Arab Academy for Science, Technology and Maritime Transport

College of Computing & Information Technology

Subject

Design & Analysis of Parallel Algorithms

Project

Parallel Merge Sort Using MPI in C++

Submitted By

Dina Ahmed El Sayed Fathy (23123818)

Submitted To

Prof.Dr. Khaled El Shafeay

Table of Contents

Chapter 1 Introduction 3

1.1. Introduction 3

1.2. Goal 3

1.3. Challenges 3

Chapter 2 Parallel Merge Sort Algorithm 4

2.1 How it Works 4

Chapter 3 What is Message Passing Interface (MPI) 5

3.1 Definition of MPI 5

3.2 Basic MPI Operations 5

Chapter 4 Implementation 6

4.1 Main Functions of the Parallel Sorting Program 6

4.2 C++ Code 6

Chapter 5 Experimental Results 10

5.1 User Input 10

5.2 Output 10

Chapter 6 11

6.1 Applications of Parallel Merge Sort 11

Chapter 7 12

7.1 Conclusion 12

Table of Figures

Figure 1-Parallel Merge Sort 4

Figure 2-C++ Console (I/p and O/p) 10

Chapter 1

Introduction

1.1. Introduction

Parallel Merge Sort is **a scalable sorting algorithm designed for distributed systems**. It divides a large dataset into smaller chunks, distributes them across multiple processors, sorts each chunk independently, and merges the results into a single sorted array. **Using MPI (Message Passing Interface), data is scattered, processed in parallel, and gathered efficiently.**

1.2. Goal

The goal of the program is to **sort an array of numbers in ascending order** using MPI.

1.3. Challenges

1. **Load Balancing**

   Ensuring all processes have an equal amount of work. If the number of elements isn't divisible by the number of processes, some processes may be idle or overloaded.

2. **Communication Overhead**

   The performance of parallel sorting depends on efficient communication between processes. Excessive communication during merging can negate the benefits of parallelism.

3. **Memory Usage**

   Sorting and merging operations may require additional memory, especially when handling large datasets.

4. **Merging Complexity**

Combining sorted chunks in the root process can become a bottleneck. This step can be parallelized further but requires more complex implementation.

5. **Scalability**

While parallel merge sort is efficient for moderately sized data, its performance may degrade with a very high number of processes due to communication overhead.

Chapter 2

Parallel Merge Sort Algorithm

2.1 How it Works

The main concept behind parallel merge and sort is **divide-and-conquer**, applied in a distributed environment

1. **Divide**

   Split the input array into chunks and distribute them among processes.

2. **Conquer**

   Each process sorts its assigned chunk independently using a sequential sorting algorithm (e.g., merge sort).

3. **Merge**

   The sorted chunks are gathered back and merged iteratively or hierarchically to form the final sorted array.

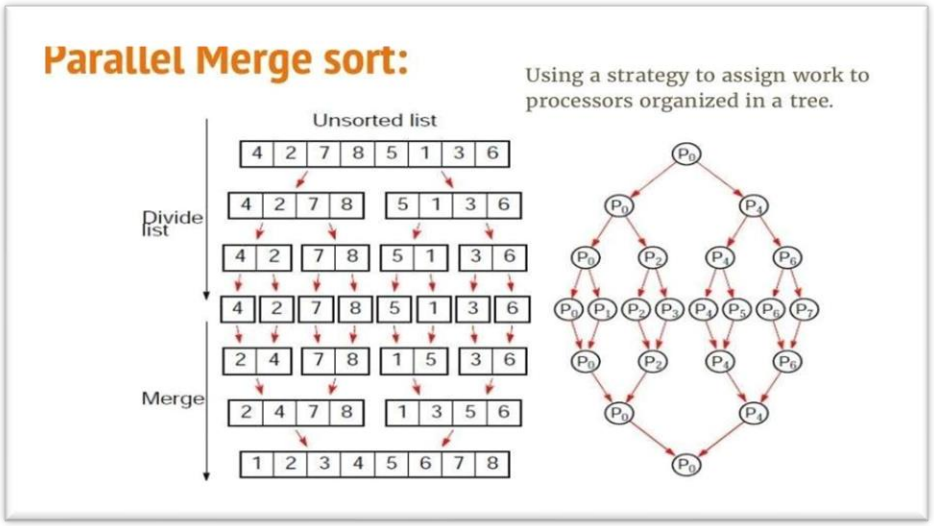

Figure 1-Parallel Merge Sort

Chapter 3

What is Message Passing Interface (MPI)

3.1 Definition of MPI

- Is a standardized and portable system for enabling communication **between processes in parallel and distributed computing environments**.
- **Allows processes to exchange data and synchronize their operations**, either on a single machine with multiple cores or across multiple machines in a cluster.
- Provides **various communication mechanisms**, including point-to-point and collective operations, and is widely used in high-performance computing (HPC) applications.
- **Offers scalability, efficiency, and** portability, making it essential for building parallel applications that require coordination and data sharing among multiple processes.
- **Is a library, not a programming language,** that provides the necessary tools and functions for inter-process communication in parallel computing.

3.2 Basic MPI Operations

- **MPI_Init**
  Initializes the MPI environment.
- **MPI_Finalize**
  Terminates the MPI environment.
- **MPI_Comm_rank**
  Determines the rank (ID) of the process in the communicator.
- **MPI_Comm_size**
  Determines the total number of processes in the communicator.
- **MPI_Send**
  Sends a message from one process to another.
- **MPI_Recv**
  Receives a message sent by another process.
- **MPI_Scatter**
  Distributes data from one process to all other processes.
- **MPI_Gather**
  Gathers data from all processes to a single process.

Chapter 4

Implementation

4.1 Main Functions of the Parallel Sorting Program

1. **Data Distribution**
   MPI_Scatter() divides the array into chunks and distributes them among the processes.
2. **Parallel Sorting**
   Each process sorts its chunk simultaneously.
3. **Data Collection**
   MPI_Gather() collects the sorted chunks back to the root process.
4. **Final Merge**
   The root process merges the sorted chunks.

4.2 C++ Code

```
#include <mpi.h>// Including MPI Library
#include <iostream>
#include <vector>// Container for dynamic arrays
// Merge function
void merge(std::vector<int>& arr, std::vector<int>& left, std::vector<int>& right)
{
int i = 0, j = 0, k = 0;
//Loop works as long as elements in both left and right arrays
while (i < left.size() && j < right.size()) {
//sorting
// compare elemnets of left and right to ensure elements added in  sorted order
if (left[i] < right[j]) {
arr[k++] = left[i++];
}
else {
//right <left
arr[k++] = right[j++];
}
}
//Copying remaining element
//After 1st loop ,one of subarray might remain so copy elements to arr
//ensuring all elements are merged finally
```

```
while (i < left.size()) arr[k++] = left[i++];
while (j < right.size()) arr[k++] = right[j++];
}

// MergeSort function
//Recursively sort an array using divide and conquer approach
void mergeSort(std::vector<int>& arr) {
if (arr.size() <= 1) return;//if less than or =1 ,already soteed
int mid = arr.size() / 2;//divdie arr into two halves(left and right)
//sorting
std::vector<int> left(arr.begin(), arr.begin() + mid);
std::vector<int> right(arr.begin() + mid, arr.end());
//by recursive
//sort two halves
mergeSort(left);
mergeSort(right);
//merge two sorted halves by using merge function
merge(arr, left, right);
}
//Main
int main(int argc, char* argv[]) {
MPI_Init(&argc, &argv);//MPI Intializaton, All MPI programs must start with this
call.

int rank, size;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);//rank(ID) of processsors
MPI_Comm_size(MPI_COMM_WORLD, &size);//total number of processors

int n;// Size of I/P the array
std::vector<int> global_array;//Holds the array elements (used only by the root
process).
//Root Process Logic
//Gets input from the user (array size n and its elements).
//& Prints the unsorted array.
if (rank == 0) {
std::cout << "Enter the number of elements in the array: ";
std::cin >> n;
//entering elemnts of array
global_array.resize(n);
std::cout << "Enter " << n << " elements for the array:\n";
```

```
for (int i = 0; i < n; ++i) {
std::cin >> global_array[i];
}
//printing elements unsorted user entered
std::cout << "Unsorted array: ";
for (int num : global_array) std::cout << num << " ";
std::cout << std::endl;
}

// Broadcast size of the array
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);//The array size n is
broadcast from the root process (rank 0) to all other processes.
// Ensures all processes know the size of the global array to be sorted.

//Divde array  into chunks for each process
// To ensure load balancing, the array is divided as evenly as possible
// Calculate local sizes
int base_size = n / size;//gives the basic size each process will handle.
int remainder = n % size;//ccounts for the extra elements when n isn't divisible by
the number of processes.
std::vector<int> send_counts(size);// stores the number of elements each process
will receive.
std::vector<int> displacements(size);//calculates the starting index in the global
array for the chunk assigned to process i.

for (int i = 0; i < size; ++i) {
send_counts[i] = base_size + (i < remainder ? 1 : 0);
displacements[i] = (i == 0) ? 0 : (displacements[i - 1] + send_counts[i - 1]);
}

// Allocate local array
std::vector<int> local_array(send_counts[rank]);

// Data Dsistribution ,Divides the array into chunck and distributes them among
processors
MPI_Scatterv(global_array.data(), send_counts.data(), displacements.data(),
MPI_INT,
local_array.data(), send_counts[rank], MPI_INT, 0, MPI_COMM_WORLD);

// Perform local sort
```

```
mergeSort(local_array);

// Collects the sorted chunks back to root
MPI_Gatherv(local_array.data(), send_counts[rank], MPI_INT,
global_array.data(), send_counts.data(), displacements.data(), MPI_INT, 0,
MPI_COMM_WORLD);

// Final merge step at root
if (rank == 0) {
int step = send_counts[0];
for (int i = 1; i < size; ++i) {
std::vector<int> temp(global_array.begin(), global_array.begin() + step +
send_counts[i]);
std::vector<int> right(global_array.begin() + step, global_array.begin() + step +
send_counts[i]);
merge(temp, temp, right);
std::copy(temp.begin(), temp.end(), global_array.begin());
step += send_counts[i];
}

// Output sorted array
std::cout << "Sorted array: ";
for (int num : global_array) std::cout << num << " ";
std::cout << std::endl;
}

MPI_Finalize();//End of MPI Program
return 0;
}
```

Chapter 5

Experimental Results

5.1 User Input

The user specifies **the number of elements they want to sort** (e.g. 8)and then provides the elements themselves. These inputs are entirely user-defined, meaning they **can enter any set of values based on their requirements**. For instance, the user could input set of values (e.g. 12 7 9 14 3 5 8 10), including negative, positive, or zero values.

The program is designed to **adapt dynamically to the user's input**, ensuring flexibility and usability for a wide range of scenarios.

5.2 Output

Shows **the interaction between the user and the program**. The user is first prompted to enter the number of elements in the array, user imputed 8. Next, the user is asked to enter 8 elements, and they provide the sequence 12 7 9 14 3 5 8 10. The program then displays the unsorted array exactly as entered by the user: 12 7 9 14 3 5 8 10. After processing this input using a sorting algorithm, the program outputs the sorted array: 3 5 7 8 9 10 12 14, showing the numbers in ascending order. **This output verifies that the program successfully sorts the user-provided data and handles input dynamically as entered by the user.**

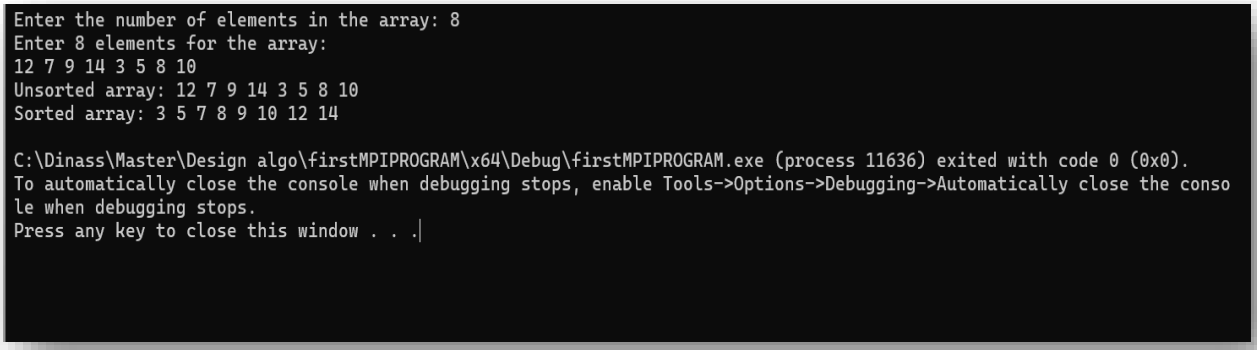

Figure 2-C++ Console (I/p and O/p)

Chapter 6

Applications of Parallel Merge Sort Using MPI

6.1 Applications of Parallel Merge Sort

1. **Big Data Analytics**

Sorting massive datasets distributed across multiple nodes in a cluster.

2. **Distributed Databases**

Efficiently organizing data during query optimization and indexing.

3. **Scientific Simulations**

Sorting intermediate results in large-scale numerical computations.

4. **Real-Time Systems**

Handling time-sensitive sorting tasks in parallel to meet deadlines.

5. **Parallel File Systems**

Sorting data blocks for faster file read/write operations.

6. **Financial Modeling**

Sorting market data for real-time trading and risk assessment.

Chapter 7

Conclusion

7.1 Conclusion

This parallel merge sort case study demonstrates the efficient use of MPI for distributed sorting by splitting the input array among multiple processes, allowing them to sort their respective subarrays independently. The root process then gathers the sorted subarrays and performs a final merge step to obtain the fully sorted array. The program highlights **the scalability of parallel sorting algorithms, where the computation is divided among available processes to speed up sorting for large datasets.**

While the local sorting is handled in parallel, the final merger step remains sequential, indicating a potential area for further optimization.

This approach effectively shows how parallel computing can be leveraged to tackle computationally intensive tasks like sorting in a distributed environment.