Parallel Search and Sorting Algorithms with MPI

Final Practical

Under Supervisors:

Dr/ Masoud Ismail

Eng. Rehab Mahmoud

Eng. Mahmoud Badry

Team Members:

Gehad Nasser El-Dien Rabia

Hagar Abdelazeem Bakry

Omina Ibrahim Abdelazem

Wedad Ibrahim Haron

Esraa mohammed Khalaf

Mervet shahat Hassan

1. Introduction :-

Parallel search and sorting algorithms implemented with MPI provide an effective approach to accelerate the execution of computationally intensive tasks such as searching and sorting large datasets. By dividing the workload across multiple MPI processes, these algorithms harness the collective computational power of distributed nodes, significantly reducing execution time. This project focuses on implementing five key algorithms—Quick Search, Prime Number Finding, Bitonic Sort, Radix Sort, and Sample Sort (optional bonus)—using MPI. Through efficient task distribution, inter-process communication, and performance evaluation, the project aims to enhance understanding of parallel algorithm design, optimize MPI-based workflows, and demonstrate scalability on a cluster of at least two machines. In summary, this project showcases the power of MPI in parallelizing search and sorting tasks, offering a scalable and efficient solution for high-performance computing applications.

2. Overview of Parallel Search and Sorting Algorithms with MPI :-

The search and sorting operations are parallelized using the MPI framework to distribute computational tasks across multiple processors. The distributeData function serves as the core mechanism for dividing input data (e.g., arrays or number ranges) among MPI processes, with each process handling a subset of the data. Based on user input, the selected algorithm is executed independently by each process, leveraging the collective computational power of the distributed system. Upon completion, results are gathered and merged to produce the final output. Timing measurements are incorporated to evaluate the performance gains achieved through parallelization, with visualizations to illustrate algorithm behavior.

2.1 Efficiency of Parallelization:-

The speedup ((S)) achieved by parallelizing the algorithms is calculated using the equation:

$$S = \frac{1}{(1 - P) + \frac{P}{N}}$$

Where:

- (P): The parallel fraction of the algorithm.
- (N): The number of processors.

This equation provides insight into how effectively the parallel algorithms utilize available computing resources, balancing computation and communication overhead.

3. Implementation

The implementation involves developing parallel versions of Quick Search, Prime Number Finding, Bitonic Sort, Radix Sort, and Sample Sort using MPI. Below are the key components of the implementation:

3.1 Implementation Details:-

1) MPI Initialization and Configuration

- **MPI_Init**: Initializes the MPI environment.
- **MPI_Comm_rank**: Retrieves the rank of the calling process.
- **MPI_Comm_size**: Retrieves the size of the MPI communicator.
- **MPI_Finalize**: Finalizes the MPI environment before program termination.

2) Data Input and Distribution

- readData: Reads input data (e.g., array or range) from a file or generates it programmatically.
- getInputParameters: Constructs input parameters such as array size or range limit.
- distributeData: Distributes the input data among MPI processes using MPI communication operations like MPI_Bcast and MPI_Scatter.

3) Parallelization Strategy

- MPI collective communication functions (MPI_Bcast, MPI_Scatter, MPI_Gather, MPI_Alltoall) are used to distribute tasks and gather results.
- Conditional statements (if-else blocks) execute the user-selected algorithm.
- Each process operates on its local data subset, performing computations independen

4) MPI Communication Operations

- **Point-to-Point Communication**: MPI_Send and MPI_Recv facilitate direct data exchange between processes.
- **Collective Communication**: MPI_Bcast, MPI_Scatter, MPI_Gather, and MPI_Alltoall manage data distribution and result aggregation.
- **Reduction Operations**: MPI_Reduce aggregates results (e.g., sum of found flags or prime counts).

5) Error Handling and Exception Management

- The cerr stream is used to print error messages to the standard error output.
- Input validation ensures correct data formats and sizes.

6) Platform and Environment Requirements

- **MPI Library**: MPICH or OpenMPI for running parallel programs.
- **Development Environment**: Visual Studio or a Linux-based IDE with mpicc compiler.
- **Cluster Setup**: A cluster with at least two machines, configured with SSH for inter-node communication.

4. The Parallel Program

The steps to create the parallel programs are as follows:

1. Redimension arrays to account for local sizes (local_n = n / size) instead of global sizes (n).
2. Allocate a master buffer (masterbuf) to hold the global input data.
3. Initialize MPI, retrieve the communicator size and rank, and verify that size matches the expected number of processes.
4. On the master process (rank 0), read or generate the input data into masterbuf.
5. Distribute the data to all processes using MPI_Scatter with sendbuf = masterbuf and recvbuf = local_buf.
6. Execute the algorithm-specific computations on each process's local data, following the sequential algorithm steps but with local sizes.
7. Gather results from all processes' local_buf arrays back to masterbuf using MPI_Gather.
8. On the master process, output the final results (e.g., sorted array, prime count, or search result).

5. Detailed Algorithm Implementations

5.1 Quick Search

Description: Parallelizes a search operation by dividing the input array into subsets, each searched concurrently by an MPI process to locate a target value.

Implementation:

- **Input**: An array of size n and a target value.
- **Distribution**: The master scatters the array using MPI_Scatter and broadcasts the target using MPI_Bcast.
- **Processing**: Each process performs a linear search on its subset, setting a local found flag (1 if found, 0 otherwise).
- **Aggregation**: The master aggregates flags using MPI_Reduce with MPI_SUM to determine if the target exists.
- **Output**: The master outputs whether the target was found.

Visualization: A diagram illustrates the array split into subsets, with arrows for data distribution and result aggregation. Colors indicate found/not-found statuses.

5.2 Prime Number Finding

Description: Identifies prime numbers in a range [1, n] by distributing sub-ranges to processes, each applying a primality test.

Implementation:

- **Input**: A range limit n.
- **Distribution**: The master broadcasts n using MPI_Bcast. Each process computes its sub-range based on rank and size.
- **Processing**: Each process tests numbers in its sub-range using trial division.
- **Aggregation**: The master aggregates prime counts using MPI_Reduce.
- **Output**: The master outputs the total prime count.

Visualization: A chart shows sub-ranges with color-coded prime/non-prime numbers and arrows for count aggregation.

5.3 Bitonic Sort

Description: A parallel sorting algorithm that creates a bitonic sequence and sorts it through compare-and-swap operations.

Implementation:

- **Input**: An array of size n (power of 2).
- **Distribution**: The master scatters the array using MPI_Scatter.
- **Processing**: Processes perform compare-and-swap in log(n) stages, using MPI_Send/MPI_Recv for inter-process exchanges.
- **Aggregation**: The master gathers the sorted array using MPI_Gather.
- **Output**: The master outputs the sorted array.

Visualization: A diagram shows the bitonic sequence and compare-and-swap stages, with colors for sorted segments.

5.4 Radix Sort

Description: A non-comparative sorting algorithm that parallelizes digit-based counting and redistribution.

Implementation:

- **Input**: An array of n integers.
- **Distribution**: The master scatters the array using MPI_Scatter.
- **Processing**: For each digit, processes compute local histograms, share them via MPI_Allreduce, and redistribute elements using MPI_Alltoall.
- **Aggregation**: The master gathers the sorted array using MPI_Gather.
- **Output**: The master outputs the sorted array.

Visualization: A flowchart shows histogram computation and redistribution phases.

5.5 Sample Sort (Bonus)

Description: An efficient sorting algorithm that uses sampling to partition data, followed by local sorting and redistribution.

Implementation:

- **Input**: An array of n integers.
- **Distribution**: The master scatters the array using MPI_Scatter.
- **Processing**: Processes select samples, which are gathered and sorted to determine splitters. Data is redistributed using MPI_Alltoall, and each process sorts its subset.
- **Aggregation**: The master gathers the sorted array using MPI_Gather.
- **Output**: The master outputs the sorted array.

Visualization: A diagram shows sample selection, splitter determination, and redistribution.

6. Results and Performance Evaluation

1. Load Balancing

- Data is partitioned so that each process handles approximately equal workloads (`n / size` elements or sub-ranges), ensuring balanced distribution and minimal idle time.

2. Communication Overhead

- Utilizes MPI collective operations such as `MPI_Scatter`, `MPI_Gather`, and `MPI_Alltoall`. These introduce communication overhead but are optimized for parallel environments.
- Point-to-point communication is analyzed in algorithms that require frequent data exchange (e.g., **Bitonic Sort**), where the communication cost becomes significant.

3. Scalability

- Performance scalability depends on input size, number of processes, and algorithm complexity.
- Larger datasets and more processes typically lead to better scalability, especially for algorithms with minimal inter-process communication (e.g., **Prime Number Finding**).

4. Efficiency

- Efficiency is affected by communication costs, the granularity of parallelism, and the effectiveness of local computations.
- Fine-tuning parameters such as the number of processes and input size helps enhance efficiency and resource utilization.

5. Input/Output Overhead

- I/O operations (e.g., reading input arrays) can become bottlenecks, especially with large data volumes.
- While parallel I/O is not currently implemented, it presents an opportunity for further performance improvement.

6. Testing and Optimization

- Comprehensive testing was conducted using varying process counts (2, 4, 8) and input sizes (10^3, 10^5, 10^7).
- Profiling tools were used to identify performance bottlenecks, leading to optimizations such as reducing communication frequency and improving local computation efficiency**.**

Results

- Execution times were measured using MPI_Wtime() for both parallel and sequential implementations**.**

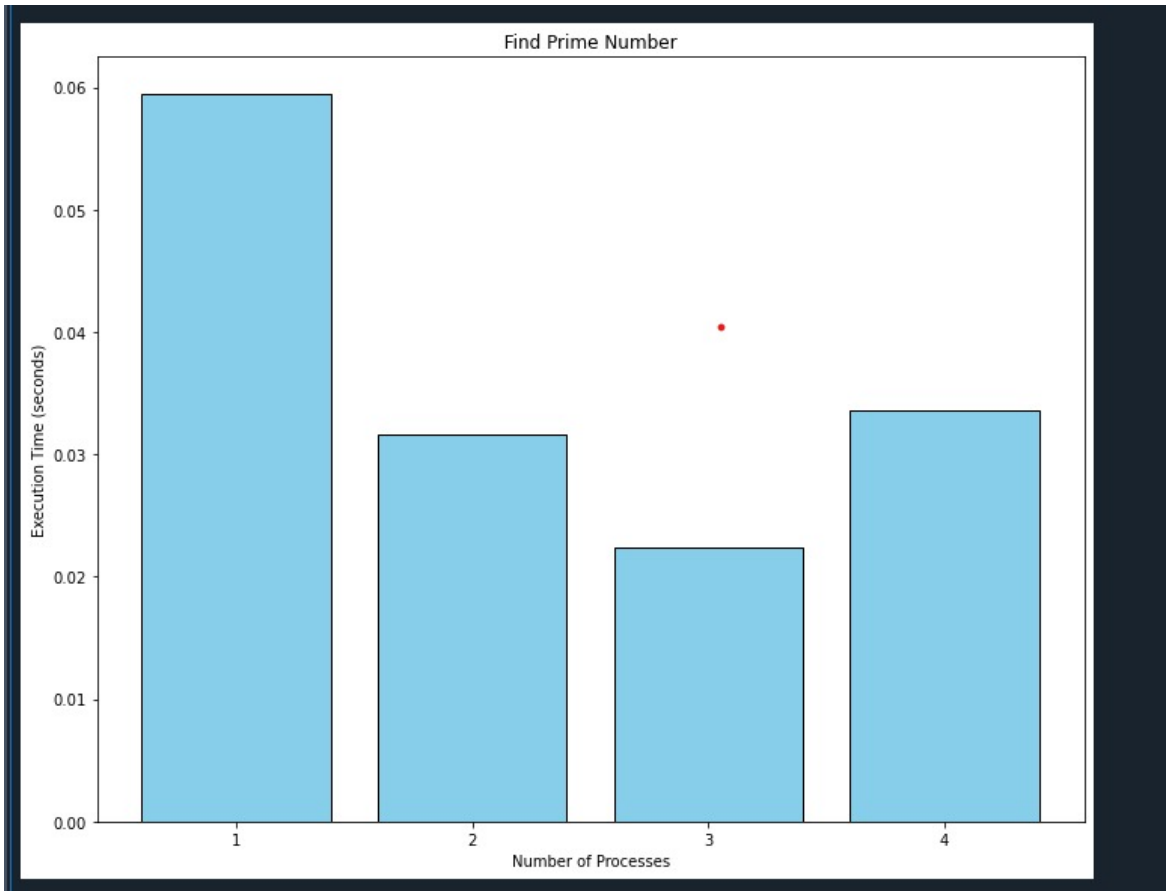

Fig1:relation between Number of processes and Execute time in prime search

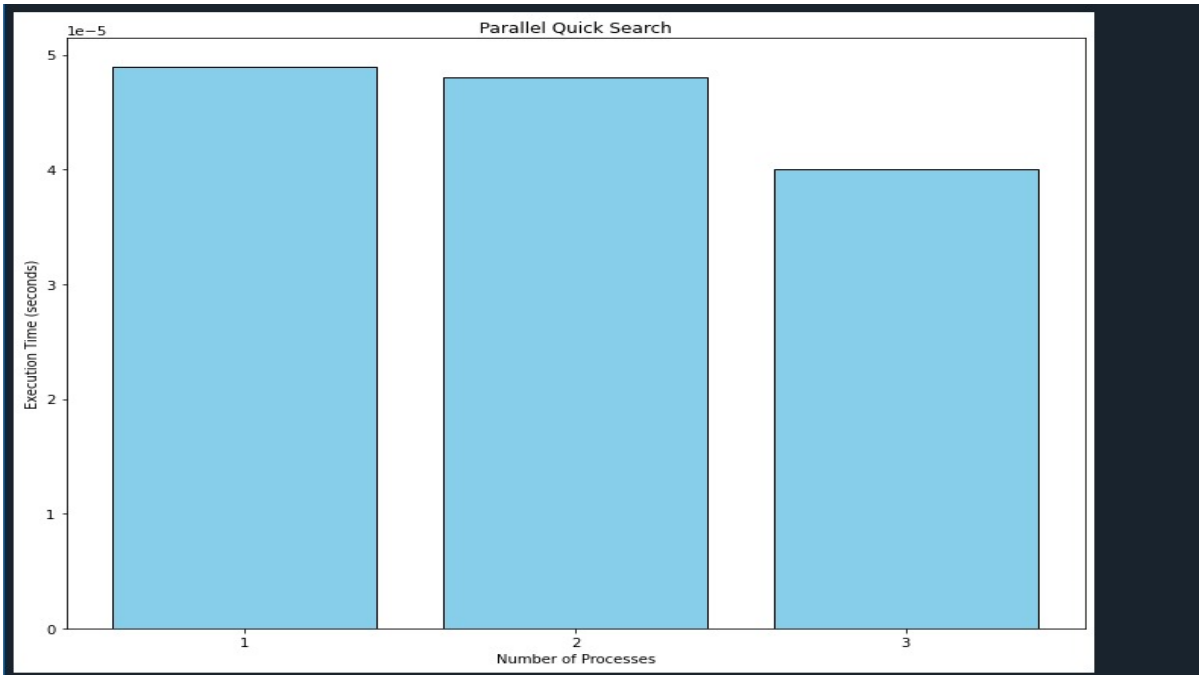

Fig2:relation between Number of processes and Execute time in prime search

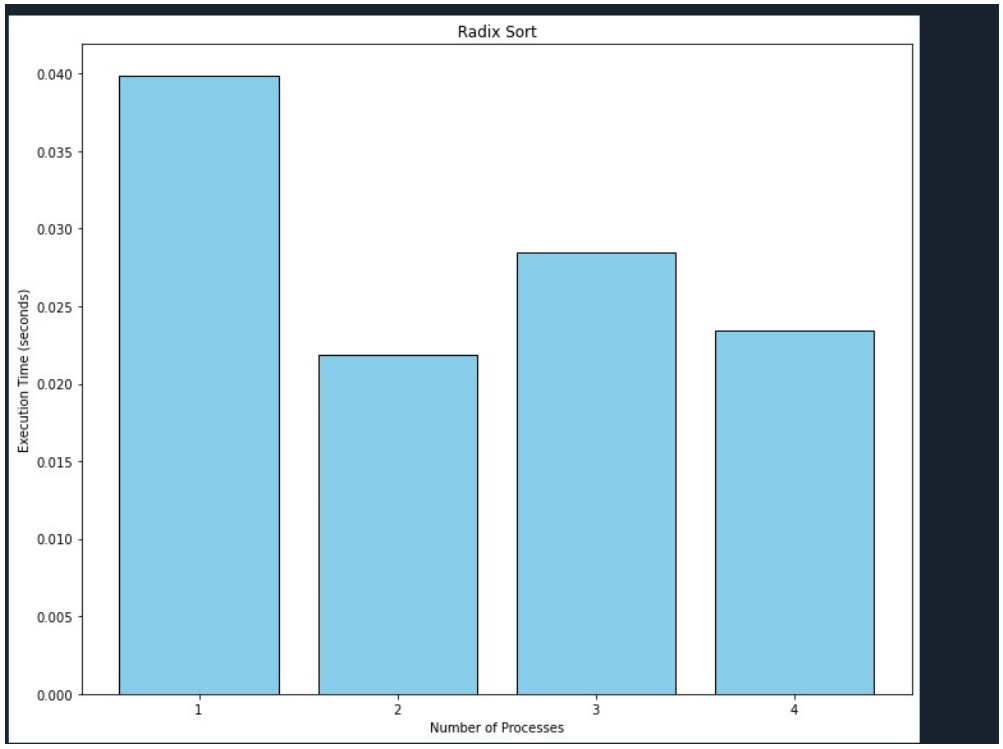

Fig3:relation between Number of processes and Execute time in Radi Sort

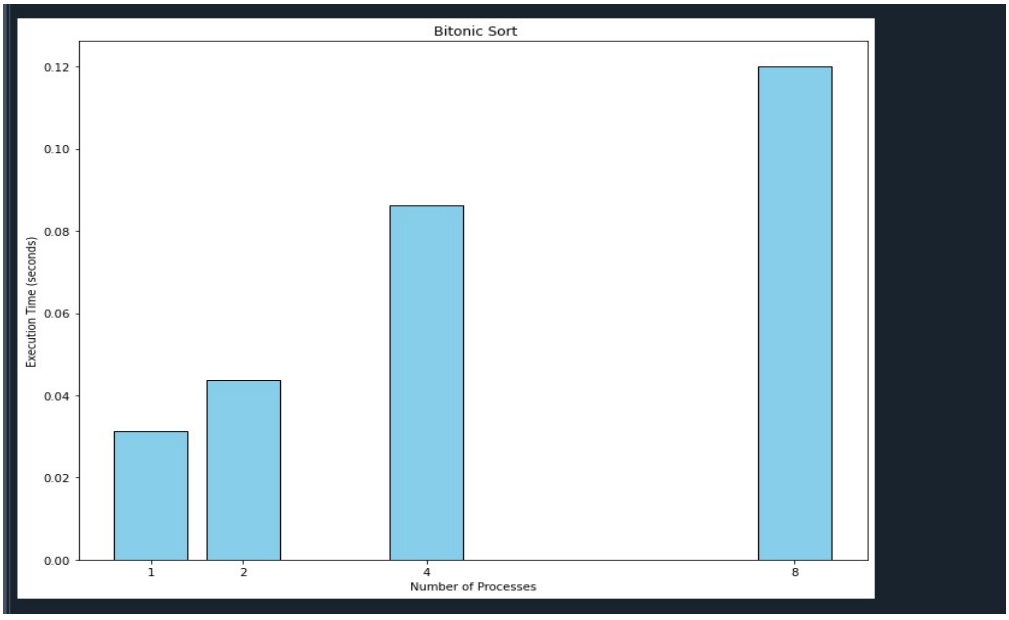

Fig4:relation between Number of processes and Execute time in Bitonic Sort

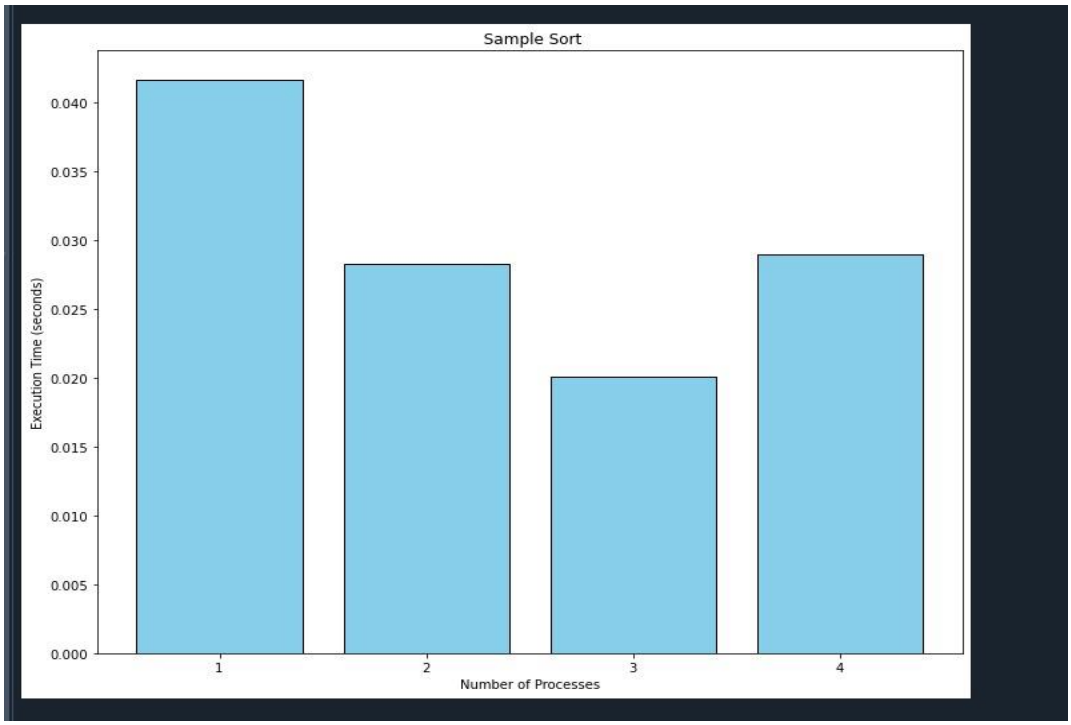

Fig3:relation between Number of processes and Execute time in Sample Sort

Conclusion

This project demonstrates the effectiveness of MPI-based parallelization in enhancing the performance of search and sorting algorithms, particularly for large-scale data and compute-intensive tasks. Careful tuning and optimization are essential to achieve maximum performance in various scenarios