Faculty of Computers and Artificial intelligence

Fayoum University

Parallel Search and Sorting Algorithms with MPI

AUTHOR

Mahmoud Adel Mahmoud

Introduction

This project focuses on the implementation and analysis of key parallel algorithms using the

Message Passing Interface (MPI).

We applied parallel computing techniques to solve computational problems efficiently, including Quick Search, Prime Number Finding, Bitonic Sort, Radix Sort, and Sample Sort.

Through hands-on development and experimentation, the project aims to reinforce MPI programming skills and promote understanding of parallel algorithm design, communication strategies, and performance

evaluation on distributed systems.

Algorithm Descriptions

Quick Search – Step-by-Step Explanation

1. Process 0 (Master) Only:

- Reads numbers from an input text file (e.g., input.txt).
- Asks the user to input the value to search for (target).
- Broadcasts the array and the target to all processes using MPI_Bcast.

2. Each Process (rank 0, 1, 2...) Gets a Data Chunk:

- If there are 100 numbers and 3 processes:
    - Rank 0 searches from index 0 to 32

- Rank 1 from 33 to 65
- Rank 2 from 66 to 99

3. Each Process Performs Linear Search on Its Chunk:

- It loops through its segment.
- If the target is found, it stores its index.
- If not found, it stores -1.

4. Results Are Gathered at Process 0:

- Using MPI_Reduce with MPI_MAX, the highest valid index is returned.
- Process 0 now knows whether the target exists and where.

5. Process 0 Prints the Final Result:

- If found → prints: "Found at index X"
- If not found → prints: "Not found"
- Each process also prints how long it took using MPI_Wtime.

```
Data:     [ 5, 7, 12, 20, 25, 30, 42, 50, 55, 60 ]
Target:   42
Processes: 3

Chunks:
- Rank 0 → [5, 7, 12]      → Not Found
- Rank 1 → [20, 25, 30]    → Not Found
- Rank 2 → [42, 50, 55, 60] → Found at index 6

MPI_Reduce → Highest index = 6
```

2. Prime Number Finding

- **Input: Integer range (start, end)**
- **Each process checks part of the range for primality**
- **Results gathered using MPI_Gather**

Prime Finder on a Specific Range – Step-by-Step Explanation

To find all prime numbers within a specific range (e.g., from start = 1 to end = 1000) using parallel processing with MPI.

1. Process 0 (Master Only):

- Asks the user to input the start and end of the range.
- Calculates the full range size.
- Broadcasts the range values to all processes.

2. Each Process Computes Its Subrange:

- The range is divided equally across all ranks:
    - For example: range = 1 to 1000, 4 ranks → each gets 250 numbers.
- Rank 0 → 1–250
  Rank 1 → 251–500
  Rank 2 → 501–750
  Rank 3 → 751–1000

3. Each Rank Performs Prime Checking on Its Part:

- Each number in the subrange is tested using basic prime-checking logic:
    - A number n is prime if it's not divisible by any number from 2 to sqrt(n)

4. All Results Are Gathered at Process 0:

- Each process stores its primes in a vector.
- MPI_Gather or MPI_Gatherv is used to collect all prime numbers to Rank **0.**

5. Process 0 Writes Output:

- All prime numbers are printed or saved to a file (e.g., output_primes.txt).
- Each rank may also print how much time its processing took.

```
Target Range: [1 ... 1000]
Number of Ranks: 4

Each Process Gets:
- Rank 0 → [1 - 250]
- Rank 1 → [251 - 500]
- Rank 2 → [501 - 750]
- Rank 3 → [751 - 1000]

Each rank checks its own range for prime numbers.
```

```
Results:
- Rank 0 → [2, 3, 5, 7, 11, ..., 241]
- Rank 1 → [251, 257, ..., 491]
...

All results sent to Rank 0 → Combined and written to output.
```

Bitonic Sort – Step-by-Step Parallel Idea

1. Process 0:

- Reads data and pads it to the next power of 2 if needed.
- Broadcasts padded array to all.

2. MPI_Scatter:

- Each rank receives a chunk of data.

3. Each Process:

- Performs bitonic sort locally on its chunk.

4. MPI_Gather:

- Collects local results at process 0.

5. Process 0:

- Performs full bitonic merge on combined data.
- Removes padding and writes result to file.

```
Original: [8, 4, 6, 2]
Padded → [8, 4, 6, 2]  → Already 2^2 elements

Chunks:
- Rank 0 → [8, 4] → Sorted → [4, 8]
- Rank 1 → [6, 2] → Sorted → [2, 6]

Gather → [4, 8, 2, 6] → Bitonic Merge → [2, 4, 6, 8]
```

Radix Sort – Step-by-Step Parallel Idea

1. Process 0:

- Reads unsorted integers from file.
- Broadcasts the full dataset to all processes.

2. MPI_Scatter:

- Splits the dataset into chunks.
- Each process receives its part (equal-sized blocks).

3. Each Process:

- Performs digit-wise countSort() on its chunk.

4. MPI_Gather:

- Sends sorted local chunks back to master (process 0).

5. Process 0:

- Merges and performs final sort on all gathered parts.
- Writes sorted result to file.

```
Data: [170, 45, 75, 90, 802, 24, 2, 66]
Steps:
- Split: Rank 0 → [170, 45, 75], Rank 1 → [90, 802, 24], Rank 2 → [2, 66]
- Local sort: Each rank uses digit-based sort (unit → tens → hundreds)
- Gather and merge on Rank 0
```

Sample Sort – Step-by-Step Parallel Idea

1. Process 0:

- Reads full array from file.
- Broadcasts data to all.

2. MPI_Scatter: All processes receive equal-sized data chunks.

3. Each Process:

- Sorts its chunk locally.
- Selects sample elements (e.g., every k-th value).

4. MPI_Gather Samples:

- Master gathers all samples and selects pivots.
- Broadcasts pivots to all processes.

5. Partition + MPI_Alltoallv:

- Each process partitions its chunk based on pivots.
- Sends buckets to appropriate processes.

6. Each Process: Receives final data chunk, sorts it, and sends back.

7. Process 0: Gathers final sorted data and writes to file.

```
Data: [15, 6, 9, 21, 3, 7, 11, 25]
Chunks:
- Rank 0 → [15, 6], Rank 1 → [9, 21], Rank 2 → [3, 7], Rank 3 → [11, 25]

Samples → Picked by each → [6, 9, 7, 11] → Sorted → Pivots = [7, 9, 11]

Partition:
- Rank 0 sends small items to Rank 0, 1...
- Alltoallv done

Final merge → Each rank sorts its final bucket
```

RUN

```
C:\PROJECT_PARALLEL\PROJECT_MPI\x64\Debug\PROJECT_MPI.exe
================================================================================
                    Welcome to Parallel Algorithm Simulation with MPI
================================================================================

    Please choose an algorithm to execute:
          1 - Quick Search
          2 - Prime Number Finding
          3 - Bitonic Sort
          4 - Radix Sort
          5 - Sample Sort

              Enter the number of the algorithm to run: 4
             Enter input file path: C:\PROJECT_PARALLEL\PROJECT_MPI\PROJECT_MPI\input_radixsort.txt

Done #
  Sorted data saved to output_radixsort.txt
********************************************************************************&

Want to try another algorithm? (Y/N): y
================================================================================
                    Welcome to Parallel Algorithm Simulation with MPI
================================================================================

    Please choose an algorithm to execute:
          1 - Quick Search
          2 - Prime Number Finding
          3 - Bitonic Sort
          4 - Radix Sort
          5 - Sample Sort

              Enter the number of the algorithm to run: 5
                  Enter input file path: C:\PROJECT_PARALLEL\PROJECT_MPI\PROJECT_MPI\input_samplesort.txt

Done #
  Sorted data saved to output_samplesort.txt
********************************************************************************&

Want to try another algorithm? (Y/N):
```

Performance Analysis

Algorithm	Input Size	Time (s) p (1)	Time (s) p (4)	Parallel Time Complexity
Quick Search	Dynamic	0.0009716	0.0001375	O(n log n)
Prime Finder	Dynamic range	0.0135441 [0 – 500]	0.001198 [0 – 500]	O(n/p)
Bitonic Sort	Dynamic	0.0015832	0.0006421	O(log^2 n)
Radix Sort	Dynamic	0.0010434	0.0008988	O(n)
Sample Sort	Dynamic	0.0063002	0.0007774	O(n log n)

THANK YOU #