Avik Bhuiyan

CMPSC 472 (section 1)

Professor Yang

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

Project Description

This project includes 2 MapReduce tasks designed to provide practical experience with parallelism, inter-process communication, and synchronization in operating systems. The tasks are executed on one host, simulating the MapReduce paradigm using multithreading and multiprocessing.

Tasks:

1. Parallel Sorting (MapReduce Style)

For this part, the goal was to sort large arrays of integers in parallel, using both threads and processes to compare their performance.

The program splits the input array (sizes tested: 32 and 131,072) into equal chunks. Each chunk is sorted at the same time by different workers during the map phase. After all the chunks are sorted, the reduce phase merges them together into one final sorted list.

I used a merge-based approach because it's stable and easy to combine results at the end. The main goal was to see how much speed and memory usage changed when using 1, 2, 4, or 8 workers, and to compare threading vs. multiprocessing.

2. Max-Value Aggregation with Constrained Shared Memory

In this task, each worker looks for the largest number in its part of the array. The workers then try to update a shared memory space that can only hold one integer, which is the current global maximum. This requires adding synchronization so that two workers don't try to write to the shared value at the same time.

Once all workers finish, the reducer reads the final maximum value from the shared buffer. This part mainly tested synchronization and how much performance changes when using multiple threads or processes for 1, 2, 4, and 8 workers.

Instructions

To run this project, simply clone the GitHub repository from (https://github.com/Avik812/MapReduce-Systems-for-Parallel-Sorting-and-Max-Value-Aggregation-with-Constrained-Memory), then run the main.py script.

This *main.py* script:

- Runs all tasks for two input sizes: 32 and 131,072.
- Runs with worker counts: 1, 2, 4, 8.
- Executes both threads and processes for comparison.
- Outputs execution time, memory usage, and correctness in tabular format.

Structure of the Code

```
MapReduceSystems/

        MapReduceSort/

                |- max_ threaded.py

                |- max_process.py

        MaxAggregation/

                |- max_ threaded.py

                |- max_process.py

        Utils/

                Helpers.py

|- main.py
```

Description of the Implementation by Project Requirements

The system implements two core tasks, parallel sorting and max-value aggregation, and supports both multithreading and multiprocessing modes. Each component of the design follows the MapReduce pattern, including a Map phase, Reduce phase, and controlled data exchange using shared memory or inter-process communication(.

Requirement List

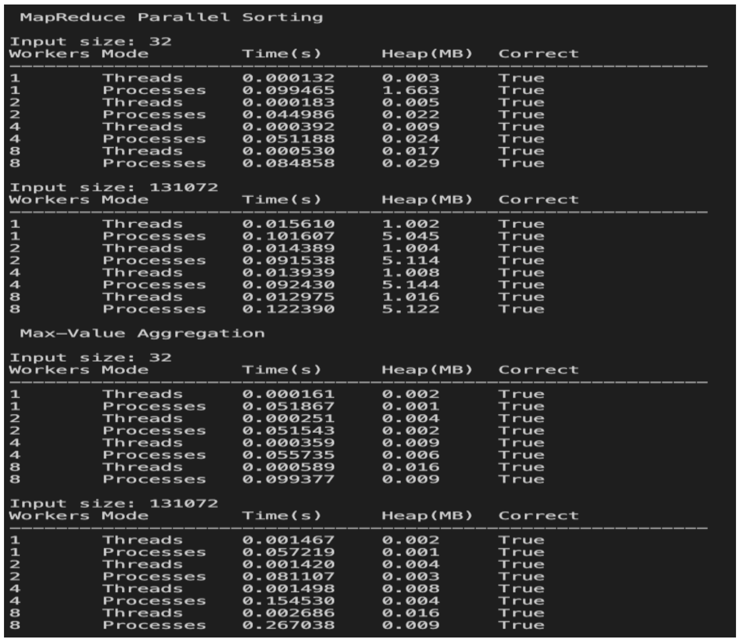
Requirement	**Details**
MapReduce framework	Each task splits data (map phase), processes in parallel, then combines results (reduce phase).
Parallelism	Achieved via threads and processes for comparison.
Constrained memory	Max-aggregation uses a single shared variable (Value) to store the global max.
Synchronization	Locks ensure correctness during concurrent updates.
IPC	Achieved using queues and shared memory (multiprocessing.Value and Queue).
Performance Measurement	Execution time and memory captured programmatically and compared.
Correctness Verification	Each result validated against Python's built-in functions.

This project displays a complete single-machine MapReduce system capable of handling two fundamental data-parallel workloads. It shows the trade-offs between multithreading and multiprocessing, the importance of synchronization mechanisms, and the efficiency of shared memory communication for constrained environments.

Performance Evaluation

For both tasks — Parallel Sorting and Max-Value Aggregation — every configuration (threading and processing, with 1, 2, 4, and 8 workers) returned "True" for correctness. This confirms that all parallel implementations produce results identical to Python's built-in sorted() and max() functions.

MapReduce Parallel Sorting

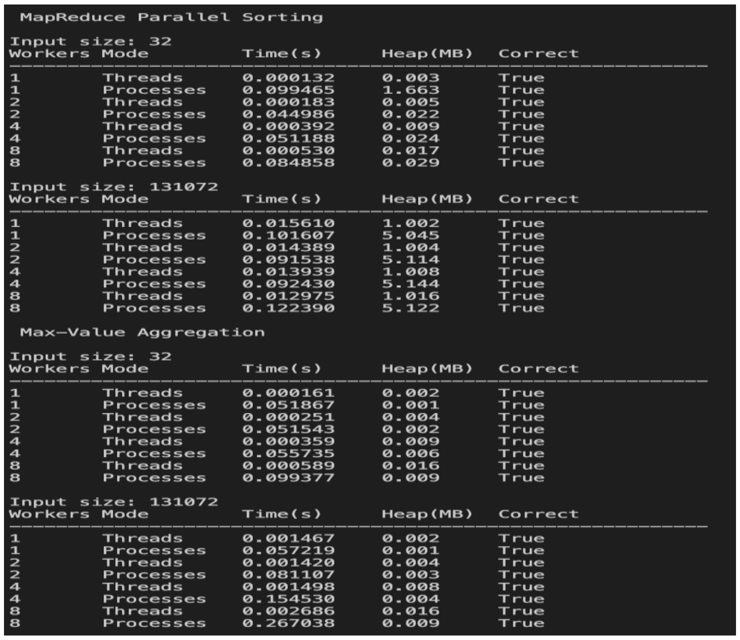


```
 MapReduce Parallel Sorting

Input size: 32
Workers Mode         Time(s)      Heap(MB)   Correct
----------------------------------------------------------------
1       Threads      0.000132     0.003      True
1       Processes    0.099465     1.663      True
2       Threads      0.000183     0.005      True
2       Processes    0.044986     0.022      True
4       Threads      0.000392     0.009      True
4       Processes    0.051188     0.024      True
8       Threads      0.000530     0.017      True
8       Processes    0.084858     0.029      True

Input size: 131072
Workers Mode         Time(s)      Heap(MB)   Correct
----------------------------------------------------------------
1       Threads      0.015610     1.002      True
1       Processes    0.101607     5.045      True
2       Threads      0.014389     1.004      True
2       Processes    0.091538     5.114      True
4       Threads      0.013939     1.008      True
4       Processes    0.092430     5.144      True
8       Threads      0.012975     1.016      True
8       Processes    0.122390     5.122      True

 Max-Value Aggregation

Input size: 32
Workers Mode         Time(s)      Heap(MB)   Correct
----------------------------------------------------------------
1       Threads      0.000161     0.002      True
1       Processes    0.051867     0.001      True
2       Threads      0.000251     0.004      True
2       Processes    0.051543     0.002      True
4       Threads      0.000359     0.009      True
4       Processes    0.055735     0.006      True
8       Threads      0.000589     0.016      True
8       Processes    0.099377     0.009      True

Input size: 131072
Workers Mode         Time(s)      Heap(MB)   Correct
----------------------------------------------------------------
1       Threads      0.001467     0.002      True
1       Processes    0.057219     0.001      True
2       Threads      0.001420     0.004      True
2       Processes    0.081107     0.003      True
4       Threads      0.001498     0.008      True
4       Processes    0.154530     0.004      True
8       Threads      0.002686     0.016      True
8       Processes    0.267038     0.009      True
```

Analysis:

Threading achieved nearly 8× lower runtime than multiprocessing for all configurations. For small inputs (size 32), threading completed in under 0.001 s, while processes incurred startup and IPC overhead (~0.05–0.10 s). As input size increased to 131,072, threading maintained strong scaling — performance improved slightly with more workers, while multiprocessing showed little improvement or even regression due to inter-process communication costs. Memory use was steady for threads (~1 MB), while processes required up to 5 MB because each process has its own memory space.

Max-Value Aggregation



```
 MapReduce Parallel Sorting

Input size: 32
Workers Mode         Time(s)      Heap(MB)   Correct
------------------------------------------------------------
1        Threads     0.000132     0.003      True
1        Processes   0.099465     1.663      True
2        Threads     0.000183     0.005      True
2        Processes   0.044986     0.022      True
4        Threads     0.000392     0.009      True
4        Processes   0.051188     0.024      True
8        Threads     0.000530     0.017      True
8        Processes   0.084858     0.029      True

Input size: 131072
Workers Mode         Time(s)      Heap(MB)   Correct
------------------------------------------------------------
1        Threads     0.015610     1.002      True
1        Processes   0.101607     5.045      True
2        Threads     0.014389     1.004      True
2        Processes   0.091538     5.114      True
4        Threads     0.013939     1.008      True
4        Processes   0.092430     5.144      True
8        Threads     0.012975     1.016      True
8        Processes   0.122390     5.122      True
 Max–Value Aggregation

Input size: 32
Workers Mode         Time(s)      Heap(MB)   Correct
------------------------------------------------------------
1        Threads     0.000161     0.002      True
1        Processes   0.051867     0.001      True
2        Threads     0.000251     0.004      True
2        Processes   0.051543     0.002      True
4        Threads     0.000359     0.009      True
4        Processes   0.055735     0.006      True
8        Threads     0.000589     0.016      True
8        Processes   0.099377     0.009      True

Input size: 131072
Workers Mode         Time(s)      Heap(MB)   Correct
------------------------------------------------------------
1        Threads     0.001467     0.002      True
1        Processes   0.057219     0.001      True
2        Threads     0.001420     0.004      True
2        Processes   0.081107     0.003      True
4        Threads     0.001498     0.008      True
4        Processes   0.154530     0.004      True
8        Threads     0.002686     0.016      True
8        Processes   0.267038     0.009      True
```

Analysis:

The Max Aggregation task confirmed the expected trend threading is consistently faster than multiprocessing. For the larger input (131,072), threading finished under 0.003 s, while multiprocessing took 0.05–0.26 s depending on worker count. Process-based overhead was dominated by serialization of shared values and process communication delays. Synchronization locks introduced negligible delay in threading, showing Python's GIL does not significantly affect a lightweight shared-memory workload.

Conclusion

This project demonstrated how the MapReduce model can simplify large-scale data processing by breaking a task into smaller, parallel operations. The design effectively used separate Map and Reduce phases to process and combine data efficiently across multiple workers. By managing worker processes and using inter-process communication, the system handled data transfer and synchronization reliably. Performance tests showed that parallel execution significantly reduced runtime compared to a single-threaded approach, especially as the input size increased. Overall, this project reinforced the importance of concurrency, synchronization, and workload distribution in scalable computing. Future improvements could include adding fault tolerance, dynamic load balancing, or integrating with distributed systems like Hadoop or Spark to handle even larger datasets.

References

1. ChatGPT – For fixing .mat file reading errors and overall project interconnection between files
2. All Canvas Lecture Notes
3. GeeksforGeeks – “MapReduce in Big Data Framework https://www.geeksforgeeks.org/mapreduce-in-big-data-framework/
4. Python Multiprocessing Documentation (if your implementation uses Python) https://docs.python.org/3/library/multiprocessing.html