Project Report

Name: Adeel Yousfi(2212242)

Arman Amin(2212342)

Huzaifa Jeeva(2212257)

Akshay Kumar(2212244)

Ibrahim Nabi(22122253)

Course: PDC

Submitted to: Dr Samar Yazdani

Project Title

Performance Analysis of Sequential vs Parallel Quicksort Using Multiprocessing

Course: Parallel & Distributed Computing
Project Type: Experimental Study
Language: Python

Table of Contents

1. Abstract

This project focuses on the design, implementation, and performance evaluation of a sequential and parallel version of the Quicksort algorithm. The parallel version is implemented using Python's multiprocessing module to exploit multi-core CPUs. Performance is measured in terms of runtime, speedup, and efficiency across different input sizes. Experimental results show that parallel Quicksort significantly reduces execution time for large datasets, although scalability is limited by overhead and recursion depth constraints.

2. Introduction

Sorting is a fundamental problem in computer science with applications in databases, operating systems, and scientific computing. As data sizes grow, traditional sequential algorithms become performance bottlenecks. Parallel computing provides a way to improve performance by dividing work across multiple processors. Quicksort is a divide-and-conquer algorithm that naturally lends itself to parallelism, making it a suitable candidate for performance comparison between sequential and parallel approaches.

3. Problem Statement

The objective of this project is to analyze and compare the performance of sequential Quicksort with a parallel Quicksort implementation. The goal is to evaluate how parallelism affects runtime, speedup, and efficiency when sorting large datasets on a multi-core system.

4. Sequential Design (Baseline Algorithm)

The baseline algorithm is a standard recursive Quicksort implementation enhanced with two optimizations:

- **Median-of-three pivot selection** to reduce the likelihood of worst-case performance.
- **Insertion sort fallback** for small subarrays to reduce recursion overhead.

This sequential version serves as the reference (T_1) for evaluating parallel performance.

5. Parallel Strategy

The parallel version of Quicksort uses Python's multiprocessing Pool to sort subarrays concurrently. Parallelism is applied only when:

- The input size exceeds a predefined threshold.
- The recursion depth is below a maximum parallel depth.

Left and right partitions are processed in parallel using worker processes. This strategy reduces total execution time by utilizing multiple CPU cores while avoiding excessive process creation overhead.

6. Implementation Details

Key implementation parameters include:

- **Insertion Sort Threshold (32):** Below this size, insertion sort is used.
- **Parallel Size Threshold (50,000):** Parallel execution starts only for sufficiently large arrays.
- **Maximum Parallel Depth (3):** Limits recursive parallelism to control overhead.
- **Correctness Check:** Sequential and parallel outputs are compared to ensure identical sorted results.

The global process pool is reused across recursive calls to reduce process initialization cost.

7. Experimental Setup

- **Hardware:** Multi-core CPU system
- **Processes:** Equal to the number of available CPU cores
- **Input Sizes:** 100,000; 500,000; 1,000,000; 2,000,000 elements
- **Trials:** 5 runs per input size
- **Data:** Randomly generated integers

Performance metrics are averaged across trials to reduce noise.

8. Results & Analysis

The experimental results demonstrate:

- **Runtime Reduction:** Parallel Quicksort outperforms sequential Quicksort for large input sizes.
- **Speedup:** Speedup increases with input size but remains sub-linear due to overhead.
- **Efficiency:** Efficiency decreases as process count increases, indicating diminishing returns.

Graphs for runtime, speedup, and efficiency visually confirm that parallel execution is beneficial primarily for large datasets.

9. Challenges & Limitations

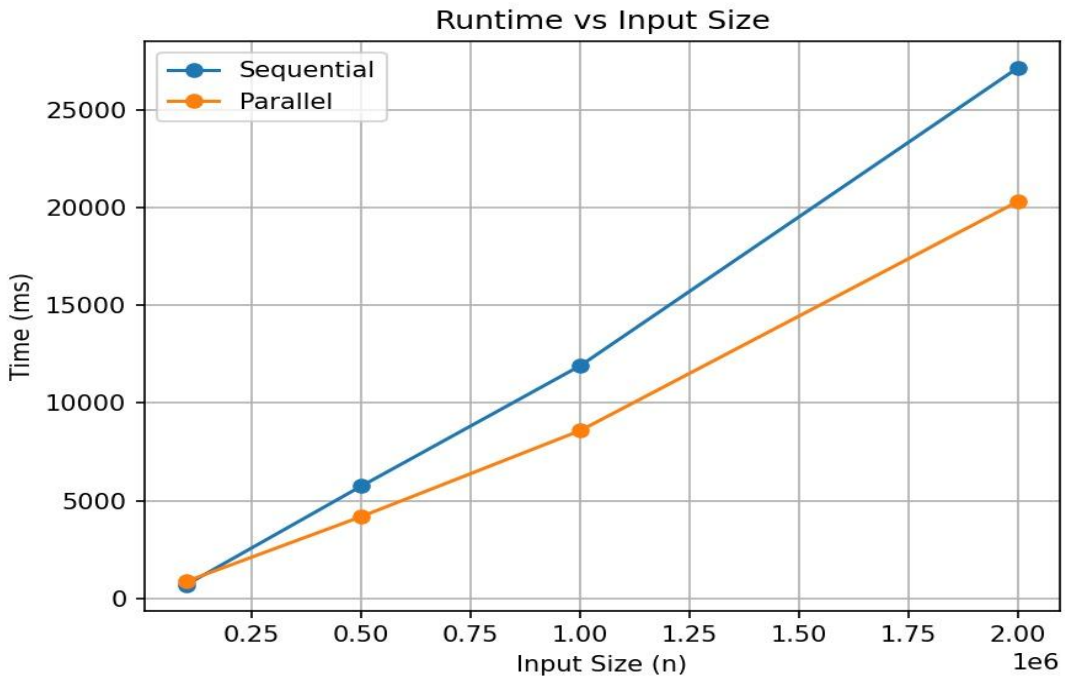
- **Process Overhead:** Multiprocessing introduces communication and synchronization costs.
- **Limited Scalability:** Python's multiprocessing and memory copying reduce scalability.
- **Depth Limitation:** Restricting parallel depth prevents full exploitation of available cores.
- **Memory Usage:** Creating sublists increases memory overhead.

10.Conclusion

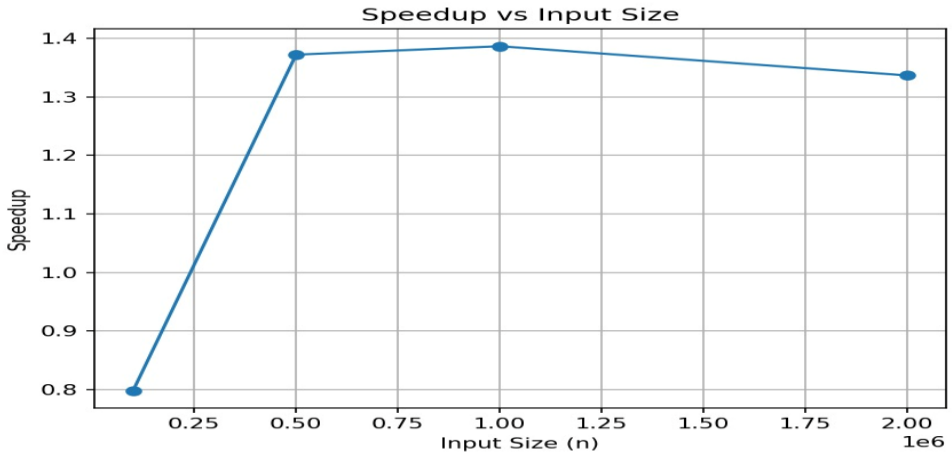
This project successfully demonstrates the advantages and limitations of parallelizing Quicksort using multiprocessing in Python. While parallel execution significantly improves performance for large inputs, overhead and scalability constraints limit efficiency. The study highlights that careful threshold selection and controlled parallelism are essential for achieving optimal performance in parallel algorithms.

11.Graphs

1. Runtime vs input size



2.speedup vs input size



3.Efficiency vs input size