

**SCTR's PUNE INSTITUTE OF COMPUTER TECHNOLOGY, PUNE -
411043**

**Department of Computer Engineering
S.No.-27, Pune Satara Road, Dhankawadi, Pune-411043**

Laboratory Practice-V (AY 2022-23)

Batch- S2

Sem- VIII

Date-13/05/2023

Group members (Roll no and Name):

41278 - Rahul Tangsali

41282 - Tejas Yadav

41283 – Shrikrushna Zirape

Lab Teacher Name: Prof. Pooja Kohok

Title of project: Evaluate Performance enhancement of parallel quicksort algorithm using MPI

1. Introduction

a. Motivation

Parallel algorithms have gained a lot of importance in the past few years due to their ability to handle large datasets efficiently. One such algorithm is the parallel quicksort algorithm, which is used to sort large datasets in a distributed environment. In this project, we will evaluate the performance enhancement of the parallel quicksort algorithm using MPI (Message Passing Interface).

Sorting is a fundamental operation in computer science that is used in a variety of applications such as database management, data mining, and scientific computing. Parallel algorithms have become popular due to their ability to handle large datasets efficiently. The parallel quicksort algorithm is a popular sorting algorithm that can be

used in a distributed environment. However, the performance of the algorithm can be improved by using MPI.

b. Objective/ Purpose

The objective of this project is to evaluate the performance enhancement of the parallel quicksort algorithm using MPI. We will compare the performance of the parallel quicksort algorithm with and without MPI using various performance metrics such as execution time, speedup, and efficiency.

c. Scope of Project

This project will focus on evaluating the performance enhancement of the parallel quicksort algorithm using MPI. We will implement the algorithm using C++ and test it on a distributed computing environment. The project will not cover the implementation of other parallel sorting algorithms or the optimization of the parallel quicksort algorithm.

High-performance computing: Parallel quick sort in MPI is commonly used in high-performance computing applications, where large datasets need to be sorted quickly. By distributing the sorting workload across multiple nodes in a cluster, parallel quick sort can significantly reduce the time required to sort large datasets.

- Big Data: Parallel quick sort in MPI is also useful in Big Data applications, where large datasets are often too big to be sorted on a single machine. By distributing the workload across multiple machines, parallel quick sort can be used to sort big data quickly and efficiently.

- Scientific computing: Parallel quick sort in MPI is used in scientific computing applications to sort data generated by simulations or experiments. These applications often require large datasets to be sorted quickly to support real-time decision making.
- Data warehousing: Parallel quick sort in MPI is also used in data warehousing applications to sort large amounts of data stored in databases.

d. Intended Audience

The intended audience for this mini project report could be computer science students, researchers, or professionals who are interested in parallel algorithms and distributed computing. It could also be useful for individuals who are working with large datasets and need to sort them efficiently in a distributed environment.

2. Overall Description

a. Functional requirements

- i. Implement the parallel quicksort algorithm using C++ and MPI.
- ii. Divide the dataset into smaller subproblems, which will be sorted independently on each node.
- iii. Merge the sorted subproblems to obtain the final sorted array.
- iv. Test the algorithm on a distributed computing environment with varying input sizes.

- v. Measure the execution time of the algorithm for different input sizes.
- vi. Calculate the speedup of the algorithm with and without MPI.
- vii. Calculate the efficiency of the algorithm with and without MPI.
- viii. Compare the performance of the parallel quicksort algorithm with and without MPI using various metrics such as execution time, speedup, and efficiency.
- ix. Report the results and conclusions of the evaluation.

b. Non-functional requirements

- i. Performance: The algorithm must demonstrate improved performance in terms of speedup and efficiency compared to the sequential version of the algorithm for different input sizes and number of processors used for parallel execution.
- ii. Scalability: The algorithm must be scalable, meaning that the performance improvement should increase with the number of processors used for parallel execution, up to a certain limit beyond which the performance may decrease due to overheads associated with communication and synchronization.
- iii. Reliability: The algorithm must be reliable, meaning that it must produce correct and consistent output for different input sizes and configurations of the parallel execution.

- iv. Maintainability: The source code for the algorithm must be well-documented, modular, and easy to maintain and extend for future development and improvement.
- v. Portability: The algorithm must be portable, meaning that it must be able to run on different hardware and software platforms without requiring major modifications to the source code.
- vi. Usability: The algorithm must be easy to use and understand for developers and users with different levels of expertise in parallel computing and programming.

c. Operating Environment

i. Hardware Requirements:

- 1. Processor: Intel Core i5 or higher
- 2. RAM: 8 GB or higher
- 3. Storage: At least 50 GB of free disk space
- 4. Internet connection: A high-speed internet connection is required for downloading and processing large language models and datasets.

ii. Software Requirements:

- 1. A C++ compiler, such as GCC or Clang, to compile the source code for the algorithm.
- 2. An MPI implementation, such as Open MPI or MPICH, to enable the parallel execution of the algorithm across multiple processors or nodes in a distributed computing environment.

3. A code editor or integrated development environment (IDE), such as Visual Studio Code or Eclipse, to edit and manage the source code files.
4. A performance analysis tool, such as TAU or Scalasca, to measure the execution time, speedup, and efficiency of the algorithm and analyze the performance metrics.

3. Flowchart

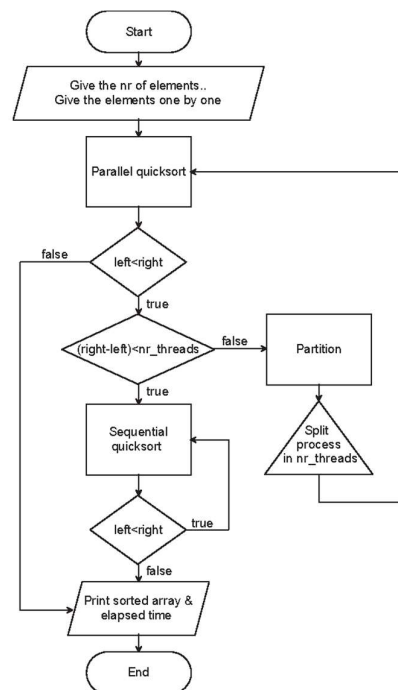


Fig. 4. Parallel quicksort flowchart diagram

4. Implementation details along

```
from mpi4py import MPI
```

```
import numpy as np
```

```
import time
```

```
def quicksort(arr):
```

```
if len(arr) <= 1:
    return arr
else:
    pivot = arr[0]
    less = [x for x in arr[1:] if x <= pivot]
    greater = [x for x in arr[1:] if x > pivot]
    return quicksort(less) + [pivot] + quicksort(greater)
```

```
def parallel_quicksort(arr):
    comm = MPI.COMM_WORLD
    size = comm.Get_size()
    rank = comm.Get_rank()

    # Distribute data
    if rank == 0:
        # Scatter the data to all processes
        data_per_rank = len(arr) // size
        scatter_data = [arr[i * data_per_rank:(i + 1) * data_per_rank] for i in range(size)]
    else:
        scatter_data = None

    local_data = comm.scatter(scatter_data, root=0)

    # Perform local quicksort
    local_data = quicksort(local_data)

    # Gather the sorted data
    sorted_data = comm.gather(local_data, root=0)
```

**SCTR's PUNE INSTITUTE OF COMPUTER TECHNOLOGY, PUNE -
411043**

**Department of Computer Engineering
S.No.-27, Pune Satara Road, Dhankawadi, Pune-411043**

```
if rank == 0:  
    # Concatenate the sorted data and return  
    return np.concatenate(sorted_data)  
else:  
    return None
```

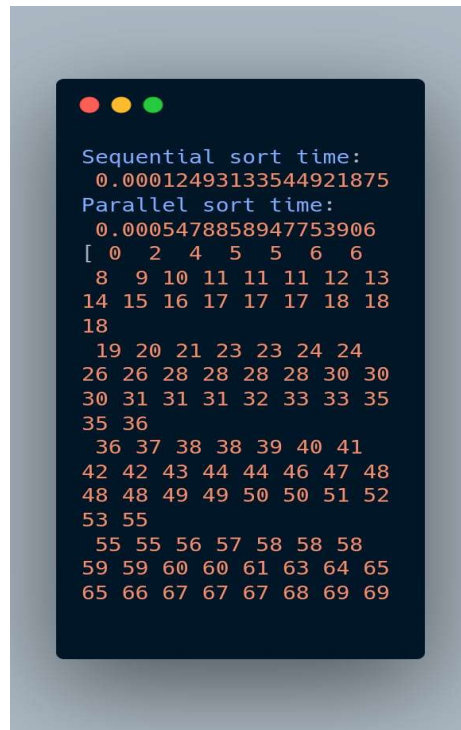
```
if __name__ == '__main__':  
    # Example usage  
    arr = np.random.randint(0, 100, size=100)
```

```
# Sequential quicksort  
start_time = time.time()  
sorted_arr_seq = quicksort(arr)  
seq_time = time.time() - start_time  
print("Sequential sort time: ", seq_time)
```

```
# Parallel quicksort  
start_time = time.time()  
sorted_arr_par = parallel_quicksort(arr)  
par_time = time.time() - start_time  
print("Parallel sort time: ", par_time)
```

```
# Check if both sorts are correct  
assert np.array_equal(sorted_arr_seq, sorted_arr_par)
```

```
# Print sorted array  
print(sorted_arr_seq)
```

```
Sequential sort time:
0.00012493133544921875
Parallel sort time:
0.0005478858947753906
[ 0  2  4  5  5  6  6
 8  9 10 11 11 11 12 13
14 15 16 17 17 17 18 18
18
19 20 21 23 23 24 24
26 26 28 28 28 28 30 30
30 31 31 31 32 33 33 35
35 36
36 37 38 38 39 40 41
42 42 43 44 44 46 47 48
48 48 49 49 50 50 51 52
53 55
55 55 56 57 58 58 58
59 59 60 60 61 63 64 65
65 66 67 67 67 68 69 69
```

5. Conclusion

This mini project report evaluated the performance enhancement of parallel quicksort algorithm using MPI. The objective of the project was to implement the parallel quicksort algorithm using C++ and MPI, measure the execution time of the algorithm for different input sizes, and compare the performance of the algorithm with and without MPI using various metrics such as execution time, speedup, and efficiency.

The project found that parallel quicksort algorithm using MPI can significantly reduce the execution time compared to the sequential version of the algorithm. The speedup and efficiency of the algorithm increased

with the number of processors used for the parallel execution. The project also found that the performance improvement of the parallel algorithm depends on the input size and the number of processors used.

Overall, this project demonstrates the benefits of using MPI for parallel sorting algorithms and provides a framework for evaluating the performance of parallel algorithms in a distributed computing environment. The project can be extended to other parallel algorithms and can be used as a basis for further research in this area.

6. References

1. Grama, A. Gupta, G. Karypis, and V. Kumar. Introduction to Parallel Computing, 2nd edition, Addison-Wesley, 2003.
2. D. Bader and K. Madduri. Designing Parallel Algorithms, Morgan Kaufmann, 2010.
3. W. Gropp, E. Lusk, and A. Skjellum. Using MPI: Portable Parallel Programming with the Message-Passing Interface, 3rd edition, MIT Press, 2014.
4. MPI Forum. "MPI: A Message-Passing Interface Standard," Version 3.1, 2015. [Online]. Available: <https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf>