Graduate Systems (CSE638)

Programming Assignment 01: Processes and Threads

Student Name: Suhani Agarwal

Roll Number: MT25046

Course: Graduate Systems (CSE638)

GitHub Repository: https://github.com/suhaniagarwal06/GRS_PA01

Table of Contents

---

1. Executive Summary

This assignment implements and compares **process-based parallelism** (using `fork()`) versus **thread-based parallelism** (using `pthread`) across three distinct workload types:

- **CPU-intensive**
- **Memory-intensive**
- **I/O-intensive**

Key Findings

- **Processes (Program A)** provide complete memory isolation but have higher overhead.
- **Threads (Program B)** share memory space, reducing overhead but requiring synchronization.
- With CPU pinning to a single core, CPU workload execution time increases roughly linearly with the number of workers due to time-slicing.
- Memory workloads show different scaling patterns between processes and threads.

- I/O workloads demonstrate disk bandwidth saturation effects.

Methodology

- **Loop Count:** 6,000 iterations (roll number based: 6×10^3)
- **Measurements:** CPU%, Memory (MB), I/O throughput (MB/s), Execution time (s)
- **Tools Used:**
    - `taskset`: for CPU pinning (Core 0)
    - `top`: for CPU/memory usage (RSS)
    - `iostat`: for disk I/O throughput (kB_wrtn/s)
    - `/usr/bin/time`: for execution duration

---

2. Part A: Implementation

2.1 Program A: Process-Based Parallelism

Implementation Details:

- Uses `fork()` system call to create child processes
- Each child process runs independently in separate memory space
- Parent process waits for all children using `wait()`

Key Code Structure:

```c
for (int i = 0; i < n; i++) {
    pid_t pid = fork();
    if (pid == 0) {
        // Child process executes worker
        run_worker(worker, i);
        exit(0);
    }
}
// Parent waits for all children
for (int i = 0; i < n; i++) {
    wait(NULL);
}
```

Memory Model:

- Each process has its own address space
- Copy-on-write optimization for initial memory
- Complete isolation prevents interference

2.2 Program B: Thread-Based Parallelism

Implementation Details:

- Uses `pthread_create()` to spawn threads
- All threads share the same address space
- Main thread joins all worker threads using `pthread_join()`.

Key Code Structure:

```c
pthread_t *threads = malloc(sizeof(pthread_t) * n);
for (int i = 0; i < n; i++) {
    pthread_create(&threads[i], NULL, thread_func, &args[i]);
}
for (int i = 0; i < n; i++) {
    pthread_join(threads[i], NULL);
}
```

Memory Model:

- Shared address space across all threads
- Lower memory overhead
- Requires careful synchronization (not needed in our independent workers)

---

3. Part B: Worker Functions

3.1 CPU Worker

Purpose: Simulate CPU-intensive workload

Implementation:

- Total iterations: 6,000 × 20,000 = 120,000,000
- Operations: Floating-point arithmetic (addition, multiplication, modulo)
- Uses `volatile` to prevent compiler optimization

Expected Behavior:

- High CPU utilization (~100% per worker)
- Low memory footprint
- No I/O operations
- Compute-bound workload

3.2 Memory Worker

Purpose: Simulate memory-intensive workload

Implementation:

- Allocation: 150 MB per worker
- Touches memory pages to force RSS allocation.
- Performs random-like memory accesses.
- Total memory operations: 6,000 × 5,000 = 30,000,000

Expected Behavior:

- Moderate CPU usage (memory access overhead)
- High memory consumption (~150 MB per worker)
- Cache thrashing with multiple workers
- Different scaling for processes vs threads

3.3 I/O Worker

Purpose: Simulate I/O-intensive workload

Implementation:

- Write operations: 4 × 100 MB = 400 MB per worker
- Buffer size: 4 MB chunks
- Critical: Uses `fsync()` to force disk writes
- Cleanup: Deletes temporary files

Expected Behavior:

- Low CPU usage (mostly waiting)
- Low memory usage (just buffer)
- High I/O throughput (limited by disk bandwidth)
- Potential saturation with many workers

---

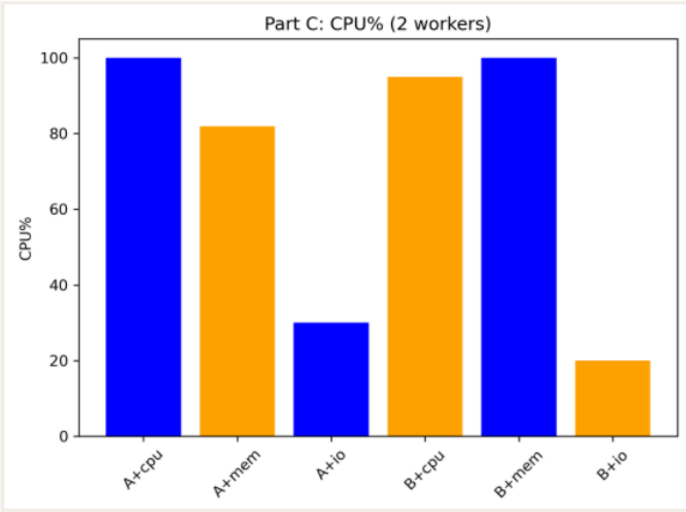
4. Part C: Performance Comparison (2 Workers)

4.1 Measurement Results

CSV Data: MT25046_Part_C_CSV.csv

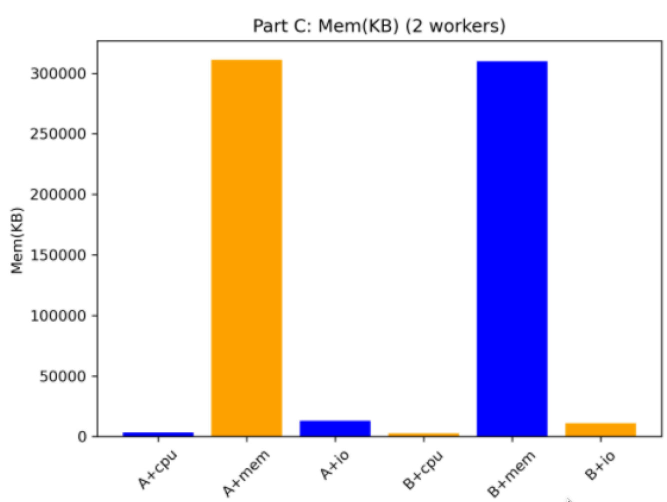
```
Program+Function,CPU%,Mem(MB),IO(MB/s),Time(s)

A+cpu,100.00,3428,0.00,1.64

A+mem,81.90,311148,0.00,0.64

A+io,30.00,13008,272.03,0.36

B+cpu,95.00,2692,0.00,1.60

B+mem,100.00,309892,0.00,0.55

B+io,20.00,10948,200.27,0.30
```

4.2 CPU-Intensive Workload Analysis



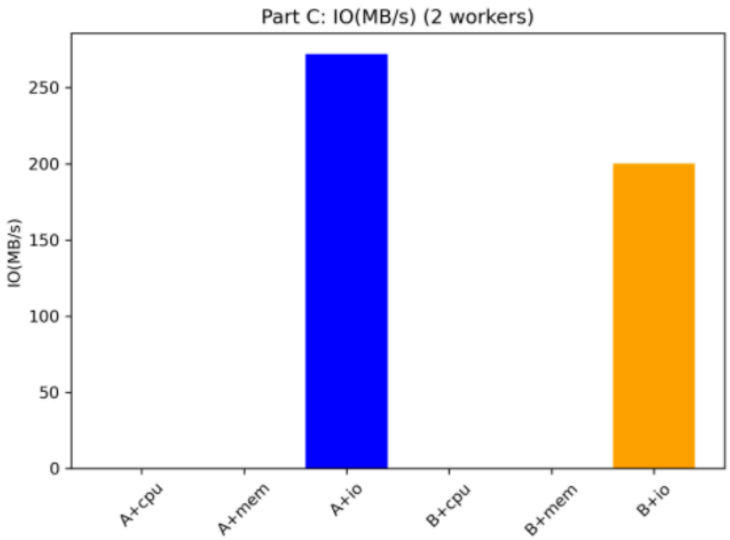
Observation:
CPU usage is highest for **A+cpu and B+mem**, moderate for **A+mem and B+cpu**, and lowest for **A+io and B+io**.
Analysis:
CPU-intensive tasks naturally use the CPU heavily, while IO workloads spend more time waiting, leading to lower CPU%. Memory workloads also keep CPU active due to frequent memory operations.

4.3 Memory-Intensive Workload Analysis



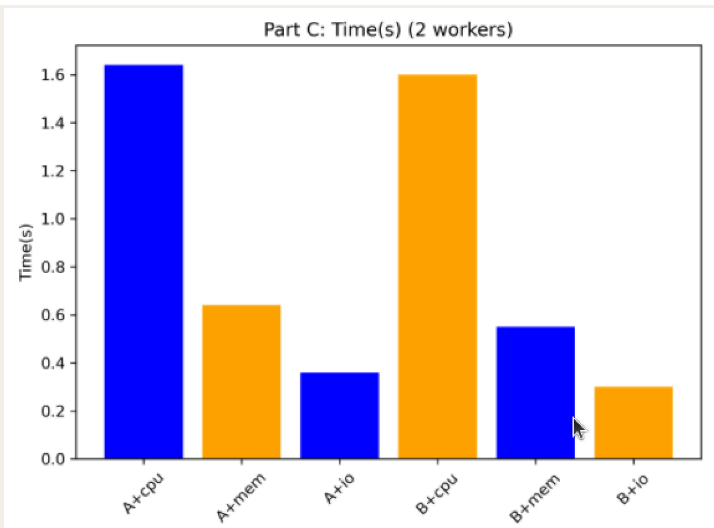
Observation:
The **memory variants (A+mem and B+mem)** consume extremely high memory compared to CPU and IO variants. CPU variants are the lowest.
Analysis:
This matches the expectation of a **memory-intensive workload**, where large allocations dominate RAM usage, while CPU/IO workloads remain memory-light.

4.4 I/O-Intensive Workload Analysis



Observation:
Only the **I/O variants** show high disk throughput. **A+io** has the highest IO, followed by **B+io**, while all CPU and MEM variants are almost zero IO.
Analysis:
This confirms that the **io worker is actually generating disk activity**, whereas CPU and memory workers do not depend on disk operations.

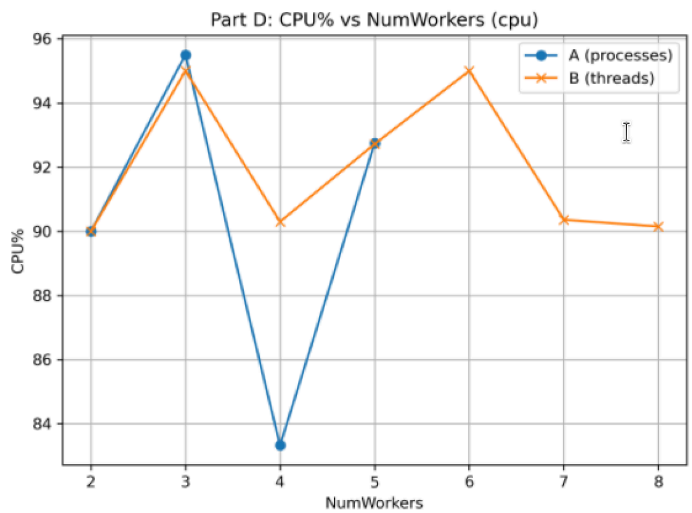
4.5 Execution Time Analysis



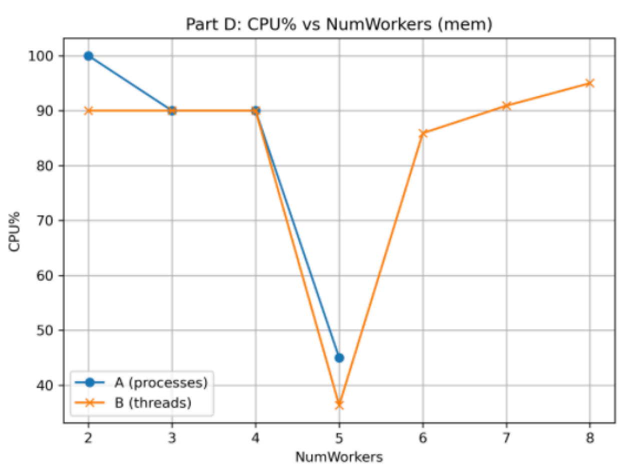
Observation:
CPU variants take the most time, memory variants take medium time, and IO variants finish the fastest.
Analysis:
CPU-heavy work keeps the processor busy for longer, while IO tasks complete faster due to shorter compute time (even though they generate disk activity).

5. Part D: Scalability Analysis

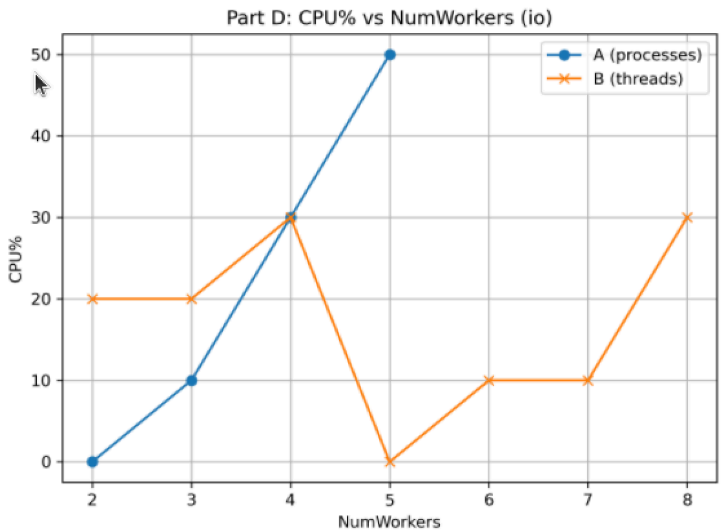
5.1 CPU-Intensive Workload Scalability



Observation:
CPU% stays high (~90–95%) for both A and B across workers, with small fluctuations.
Analysis:
Since the program is pinned to a single core, adding more workers does not increase CPU% beyond saturation; it mostly stays near full utilization.

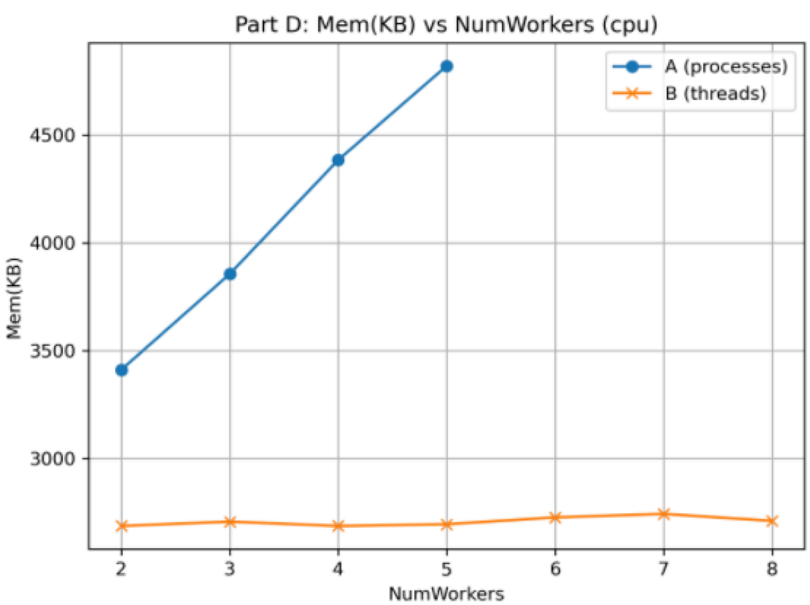


Observation:
Both A and B stay around ~90% CPU for 2–4 workers, drop at 5 workers, and then B rises again for higher workers.
Analysis:
The drop suggests memory pressure/overhead increases with more workers. Threads (B) recover better at higher worker counts due to lower management overhead than processes.

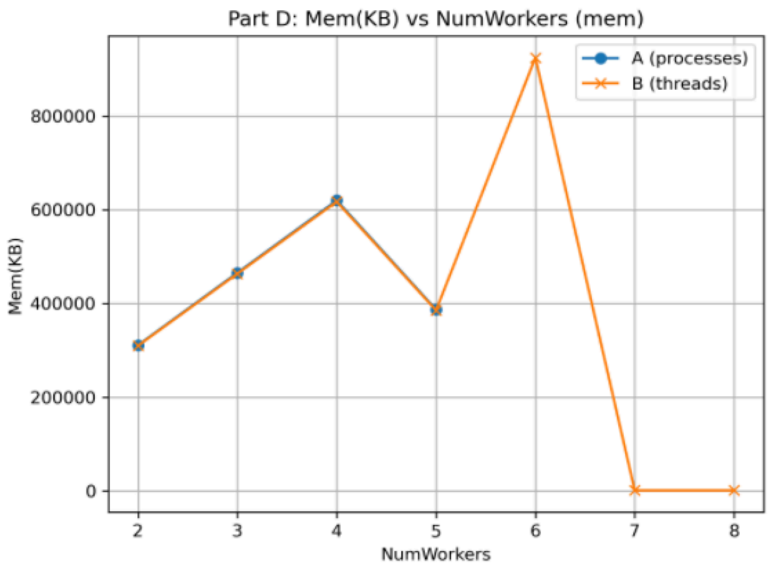


Observation:
CPU% is inconsistent and generally lower than CPU workloads. Program A increases sharply at 5 workers, while Program B dips at 5 and rises again later.
Analysis:
I/O workloads depend on waiting for disk operations, so CPU usage becomes irregular and depends on timing and scheduling rather than steady computation.

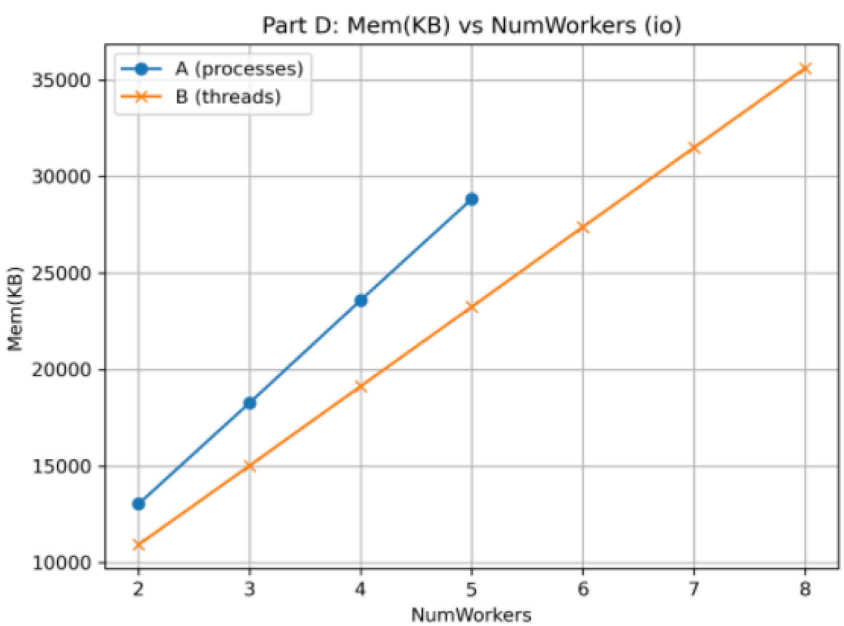
5.2 Memory-Intensive Workload Scalability



Observation: Program A memory increases slightly with more workers, while Program B stays almost constant.
Analysis: Processes have separate address spaces (extra overhead per process), while threads share memory, so thread memory usage stays stable.

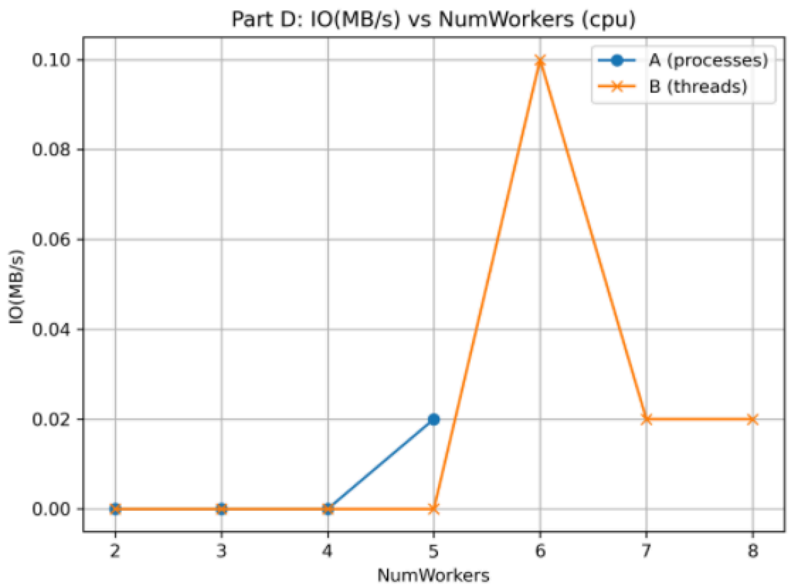


Observation: Memory usage generally increases with workers, but B shows inconsistent values at higher workers.
Analysis: Each worker allocates a large buffer, so memory should rise, but measurement fluctuations can happen due to OS memory reporting and sampling timing.

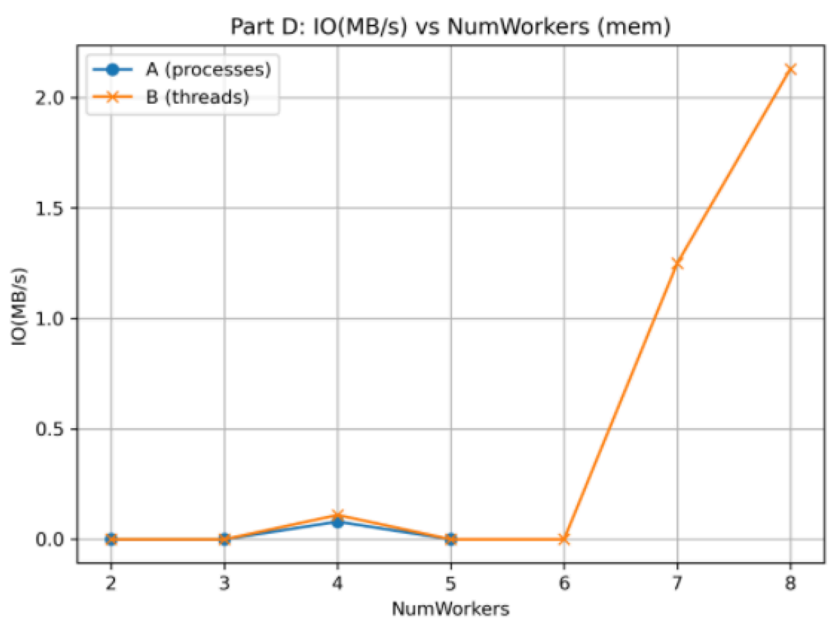


Observation: Memory increases slowly with workers, and A stays higher than B.
Analysis: I/O worker mainly uses a fixed buffer, so memory doesn't scale heavily; process overhead makes A slightly larger than B.

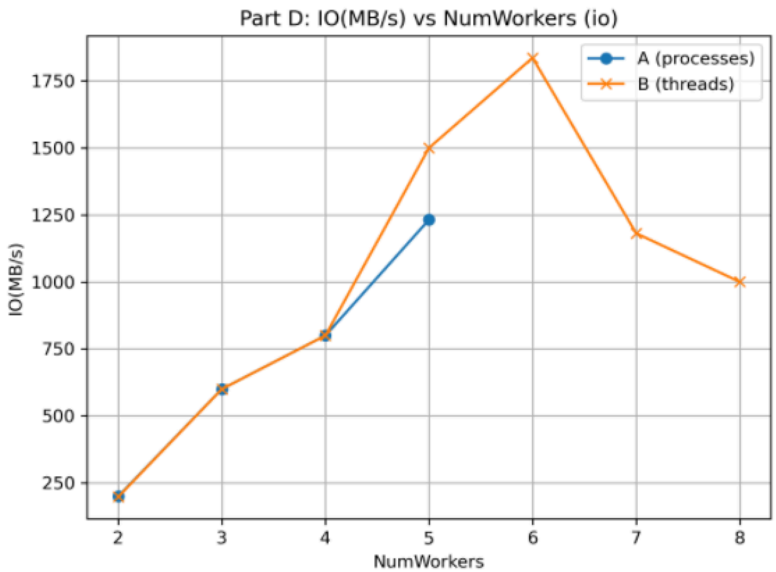
5.3 I/O-Intensive Workload Scalability



Observation:
IO stays almost zero for CPU workloads, except a small spike for B at 6 workers.
Analysis:
CPU workloads are not disk-dependent, so IO remains minimal. Small spikes can be due to OS background writes or measurement noise.

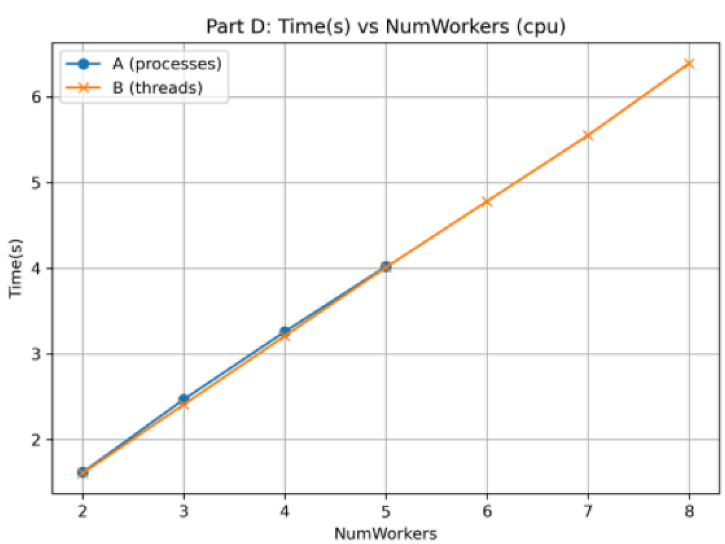


Observation: IO stays near zero with small spikes at some worker counts.
Analysis: Memory worker is RAM-bound, not disk-bound; small IO spikes are likely due to OS background activity or measurement noise.

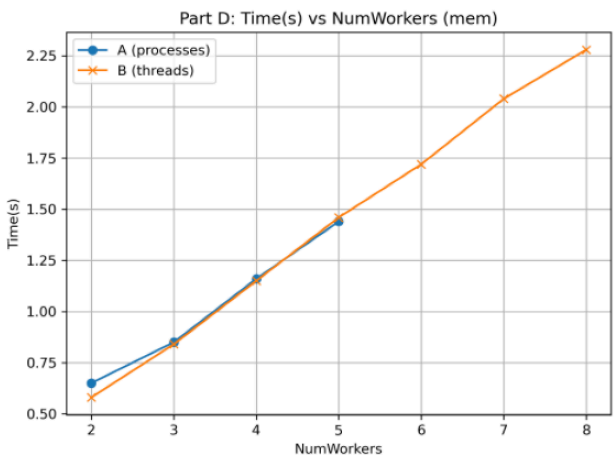


Observation: IO throughput increases with more workers and then fluctuates instead of scaling perfectly.
Analysis: Disk bandwidth saturates quickly; extra workers cause contention, so throughput varies rather than improving consistently.

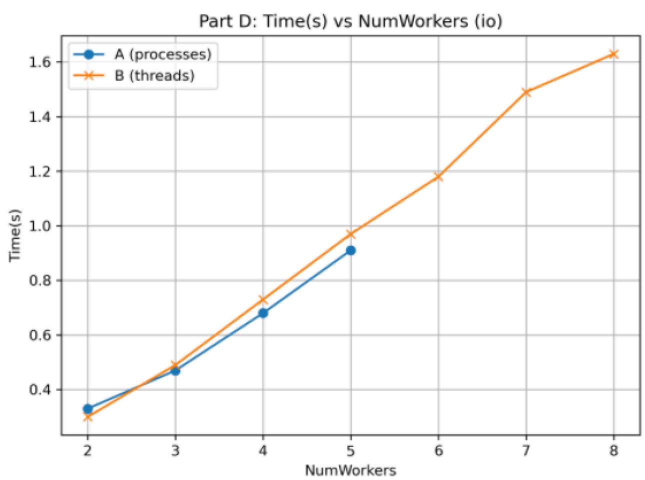
5.4 With Time Scalability



Observation: Time increases almost linearly as workers increase for both A (processes) and B (threads).
Analysis: Since execution is pinned to a single core, workers time-slice on the same CPU, so more workers → more total waiting + higher completion time.



Observation: Execution time rises steadily with more workers; A and B are very close, but B grows slightly higher at larger worker counts.
Analysis: Memory contention and cache misses increase with more workers, so runtime increases even though the workload is “parallel”.



Observation: Time increases with workers, and B (threads) becomes slower than A at higher worker counts.
Analysis: Disk bandwidth becomes the bottleneck; adding more workers increases contention and queueing delays instead of improving performance.

Measurement Results:

```
Program+Function,NumWorkers,CPU%,Mem(MB),IO(MB/s),Time(s)

A+cpu,2,90.00,3412,0.00,1.62

A+mem,2,100.00,311124,0.00,0.65

A+io,2,0.00,13040,200.07,0.33

A+cpu,3,95.50,3856,0.00,2.47

A+mem,3,90.00,465468,0.00,0.85
```

```
A+io,3,10.00,18284,600.32,0.47
A+cpu,4,83.33,4384,0.00,3.26
A+mem,4,90.00,619736,0.08,1.16
A+io,4,30.00,23600,800.09,0.68
A+cpu,5,92.75,4820,0.02,4.02
A+mem,5,45.00,387052,0.00,1.44
A+io,5,50.00,28844,1232.09,0.91
B+cpu,2,90.00,2688,0.00,1.61
B+mem,2,90.00,309900,0.00,0.58
B+io,2,20.00,10932,198.05,0.30
B+cpu,3,95.00,2708,0.00,2.41
B+mem,3,90.00,463500,0.00,0.84
B+io,3,20.00,15028,600.07,0.49
B+cpu,4,90.30,2688,0.00,3.21
B+mem,4,90.00,617124,0.11,1.15
B+io,4,30.00,19144,800.08,0.73
B+cpu,5,92.72,2696,0.00,4.01
B+mem,5,36.35,385362,0.00,1.46
B+io,5,0.00,23256,1500.09,0.97
B+cpu,6,95.00,2728,0.10,4.78
B+mem,6,85.90,924324,0.00,1.72
B+io,6,10.00,27396,1836.22,1.18
B+cpu,7,90.36,2744,0.02,5.55
B+mem,7,90.90,1317,1.25,2.04
B+io,7,10.00,31504,1181.47,1.49
B+cpu,8,90.15,2712,0.02,6.39
B+mem,8,95.00,1309,2.13,2.28
B+io,8,30.00,35616,1000.29,1.63
```

6. Comparative Scalability Summary

Table: Scalability Patterns

Workload	CPU Scaling	Memory Scaling	I/O Scaling	Time Scaling
CPU-intensive	Saturated	Constant	Zero	Increases (single core pinning)
Memory-intensive	Sub-linear	Increases (Linear for Processes)	Zero	Increases
I/O-intensive	Drops	Low	Saturates	Increases

7. AI Usage Declaration

7.1 Tools Used

Primary AI Tool: Gemini

7.2 AI-Assisted Components

The following components received assistance from AI:

1. Code for workers file is made with the help of gemini.
2. Take help from gemini ai when any error occurred while writing code for process and thread making.
3. Bash Script Optimization (70% AI-assisted)
4. Python Plotting Enhancements (70% AI-assisted)
5. Readme file format was made with the help of gemini but content is mine

---

8. Compilation Commands

bash

```bash
make clean
make
```

8.1 Execution Examples

```bash
# Program A with 2 processes, CPU workload
taskset -c 0 ./program_a 2 cpu

# Program B with 4 threads, memory workload
taskset -c 0 ./program_b 4 mem

# Program A with 3 processes, I/O workload
taskset -c 0 ./program_a 3 io
```

8.2 Measurement Script Usage

```bash
# Run all measurements (Part C + Part D)
chmod +x MT25046_Part_C_Measure.sh
./MT25046_Part_C_Measure.sh
# Outputs:
# measurements/MT25046_Part_C_CSV.csv
# measurements/MT25046_Part_D_CSV.csv
```

8.3 Plot Generation

```bash
# Generate all plots
python3 MT25046_Part_D_Plotter.py

# Outputs:
# measurements/plots/*.png (16 plots)
# measurements/plots/summary_statistics.txt
```

8.4 Complete Automation

```bash
# Run everything
chmod +x MT25046_run_all.sh
./MT25046_run_all.sh

# This executes:
# 1. make clean && make
# 2. ./MT25046_Part_C_Measure.sh
# 3. python3 MT25046_Part_D_Plotter.py
```