Lab 0 Report

Foundations of Parallel Computing

Name: Ahmad Bin Rashid

Roll No: 2023-CS-53

Instructor: Sir Waqas Ali

Course: Distributed and Parallel Computing

University of Engineering and Technology

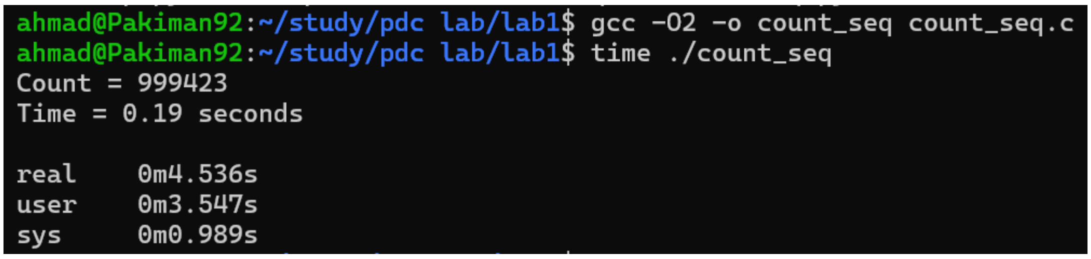
February 17, 2026

1 Introduction

This lab explores the fundamentals of parallel and distributed computing using threads and processes. Sequential and parallel implementations of a counting problem were developed in C and Python. Performance was measured to understand speedup, overheads, and the impact of Python's Global Interpreter Lock (GIL).

2 Screenshots of Program Execution

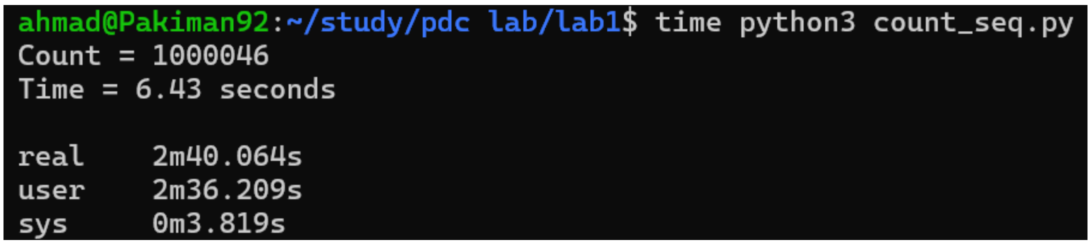
Sequential C Program



```
ahmad@Pakiman92:~/study/pdc lab/lab1$ gcc -O2 -o count_seq count_seq.c
ahmad@Pakiman92:~/study/pdc lab/lab1$ time ./count_seq
Count = 999423
Time = 0.19 seconds

real    0m4.536s
user    0m3.547s
sys     0m0.989s
```

Figure 1: Sequential C execution

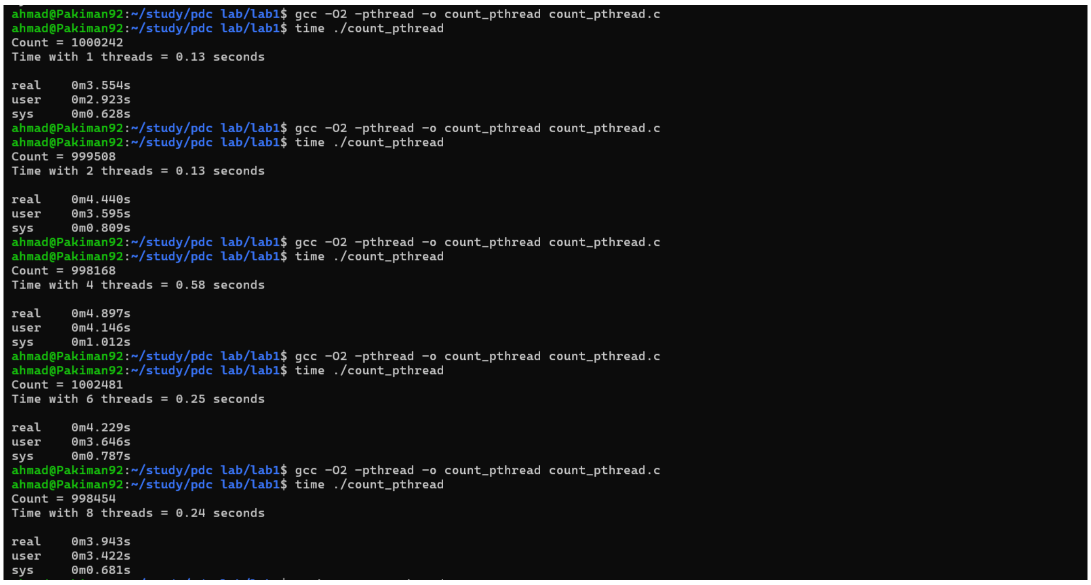
Sequential Python Program



```
ahmad@Pakiman92:~/study/pdc lab/lab1$ time python3 count_seq.py
Count = 1000046
Time = 6.43 seconds

real    2m40.064s
user    2m36.209s
sys     0m3.819s
```

Figure 2: Sequential Python execution

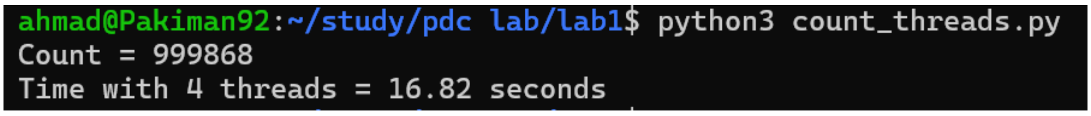
C Pthreads Execution



```
ahmad@Pakiman92:~/study/pdc lab/lab1$ gcc -O2 -pthread -o count_pthread count_pthread.c
ahmad@Pakiman92:~/study/pdc lab/lab1$ time ./count_pthread
Count = 1000242
Time with 1 threads = 0.13 seconds

real    0m3.554s
user    0m2.923s
sys     0m0.628s
ahmad@Pakiman92:~/study/pdc lab/lab1$ gcc -O2 -pthread -o count_pthread count_pthread.c
ahmad@Pakiman92:~/study/pdc lab/lab1$ time ./count_pthread
Count = 999508
Time with 2 threads = 0.13 seconds

real    0m4.440s
user    0m3.595s
sys     0m0.809s
ahmad@Pakiman92:~/study/pdc lab/lab1$ gcc -O2 -pthread -o count_pthread count_pthread.c
ahmad@Pakiman92:~/study/pdc lab/lab1$ time ./count_pthread
Count = 998168
Time with 4 threads = 0.58 seconds

real    0m4.897s
user    0m4.146s
sys     0m1.012s
ahmad@Pakiman92:~/study/pdc lab/lab1$ gcc -O2 -pthread -o count_pthread count_pthread.c
ahmad@Pakiman92:~/study/pdc lab/lab1$ time ./count_pthread
Count = 1002481
Time with 6 threads = 0.25 seconds

real    0m4.229s
user    0m3.646s
sys     0m0.787s
ahmad@Pakiman92:~/study/pdc lab/lab1$ gcc -O2 -pthread -o count_pthread count_pthread.c
ahmad@Pakiman92:~/study/pdc lab/lab1$ time ./count_pthread
Count = 998454
Time with 8 threads = 0.24 seconds

real    0m3.943s
user    0m3.422s
sys     0m0.681s
```

Figure 3: C pthread executions

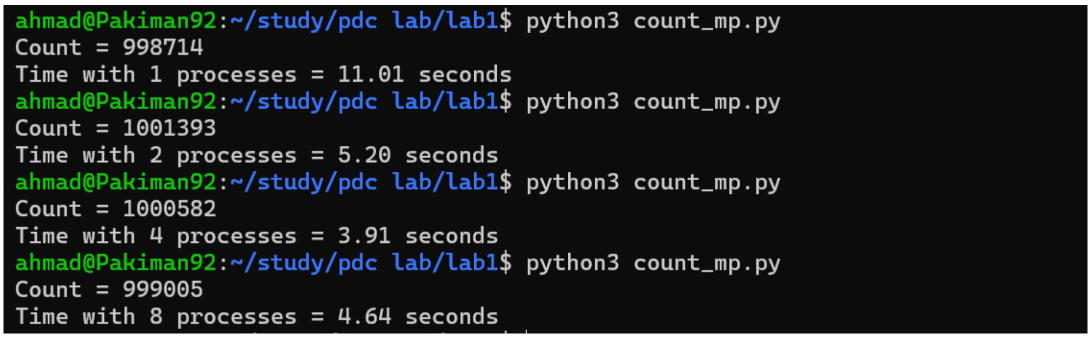
Python Threading Execution



```
ahmad@Pakiman92:~/study/pdc lab/lab1$ python3 count_threads.py
Count = 999868
Time with 4 threads = 16.82 seconds
```

Figure 4: Python threading execution

Python Multiprocessing Execution



```
ahmad@Pakiman92:~/study/pdc lab/lab1$ python3 count_mp.py
Count = 998714
Time with 1 processes = 11.01 seconds
ahmad@Pakiman92:~/study/pdc lab/lab1$ python3 count_mp.py
Count = 1001393
Time with 2 processes = 5.20 seconds
ahmad@Pakiman92:~/study/pdc lab/lab1$ python3 count_mp.py
Count = 1000582
Time with 4 processes = 3.91 seconds
ahmad@Pakiman92:~/study/pdc lab/lab1$ python3 count_mp.py
Count = 999005
Time with 8 processes = 4.64 seconds
```

Figure 5: Python multiprocessing execution

3 Execution Time Comparison

3.1 Measured Execution Times

Program Version	**Execution Time (seconds)**
C Sequential	0.19
C Pthreads (1 thread)	0.13
C Pthreads (2 threads)	0.13
C Pthreads (4 threads)	0.58
C Pthreads (6 threads)	0.25
C Pthreads (8 threads)	0.24
Python Sequential	6.43
Python Threads (4 threads)	16.82
Python Multiprocessing (1 process)	11.01
Python Multiprocessing (2 processes)	5.20
Python Multiprocessing (4 processes)	3.91
Python Multiprocessing (8 processes)	4.64

Table 1: Execution time comparison of different implementations

4 Speedup Analysis for C Pthreads

Speedup is calculated as:

$$\text{Speedup} = \frac{T_1}{T_N}$$

Where T_1 is the execution time with one thread.

Threads	**Speedup**
1	1.00
2	1.00
4	0.22
6	0.52
8	0.54

Table 2: Speedup of C pthread implementation

The speedup is not linear due to thread management overhead, memory bandwidth saturation, and the sequential portions of the program as explained by Amdahl's Law.

5 Speedup Graph for C Pthreads

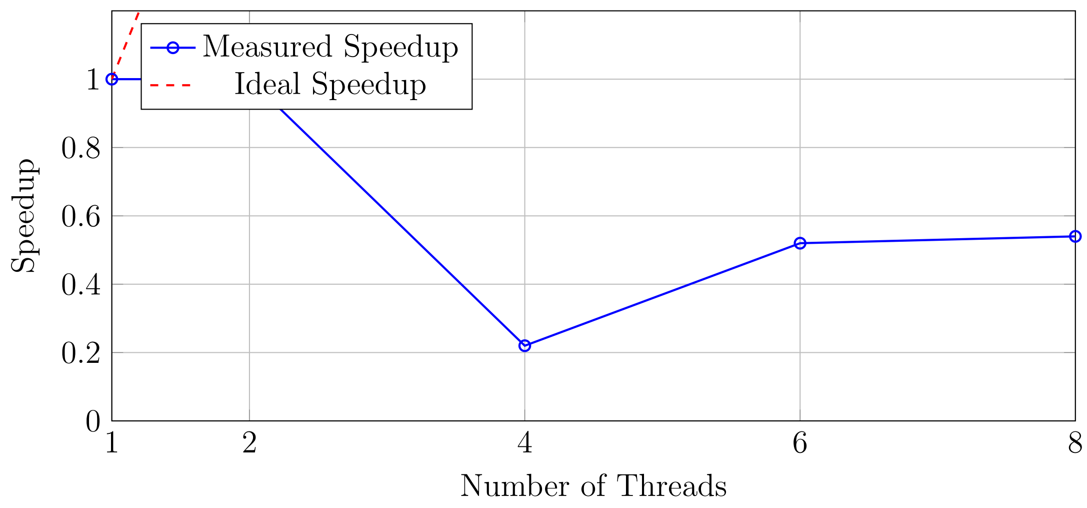


Figure 6: Speedup vs number of threads for C pthread implementation

6 Self-Assessment Questions

1. Difference between a process and a thread

A process has its own separate memory space, while threads share the same address space within a process. Process creation is expensive due to memory and resource allocation. Threads are lightweight and faster to create. Communication between threads is easier but requires synchronization.

2. Why is a mutex required?

A mutex prevents race conditions when multiple threads update a shared global counter. Without mutual exclusion, updates can overlap and produce incorrect results. The mutex ensures only one thread modifies the shared variable at a time.

3. Explanation of `lscpu` output

The `CPU(s)` field shows the total logical CPUs available. `Core(s) per socket` indicates physical cores. `Thread(s) per core` shows hardware multithreading (hyper-threading). Total logical CPUs are computed from these values.

4. Speedup behavior

The observed speedup is not linear. Thread creation overhead, synchronization, memory contention, and the sequential fraction of the program limit scalability. According to Amdahl's Law, these factors cap maximum achievable speedup.

5. Python threading performance

Python threads are slower for CPU-bound tasks due to the Global Interpreter Lock (GIL). The GIL allows only one thread to execute Python bytecode at a time. Threading overhead further increases execution time.

6. False sharing

False sharing occurs when threads modify different variables that reside on the same cache line. Cache coherence mechanisms cause unnecessary invalidations. This leads to increased memory traffic and degraded performance.

7. Private counters optimization

Using private counters allows threads to avoid frequent locking. Each thread updates its own counter and locks only once to update the global value. This reduces lock contention and improves performance.

8. Threads vs processes in Python

Threads are suitable for I/O-bound tasks because the GIL is released during I/O. Processes are preferred for CPU-bound tasks as they bypass the GIL and allow true parallel execution. Multiprocessing provides better scalability for compute-heavy workloads.

7 Reflection

This lab demonstrated that parallel programming introduces both performance benefits and complexity. While C with pthreads achieved significant speedups, the gains were limited by memory bandwidth and synchronization overhead. Python threading performed poorly for CPU-bound tasks due to the GIL, confirming theoretical expectations. Multiprocessing improved performance but introduced inter-process communication overhead. The experiments highlighted the importance of choosing the correct parallel model based on workload characteristics. Overall, the lab provided practical insight into real-world limitations of parallel execution.