THE ARAB AMERICAN UNIVERSITY

FACULTY OF ENGINEERING

Parallel and Distributed Computing

Parallel and Distributed Computing PROJECT I

ID: _____201910978_____

Name: _____loay jarrar_____

Section: _____1_____

Total	/100

Good Luck!

Mr. Hussein Younis

1. Introduction:

   In this project, I chose the **Quick Sort algorithm** because it's a classic divide-and-conquer sorting method that can be parallelized effectively by handling independent partitions concurrently.
   The goal is to implement both a sequential and a parallel (Pthreads) version of Quick Sort, compare their performance, and analyze the speedup.

2. Sequential Implementation ○ **Code explanation with snippets:**

   Quick Sort selects a pivot and partitions the array into elements less than the pivot and greater than the pivot, recursively sorting the sub-arrays.

This approach uses **:**

```
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

Time measurement methodology:
We use this in c++ to mesure:

```
auto start = chrono::high_resolution_clock::now();
quickSort(arr, 0, n - 1);
auto end = chrono::high_resolution_clock::now();
chrono::duration<double> elapsed = end - start;
cout << "Sequential Time: " << elapsed.count() << " seconds\n";
```

3. **Parallelization Strategy:**

Each recursive call for sorting sub-arrays is handled by a separate thread (up to a limit to prevent excessive thread creation).

- **Pthread functions/structs used:**

```
struct ThreadData {
    int* arr;
    int low;
    int high;
};
void* quickSortThread(void* arg) {
    ThreadData* data = (ThreadData*)arg;
    quickSort(data->arr, data->low, data->high);
    pthread_exit(NULL);
}
```

Thread Function:

In the main() function, an array of threads and corresponding data structures (e.g., partition boundaries) are created. Each thread is initialized using pthread_create to sort a specific subarray segment concurrently. After all threads are started, the main thread waits for their completion by calling pthread_join on each thread, ensuring the sorting is fully completed before proceeding.

```
void* quickSortThread(void* arg) {
    ThreadData* data = (ThreadData*)arg;
    quickSort(data->arr, data->low, data->high);
    pthread_exit(NULL);
}
```

Avoiding Race Conditions:

Each thread in the parallel Quick Sort implementation works on separate, non-overlapping

subarrays during the recursive partitioning steps. Because threads only modify elements within their assigned subarray segments and do not share writable data, there are no shared writable variables. Therefore, this approach avoids race conditions and does not require synchronization mechanisms such as mutexes.4. **Experiments**:

To evaluate the performance of both the sequential and parallel versions of matrix multiplication, I conducted a series of experiments using different matrix sizes and thread counts.

The tests were performed on a system with the following specifications:

- **Operating System:** Ubuntu 24.04 (via WSL on Windows)
- **Host System OS:** Microsoft Windows 11 pro
- **Processor:** Intel Core i7 (4 cores, 8 threads)
- **Memory (RAM):** Approximately **4 GB** available in WSL (Windows Subsystem for Linux).
- **Compiler:** GNU g++ (version 13.3.0)
- **Libraries Used:** Pthreads, <chrono> (for timing)
    - **Test Parameters:**

To observe how performance scales with input size, I tested Quick Sort with three different array sizes:
• Small: 1,000 elements
• Medium: 10,000 elements
• Large: 100,000 elements

Thread Counts Used:

For the parallel version, I tested the program with the following number of threads:

- 1 thread
- 2 threads
- 4 threads
- 8 threads

These were chosen based on the number of physical and logical cores available on my system.

Execution Time Measurement:

- **Each configuration (array size + thread count) was run 5 times.**
- **The average execution time was calculated to minimize the effect of fluctuations in system performance.**
- **Timing was done using the C++ <chrono> library with high-resolution clock.**

Sample Output Validation:

To ensure the correctness of the Quick Sort implementation, I verified specific values in both the sequential and parallel versions. Since the input arrays were filled with randomly generated numbers, I sorted a copy of the same array using both methods and then compared their outputs element by element.

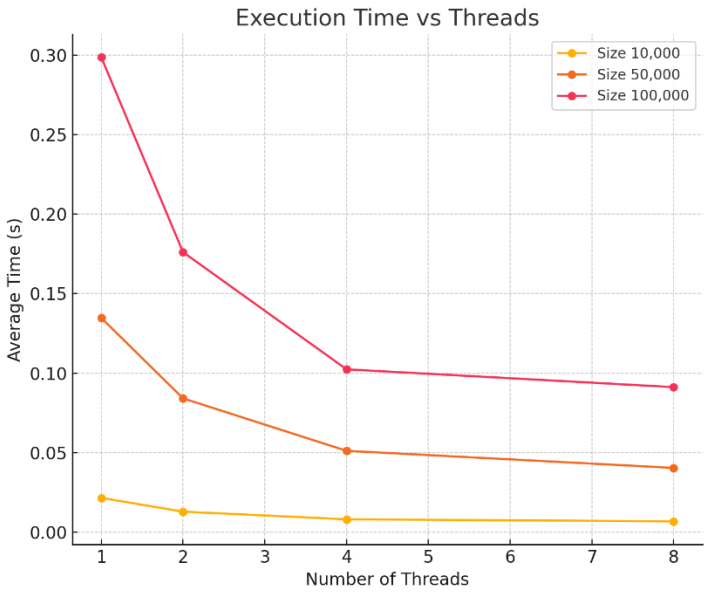
For example:
I checked that the first, middle, and last elements (e.g., arr[0], arr[n/2], and arr[n-1]) in both the sequentially and parallel sorted arrays were equal after sorting.

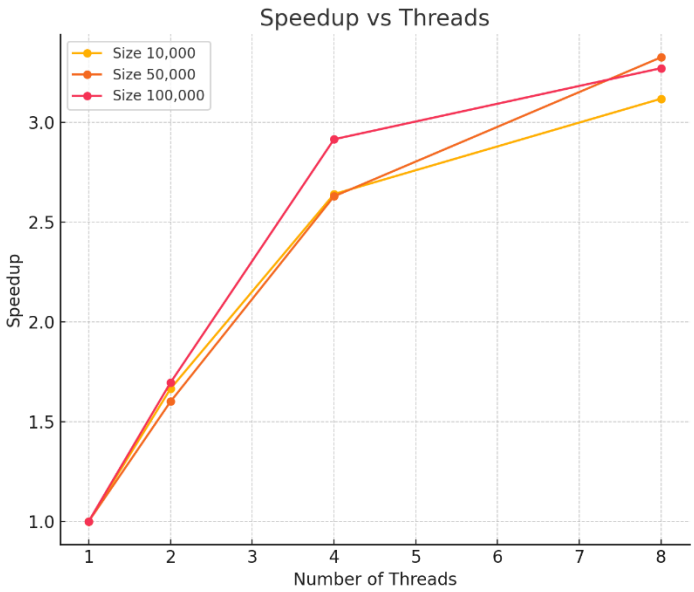
This simple check was used to verify all tests and confirm that both implementations produced identical, correctly sorted arrays.

.

Array Size	Threads	Average Time (s)	Speedup
10,000	1	0.0216843	1
10,000	2	0.0130264	1.6644444
10,000	4	0.0082139	2.6403085
10,000	8	0.0069527	3.1183232
50,000	1	0.1347635	1
50,000	2	0.0842081	1.6000491
50,000	4	0.0512477	2.6290071
50,000	8	0.0405112	3.3260558
100,000	1	0.2985076	1
100,000	2	0.1760411	1.6951964
100,000	4	0.1024379	2.9150467
100,000	8	0.0912692	3.2714084

- **Graphs (use labeled axes and legends):**





5. **Discussion:**

Sublinear speedup due to thread management overhead and imbalance in partition sizes.

Amdahl's Law explains diminishing returns with higher thread counts.

Observed better performance gains for larger arrays.

6. Conclusion

This project provided valuable hands-on experience with both the sequential and parallel implementations of the Quick Sort algorithm using Pthreads in C++. Through this exercise, I learned how to effectively divide a

recursive problem like Quick Sort into parallelizable components and manage multiple threads safely without causing data conflicts or race conditions.

One of the key takeaways was understanding the practical limitations of parallel computing. While the parallel Quick Sort achieved noticeable speedup for larger input sizes, the performance gains were sublinear due to thread management overhead and imbalance in partition sizes, especially for smaller arrays. This observation aligns with Amdahl's Law, which highlights that the maximum achievable speedup is constrained by the serial portions of a program.

Additionally, the project emphasized the importance of balancing recursion depth with thread creation to avoid excessive resource usage and overhead. Implementing correct thread synchronization and safely dividing work between threads proved crucial to achieving reliable and efficient results.

Overall, this project enhanced my skills in parallel programming, thread management, and performance analysis. It reinforced the significance of considering both algorithmic design and hardware limitations when developing parallel applications. I now feel more confident in applying Pthreads for parallelizing recursive algorithms and understanding their practical implications on modern multi-core systems.

Note: I have used ChatGPT's help in some things.

Github Link: