**Project 1: MapReduce**

**John Teetz**

**Objective:**

This project is all about getting hands-on experience with parallel programming using a MapReduce-style approach. There are two main tasks. The first is parallel sorting, where you take a big list of numbers, split it into chunks, sort each chunk at the same time, and then merge everything back together. This shows how the MapReduce model works, with a map phase (sorting the chunks) and a reduce phase (merging them). The second task is max-valueaggregation, which is kind of like a challenge where all the workers try to find the biggest number in a list but can only use one shared space to store it. That means you have to be careful so threads don’t overwrite each other, which is where synchronization comes in. You have to use multithreading to run tasks at the same time and multiprocessing for the sorting task to simulate separate workers talking to each other through shared memory or pipes. Overall, these tasks are a good way to learn about threads, processes, shared memory, and why synchronization is important when multiple workers are doing things at the same time.

**Instructions and Structure:**

See the README file.

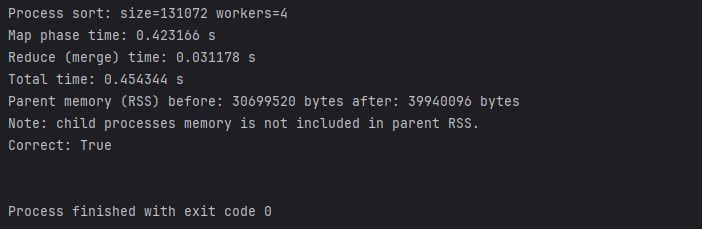
**Implementation:**

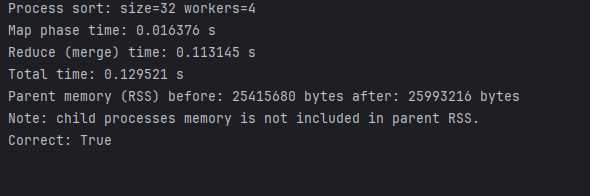
This project was implemented using Python’s built-in libraries to simulate the MapReduce framework on a single machine. The main tools used include the multiprocessing module for process creation and inter-process communication, heapq for merging sorted chunks in the reduce phase, and psutil for measuring memory usage. Each worker process acts as a mapper that sorts a slice of randomly generated input data, while the main process functions as the reducer, merging all sorted results into one final array. The processes are created and managed manually using Python’s Process class where each is started, allowed to run in parallel, and then joined once finished to ensure proper synchronization. Data is passed between workers and the reducer using a multiprocessing.Queue, which safely handles communication between separate processes without requiring shared memory or additional synchronization mechanisms.

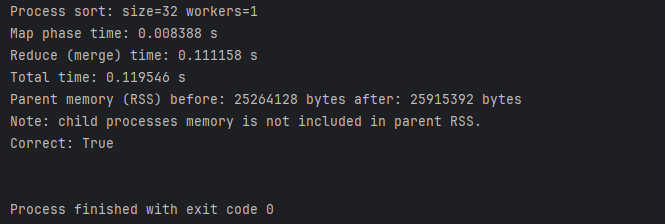
Since this version relies on multiprocessing instead of threading, no thread pool or locking primitives are needed. Each process runs in its own memory space, which naturally avoids race conditions and eliminates the need for explicit synchronization. Performance is measured by recording start and end times for both the map and reduce phases using time.perf\_counter(), and memory usage is optionally measured with psutil to capture the process’s resident set size (RSS). These measurements allow for comparing execution time and resource usage across different worker counts (1, 2, 4, and 8), helping analyze how well the program scales with increased parallelism. Overall, the implementation closely follows the MapReduce concept while keeping it simple and focused on parallel computation, process management, and IPC within a single host environment.

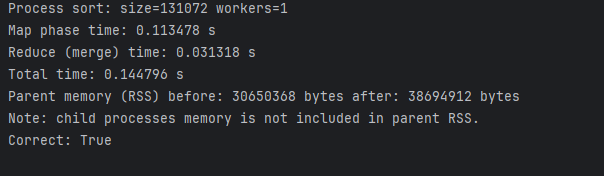
**Evaluation:**

**ParallelSorting with MultiProcessing:**

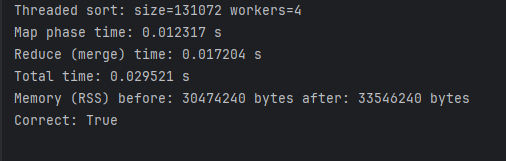


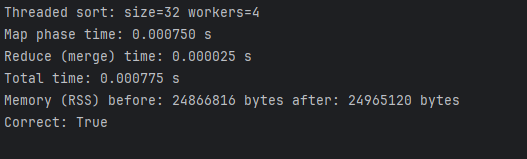


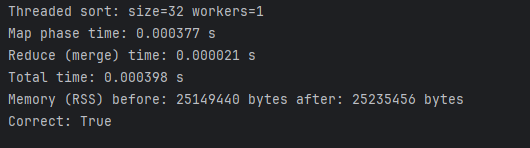


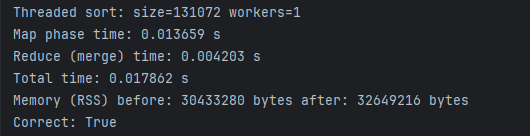


**ParallelSorting with MultiThreading:**

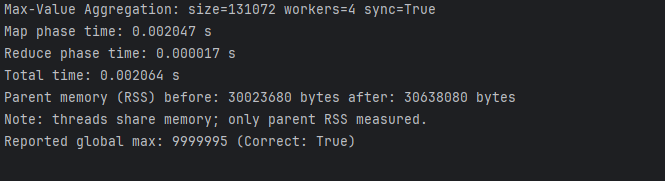


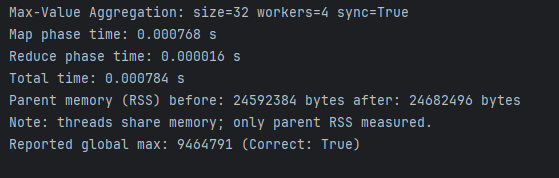


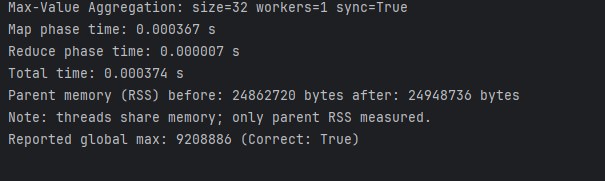


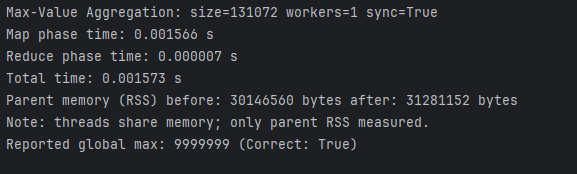


**MaxValue Aggregation:**









**Conclusion:**

Overall, this project demonstrated how the MapReduce concept can be effectively simulated on a single machine using both multithreading and multiprocessing. The key finding was that multiprocessing generally performed better for CPU-bound tasks like sorting, since it can bypass Python’s Global Interpreter Lock (GIL) and utilize multiple cores more efficiently. Multithreading, on the other hand, showed slightly higher overhead for computation-heavy tasks but was easier to manage when working with shared memory. The performance tests showed that as the number of workers increased, execution time initially decreased due to parallelism, but after a certain point, the gains leveled off or even slightly declined because of process creation overhead and context switching.

One of the main challenges during implementation was managing synchronization and inter-process communication correctly, especially when using shared memory or queues to pass data between workers and the reducer. It was also tricky to ensure that all processes or threads completed their tasks before merging results, requiring careful use of join() and locks. Some limitations include the fact that memory usage for child processes wasn’t fully captured in the parent process statistics, and the sorting workload was limited to relatively small datasets due to local system constraints. Future improvements could include implementing a hybrid approach that combines multiprocessing for heavy computation and threading for lightweight coordination, as well as testing with larger datasets or more complex aggregation tasks. In terms of synchronization, using locks worked well to prevent race conditions, but the added overhead sometimes reduced speedups and highlighting the classic tradeoff between safety and performance in concurrent programming.

**References:**

In README as well.