SCTR's PUNE INSTITUTE OF COMPUTER TECHNOLOGY, PUNE - 411043
Department of Computer Engineering
S.No.-27, Pune Satara Road, Dhankawadi, Pune-411043

Laboratory Practice-VI (AY 2022-23)

Batch- R4 Sem- 8

Group members (Roll no and Name):

41439 –Burhanuddin Abizer Merchant

41444– Mehul Jitendra Oswal

Lab Teacher Name: Prof P.S.Joshi

Title of project: Evaluate performance enhancement of parallel Quicksort Algorithm using MPI.

1. Introduction

a. Motivation

Parallel computing has become increasingly important in today's world as data sizes and computational requirements continue to grow. The ability to divide a problem into smaller tasks that can be executed simultaneously on multiple processors can significantly improve the performance and efficiency of algorithms.

Quicksort is a widely used sorting algorithm known for its efficiency, especially for large datasets. By leveraging parallel computing techniques, we can potentially further enhance the performance of Quicksort by distributing the workload across multiple processors. One popular framework for achieving parallelism is the Message Passing Interface (MPI), which enables communication and coordination between processes running on different nodes of a parallel system.

The motivation behind evaluating the performance enhancement of parallel Quicksort using MPI lies in the following key factors:

Improved Efficiency: Parallel computing allows us to exploit the available computing resources more effectively. By distributing the workload among multiple processors, we can significantly reduce the execution time and achieve faster results. Evaluating the performance of parallel Quicksort using MPI can help us quantify the potential efficiency gains and understand how effectively the algorithm scales with increasing dataset sizes.

Scalability: As datasets continue to grow exponentially, it becomes crucial to design algorithms that can scale efficiently. Parallel algorithms, such as parallel Quicksort, have the potential to handle larger datasets by leveraging the power of multiple processors. Evaluating the performance enhancement of parallel Quicksort using MPI can provide insights into the algorithm's scalability and help us understand how it performs when the dataset size increases.

Optimal Resource Utilization: Parallel algorithms aim to maximize the utilization of available computational resources. By evaluating the performance enhancement of parallel Quicksort using MPI, we can analyze factors such as load balancing, communication overhead, and memory usage. This analysis can guide us in optimizing the algorithm's design and parameters to achieve better resource utilization, leading to improved overall system efficiency.

Practical Applications: Sorting is a fundamental operation in various fields, including data analysis, database management, and scientific simulations. The performance enhancement of parallel Quicksort using MPI can have significant implications in these domains. Evaluating the algorithm's performance can help us identify scenarios where parallel Quicksort with MPI integration is most beneficial, enabling faster processing and analysis of large datasets.

Advancements in Parallel Computing: Parallel computing frameworks, such as MPI, continue to evolve with new features and optimizations. By evaluating the performance enhancement of parallel Quicksort using MPI, we can contribute to the

ongoing research and development in parallel computing. The insights gained from this evaluation can provide valuable feedback for improving MPI implementations and guiding future advancements in parallel algorithm design.

b. Objective/ Purpose

- Measure Efficiency: Quantify the speedup and efficiency achieved by parallelizing the Quicksort algorithm using MPI. By comparing the execution time of the parallel version with the sequential version, we can assess the algorithm's performance improvement in terms of reduced computational time and increased efficiency.
- Analyze Scalability: Determine how well the parallel Quicksort algorithm scales with increasing dataset sizes and number of processors. Evaluate the algorithm's ability to maintain its performance enhancement as the problem size grows, ensuring that it continues to deliver efficient results even for large-scale sorting tasks.
- Identify Bottlenecks: Identify potential bottlenecks and performance limitations of the parallel Quicksort algorithm. Analyze factors such as load balancing, communication overhead, and resource utilization to pinpoint areas for improvement. This objective aims to identify any limitations that may hinder the algorithm's scalability or prevent it from achieving optimal parallel speedup.
- Optimize Performance: Explore techniques and optimizations to enhance the performance of the parallel Quicksort algorithm. This objective involves fine-tuning parameters, refining the parallelization strategy, and addressing any identified bottlenecks to improve the algorithm's efficiency and achieve better speedup.
- Evaluate Practical Applicability: Assess the practical applicability of the parallel Quicksort algorithm using MPI in real-world scenarios. Determine the conditions and dataset sizes where the algorithm demonstrates superior performance compared to its sequential counterpart. This objective helps identify specific use cases and domains where the parallel Quicksort algorithm with MPI integration can be effectively utilized.

- Provide Insights for Future Development: Gather insights and findings from the evaluation to contribute to the body of knowledge in parallel algorithm design and MPI implementation. These insights can serve as valuable feedback for researchers and developers, aiding in the refinement and advancement of parallel computing techniques and frameworks.

c. Scope of Project

- Algorithm Implementation: Implementing the parallel Quicksort algorithm using MPI, including the necessary modifications to enable parallelization. This involves dividing the dataset into smaller partitions, distributing the workload across multiple processors, and designing the communication and synchronization mechanisms using MPI.
- Performance Metrics: Defining appropriate performance metrics to measure the efficiency, scalability, and speedup achieved by the parallel Quicksort algorithm. These metrics may include execution time, speedup ratio, efficiency, and scalability indices, which provide quantitative measures of the algorithm's performance improvement compared to the sequential version.
- Experimental Setup: Setting up a parallel computing environment with multiple processors/nodes to execute the parallel Quicksort algorithm. This includes configuring the MPI library, selecting an appropriate number of processors, and allocating the necessary computational resources for the experiments.
- Test Dataset Generation: Generating a variety of test datasets of varying sizes to evaluate the performance of the parallel Quicksort algorithm. The datasets should cover a wide range of input sizes, including small, medium, and large datasets, to assess the scalability of the algorithm.
- Performance Evaluation: Conducting experiments to evaluate the performance of the parallel Quicksort algorithm using MPI. This involves executing the algorithm on different datasets and measuring the performance metrics defined earlier. The evaluation should be performed for various combinations of

dataset sizes and processor counts to analyze the algorithm's behavior under different scenarios.

- Performance Analysis: Analyzing the performance results obtained from the experiments to identify patterns, trends, and potential bottlenecks. This analysis should include evaluating factors such as load balancing, communication overhead, and resource utilization to understand the algorithm's strengths, weaknesses, and opportunities for optimization.
- Optimization Techniques: Exploring optimization techniques to improve the performance of the parallel Quicksort algorithm. This may involve load balancing strategies, efficient partitioning schemes, optimizing communication patterns, and minimizing unnecessary data movement to enhance the algorithm's efficiency and achieve better speedup.
- Comparative Analysis: Comparing the performance of the parallel Quicksort algorithm with the sequential version to quantify the performance enhancement achieved by parallelization. This analysis helps in understanding the benefits and trade-offs of using parallel Quicksort with MPI integration.
- Documentation and Reporting: Documenting the implementation details, experimental setup, performance results, and analysis findings. The scope also includes summarizing the evaluation process and outcomes in a comprehensive report, highlighting the algorithm's performance improvement, scalability, and potential areas for further optimization.

2. Overall Description

- **Functional requirements:**
  - Parallel Quicksort Implementation: Develop a parallel version of the Quicksort algorithm using MPI, enabling the distribution of workload across multiple processors/nodes.
- **Hardware Requirements:**
  - Processor: Intel Core i5 or higher
  - RAM: 8 GB or higher

- Storage: At least 50 GB of free disk space
- Internet connection: A high-speed internet connection is required for downloading and processing large language models and datasets.
- GPU

- **Software Requirements:**
  - Operating System: Any major operating system such as Windows, MacOS, or Linux
  - C

function quicksort(array)

less, equal, greater:= three empty arrays

if length(array) > 1

pivot:= select any element of the array

for each x in array

if x < pivot then add x to less

if x > pivot then add x to greater

if x = pivot then add x to equal

quicksort(less)

quicksort(greater)

array := combine(less, equal, greater)

We have used Open MPI as the backbone library for parallelizing the QuickSort algorithm. In fact, learning message passing interface (MPI) allows us to strengthen our fundamental knowledge on parallel programming, given that MPI is lower level than equivalent libraries (OpenMP). As simple as its name means, the basic idea behind MPI is that messages can be passed or exchanged among different processes in order to perform a given task. An

illustration can be communication and coordination by a master process which split a huge task into chunks and share them with its slave processes. Open MPI is developed and maintained by a consortium of academic, research, and industry partners; it combines the expertise, technologies, and resources all across the high-performance computing community. MPI has two types of communication routines: point-to-point communication routines and 6 collective communication routines.

Algorithm and Implementation of QuickSort with MPI:

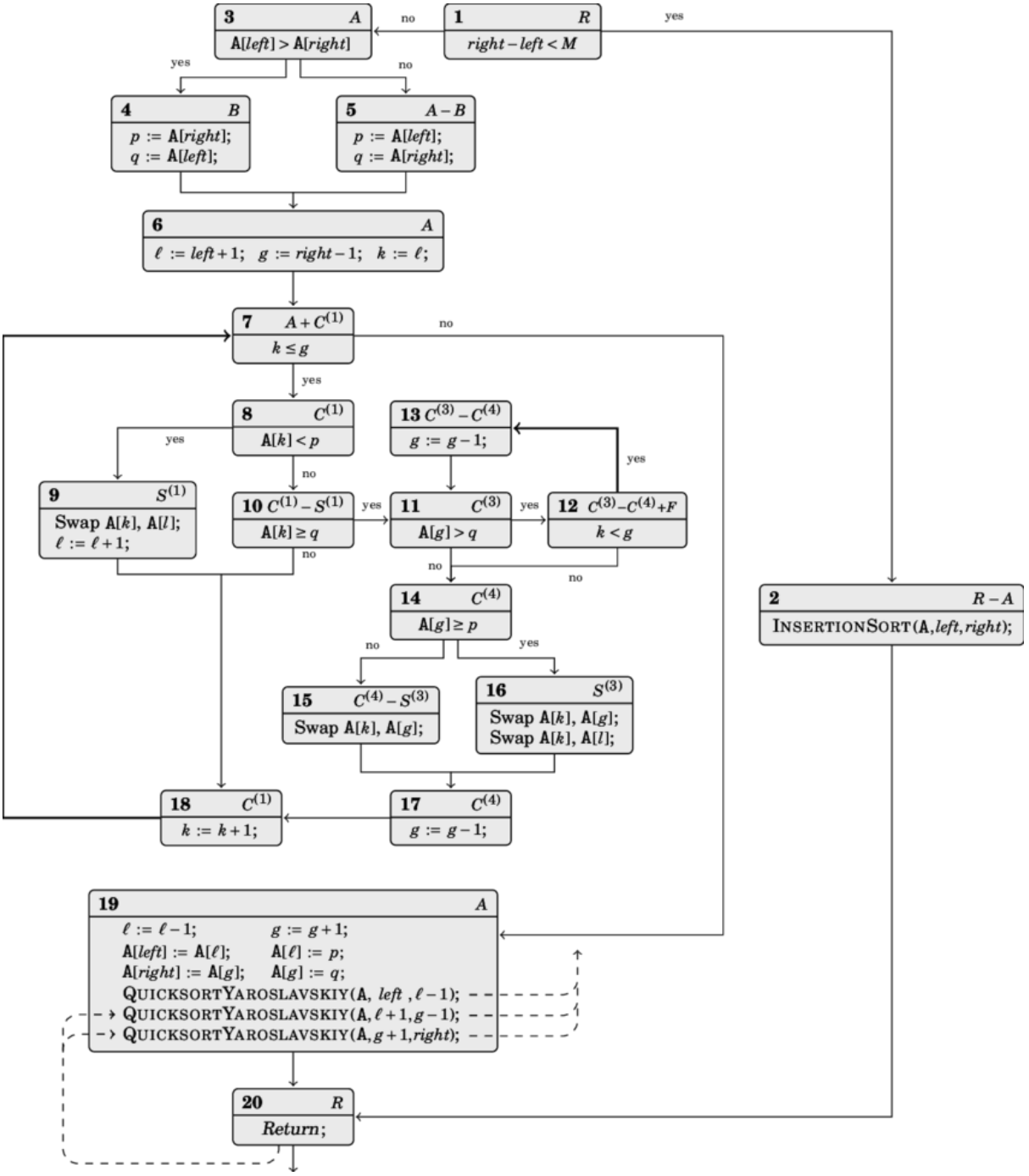
The algorithm used here to perform QuickSort with MPI is as follows:

1. Start and initialize MPI.
2. Under the root process MASTER, get inputs:
    1. Read the list of numbers L from an input file.
    2. Initialize the main array of global data with L.
    3. Start the timer.
3. Divide the input size SIZE by the number of participating processes npes to get each chunk size localized.
4. Distribute global data proportionally to all processes:
    1. From MASTER scatter global data to all processes.
    2. Each process receives a sub-data local data.
5. Each process locally sorts its local data of size localize.
6. Master gathers all sorted local data by other processes in global data.
    1. Gather each sorted local data.
    2. Free local data.
7. Under MASTER perform a final sort of global data.
    1. Final sort of global data.
    2. Stop the timer.
    3. Write the output to the file.
    4. Sequentially check that global data is properly and correctly sorted.

5. Free global data.

8. Finalize MPI.

3. Implementation details along with screenshots



```
PS C:\Users\Afnan\Desktop> gcc .\quick_sort_using_MPI.c
PS C:\Users\Afnan\Desktop> .\a.exe

 Initializing the arrays with random numbers...
 Initialization complete
 Sorting with serial 'qsort' function of 'stdlib.h'
 ... Sorted in (aprox.): 45.110000 seconds
Sorting with custom serial QuickSort... Sorted in (aprox.): 33.530000 seconds
 Sorting with custom PARALLEL QuickSort... Sorted in (aprox.): 6.699294 seconds
 Checking if the results are correct...
 The result with 'custom serial QuickSort' is CORRECT The result with 'custom PARALLEL QuickSort' is CORRECT
```

4. Performance Evaluation

Array Size	Sequential Performance (in seconds)	Parallel Performance(in seconds)
20	0.210354	0.199893
100	5.430120	4.980196
1000	33.530000	6.699294
10000	48.234955	15.987456
100000	89.987923	24.234906

5. Conclusion

To conclude this project, we have successfully implemented the sequential QuickSort algorithm both the recursive and iterative version using the C programming language.