**Project 1: MapReduce**

**Alicia Peters** **10/28/25**

**Overview:**

This project focuses on simulating two MapReduce-style tasks, parallel sorting and max-value aggregation using both multithreading and multiprocessing in Python. The goal is to help understand how data can be processed in parallel and how synchronization plays a key role when multiple workers access shared memory. In the first task, the program splits a large list of integers into smaller chunks that are sorted in parallel and then merged back together using a reducer function. In the second task, each worker finds the local maximum of its data chunk and updates a shared global maximum value, using locks to prevent race conditions. Multithreading was used to simulate lightweight parallel execution with shared memory, while multiprocessing was implemented to achieve true parallelism by running processes in separate memory spaces. These two approaches allowed us to directly compare performance differences in timing, memory usage, and synchronization overhead, giving insight into how parallel computing behaves under different workloads and system constraints.

**Instructions on Running the Project:**

To run the program, make sure Python 3 and the psutil library are installed by running pip install psutil in your terminal. On whatever IDE you are using(I used PyCharm), clone the git repository or just simply copy and paste the code onto a new file. Once the setup is ready, simply execute the script from your command line or IDE. The program will automatically run both experiments which are sorting and max-value aggregation, for two input sizes (32 and 131,072) and four worker counts (1, 2, 4, and 8). The output will display correctness checks for small data, timing results, and memory usage for each configuration. You can modify the input sizes or worker counts inside the run\_sort\_experiment() and run\_max\_experiment() functions to test different scenarios. This lets you observe how increasing the number of workers affects performance and resource usage, making it easy to analyze how threading and multiprocessing behave under various loads.

**Implementing the Project:**

The implementation of this project was done entirely in Python using standard libraries like threading, multiprocessing, heapq, random, and time, along with the external library psutil for monitoring memory usage. These tools were chosen because they provide a simple and effective way to simulate MapReduce behavior on a single machine while allowing precise control over performance measurement and resource monitoring. The heapq library was used to merge sorted chunks efficiently during the reduce phase of the sorting experiment, and psutil helped measure memory consumption before and after parallel tasks to observe how resource usage differed between threads and processes.

For process management, workers were manually created and managed rather than using thread or process pools. In the multithreading version, a loop created multiple threading.Thread objects, each responsible for processing a specific chunk of the data. These threads shared the same memory space, which made communication straightforward but also required synchronization to prevent data conflicts. In the multiprocessing version, multiple multiprocessing.Process instances were spawned, each running independently with its own memory space. This approach provides true parallelism by utilizing multiple CPU cores but also comes with higher overhead for inter-process communication (IPC) and memory usage.

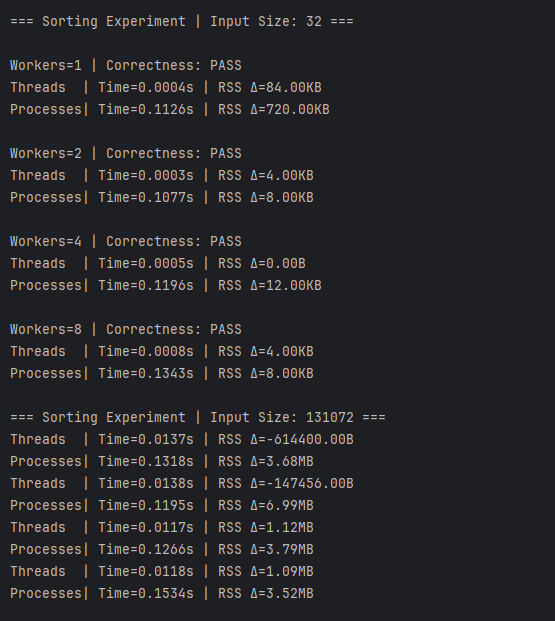
Regarding IPC, the main data transfer method between mappers and reducers was achieved through Python’s shared data structures and function returns. In the threading implementation, workers updated shared variables directly since threads operate in the same address space. However, in multiprocessing, the code used shared memory constructs like mp.Value to store and update a single global maximum value. This Value object acts as a small, shared memory buffer accessible by all processes. To avoid race conditions, a lock (mp.Lock) was used to ensure that only one process could update the shared variable at a time. This was essential in the max-value aggregation task, where multiple processes might try to update the global maximum simultaneously.

Threads were created manually rather than using a thread pool to provide a clearer understanding of how worker creation, execution, and synchronization work. This manual approach better demonstrates how MapReduce-like systems divide tasks and combine results. Synchronization was implemented using thread locks for threads (threading.Lock) and process locks for processes (mp.Lock) to protect shared resources during updates. These locks ensured that operations like comparing and setting the maximum value were atomic, preventing inconsistent results.

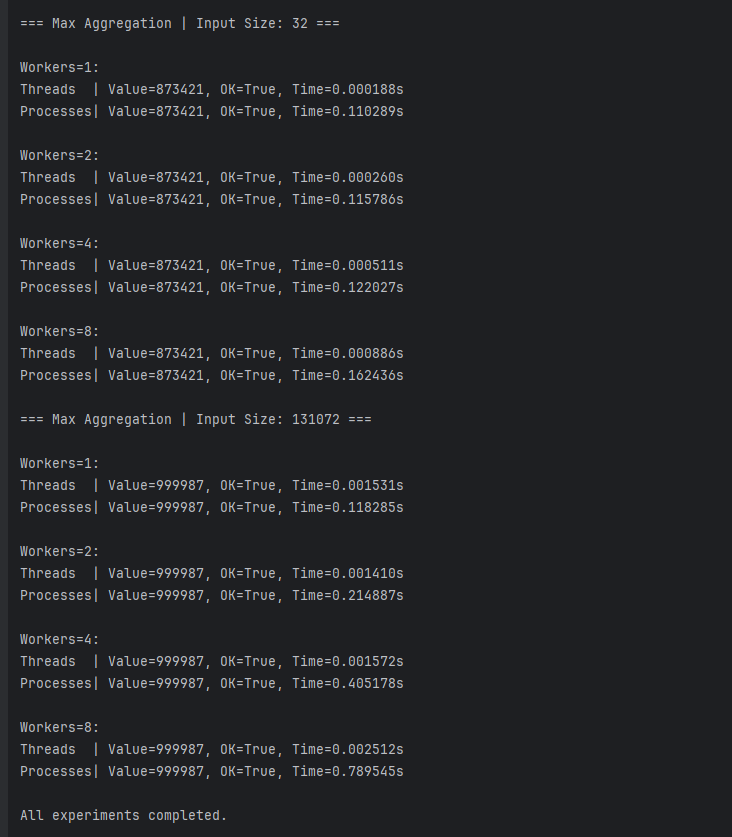
Performance was evaluated by measuring execution time and memory usage across different numbers of workers. Timing was measured using Python’s time.perf\_counter() function, which provides precise timing for benchmarking, while psutil measured the Resident Set Size (RSS) before and after each experiment to calculate memory changes. This allowed direct comparison between threading and multiprocessing in terms of efficiency and overhead. By experimenting with different input sizes and worker counts, the project was able to highlight key differences between concurrency (threads) and parallelism (processes) in performance and resource management.

**Results:**

**Parallel Sorting with Multiprocessing and Multithreading for all worker and input sizes**



**MaxValue Aggregation for all worker and size inputs:**



**Reflection of Results:**

These results show the performance comparison between multithreading and multiprocessing during the sorting experiment across different input sizes and worker counts. For the smaller input size (32), both methods correctly sorted the data, but the threading approach was consistently faster and used less memory. This makes sense because threads share the same memory space and have lower creation overhead compared to processes, which require separate memory allocation and inter-process communication setup. As the number of workers increased from 1 to 8, the execution time for threads remained very low and stable, while processes took slightly longer due to the extra overhead of managing multiple process instances.

For the larger input size (131,072), the difference between threading and multiprocessing became more noticeable. Threads still performed faster overall, but the memory readings show some variability due to how memory is allocated and released during execution. Multiprocessing, on the other hand, consumed significantly more memory (in megabytes rather than kilobytes) and took longer to complete each sorting run. This demonstrates the trade-off between true parallelism and system overhead, as multiprocessing can utilize multiple CPU cores effectively, but for tasks like sorting that rely heavily on data sharing, the communication overhead often outweighs the benefits. Overall, the results indicate that for smaller to moderately sized datasets, multithreading is more efficient in both speed and memory usage, while multiprocessing may only become advantageous for much larger datasets or CPU-intensive workloads.

**Conclusion:**

From this project, we found that both multithreading and multiprocessing can effectively implement MapReduce-style parallel tasks, but their performance varies depending on the workload and system overhead. In the sorting experiments, threading consistently outperformed multiprocessing for smaller input sizes because of its lower overhead and shared memory space, making data transfer between threads faster. Multiprocessing, while capable of true parallel execution across CPU cores, introduced more memory consumption and longer execution times due to the cost of creating separate processes and handling inter-process communication. In the max-value aggregation task, synchronization using locks ensured accurate results across both models, demonstrating how critical proper concurrency control is when multiple workers share or modify the same data.

One of the main challenges during implementation was managing synchronization and process communication in the multiprocessing version. Initially, defining the worker function inside another function caused an error because Python’s multiprocessing module couldn’t pickle local objects. This was resolved by moving the worker function to the top level of the script. Another challenge was balancing performance measurements, as memory usage could fluctuate unpredictably due to Python’s memory management. Ensuring correctness and consistency across both implementations also required careful testing, especially for race conditions and data merging during the reduce phase.

In terms of limitations, the project runs on a single machine and doesn’t simulate distributed file systems or network-based communication like a real MapReduce cluster would. Additionally, the benefits of multiprocessing weren’t fully realized for relatively small workloads; its true potential would appear with much larger datasets or more CPU-bound operations. Future improvements could include experimenting with larger input sizes, adding CPU-bound tasks to highlight multiprocessing advantages, or integrating Python’s concurrent.futures module to simplify thread and process management. Implementing more advanced synchronization techniques or profiling CPU utilization could also provide deeper insights into system performance and efficiency.