuring execution time precisely for small workloads where timing differences were minimal.
- Dealing with Python's inherent threading limitations on CPU-bound tasks.

## Limitations and Possible Improvements

One limitation is that all experiments were done using **threads** only. Running the same tasks with **multiprocessing** would show true parallel execution across cores.
Future improvements could include:

- Implementing a **multiprocessing version** for comparison.

- Using **shared memory arrays** or `multiprocessing.Queue` to reduce data transfer overhead.
- Adding **CPU utilization tracking** to quantify parallel efficiency.
- Integrating **thread pools** (`concurrent.futures.ThreadPoolExecutor`) for more scalable and cleaner worker management.

Overall, this project deepened my understanding of how MapReduce-style computation, parallelism, and synchronization interact in practice.
It showed me that parallel speedup depends not just on splitting work but also on how efficiently threads or processes share and synchronize data.