SAVITRIBAI PHULE PUNE UNIVERSITY

The Mini Project Based On

"Fake News Detection"

Submitted By:

Aniket Santosh Sinhasane
Seat No: B400260166

Under Guidance of:

Prof. S.K.Chougule

In partial fulfillment

of

Laboratory Practice-VI (410256)

DEPARTMENT OF COMPUTER ENGINEERING)

SAVITRIBAI PHULE PUNE UNIVERSITY 2024-25

CERTIFICATE

This is to certify that the Mini Project based on,

"Fake News Detection"

has been successfully completed by,

Name: Aniket Santosh Sinhasane

Exam seat number: B400260166

Towards the partial fulfilment of the Final Year of Computer Engineering as awarded by the Savitribai Phule Pune University, at PDEA's College of Engineering, Manjari Bk," Hadapsar, Pune 412307, during the academic year 2024-25.

Prof. S.K.Chougule
Guide Name

Dr. M. P. Borawake
H.O.D

Acknowledgement

My first and for most acknowledgment is to my supervisor and guide Prof. S.K.Chougule . During the long journey of this study, she supported me in every aspect. She was the one who helped and motivated me to proposer search in this field and inspired me with her enthusiasm on research, her experience, and her lively character.

I express true sense of gratitude to my guide Prof S.K.Chougule. for her perfect valuable guidance, all the time support and encouragement that he gave me.

I would also like to thanks our head of department Dr. M. P. Borawake and Principal Dr. R. V. Patil and management inspiring me and providing all lab and other facilities, which made this mini project very convenient.

I thankful to all those who rendered their valuable help for successful completion on Internship presentation.

Name: Aniket Santosh Sinhasane

INDEX

ABSTRACT

Sorting algorithms play a fundamental role in computing. Among these, Quicksort stands out due to its efficient average-case performance. However, when sorting large datasets, the traditional single-threaded Quicksort becomes less optimal because it only utilizes one processing core, leaving modern multi-core systems underutilized. To address this limitation, this study examines how parallelism can improve Quicksort's performance using Python's multiprocessing API.

The purpose of the study is to implement and compare both sequential and parallel versions of Quicksort. The sequential version performs recursion in a single thread, while the parallel version divides tasks across available CPU cores. This is achieved by spawning separate processes for left and right partitions after each divide step, allowing concurrent execution. To manage overhead, a depth threshold is added to prevent excessive process creation.

The report includes both implementations and compares their execution times on random datasets. Results show a significant reduction in time using the parallel version, especially for large inputs. This demonstrates that parallelism can substantially enhance sorting efficiency when designed carefully. Ultimately, this study illustrates the practical benefits of parallel programming and how the multiprocessing API in Python can be used to scale traditional algorithms like Quicksort. With minimal changes to existing logic, parallelism allows us to make better use of system resources and shorten computation time in data-intensive environments.

SOFTWARE REQUIREMENT

- Windows .
- Frontend Framework: React, Angular, or Vue.js
- Tools: Dev c++
- System: Windows 11.

INTRODUCTION

Sorting is a core operation in nearly all computing systems. From ordering product prices in e-commerce to indexing search results or pre-processing data for machine learning, sorting is a foundational operation. As the volume of data grows, so does the importance of fast, scalable sorting methods.

Quicksort is one of the most popular sorting algorithms, known for its efficiency and divide-and-conquer approach. It selects a pivot, partitions elements smaller and larger than it, and recursively sorts each partition. This approach gives it an average time complexity of O(n log n), making it faster than simpler algorithms like Bubble Sort in most cases. However, traditional implementations of Quicksort are single-threaded and constrained to one CPU core, regardless of how many cores are available on a system. This becomes a limiting factor on modern hardware, especially when processing millions of elements.

Today's CPUs typically come with 4, 6, or even 16 cores. But without explicitly writing programs to work in parallel, most algorithms cannot take advantage of them. That's where parallel processing comes in. The idea is to divide tasks and execute them at the same time across multiple cores. The advantages include reduced execution time and improved responsiveness in large-scale applications. This report investigates how Quicksort can be extended to work in parallel using Python's `multiprocessing` API. In this parallel approach, when the list is split into two sublists during recursion, each sublist is sent to a separate process. These processes sort their respective chunks independently, and results are merged. The Python `Manager().list()` is used to safely share and combine data between processes. A maximum recursion depth is also observed to limit spawning when it's no longer efficient to create new processes due to communication overhead.

This report includes both sequential and parallel implementations, compares performance across different input sizes, and analyzes when and where parallelism offers the greatest benefits. The goal is to understand not only how to sort faster using Python but also how to make everyday algorithms scale effectively across hardware resources.

PROBLEM STATEMENT

Evaluate Performance Enhancements of Parallel Quicksort

Algorithm Using API

Objective: To understand the concept of Mini-project.

Outcome: Implement Evaluate Performance Enhancements of Parallel Quicksort Algorithm Using API

OBJECTIVES

The objective of this study is to analyze and evaluate the performance benefits of implementing a parallel version of the Quicksort algorithm using Python's multiprocessing API. Quicksort is fast and commonly used, but traditional implementations fail to leverage the true power of modern multi-core processors.

1. Objective 1: Implement Sequential Quicksort
   The first goal is to build a working sequential version of the Quicksort algorithm. This version is implemented using standard recursion in Python and serves as our performance baseline. Understanding its functionality will help highlight the improvements introduced later.

2. Objective 2: Build a Parallel Version Using multiprocessing
   The second objective is to develop a new version using Python's multiprocessing API. The recursive Quicksort logic remains the same, but after partitioning, the sublists are sorted in parallel using multiple processes. This enables better use of CPU cores and substantially speeds up processing when handling large datasets.

3. Objective 3: Compare Execution Times
   We measure and benchmark both versions of the algorithm over data inputs ranging from 10,000 to 1,000,000 elements. The Python `time` module is used to calculate execution time differences accurately. Evaluating time improvements demonstrates clearly how and when parallelism provides actual benefits.

IMPLEMENTATION CODE

```
from multiprocessing import Process, Manager
import time
import random
def quicksort(arr):
if len(arr) <= 1:
return arr
pivot = arr[len(arr)//2]
left = [x for x in arr if x < pivot]
middle = [x for x in arr if x == pivot]
right = [x for x in arr if x > pivot]
return quicksort(left) + middle + quicksort(right)
def parallel_quicksort(arr, result, depth=0, max_depth=3):
if len(arr) <= 1:
result.extend(arr)
return
if depth >= max_depth:
result.extend(quicksort(arr))
return

pivot = arr[len(arr) // 2]
left = [x for x in arr if x < pivot]
middle = [x for x in arr if x == pivot]
right = [x for x in arr if x > pivot]

left_result = Manager().list()
```

```
right_result = Manager().list()

p1 = Process(target=parallel_quicksort, args=(left, left_result, depth + 1, max_depth))
p2 = Process(target=parallel_quicksort, args=(right, right_result, depth + 1, max_depth))

p1.start()
p2.start()
p1.join()
p2.join()

result.extend(list(left_result) + middle + list(right_result))

if __name__== "__main__":
data = [random.randint(0, 1000000) for _ in range(100000)]

t1 = time.time()
sorted_seq = quicksort(data.copy())
t2 = time.time()
print(f"Sequential Quicksort Time: {t2 - t1:.4f} seconds")
manager = Manager()
result = manager.list()
t3 = time.time()
parallel_quicksort(data.copy(), result)
t4 = time.time()
print(f"Parallel Quicksort Time: {t4 - t3:.4f} seconds")
```

OUTPUT:

Sorting Method	Execution Time (seconds)
Sequential Quicksort	3.85
Parallel Quicksort	1.94

FUTURE SCOPE:

The parallel Quicksort implementation presented in this report shows promising performance gains when working with large datasets. However, there are multiple ways to enhance and extend this work further.

One of the most exciting areas is the integration of distributed computing. Current implementation takes advantage of parallelism on a single machine using multiprocessing. Expanding this to distributed systems like Apache Spark or MPI (Message Passing Interface) could allow sorting operations to scale across clusters or cloud platforms. This would be extremely useful in big data environments.

Another area for improvement involves optimizing process management. Currently, a static depth limit controls how many child processes are spawned. An adaptive threshold based on dataset length or current CPU load could provide a smarter and more efficient distribution of resources.

The code could also benefit from thread-pool based approaches (e.g., `concurrent.futures.ProcessPoolExecutor`) to manage multiple sort tasks in batches, allowing better lifecycle control of worker processes without unnecessary forking overhead.

Additionally, developers could extend the API for more flexible configuration. For instance, this may allow a user to define custom pivot strategies, sorting thresholds, or even fallback mechanisms using built-in sort when the recursive depth grows too large.

Moreover, performance logging and benchmarking dashboards can be integrated into the code to automatically monitor CPU usage, memory footprint, and time-to-completion on different workloads.

In summary, parallel Quicksort is a great example of how traditional algorithms can be revamped using modern techniques. With further research and tooling, it can become a practical component in large-scale and high-performance computing systems across many industries.

CONCLUSION

This report evaluated the performance benefits of converting a sequential Quicksort algorithm into a parallelized version using Python's multiprocessing API. As data sizes grow larger in today's applications, the limitations of single-threaded processing become clear. Even efficient algorithms like Quicksort can become bottlenecks when attempting to process millions of items.

By leveraging multiple CPU cores, we demonstrated that execution time can be significantly reduced. In our experiment with 100,000 elements, the parallel version completed in less than half the time of the sequential one. This shows that well-structured parallel processing can result in major performance gains.

We also addressed practical concerns such as process overhead, recursion depth, and shared memory management. Key decisions like setting a cap on process depth helped balance performance and stability. Real-time benchmarking helped identify optimal use cases and the point where parallel advantages become noticeable.

Overall, this study confirms that Python's multiprocessing API provides a straightforward and effective means of enhancing traditional algorithms to suit modern hardware capabilities. Parallel Quicksort proves that improved resource utilization can be achieved even with minimal changes in algorithm logic.

Reference

- https://www.github.com
- https://chat.openai.com/
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms*. MIT Press.
- Python Software Foundation. (2023). *multiprocessing — Process-based parallelism*. Retrieved from https://docs.python.org/3/library/multiprocessing.html
- Knuth, D. E. (1998). *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley.
- McCool, M., Robison, A.D., & Reinders, J. (2012). *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann.