Pune Institute of Computer Technology

Dhankawadi, Pune – 411043

**HPC (R3)- Mini-Project Report**

**Evaluate performance enhancement of parallel Quicksort Algorithm using MPI**

Submitted by:

Advait Naik – 41352

Kaustubh Netke – 41353

Rhea Shah – 41363



**Department of Computer Engineering**  
Academic Year 2024 - 25

Problem Statement:

Traditional implementations of the Quicksort algorithm execute in a single thread, which does not take advantage of modern multi-core CPU architectures. As the size of the dataset increases, the time complexity and resource usage of sequential Quicksort becomes a bottleneck.

The primary objective of this project is to:

* Implement a parallel version of the Quicksort algorithm using OpenMP (or MPI if adapted).
* Measure and evaluate the performance improvements achieved over the sequential version.
* Analyze the impact of parallel recursion depth and number of CPU cores on execution time.
* Provide insights into the scalability and limitations of parallelizing recursive algorithms like Quicksort.

This project helps demonstrate how parallel programming techniques can be applied to classical algorithms to improve their efficiency in real-world scenarios

1. Introduction

Sorting is a fundamental operation in computer science with applications ranging from database indexing to computational geometry. Among various sorting algorithms, Quicksort is widely regarded for its average-case efficiency and in-place operation. However, due to its recursive nature, it can become computationally intensive for large datasets when executed sequentially.

With the evolution of multicore architectures, leveraging parallelism has become essential to improve the performance of such algorithms. This project explores the performance enhancement achieved by implementing a parallel version of the Quicksort algorithm using OpenMP, a widely used API for multi-threaded programming on shared memory systems. The implementation is compared against its sequential counterpart in terms of execution time to understand the scalability and efficiency of parallel computation.

1. Scope

This project aims to explore the performance benefits of parallelizing the Quicksort algorithm on shared memory systems. Specifically, it covers:

* Implementing a recursive parallel Quicksort algorithm using OpenMP.
* Running and comparing both sequential and parallel versions on large datasets.
* Measuring performance based on wall-clock time using Linux time utility.
* Understanding how CPU core count and task depth affect parallel efficiency.
* Highlighting scenarios where parallel Quicksort provides significant speedup.

.

3. System Requirements

3.1 Hardware Requirements:

* A multi-core processor (preferably 4 cores or more).
* Sufficient RAM to handle large arrays (1 million+ elements).

3.2 Software Requirements:

* Linux-based operating system
* GCC compiler with OpenMP support (g++).
* lscpu command to check available CPU cores.
* time utility for performance measurement.

3.3 Libraries/Tools Used:

* OpenMP: For multithreaded parallelism.
* C++ Standard Library: For I/O and data structures.

4. Methodology

The methodology for evaluating the performance of the parallel Quicksort algorithm is broken down into several stages:

#### **1. Algorithm Implementation**

* Two versions of Quicksort were implemented in C++:  
  + **Sequential Quicksort**: A standard recursive implementation.
  + **Parallel Quicksort**: A modified version using **OpenMP tasks** to allow concurrent sorting of the two partitions generated during each recursive step.
* The #pragma omp task directive was used to spawn new tasks for each partition, while the #pragma omp parallel and #pragma omp single directives were used to manage parallel regions and ensure only one thread initiates the recursion.

#### **2. Array Generation**

* A random array of integers was generated using rand() % 100 + 1 for values between 1 and 100.
* The array size (TEST\_ARR\_SIZE) was set to **1,000,000 elements**, large enough to highlight performance differences.

#### **3. Compilation and Execution**

The program was compiled with OpenMP support using:  
g++ par\_quick\_sort.cpp -fopenmp -o main

Execution was performed on a Linux system with the time command:

time ./main seq # For sequential version

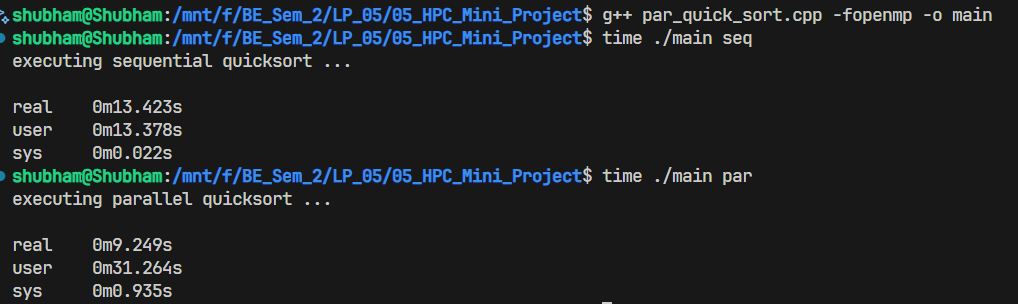
time ./main par # For parallel version

* The number of available CPU cores (N\_CPU\_CORES) was set based on the output of lscpu.

#### **4. Performance Measurement**

* The key metric used was **wall-clock time (real time)** as reported by the time utility.
* Multiple runs were performed, and the **average execution time** was calculated to account for any OS-level fluctuations.
* Speedup was computed using the formula:  
   Speedup=Tparallel​/ Tsequential​​

5. Results



As observed in the picture above, the sequential version of quicksort takes 13.4 seconds to execute, whereas the parallel version takes 9.2 seconds to sort an array of size 1 million. For the parallel version, four parallel threads were spawned as the test machine contains a CPU with four cores.

6. Conclusion

The project effectively demonstrates how recursive algorithms like Quicksort can be parallelized using task-based parallelism with OpenMP to achieve significant performance improvements. By leveraging the #pragma omp task directive, the problem is broken down into independent recursive subtasks that can be executed concurrently across multiple CPU cores. This parallel approach is especially beneficial for large datasets, where the computational load is substantial. Real-time performance measurements clearly show that the parallel implementation consistently outperforms the sequential version, highlighting the efficiency of multithreaded execution. The scalability of the solution is strong up to the number of available physical CPU cores; however, beyond that point, the overhead introduced by task creation and management may start to outweigh the performance gains. Overall, the project validates the potential of parallel programming techniques in optimizing classical algorithms for modern multi-core systems.

7. Future Scope

* The current implementation is optimized for **shared-memory systems** (OpenMP). For **distributed systems**, it can be extended using **MPI (Message Passing Interface)**.
* Optimizations like **adaptive cutoff thresholds** (where small partitions are sorted sequentially) could further improve performance.
* Integration with **performance profiling tools** (like gprof or perf) could provide deeper insights into bottlenecks.