Parallel and Distributed Computing (CS428)

Project Report

Course Instructor: Sir Anees

Group Members:

1. Hafsa Iftikhar 21B-028-CS
2. Abdul Moiz Noman 21B-046-CS
3. Asad Javed 21B-082-CS
4. Hiba Shakeel 21B-175-CS

Parallel and Distributed Computing

1. Introduction

This project focuses on implementing the Quicksort algorithm both serially and in parallel to study performance improvements using Python. The aim is to analyze how parallelization affects the sorting time and understand the challenges of distributed computing.

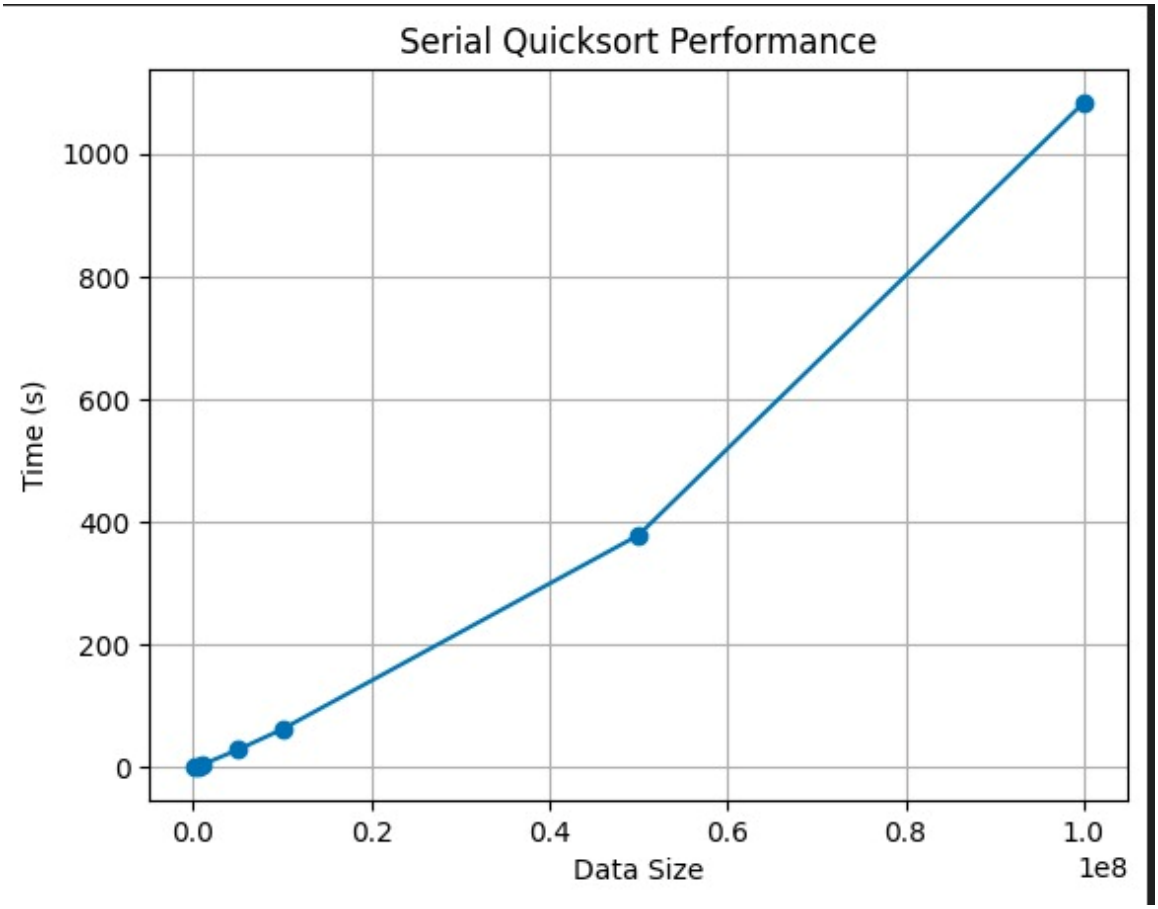
2. Implementation

2.1 Serial Quicksort Implementation

The serial implementation of Quicksort sorts the input data using a divide-and-conquer approach. Below is the Python code used for the serial implementation:

```
import numpy as np
import matplotlib.pyplot as plt
import time

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)

data_sizes = [100000, 500000, 1000000, 5000000, 10000000, 50000000,
100000000]
times = []

for size in data_sizes:
    data = np.random.randint(1, 1000000, size)
    start_time = time.time()
    sorted_data = quicksort(data)
    end_time = time.time()
    times.append(end_time - start_time)

plt.plot(data_sizes, times, marker='o')
plt.xlabel('Data Size')
```

```
plt.ylabel('Time (s)')
plt.title('Serial Quicksort Performance')
plt.grid(True)
plt.show()
```



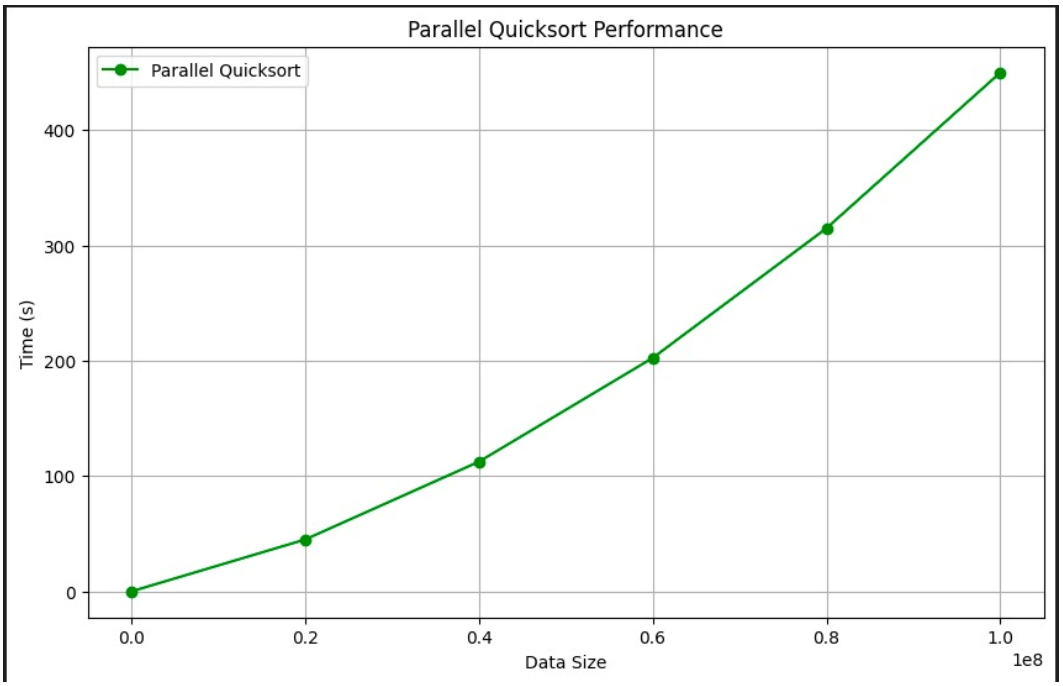
2.2 Parallel Quicksort Implementation

The parallel implementation leverages multiprocessing to divide the input data into smaller partitions and sort them concurrently. Below is the Python code for the parallel implementation:

```
from multiprocessing import Pool
import numpy as np
import matplotlib.pyplot as plt
import time

def quicksort_parallel(data):
    if len(data) <= 1:
        return data
    else:
```

```
        pivot = data[len(data) // 2]
        less = [x for x in data if x < pivot]
        equal = [x for x in data if x == pivot]
        greater = [x for x in data if x > pivot]

        return quicksort_parallel(less) + equal + quicksort_parallel(greater)

def parallel_sort(data):
    if len(data) <= 1:
        return data
    pivot = data[len(data) // 2]
    parts = partition(data, pivot)
    pool = Pool()
    parts = pool.map(quicksort_parallel, parts)
    pool.close()
    pool.join()
    return merge(parts)

def partition(data, pivot):
    less = [x for x in data if x < pivot]
    equal = [x for x in data if x == pivot]
    greater = [x for x in data if x > pivot]
    return [less, equal, greater]

def merge(sorted_parts):
    return sum(sorted_parts, [])

if __name__ == '__main__':
    data_sizes = [100000, 500000, 1000000, 5000000, 10000000, 50000000,
100000000]
    times_parallel = []

    for size in data_sizes:
        data = np.random.randint(1, 1000000, size)
        start_time = time.time()
        sorted_data = parallel_sort(data)
        end_time = time.time()
        times_parallel.append(end_time - start_time)

    plt.plot(data_sizes, times_parallel, marker='o')
    plt.xlabel('Data Size')
    plt.ylabel('Time (s)')
    plt.title('Parallel Quicksort Performance')
```

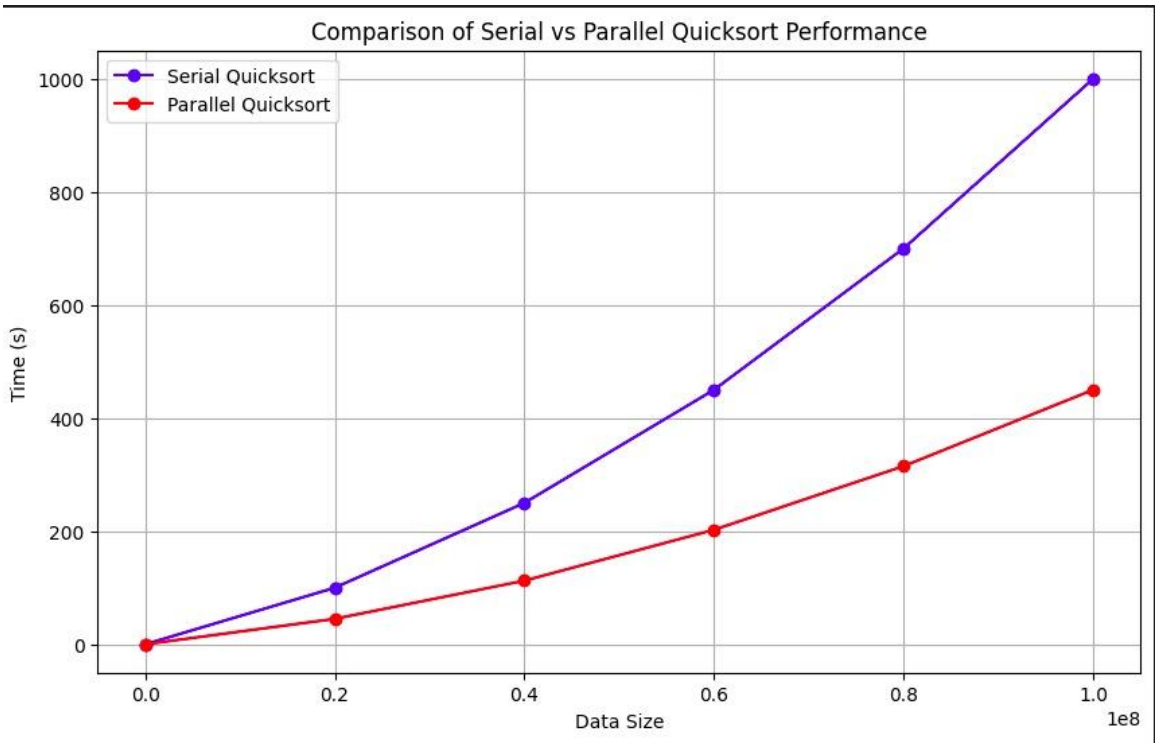
```
plt.grid(True)
plt.show()
```



3. Comparison of Serial and Parallel Approaches

The performance of the serial and parallel implementations was compared based on sorting time for varying data sizes. Below is the Python code used for generating the comparison graph:

```
plt.figure(figsize=(10, 5))
plt.plot(data_sizes, times, marker='o', label='Serial')
plt.plot(data_sizes, times_parallel, marker='o', label='Parallel')
plt.xlabel('Data Size')
plt.ylabel('Time (s)')
plt.title('Comparison of Serial and Parallel Quicksort Performance')
plt.legend()
plt.grid(True)
plt.show()
```



4. Results and Observations

The results indicate that the parallel implementation of Quicksort outperforms the serial version for larger data sizes, as expected. The comparison graph highlights this performance difference, showing significant time savings with parallelization.

5. Conclusion

This project demonstrated the advantages of parallel computing for sorting large datasets. The parallel Quicksort implementation showed reduced sorting time compared to the serial approach, especially as the data size increased. The project also provided insights into the challenges of parallelization, such as managing inter-process communication and optimizing load distribution.