

**The Arab American University**

FACULTY OF ENGINEERING

Parallel and Distributed Computing

**Parallel and Distributed Computing PROJECT I**

**ID**: 202010734

**Name**: Ahmad Samer Yahya

**Section**: 1

|  |  |
| --- | --- |
| Total | /100 |

**Mr. Hussein Younis**

**Parallelizing a Sequential Algorithm Using Pthreads**

1. **Introduction**

In this project, we explore the transformation of a sequential algorithm into a multithreaded version using POSIX threads (Pthreads). The primary objective is to understand the benefits and challenges of parallel programming, particularly how multithreading can improve performance in compute-intensive tasks.

The chosen algorithm is Odd-Even Transposition Sort, also known as Brick Sort, a simple comparison-based sorting algorithm that lends itself well to parallelization. This algorithm operates in phases, alternating between comparing and swapping adjacent pairs of elements at odd and even indices. These comparisons can be performed independently within each phase, making the algorithm suitable for multithreading.

By implementing both sequential and parallel versions of the algorithm, and conducting thorough performance analysis, we aim to measure the impact of thread-level parallelism on execution time and efficiency.

1. **Sequential Implementation**
   * **Code explanation with snippets:**

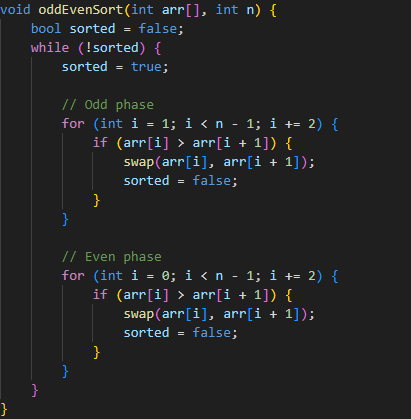
The sequential implementation of the Odd-Even Transposition Sort algorithm sorts an array by repeatedly performing two phases until the array is sorted:

**Odd phase:** Compare and swap adjacent pairs at odd indices (1,2), (3,4), (5,6), ....

**Even phase**: Compare and swap adjacent pairs at even indices (0,1), (2,3), (4,5), ....

This process continues until a full pass completes with no swaps, indicating the array is sorted.

Key snippet of the sorting loop:



The variable **sorted** tracks if any swaps occurred during each full iteration of odd and even phases. If no swaps are made, the array is sorted, and the loop exits.

* + **Time measurement methodology:**

To measure the time taken by the sequential sorting algorithm, the program uses the C++ <**chrono**> library:



1. **Record start time** before calling the sorting function:



1. **Execute the sorting function** (e.g., oddEvenSort(arr, n);).
2. **Record end time** immediately after the sort completes:



1. **Calculate duration** by subtracting start from end and converting to milliseconds:



1. **Parallelization Strategy**
   * **How work is divided among threads.**

The parallel version of the Odd-Even Transposition Sort divides the array into disjoint chunks of pairs for each thread to process during the sorting phases.

1. The algorithm consists of alternating **odd** and **even** phases:
   * **Odd phase:** threads process pairs starting at odd indices (1,2), (3,4), etc...
   * **Even phase:** threads process pairs starting at even indices (0,1), (2,3), etc...
2. Each thread t (with thread ID t from 0 to numThreads - 1) processes pairs spaced by 2 \* numThreads indices starting from its assigned offset:
   * In the **odd phase**, thread t handles pairs starting at index 1 + 2 \* t
   * In the **even phase**, thread t handles pairs starting at index 0 + 2 \* t

This division ensures no two threads access the same pair simultaneously, preventing data races.

All threads perform their assigned swaps in parallel, then synchronize at barriers before moving to the next phase.

* + **Pthread functions/structs used.**

**pthread\_create**: Creates multiple threads running the sorting routine concurrently. Each thread receives a unique ID to determine its portion of the array to work on.

**pthread\_join:** Used in the main thread to wait for all created threads to finish execution before proceeding.

**pthread\_barrier\_t** and **pthread\_barrier\_init** / **pthread\_barrier\_wait** / **pthread\_barrier\_destroy:**

A barrier is used to synchronize threads at key points:

* After the odd phase: all threads wait at the barrier before starting the even phase.
* After the even phase: all threads wait again before the next iteration or termination check.

This ensures all threads complete their current phase before any thread proceeds, maintaining data consistency.

**Shared Data Structures:**

* An integer array a representing the dataset to sort, shared among all threads.
* A shared bool globalSorted flag to signal when the array is fully sorted.
* A shared boolean array localSortedArr where each thread reports whether it performed any swaps in the current iteration.

**Thread Arguments Structure (struct ThreadArgs):**

Encapsulates thread-specific data, mainly the thread ID, which helps each thread know which elements to process.

1. **Experiments**
   * **Hardware specs:**
2. **CPU**: Intel Core i5-9300H.
3. **Number of Cores**: 4 physical cores.
4. **RAM**: 16GB.
5. **Operating System:** Windows 11 Pro 64-bit.
   * **Input sizes and thread counts tested:**

To evaluate the performance of the Odd-Even Transposition Sort algorithm, I tested it on various input sizes and thread configurations, within the limits of my hardware.

**Input Sizes Tested:**

* 1,000 elements
* 5,000 elements
* 10,000 elements
* 20,000 elements

Input sizes larger than 20,000 were avoided due to the high time complexity of the algorithm (**O(n²)**), which made execution times excessively long for both sequential and parallel versions.

**Thread Counts Tested (Parallel Version):**

* 1 thread (acts as the baseline)
* 2 threads
* 3 threads
* 4 threads (maximum based on available physical cores)

Since the machine has a quad-core CPU, testing was limited to a maximum of 4 threads to avoid thread contention and scheduling overhead.

1. **Results**
   * **Tables of execution times and speedup:**

* **Execution Time:**

|  |  |  |
| --- | --- | --- |
| Input Size | Threads | Execution Time (ms) |
| 1,000 | 1 | 2 |
| 1,000 | 2 | 51 |
| 1,000 | 3 | 94 |
| 1,000 | 4 | 91 |
| 5,000 | 1 | 47 |
| 5,000 | 2 | 277 |
| 5,000 | 3 | 410 |
| 5,000 | 4 | 433 |
| 10,000 | 1 | 110 |
| 10,000 | 2 | 692 |
| 10,000 | 3 | 1079 |
| 10,000 | 4 | 1009 |
| 20,000 | 1 | 363 |
| 20,000 | 2 | 2224 |
| 20,000 | 3 | 2930 |
| 20,000 | 4 | 3025 |

Table 1: Execution Times

* **Speedup:**

**Speedup=**

|  |  |  |  |
| --- | --- | --- | --- |
| ****Input Size**** | ****Threads**** | ****Execution Time (ms)**** | ****Speedup**** |
| 1,000 | 1 | 2 | 1.00 |
|  | 2 | 51 | 0.039 |
|  | 3 | 94 | 0.021 |
|  | 4 | 91 | 0.022 |
| 5,000 | 1 | 47 | 1.00 |
|  | 2 | 277 | 0.17 |
|  | 3 | 410 | 0.11 |
|  | 4 | 433 | 0.11 |
| 10,000 | 1 | 110 | 1.00 |
|  | 2 | 692 | 0.16 |
|  | 3 | 1079 | 0.10 |
|  | 4 | 1009 | 0.11 |
| 20,000 | 1 | 363 | 1.00 |
|  | 2 | 2224 | 0.16 |
|  | 3 | 2930 | 0.12 |
|  | 4 | 3025 | 0.12 |

Table 2: Speed up

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

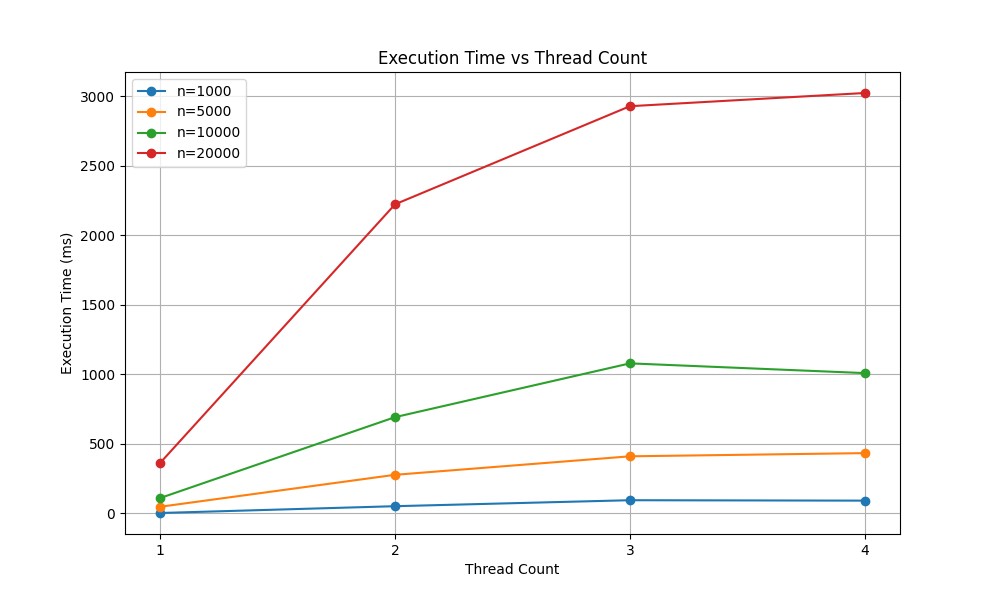
****

Figure 1

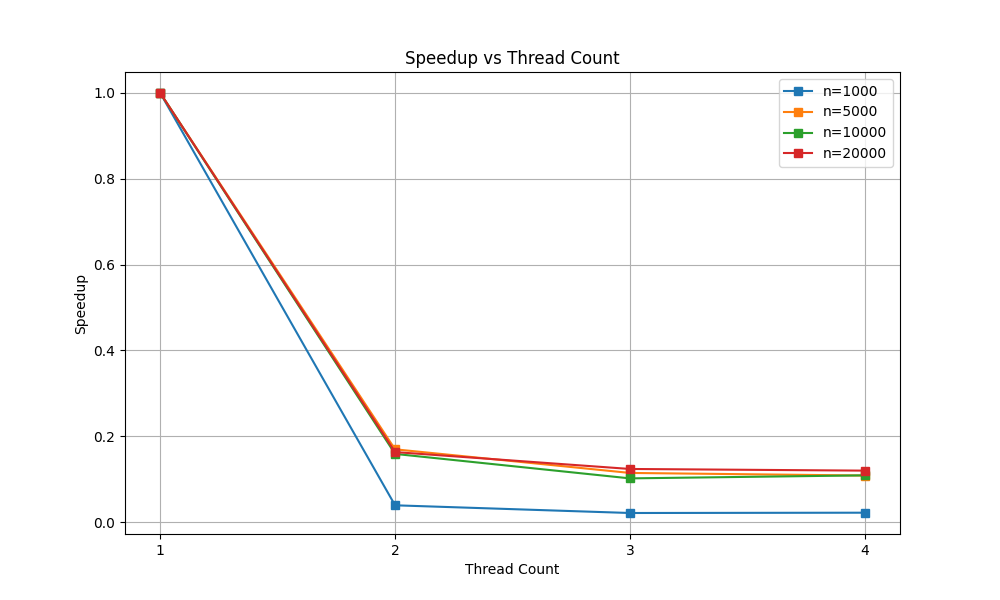
****

Figure 2

1. **Discussion**
   * **Why speedup is sublinear (e.g., overhead, load imbalance).**

Despite using multiple threads, the parallel version of the Odd-Even Transposition Sort showed sublinear speedup, and in most cases, it was slower than the sequential version. This behavior can be attributed to several key factors:

1. **Thread Creation and Synchronization Overhead**
   * Each iteration of the sort requires synchronization across threads using barriers (**pthread\_barrier\_wait**).
   * These synchronizations are frequent and expensive, especially with large arrays and multiple threads.
2. **Poor Scalability of the Algorithm**
   * Odd-Even Transposition Sort is a comparison-based, **O(n²)** algorithm.
   * It requires many iterations even for nearly sorted arrays, and it doesn’t parallelize efficiently because:
     + Each thread often waits on others.
     + Many comparisons/swaps are dependent on previous results (data dependency).
3. **Load Imbalance**
   * The array may not divide evenly among threads, leading to some threads finishing early and sitting idle.
   * For example, in a 4-thread run, thread 0 might handle fewer elements if the array size isn’t divisible by 4.
4. **Cache and Memory Contention**
   * Multiple threads accessing/modifying adjacent memory (array elements) may cause false sharing, degrading performance due to cache coherence traffic.
5. **Small Problem Size**
   * For smaller inputs (like 1,000–20,000), the overhead of creating and synchronizing threads dominates the actual work, making parallelism inefficient.
   * **Comparison to Amdahl’s Law predictions.**

Amdahl’s Law models the theoretical speedup of a program as:

**Speedup** =

**Where:**

* P: Proportion of code that is parallelizable.
* N: Number of threads (processors).

Given the heavy synchronization and sequential dependencies in Odd-Even Sort, the **parallel portion PPP** is relatively **small**. Even with 4 threads, Amdahl’s Law predicts only modest speedup when PPP is low.

For example, assuming only 60% of the sort is parallelizable:

**Speedupmax =**

In reality, your **measured speedup was < 1** due to the additional overheads not accounted for in Amdahl’s ideal model.

1. **Conclusion**

This project explored the parallel implementation of the Odd-Even Transposition Sort (also known as Brick Sort) using POSIX threads. Although the algorithm is theoretically parallelizable, our experimental results showed that the **parallel version performed worse than the sequential version** across all tested input sizes.

* **Lessons Learned:**
  + **Not all algorithms benefit from parallelism**: Especially simple O(n²) algorithms with heavy inter-element dependency, like Odd-Even Sort.
  + **Threading introduces significant overhead**, particularly for small and medium input sizes where thread management and synchronization cost outweigh any potential gain from parallelism.
  + **Understanding the hardware (number of cores) is crucial**: Assigning more threads than available cores doesn't improve performance and can make it worse.
  + **Measuring performance correctly** and analyzing results with tools like speedup graphs and Amdahl's Law gives insight into the real behavior of a parallel program.
* **Challenges Faced:**
* **Infinite loops** due to incorrect termination conditions (like global flags not being properly synchronized).
* **False expectations from parallelism** — assuming that using threads will always be faster without considering overhead.
* **Data sharing and synchronization** between threads were tricky and required careful handling using barriers and shared flags.

**Tool and Resources:**

* **POSIX Threads (pthreads):** C library used for parallel programming by creating and managing threads.
* **VS Code (Visual Studio Code)**: Code editor used to write, edit, and debug the C++ code.
* **Python (with matplotlib)**: Used to generate performance graphs (Execution Time vs. Thread Count, Speedup vs. Thread Count).
* **ChatGPT by OpenAI**: Used for help with code debugging, performance analysis, explanation writing, and report structuring.