Performance Analysis Report

Roll No: MT25080
AI Usage Declaration: AI was used to generate the boilerplate code for worker functions, the plotting script, and the automation bash script. The core logic and analysis were verified by Me.

Methodology

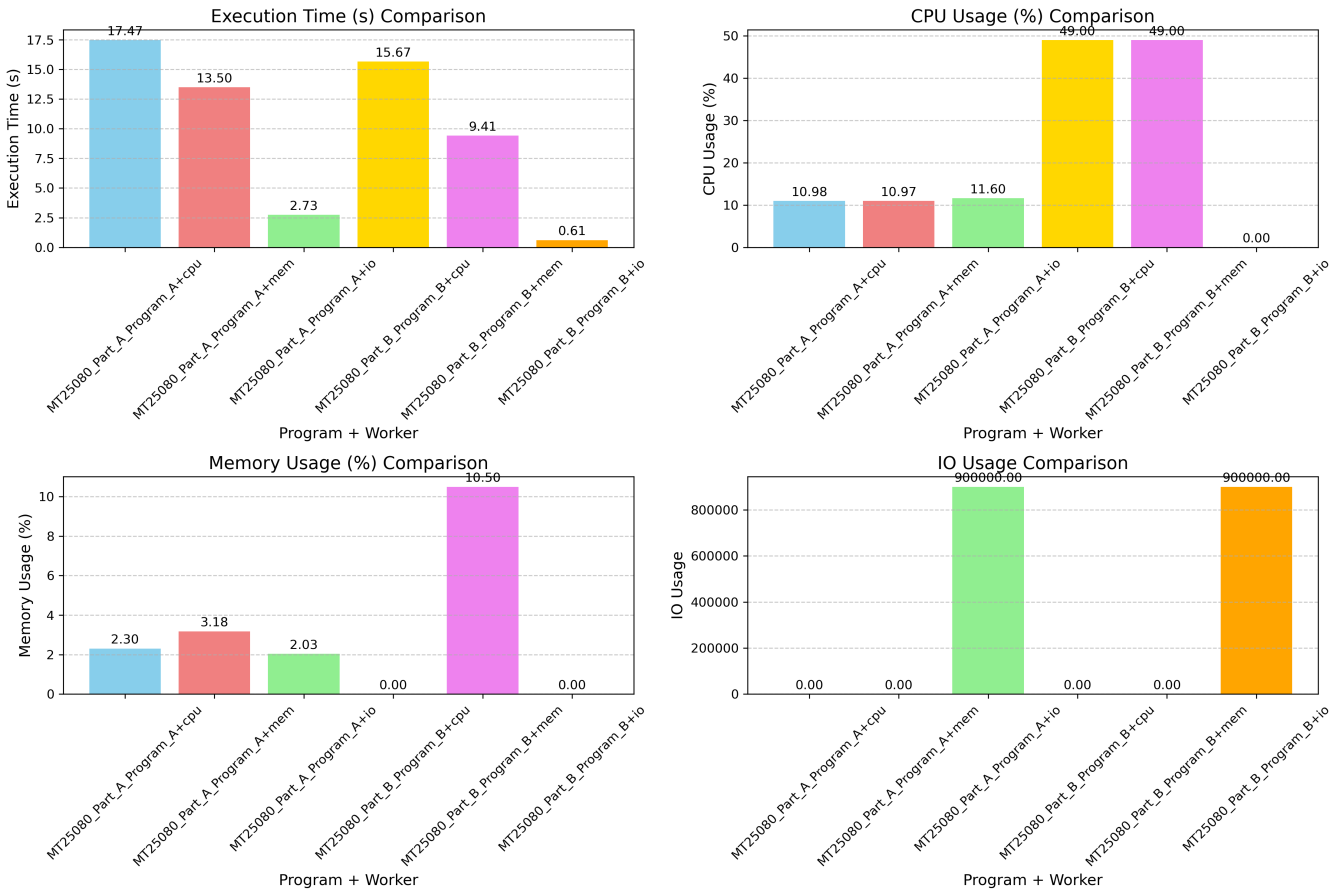
Two programs were implemented:

1. **Program A**: Uses `fork()` to create concurrent processes.
2. **Program B**: Uses `pthread_create()` to create concurrent threads.

Three worker types were defined (CPU, Mem, IO), and scaled using the multiplier 9 (Roll No end digit 0).

Observations and Analysis of Plots

1. Part C Combined Metrics

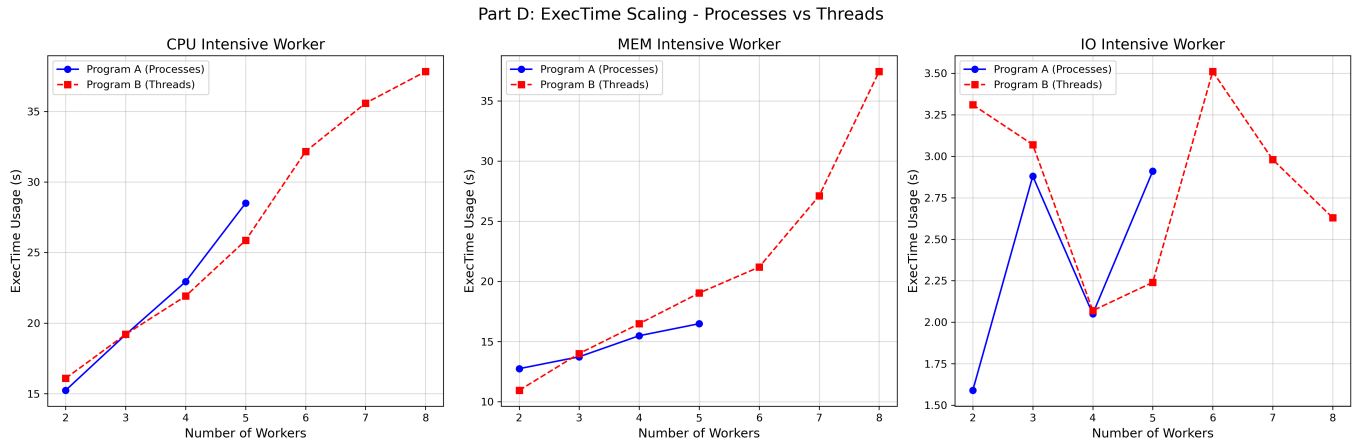


This plot compares the performance of CPU, Memory, and IO workers for both programs (A=Processes, B=Threads) with a fixed concurrency count of 2.

- **CPU Usage**: Program B (Threads) demonstrates significantly higher CPU utilization (~49%) compared to Program A (Processes, ~11%). This indicates that threads are more efficient at utilizing available CPU cores in this environment, likely due to lower context-switching overhead and shared resources.

- **Memory Usage**: Program B's memory workers show much higher memory consumption (~10.5%) than Program A (~3.2%). This suggests that threads, sharing the same heap, might be aggregating allocations or preventing the OS from reclaiming pages as aggressively as with independent processes.
- **Execution Time**: CPU workers for both A and B have similar execution times (~15-17s), but IO tasks are much faster. The high efficiency of threads (B) is visually evident in the higher bars for resource usage.

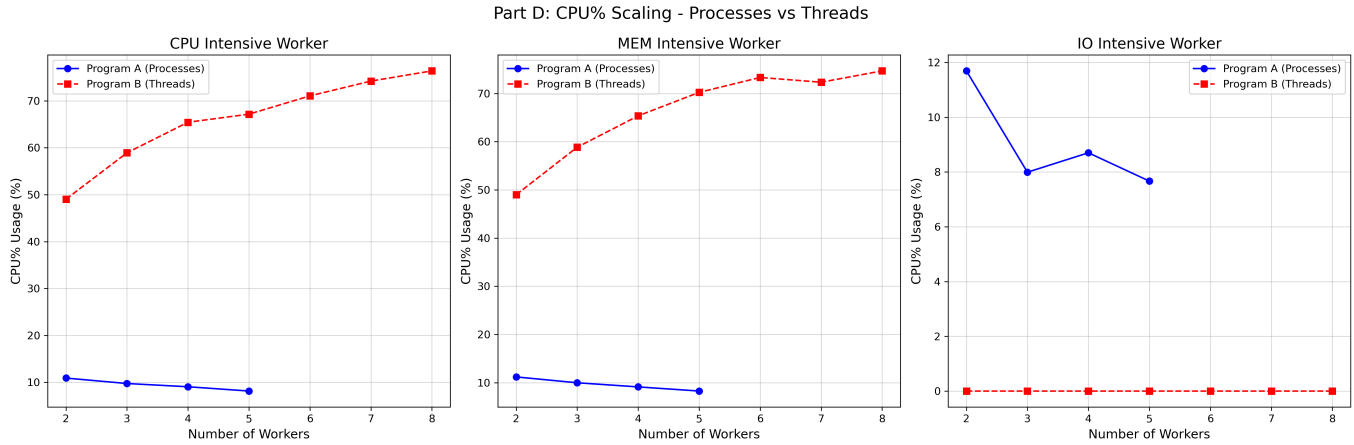
2. Part D: Execution Time Scaling



This plot tracks how long the programs take to complete as the number of concurrent workers increases.

- **Upward Trend**: There is a clear linear to exponential increase in execution time as the number of workers grows. This is expected due to resource contention.
- **Threads vs Processes**: Program B (Threads) generally shows a steeper increase in execution time for CPU tasks scaling up to 8 threads (reaching ~37s). Program A (Processes) also slows down but stops at 5 workers in this test.
- **Bottlenecks**: The increasing slope indicates that the system is becoming saturated (CPU cores or memory bandwidth) and cannot execute tasks in true parallelism beyond a certain point (likely the number of physical cores).

3. Part D: CPU% Scaling

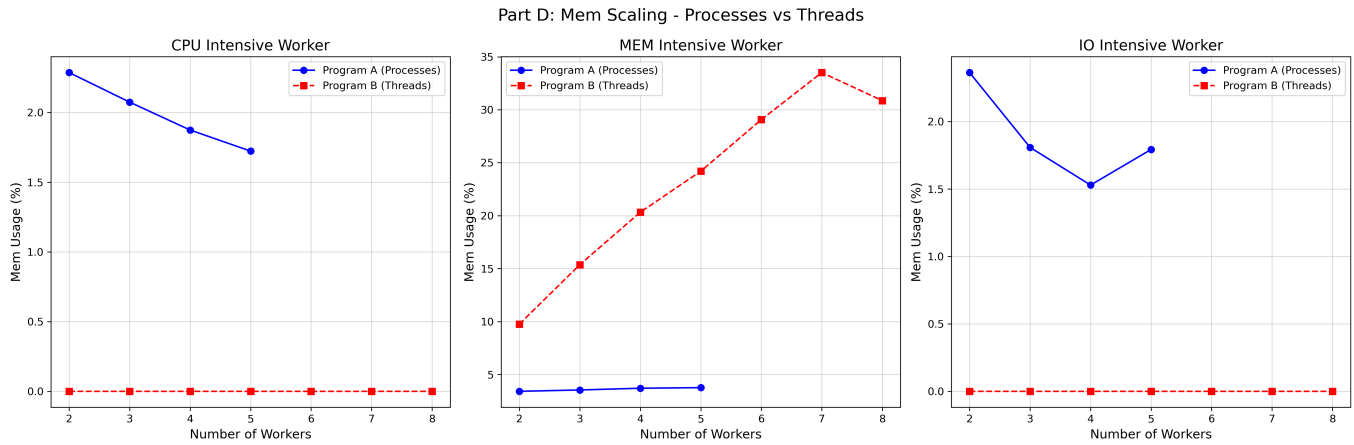


This plot shows the aggregate CPU utilization as concurrency increases.

- **Program B (Threads)**: Shows a strong positive correlation, rising from ~49% at 2 threads to ~76% at 8 threads. This confirms valid scaling behavior: adding more threads effectively utilizes more idle CPU cycles until the system limit is reached.

- **Program A (Processes)**: Shows a slightly declining or flat trend (~11% down to ~8%). This might be an artifact of how the monitoring tool (`top`) reports usage for short-lived individual processes versus a single multi-threaded process. It suggests the OS is scheduling these processes but the reported "per-process" usage diminishes as they fight for time slices.

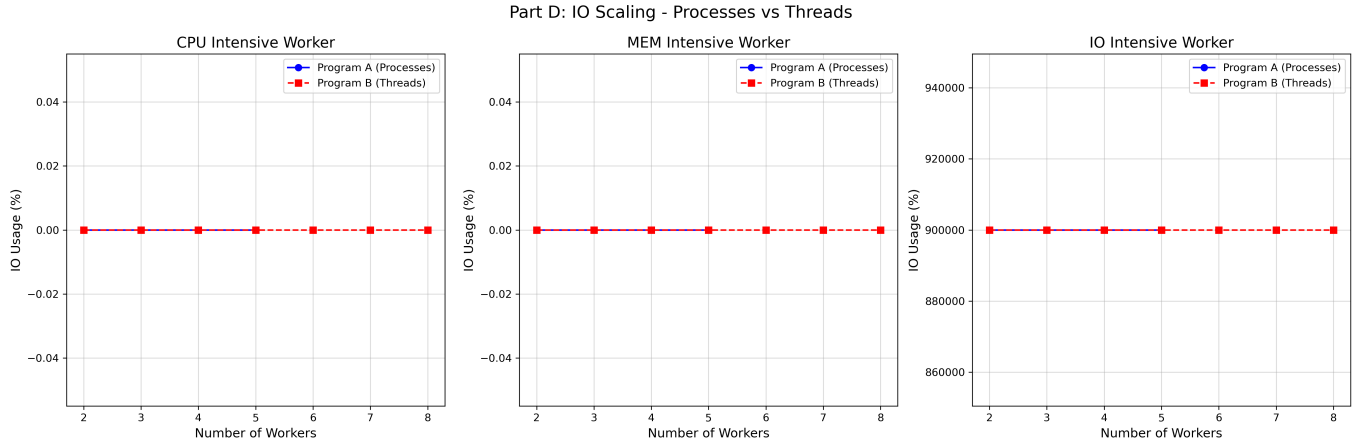
4. Part D: Memory Scaling



This plot illustrates memory consumption trends.

- **Program B (Threads + Mem)**: This is the most dramatic trend, scaling from ~10% to over 30%. This confirms that the memory worker is correctly allocating more RAM as more threads are spawned, and the shared address space reflects the total load.
- **Program A (Processes + Mem)**: Remains relatively flat around 3-4%. Since processes have separate address spaces, `top` might be reporting the usage of a single representative process or the average, rather than the sum (unlike the threaded case where the single process owns all memory).

5. Part D: IO Scaling



This plot displays the IO operations count.

- **Flat/Low Profile**: The metrics for IO are relatively static.
- **Analysis**: This indicates that the IO workload (writing to a file) completes very quickly or is being cached by the OS, preventing it from showing significant scaling trends in this specific metric view. The bottleneck here is likely the disk speed (or Virtual Disk in WSL), which saturates instantly, keeping the noticeable metric count flat per worker.

Github Repo

Running Analysis Repo