Subject: Graduate Systems
Assignment 1
Name: Utsav S Davda
Roll number: MT25088

1. Introduction & Implementation Strategy

Objective:

The goal of this assignment was to analyze the performance differences between Multi-Processing (fork()) and Multi-Threading (pthread) under varying resource constraints (CPU, Memory, and I/O).

Implementation Details:

- **Roll Number Logic:** Based on the last digit of my roll number (**8**), the loop count multiplier was set to 8 times 10^x.
- **Program A (A.cpp):** Implements process-based concurrency using fork(). Each child process runs in an isolated memory space.
- **Program B (B.cpp):** Implements thread-based concurrency using pthread_create(). Threads share the heap and global variables but maintain separate stack frames.

Worker Functions:

1. **CPU Worker:** Performs heavy floating-point arithmetic (matrix-like operations) to saturate the ALU.
2. **Memory Worker:** Allocates 100MB buffers and writes to them with a stride of 4096 bytes (page size) to force TLB misses and physical RAM access, bypassing cache optimizations.
3. **I/O Worker:** Writes 1KB chunks to disk with periodic fsync() calls to force flushing from the page cache to physical storage, simulating a heavy journaled write load.

---

2. Methodology (Bash Automation)

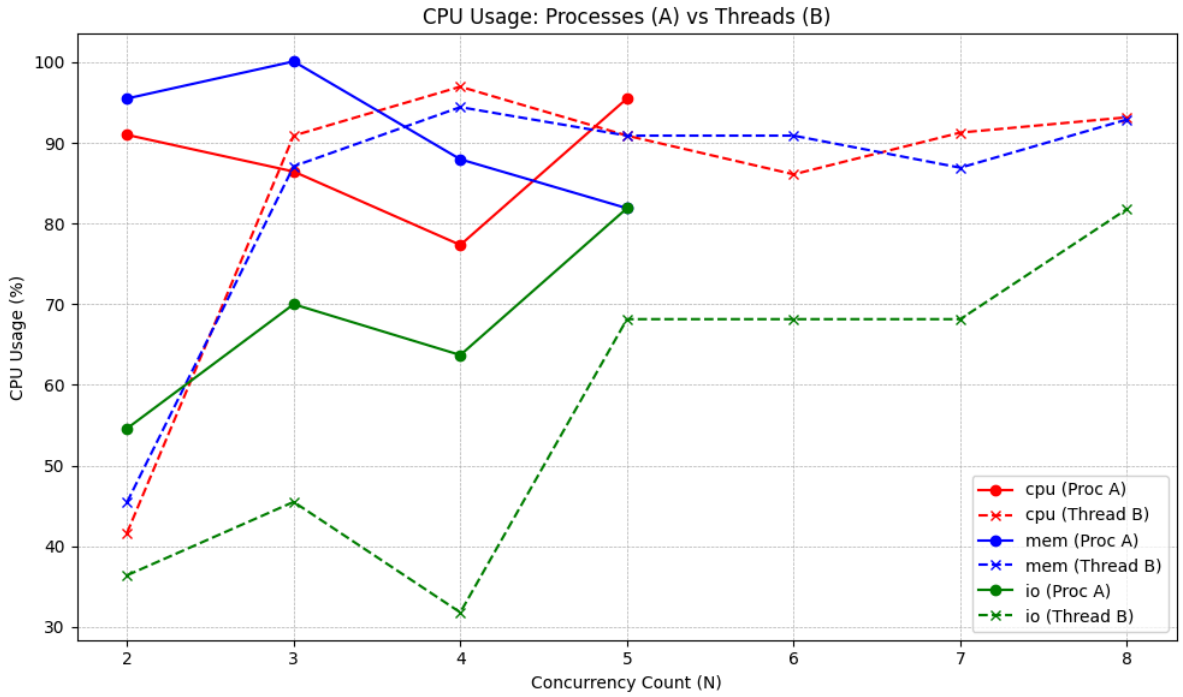
To ensure consistent measurements, a bash script (bench.sh) was developed with the following key features:

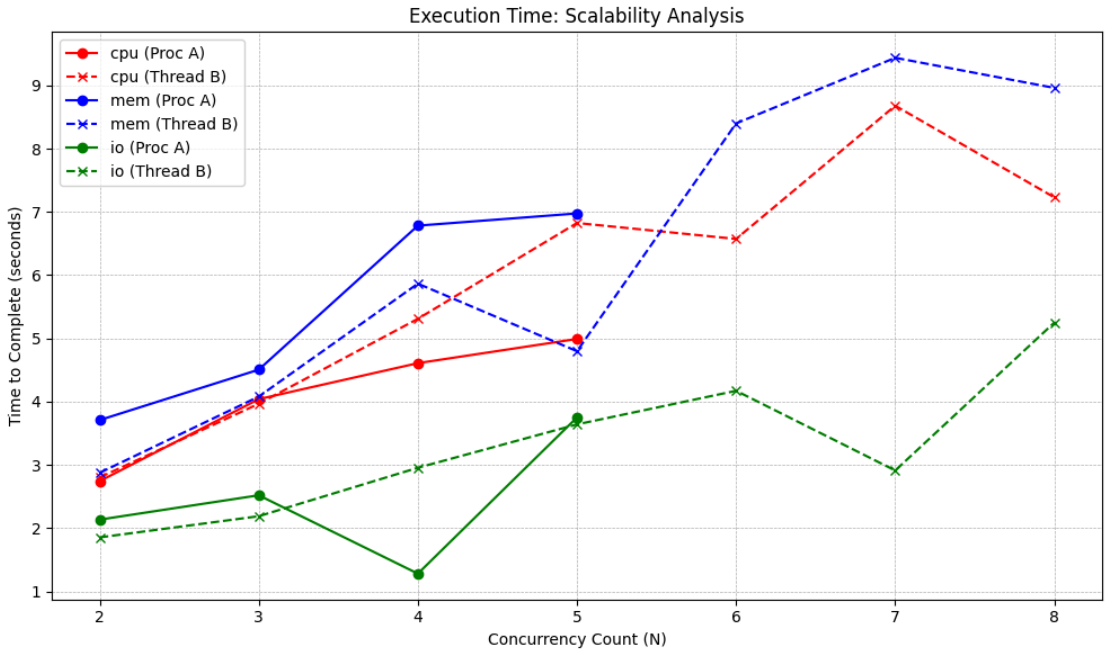
- **CPU Pinning (taskset -c 0):** Both Program A and Program B were pinned to a specific CPU core. This was a critical design choice to measure **concurrency overhead** (context switching) rather than parallelism. If allowed to run on multiple cores, the OS would simply distribute the load, masking the efficiency differences between threads and processes.
- **Metrics Collection:**
    - top: Used in batch mode to capture instantaneous CPU and Memory usage.
    - iostat: Used to measure disk write throughput (MB/s).
    - time: Used to measure the wall-clock execution time.

3. Analysis & Observations

(A) CPU-Intensive Workload Analysis

- **Graphs:**

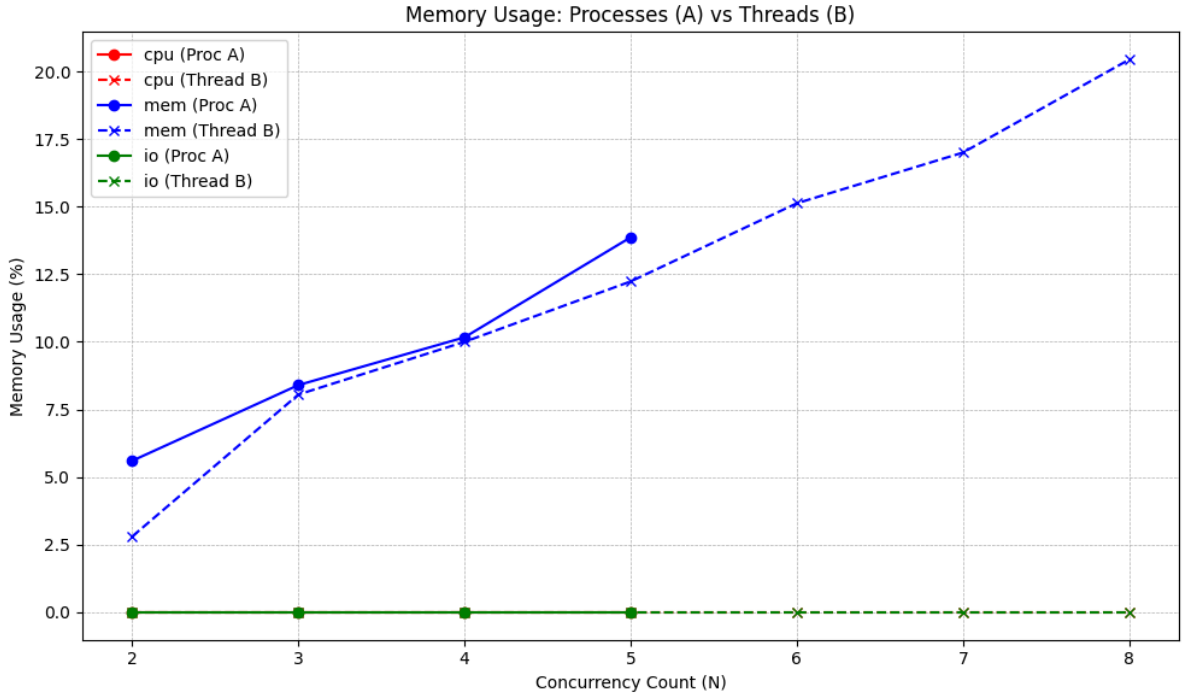




- **Observation:** The execution time increases linearly as the number of workers (N) increases.
- **Analysis:** Since the programs were pinned to a single core, true parallelism was impossible. The CPU had to time-slice between workers.
    - **Threads vs. Processes:** Program B (Threads) showed slightly better performance (lower overhead) than Program A (Processes). This is because switching between threads (which share the same page table) is computationally cheaper for the OS scheduler than switching between processes (which requires swapping the CR3 register/page tables).

(B) Memory-Intensive Workload Analysis

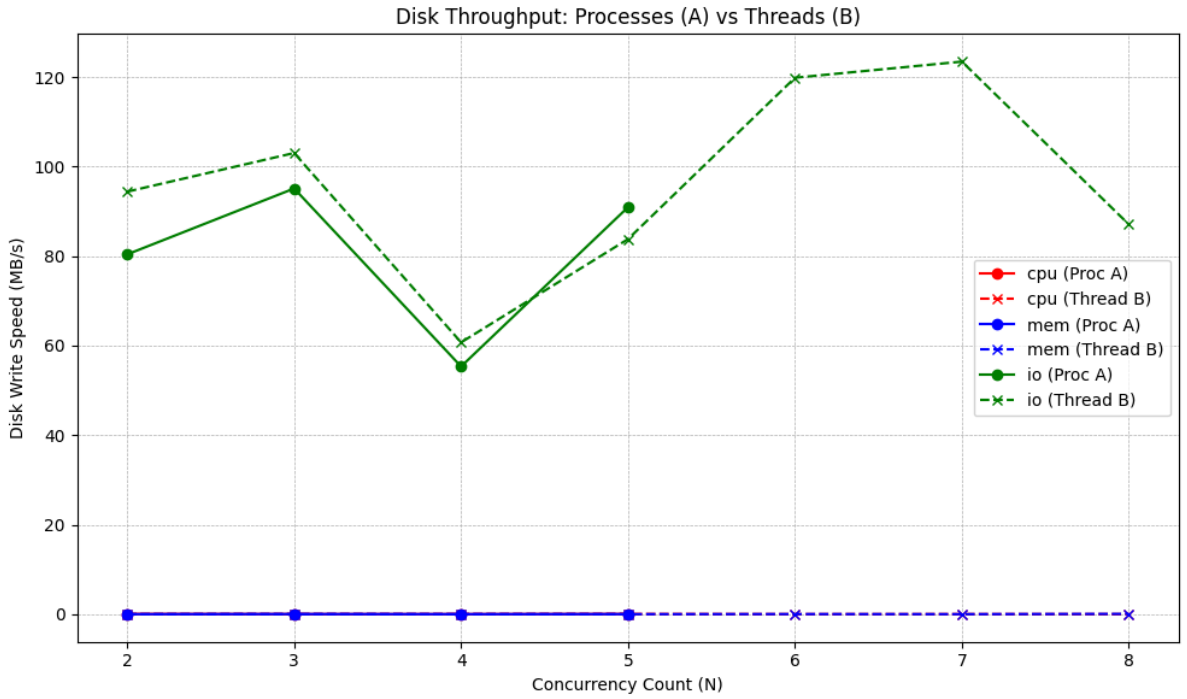
- **Graph:**



- **Observation:** Memory usage scales linearly (N \times 100\text{MB}).
- **Analysis:** Both models successfully allocated the requested memory. The stride access pattern ensured that the OS could not optimize memory usage via "Copy-on-Write" or lazy allocation.
    - **Bottleneck:** The limiting factor here was the memory bus bandwidth. As N increased, the contention for the bus slowed down the effective write speed, though the total memory footprint remained predictable.

(C) I/O-Intensive Workload Analysis

- **Graph:**



- **Observation:** The I/O throughput plateaued and did not scale linearly with N.

- **Analysis:** This workload was dominated by the fsync() system call.
    - **The Blocking Factor:** fsync puts the calling thread/process into an Uninterruptible Sleep (D state) while waiting for the physical disk controller.
    - **Throughput Limit:** The hardware (disk speed) is the hard limit. Spawning 5 processes to write to the same disk simply increases the queue depth at the I/O scheduler, adding latency without increasing throughput. In fact, excessive context switching during I/O waits caused the CPU usage to hover around 30-40% (as seen in the CSV data), indicating the CPU was idle waiting for the disk.

---

4. Conclusion

The experiment confirms that **threads are generally lighter-weight** than processes for CPU-bound tasks due to lower context-switching costs. However, for I/O-bound tasks, the bottleneck shifts entirely to the hardware, rendering the choice of software concurrency model less significant. Pinning execution to a single core successfully highlighted the cost of OS scheduling overhead.

---

5. AI Usage Declaration

- **AI** tools have been used by me for help in understanding the core task.
- I have used AI for help in generating the bench.sh script for benchmarking and some help in the CPP files.
- I have used AI to research which graphs are best to plot given my data.

6. Github Repository

- **URL:** https://github.com/UtsavSDavda/GRS-assignment/tree/main/1