Lab 0: Foundations of Parallel Computing – Threads, Processes, and Linux Tools

1. Learning Objectives

By the end of this lab, you will be able to:

- Explain the historical shift to multicore processors and why parallel programming is essential.
- Distinguish between processes, threads, concurrency, and parallelism.
- Understand shared-memory vs. distributed-memory architectures.
- Use Linux commands to inspect CPU cores, memory, and running threads/processes.
- Write, compile, and run sequential C and Python programs for counting and searching.
- Implement parallel versions using POSIX threads (Pthreads) in C.
- Use Python's `threading` and `multiprocessing` modules to parallelise a counting task.
- Observe the effect of Python's Global Interpreter Lock (GIL) on CPU-bound workloads.
- Identify race conditions and use mutexes for correct synchronisation.
- Measure and analyse speedup, and explain deviations from ideal linear scaling.

---

2. Theoretical Background (Expanded)

2.1 The End of the Free Lunch

For decades, software performance improved automatically because CPU clock speeds increased every year (from a few MHz to ~4 GHz). This "free lunch" ended around 2005 due to physical limits: higher clock speeds caused excessive heat and power consumption. Manufacturers began putting multiple cores on a single chip instead of making a single core faster. Now, to make programs faster, we must write **parallel programs** that use multiple cores simultaneously.

2.2 Processes vs. Threads

- **Process**: An independent program running in its own memory space (code, data, heap, stack). Processes do not share memory by default; they communicate via pipes, sockets, files, etc. Creating a process is relatively heavy (copying memory tables, file descriptors).
- **Thread**: A lightweight unit of execution within a process. All threads of a process share the same memory space (code, heap, global

variables) but have their own stack and registers. Thread creation is cheaper than process creation, and communication is simple because they see the same memory. However, this shared access requires careful synchronisation to avoid race conditions.

```
Process A                 Process B
+--------------------+    +--------------------+
| Code, Data, Heap   |    | Code, Data, Heap   |
| Stack              |    | Stack              |
| Registers          |    | Registers          |
+--------------------+    +--------------------+

Process X (with 2 threads)
+------------------------------------+
| Code, Data, Heap (shared)          |
+-----------+-----------+------------+
| Thread 1  | Thread 2  | ...        |
| Stack     | Stack     |            |
| Registers | Registers |            |
+-----------+-----------+------------+
```

2.3 Concurrency vs. Parallelism

- **Concurrency**: Dealing with many things at once (structuring a program as multiple independently executing tasks). Concurrency is about composition; it can be achieved on a single-core CPU by time-slicing.
- **Parallelism**: Doing many things at once (actually executing simultaneously). Parallelism requires hardware with multiple cores.

2.4 Flynn's Taxonomy

- **SISD** (Single Instruction, Single Data): traditional sequential computer.
- **SIMD** (Single Instruction, Multiple Data): vector processors, modern CPU extensions (AVX), GPUs.
- **MISD** (Multiple Instruction, Single Data): rarely used.
- **MIMD** (Multiple Instruction, Multiple Data): most modern parallel systems – multiple cores each executing their own instructions on their own data. This is our focus.

2.5 Shared Memory vs. Distributed Memory

- **Shared Memory**: All processors/cores share a single address space. Communication is implicit via loads/stores to shared variables. Programming models: Pthreads, OpenMP, Intel TBB. Synchronisation (locks, semaphores) is required to avoid races.
- **Distributed Memory**: Each processor has its own private memory. Communication is explicit via messages over a network. Programming model: MPI (Message Passing Interface). This will be covered in later labs.

In this lab we work with shared memory within a single multicore machine.

2.6 Why Parallelism is Hard

- **Race conditions**: When multiple threads access shared data and at least one access is a write, the final result depends on the unpredictable interleaving of operations.
- **Synchronisation overhead**: Using locks (mutexes) to protect critical sections introduces serialisation and overhead.
- **Load imbalance**: If work is not evenly divided, some threads finish early and wait, reducing speedup.
- **Amdahl's Law**: Even with infinite cores, the sequential portion of a program limits speedup.

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}}$$

where P is the parallelisable fraction. Example: if 90% can be parallelised ($P = 0.9$), max speedup is $10\times$. This emphasises the need to minimise serial code.

2.7 Python's Global Interpreter Lock (GIL)

CPython's memory management is not thread-safe. To protect internal data structures, the interpreter uses a **GIL** – a mutex that allows only one thread to execute Python bytecode at a time.

- **Consequence**: For CPU-bound tasks, Python threads cannot run in parallel; they actually contend for the GIL, often making multithreaded code **slower** than sequential.
- **Solution**: Use the `multiprocessing` module, which creates separate processes, each with its own Python interpreter and GIL. Communication between processes requires pickling data (overhead) but true parallelism is achieved.
- **I/O-bound tasks**: Threads are still useful because while one thread waits for I/O (e.g., network, disk), it releases the GIL, allowing another thread to run.

---

3. Linux Command Line Tools for Observing Parallelism

You will need to monitor your system while running parallel programs. Practice these commands:

Command	Description	Example Usage
`lscpu`	Display CPU architecture information	`lscpu | grep -E "Model name|CPU(s)|Thread"`
`nproc`	Print number of processing units	`nproc`
`free -h`	Show memory usage in human-readable form	`free -h`
`top`	Interactive process viewer. Press `H` to toggle thread view.	`top -H -p <PID>`
`htop`	Enhanced top (install with `sudo apt install htop`). Press `F2` to enable tree view and show threads.	`htop`
`ps -eLf`	List all processes with lightweight info (includes threads)	`ps -eLf | grep count`

Command	Description	Example Usage
`time`	Measure execution time of a command	`time ./my_program`
`perf stat`	Detailed performance counters (if `perf` installed)	`perf stat ./my_program`
`watch`	Run a command repeatedly and display output	`watch -n 1 'ps -eLf | grep count_pthread | wc -l'`

Try this: While running a parallel program, open another terminal and watch how CPU cores are utilised.

---

4. Setting Up Your Environment

4.1 Software Requirements

- **Linux** (Ubuntu 20.04 or later recommended; WSL on Windows works too).
- **GCC** with pthread support:

  ```
  sudo apt update
  sudo apt install build-essential
  ```

- **Python 3**:

  ```
  sudo apt install python3
  ```

- **Text editor**: VS Code, vim, nano, etc.
- **Git** (optional):

  ```
  sudo apt install git
  ```

4.2 Verify Installation

```
gcc --version
python3 --version
```

4.3 Compilation Flags Explained

- `-O2`: Enables optimisations (makes code faster).
- `-pthread`: Links the pthread library and enables thread-safe compilation.
- `-o <output>`: Specifies output filename.

Example:

```
gcc -O2 -pthread -o count_pthread count_pthread.c
```

---

5. Hands-On Exercises (with Full Code and Explanations)

Exercise 1: Sequential Counting – Establish Baseline

Goal: Create a large array of random integers and count occurrences of a target value. This provides a sequential baseline for comparison.

Key Concepts:

- Random number generation.
- Looping through large data.
- Measuring CPU time with `clock()` (C) or `time.time()` (Python).

C Code – `count_seq.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define N 100000000   // 100 million elements
#define TARGET 42

int main() {
    int *arr = malloc(N * sizeof(int));
    if (!arr) { perror("malloc"); return 1; }

    // Fill array with random numbers 0-99
    srand(time(NULL));
    for (long i = 0; i < N; i++)
        arr[i] = rand() % 100;

    clock_t start = clock();

    long count = 0;
    for (long i = 0; i < N; i++)
        if (arr[i] == TARGET) count++;

    clock_t end = clock();
    double elapsed = (double)(end - start) / CLOCKS_PER_SEC;
    printf("Count = %ld\n", count);
    printf("Time = %.2f seconds\n", elapsed);

    free(arr);
    return 0;
}
```

Compile and run:

```
gcc -O2 -o count_seq count_seq.c
time ./count_seq
```

Python Code – `count_seq.py`

```
import random
import time

N = 100_000_000
TARGET = 42

arr = [random.randint(0, 99) for _ in range(N)]

start = time.time()
count = sum(1 for x in arr if x == TARGET)
end = time.time()

print(f"Count = {count}")
print(f"Time = {end - start:.2f} seconds")
```

Run:

```
time python3 count_seq.py
```

Expected output: Count ≈ 1,000,000 (since 1% of numbers are 42 on average). Execution time on a modern CPU: C ~0.3–0.5 s, Python ~5–10 s.

Questions:

- Why is C so much faster?
- Use `top` while running each. How much CPU does each use? (Should be near 100% for one core.)

Exercise 2: Parallel Counting with Pthreads (C)

Goal: Parallelise the counting task using POSIX threads and observe speedup.

Key Concepts:

- Thread creation: `pthread_create()`

- Thread joining: `pthread_join()`
- Mutex: `pthread_mutex_t` to protect shared global counter.
- Work decomposition: divide array into equal chunks.
- Race condition: without mutex, updates to global counter may interleave incorrectly.

C Code – `count_pthread.c`

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>

#define N 100000000
#define TARGET 42
#define NUM_THREADS 4

int *arr;
long global_count = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

// Structure to pass start and end indices to a thread
typedef struct {
    long start;
    long end;
} ThreadArgs;

void* count_chunk(void* arg) {
    ThreadArgs *args = (ThreadArgs*) arg;
    long local_count = 0;

    // Count in this thread's chunk
    for (long i = args->start; i < args->end; i++) {
        if (arr[i] == TARGET)
            local_count++;
    }

    // Add local count to global count (critical section)
    pthread_mutex_lock(&mutex);
    global_count += local_count;
    pthread_mutex_unlock(&mutex);

    return NULL;
}

int main() {
    arr = malloc(N * sizeof(int));
    if (!arr) { perror("malloc"); return 1; }

    // Fill array with random numbers
    srand(time(NULL));
    for (long i = 0; i < N; i++)
        arr[i] = rand() % 100;

    pthread_t threads[NUM_THREADS];
    ThreadArgs args[NUM_THREADS];
    long chunk_size = N / NUM_THREADS;

    clock_t start = clock();

    // Create threads
    for (int t = 0; t < NUM_THREADS; t++) {
        args[t].start = t * chunk_size;
        // Last thread takes the remainder to handle cases where N not divisible
        args[t].end = (t == NUM_THREADS - 1) ? N : (t + 1) * chunk_size;
        pthread_create(&threads[t], NULL, count_chunk, &args[t]);
    }

    // Wait for all threads to finish
    for (int t = 0; t < NUM_THREADS; t++) {
        pthread_join(threads[t], NULL);
    }

    clock_t end = clock();
    double elapsed = (double)(end - start) / CLOCKS_PER_SEC;

    printf("Count = %ld\n", global_count);
    printf("Time with %d threads = %.2f seconds\n", NUM_THREADS, elapsed);

    free(arr);
    pthread_mutex_destroy(&mutex);
    return 0;
}
```

Compile and run:

```
gcc -O2 -pthread -o count_pthread count_pthread.c
./count_pthread
```

Experiments:

- Change `NUM_THREADS` to 1, 2, 4, 8 and record times.
- Compute speedup = T(1) / T(N). Plot speedup vs. number of threads.
- Observe with `top -H` how many threads are created.

Why not perfect linear speedup?

- Thread creation overhead.
- Mutex contention (though minimal here because each thread locks only once).
- Memory bandwidth saturation: all threads reading from the same array may exceed the memory system’s capacity.
- Cache effects.

Exercise 3: Parallel Counting with Python Threads – GIL Demonstration

Goal: Show that Python threads do **not** speed up CPU-bound work due to the GIL.

Key Concepts:

- `threading.Thread`
- Shared global variable with `threading.Lock`
- GIL limits true parallelism.

Python Code – `count_threads.py`

```python
import random
import time
import threading

N = 100_000_000
TARGET = 42
NUM_THREADS = 4

arr = [random.randint(0, 99) for _ in range(N)]
global_count = 0
lock = threading.Lock()

def count_chunk(start, end):
    local_count = 0
    for i in range(start, end):
        if arr[i] == TARGET:
            local_count += 1
    with lock:
        global global_count
        global_count += local_count

threads = []
chunk_size = N // NUM_THREADS
start_time = time.time()

for t in range(NUM_THREADS):
    s = t * chunk_size
    e = N if t == NUM_THREADS-1 else (t+1) * chunk_size
    thread = threading.Thread(target=count_chunk, args=(s, e))
    threads.append(thread)
    thread.start()

for t in threads:
    t.join()

end_time = time.time()
print(f"Count = {global_count}")
print(f"Time with {NUM_THREADS} threads = {end_time - start_time:.2f} seconds")
```

Run:

```
python3 count_threads.py
```

Expected observation: Time will be similar to or worse than sequential Python. Use `top -H` to see that only one Python thread executes at a time.

Exercise 4: Parallel Counting with Python Multiprocessing – True Parallelism

Goal: Use separate processes to bypass the GIL and achieve real parallelism.

Key Concepts:

- `multiprocessing.Pool`
- Each process has its own memory space; data is copied (pickled).
- Inter-process communication overhead.

Python Code – `count_mp.py`

```python
import random
import time
import multiprocessing as mp

N = 100_000_000
TARGET = 42
NUM_PROCESSES = 4

# Create array in main process (will be copied to children)
arr = [random.randint(0, 99) for _ in range(N)]

def count_chunk(start, end):
    local_count = 0
    for i in range(start, end):
        if arr[i] == TARGET:
            local_count += 1
    return local_count

if __name__ == "__main__":
    chunk_size = N // NUM_PROCESSES
    pool = mp.Pool(NUM_PROCESSES)
    results = []

    start_time = time.time()
    for t in range(NUM_PROCESSES):
        s = t * chunk_size
        e = N if t == NUM_PROCESSES-1 else (t+1) * chunk_size
        results.append(pool.apply_async(count_chunk, (s, e)))

    pool.close()
    pool.join()

    total = sum(r.get() for r in results)
    end_time = time.time()
    print(f"Count = {total}")
    print(f"Time with {NUM_PROCESSES} processes = {end_time - start_time:.2f} seconds")
```

Run:

```
python3 count_mp.py
```

Expected observation: This should be faster than sequential and threaded Python, though still slower than C due to Python overhead and process creation.

Exercise 5: Parallel Search (Optional Advanced)

Problem: Find the **first index** of a target value in an array, or return -1 if not found. This requires early termination: once a thread finds the target, others should stop.

Key Concepts:

- Shared `found` flag protected by a mutex.
- Periodic checking of the flag inside the search loop.
- Condition variables could be used for efficiency, but we use a simple flag.

C Code – `search_pthread.c` (simplified version)

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define N 100000000
#define TARGET 42
#define NUM_THREADS 4

int *arr;
int found_index = -1;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void* search_chunk(void* arg) {
    long start = *(long*)arg;
    long end = start + N/NUM_THREADS;
    if (start == (NUM_THREADS-1)*N/NUM_THREADS) end = N; // last chunk

    for (long i = start; i < end; i++) {
        // Check if target already found by another thread
        pthread_mutex_lock(&lock);
        int stop = (found_index != -1);
        pthread_mutex_unlock(&lock);
        if (stop) break;

        if (arr[i] == TARGET) {
            pthread_mutex_lock(&lock);
            if (found_index == -1) found_index = i; // first thread to find sets index
            pthread_mutex_unlock(&lock);
            break;
        }
    }
    return NULL;
}
```

(Full code omitted for brevity; students can complete it.)

Python version can be written similarly using `multiprocessing.Value` as a shared flag.

Exercise 6: Monitoring and Analysing Performance

Instructions:

1. Run the C sequential and parallel versions with 1, 2, 4, 8 threads.
2. Record execution times in a table.
3. Plot speedup vs. number of threads. Add a line for ideal speedup.
4. Explain discrepancies (overhead, memory contention, Amdahl's law).

Use `perf` (if installed) to get hardware counters:

```
perf stat ./count_pthread
```

Look at `cycles`, `instructions`, `cache-misses`. More threads may increase cache misses if they compete for memory bandwidth.

---

6. Common Pitfalls and Debugging Tips

Race Conditions

- **Symptoms:** Program gives different results each run.
- **Detection:** Use `valgrind --tool=helgrind ./count_pthread` to detect data races.

Deadlocks

- Occur when two threads wait for locks held by each other.
- Avoid by consistent lock ordering and using `pthread_mutex_trylock` if needed.

Excessive Locking

- Keep critical sections as short as possible.
- Use local variables to accumulate partial results, then lock once (as we did).

False Sharing

- When threads modify different variables that happen to reside on the same cache line, the cache coherency protocol causes unnecessary traffic.
- Mitigation: Pad data structures so that each thread's data is on a separate cache line (e.g., `__attribute__((aligned(64)))` in C).

Compiler Optimisation Surprises

- Without `-O2`, the counting loop may be slower.
- With `-O3 -march=native`, the compiler might vectorise the loop automatically. Try it and see if speed improves.

Python Global Interpreter Lock

- Remember: for CPU-bound tasks, `threading` is useless. Use `multiprocessing` or rewrite performance-critical parts in C (e.g., with NumPy, Cython).

---

7. Self-Assessment Questions

1. **What is the main difference between a process and a thread?**
   Hint: Think about memory sharing and creation overhead.
2. **Why do we need a mutex in the parallel counting example?**
   Hint: What happens if two threads try to update the global count at the same time?
3. **Explain the output of `lscpu` on your machine: how many cores, threads per core?**
   Hint: Look for "CPU(s)", "Core(s) per socket", "Thread(s) per core".
4. **Run the C pthread program with 1, 2, 4, 8 threads. Plot speedup. Is it linear? Why not?**
   Hint: Consider Amdahl's law, thread creation overhead, and memory bandwidth.
5. **Why is the Python threading version slower than the sequential Python version for CPU-bound work?**
   Hint: GIL.
6. **What is false sharing and how can it affect performance?**
   Hint: Cache lines and coherency protocol.
7. **Modify the C pthread code so that each thread updates its own private counter and only adds to the global once at the end.**

Why does this improve performance?
Hint: Reduces lock contention.

8. **Write a short paragraph summarising when you would use threads vs. processes in Python.**
Hint: I/O-bound vs CPU-bound.

---

8. Deliverables

Prepare a single PDF report containing:

1. Screenshots of each program running (Exercises 1–4).
2. A table comparing execution times:
   - C sequential
   - C pthreads (for 1, 2, 4, 8 threads)
   - Python sequential
   - Python threads
   - Python multiprocessing
3. A speedup graph for the C pthreads results.
4. Answers to the self-assessment questions.
5. A brief reflection (10–15 sentences) on the challenges and insights gained from parallelising a simple counting problem.

---

9. Further Reading and Resources

- POSIX Threads Programming (LLNL)
- Python `threading` — Thread-based parallelism
- Python `multiprocessing` — Process-based parallelism
- The GIL and its effects (Real Python)
- Linux `perf` examples
- Valgrind (Helgrind) for thread error detection

Parallel programming is both an art and a science. Be patient, experiment, and use tools to understand what your code is really doing.