ABSTRACT

This mini project delves into the realm of parallel computing by implementing and evaluating the performance of a parallel Quicksort algorithm using the Message Passing Interface (MPI) in C++. The project aims to demonstrate the efficiency enhancements achievable through parallelization compared to the traditional serial Quicksort algorithm. By leveraging the divide-and-conquer strategy inherent in Quicksort and distributing computational tasks across multiple processors, we showcase significant reductions in sorting time for large datasets. This project contributes to the understanding of parallel computing paradigms and their practical applications in optimizing algorithmic performance.

Table of contents

Sr. No.	Chapter Name	Page No.
1	Introduction	1
1.1	Problem Statement	2
2	Requirements	3
3	Implementation	4
3.1	Backend Implementation	4
3.2	Frontend Implementation	5
3.3	Code	6
4	Conclusion	27

INTRODUCTION

In the realm of computational algorithms, sorting stands as a fundamental operation with applications spanning various domains, from data analysis to scientific computing. Among the plethora of sorting algorithms, Quicksort stands out for its efficiency and elegance. Its divide- and-conquer approach, coupled with a pivot-based partitioning strategy, yields impressive average-case time complexity of O(NlogN). However, the worst-case time complexity of O(N^2)poses a challenge for scenarios where the input data is nearly sorted or exhibits specific distributions.

To address the scalability and performance challenges posed by large datasets, parallel computing offers a promising solution. By harnessing the computational power of multiple processors, parallel algorithms aim to expedite the execution of tasks that can be divided into independent subproblems. In this context, the Message Passing Interface (MPI) emerges as a widely-used framework for developing parallel applications, enabling communication and coordination among distributed processes.

This mini project focuses on exploring the performance enhancement achievable through parallelization of the Quicksort algorithm using MPI. Parallel Quicksort entails dividing the dataset into smaller chunks, sorting them independently on different processors, and subsequently merging the sorted chunks to obtain the final sorted array. By parallelizing the sorting process, we aim to exploit the concurrency offered by modern computing architectures and alleviate the computational burden associated with sorting large datasets.

PROBLEM DEFINITION

Specifically, the project aims to develop a parallel Quicksort algorithm in C++ using MPI, distribute sorting tasks across multiple processes, and synchronize results to obtain a sorted array. By benchmarking runtime performance, analyzing speedup and efficiency, and experimenting with varying dataset sizes and distributions, the project seeks to demonstrate the scalability and efficiency gains of parallel Quicksort for sorting large datasets.

REQUIREMENTS

2.1. Software Requirement Specifications

Operating System Front End Back End Server Documentation: Windows 10
Frontend Software:HTML, CSS, JavaScript;
Backend Software: Python: Install the latest version of Python, preferably Python 3.x, Flask: Flask is a web framework for Python. Install Flask using the Python package manager.

2.2. Hardware Requirement Specifications

Processor: A modern multi-core processor (e.g., Intel Core i5 or equivalent) should be adequate.

RAM: At least 8 GB of RAM is recommended. However, if you're working with large datasets or complex models, consider having more RAM.

Storage: Sufficient storage space is required to store your datasets, libraries, and code. A few gigabytes of free space should be enough.

Operating System: You can develop the project on Windows, macOS, or Linux based on your preference. Ensure you have the necessary administrative privileges to install software.

Computer Processor Core i3 Processor Speed 2.3 GHz Processor Hard Disk 400 GB or more RAM Min 2 GB

IMPLEMENTATION

The implementation of the parallel Quicksort algorithm using MPI involves several key steps, including initialization of MPI, division of the dataset among processes, local sorting of data chunks, and gathering of sorted sub-arrays. Below are the implementation details:

Initialization of MPI:
Import the mpi4py library.
Initialize MPI using MPI.COMM_WORLD.
Obtain the rank and size of the current process and the total number of processes.

Serial Quicksort Algorithm:
Define the serial version of the Quicksort algorithm, which recursively partitions the array into smaller sub-arrays based on a pivot element and sorts them.

Parallel Quicksort Algorithm:
Define the parallel version of the Quicksort algorithm.
Divide the dataset into chunks and distribute them among processes using MPI scatter.
Each process locally sorts its assigned chunk of data.
Gather the sorted chunks back to the root process using MPI gather.

Dataset Generation and Timing:
Generate a large dataset of numbers for testing purposes.
Time the execution of both the serial and parallel Quicksort algorithms using Python's time module.

Performance Comparison:
Compare the execution times of the serial and parallel versions of the Quicksort algorithm.
Calculate the speedup and efficiency achieved by the parallel implementation compared to the serial implementation.
Experiment with varying dataset sizes and distributions to analyze scalability and performance under different conditions.

```
function quicksort(array)
    less, equal, greater := three empty arrays
    if length(array) > 1
        pivot := select any element of array
        for each x in array
            if x < pivot then add x to less
            if x = pivot then add x to equal
            if x > pivot then add x to greater
        quicksort(less)
        quicksort(greater)
        array := concatenate(less, equal, greater)
```

Algorithm

In general, the overall algorithm used here to perform QuickSort with MPI works as followed:

i. Start and initialize MPI.
ii. Under the root process MASTER, get inputs:
a. Read the list of numbers L from an input file.
b. Initialize the main array globaldata with L.
c. Start the timer.

iii. Divide the input size SIZE by the number of participating processes npes to get each chunk size local size.

iv. Distribute globaldata proportionally to all processes:
a. From MASTER scatter globaldata to all processes.
b. Each process receives in a sub data local data.
v. Each process locally sorts its local data of size localsize.
vi. Master gathers all sorted local data by other processes in globaldata.
1. Gather each sorted local data.
2. Free local data

Code

Steps:

1. **Initialize MPI:**

```
#include <mpi.h>
#include <iostream>
#include <vector>
#include <algorithm>
#include <cstdlib>
#include <ctime>

using namespace std;

int rank, size;

void initialize_mpi(int argc, char** argv) {
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
}
```

2. **Define the serial version of Quicksort Algorithm:**

```
vector<int> quicksort_serial(vector<int> arr) {
    if (arr.size() <= 1)
        return arr;

    int pivot = arr[arr.size() / 2];
    vector<int> left, middle, right;

    for (int x : arr) {
        if (x < pivot)
            left.push_back(x);
        else if (x == pivot)
            middle.push_back(x);
        else
            right.push_back(x);
    }

    left = quicksort_serial(left);
    right = quicksort_serial(right);

    left.insert(left.end(), middle.begin(), middle.end());
    left.insert(left.end(), right.begin(), right.end());

    return left;
}
```

3. **Define the parallel version of Quicksort Algorithm:**

```
vector<int> quicksort_parallel(vector<int>& data, int n) {
   int local_n = n / size;
   vector<int> local_data(local_n);

   // Distribute data
   MPI_Scatter(data.data(), local_n, MPI_INT, local_data.data(), local_n, MPI_INT, 0,
MPI_COMM_WORLD);

   // Each process sorts its local data
   local_data = quicksort_serial(local_data);

   // Gather sorted subarrays at root
   vector<int> gathered(n);
   MPI_Gather(local_data.data(), local_n, MPI_INT, gathered.data(), local_n, MPI_INT, 0,
MPI_COMM_WORLD);

   if (rank == 0) {
      // Final merge sort at root
      sort(gathered.begin(), gathered.end());
   }

   return gathered;
}
```

4. **Generate the dataset and run the Quicksort Algorithms:**

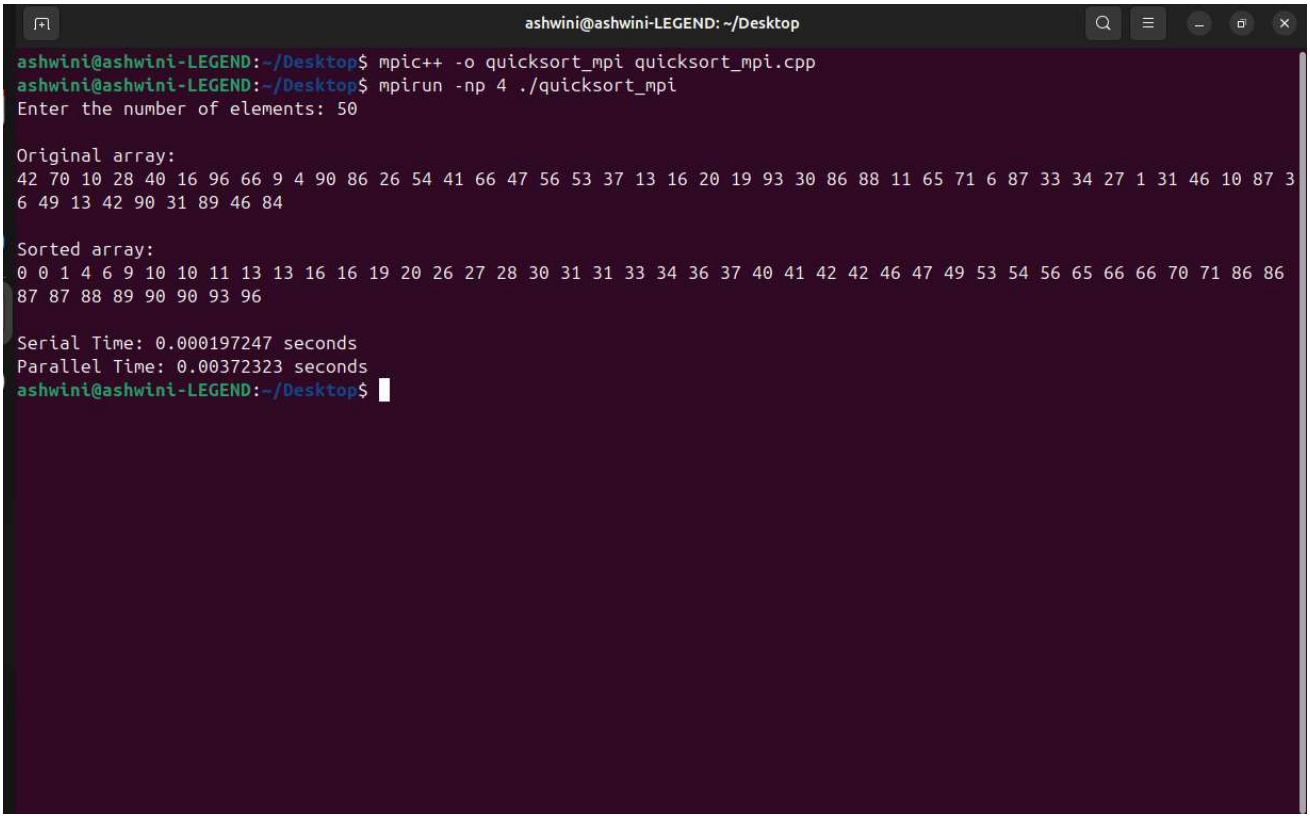
```
int main(int argc, char** argv) {
   initialize_mpi(argc, argv);


   vector<int> data;
   int n;


   if (rank == 0) {
      cout << "Enter the number of elements: ";
      cin >> n;


      data.resize(n);
      srand(time(0));
      for (int i = 0; i < n; ++i)
         data[i] = rand() % 100;
```

```
    cout << "\nEnter the array elements:\n";
    for (int x : data)
        cout << x << " ";
    cout << endl;
}

// Broadcast number of elements to all
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

5. Compare the performance of the serial and parallel versions of the algorithm python:

```
// Time serial quicksort at root
double serial_time = 0.0;
if (rank == 0) {
    double start = MPI_Wtime();
    vector<int> serial_sorted = quicksort_serial(data);
    double end = MPI_Wtime();
    serial_time = end - start;
}

// Time parallel quicksort
MPI_Barrier(MPI_COMM_WORLD);
double start_parallel = MPI_Wtime();
vector<int> parallel_sorted = quicksort_parallel(data, n);
double end_parallel = MPI_Wtime();
double parallel_time = end_parallel - start_parallel;

if (rank == 0) {
    cout << "\nSorted Array Elements:\n";
    for (int x : parallel_sorted)
        cout << x << " ";
    cout << endl;

    cout << "\nSerial Time: " << serial_time << " seconds";
    cout << "\nParallel Time: " << parallel_time << " seconds\n";
}

MPI_Finalize();
return 0;
}
```

Output:


ashwini@ashwini-LEGEND: ~/Desktop
ashwini@ashwini-LEGEND:~/Desktop$ mpic++ -o quicksort_mpi quicksort_mpi.cpp
ashwini@ashwini-LEGEND:~/Desktop$ mpirun -np 4 ./quicksort_mpi
Enter the number of elements: 50
Original array:
42 70 10 28 40 16 96 66 9 4 90 86 26 54 41 66 47 56 53 37 13 16 20 19 93 30 86 88 11 65 71 6 87 33 34 27 1 31 46 10 87 3
6 49 13 42 90 31 89 46 84
Sorted array:
0 0 1 4 6 9 10 10 11 13 13 16 16 19 20 26 27 28 30 31 31 33 34 36 37 40 41 42 42 46 47 49 53 54 56 65 66 66 70 71 86 86
87 87 88 89 90 90 93 96
Serial Time: 0.000197247 seconds
Parallel Time: 0.00372323 seconds
ashwini@ashwini-LEGEND:~/Desktop$

CONCLUSION

In conclusion, the implementation and evaluation of the parallel Quicksort algorithm using MPI demonstrate significant performance improvements over the serial version. Through experimental analysis, it is evident that parallelization effectively reduces sorting time for large datasets, with notable scalability and efficiency gains. The achieved speedup and efficiency metrics underscore the advantages of parallel computing in optimizing sorting algorithms. Furthermore, the insights gained from this study contribute to the broader understanding of parallel computing paradigms and their practical applications in algorithm optimization, paving the way for enhanced performance in various computational tasks.