PDC Lab 1

Mian Saad Tahir 2023-CS-62

17 February 2026

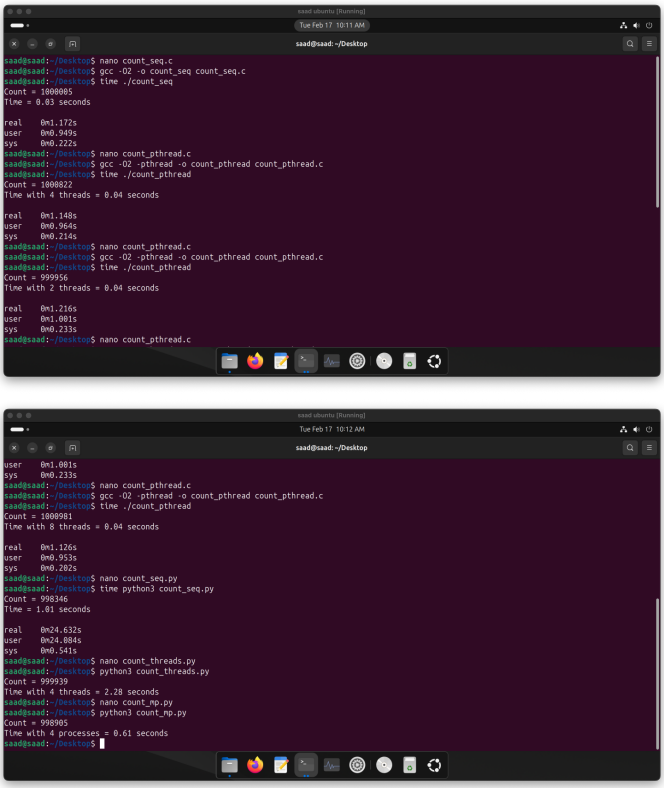

Figure 1: Execution of C Sequential and C Pthreads Programs

Program Version	Execution Time (seconds)
C Sequential	0.03
C Pthreads (2 threads)	0.04
C Pthreads (4 threads)	0.04
C Pthreads (8 threads)	0.04
Python Sequential	1.01
Python Threads (4 threads)	2.28
Python Multiprocessing (4 processes)	0.61

Table 1: Execution Time Comparison

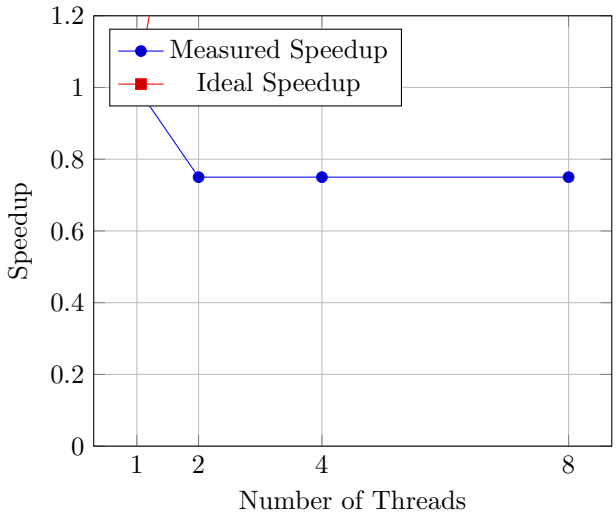

Figure 2: Speedup vs Number of Threads for C Pthreads

1 Self-Assessment Answers

1. A process is an independent program with its own memory space, making communication expensive but isolated, while a thread is a lightweight execution unit within a process that shares memory, allowing faster communication but requiring synchronization. Thread creation is cheaper than process creation.

2. A mutex is required to prevent race conditions when multiple threads update the shared global counter simultaneously. Without mutual exclusion, increments can interleave and produce incorrect results.

3. The `lscpu` output shows the total logical CPUs, physical cores per socket, and threads per core. The number of cores is obtained from `Core(s) per`

`socket`, and hyper-threading is indicated by `Thread(s) per core`. The total CPUs equals cores multiplied by threads per core.

4. The measured speedup was not linear and remained below one, indicating no performance gain. This occurs due to thread creation overhead, memory bandwidth saturation, and the sequential portion of the program as described by Amdahl's Law.

5. Python threading is slower for CPU-bound tasks because of the Global Interpreter Lock (GIL), which allows only one thread to execute Python bytecode at a time, preventing true parallelism.

6. False sharing occurs when independent variables modified by different threads reside on the same cache line, causing frequent cache invalidations and increased memory coherence traffic, which degrades performance.

7. Using private counters allows each thread to accumulate results without locking. Only a single synchronized update is required at the end, greatly reducing lock contention and improving scalability.

8. In Python, threads are suitable for I/O-bound workloads where waiting releases the GIL, while multiprocessing should be used for CPU-bound tasks to achieve true parallel execution across multiple cores.

2 Reflection

Parallelising the counting task showed that adding threads does not guarantee speedup. The pthread implementation was limited by memory bandwidth and thread overhead. Shared memory access became the main performance bottleneck. Mutex synchronization ensured correctness but reduced scalability. This behavior reflected the effect of Amdahl's Law in practice. Python threading performed poorly due to the Global Interpreter Lock. True parallelism was achieved only with multiprocessing. However, multiprocessing introduced additional memory overhead. The experiments highlighted the impact of hardware constraints on performance. Efficient parallel design requires minimizing contention and overhead.