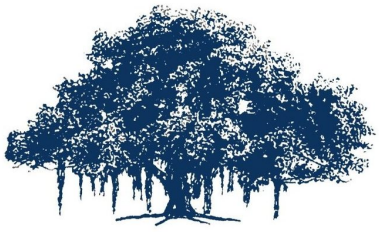
INDEPENDENT STUDY

May 2025



INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY

HYDERABAD

Guide : KISHORE KOTHAPALLI

Shantanu Pathak

(2023202019)

Rutuja Deshpande

(2023021032)

1. MOTIVATION

The motivation behind this study is to explore and compare distributed algorithms in the MapReduce and Massively Parallel Computing (MPC) models to understand their efficiency, scalability, and applicability in solving large-scale computational problems.

2. PROBLEM STATEMENT

This project aims to study the Massively Parallel Computing (MPC) model and its algorithms, implement selected fundamental problems under both the MPC and MapReduce models, and perform a comparative analysis to evaluate their efficiency and scalability.

3. ABOUT

3.1 MPI

The Message Passing Interface (MPI) is a standardized and portable message-passing system designed to enable communication between processes in a parallel computing environment. It is particularly well-suited for distributed memory architectures, where each process has its own local memory.

MPI provides a set of functions that allow processes to send and receive messages, synchronize, and coordinate tasks. These operations make it possible to divide a large computational problem into smaller tasks that run concurrently across multiple processors or computers in a cluster.

Developed in the early 1990s, MPI has become the *de facto* standard for high-performance computing (HPC). It supports multiple languages, including C, C++, and Fortran, and is implemented by various libraries such as MPICH and OpenMPI.

By offering fine-grained control over communication and computation, MPI allows developers to optimize performance and scalability in scientific simulations, data analysis, and other compute-intensive applications.

3.2 Hadoop

Apache Hadoop is an open-source framework that enables the distributed storage and processing of large datasets across clusters of computers using simple programming models. It is designed to scale up from a single server to thousands of machines, each offering local computation and storage.

Hadoop was originally developed by Doug Cutting and Mike Cafarella and is now maintained by the Apache Software Foundation. It was inspired by Google's MapReduce and Google File System (GFS) papers.

The core components of Hadoop include:

- **HDFS (Hadoop Distributed File System)** – A highly fault-tolerant distributed file system that stores data across multiple machines.
- **MapReduce** – A programming model for processing large datasets in parallel across a Hadoop cluster.

- **YARN (Yet Another Resource Negotiator)** – Manages resources and schedules tasks across the cluster.
- **Hadoop Common** – A set of shared utilities and libraries used by other Hadoop modules.

Hadoop is widely used in big data applications due to its scalability, fault tolerance, and ability to handle unstructured and semi-structured data. It plays a critical role in industries ranging from finance to healthcare for processing and analyzing massive volumes of data efficiently.

3.3 MapReduce

MapReduce is a programming model and processing technique for handling and generating large datasets with a distributed algorithm on a cluster. It was introduced by Google to simplify data processing on large-scale systems and has been widely adopted through platforms like Apache Hadoop.

The model breaks down data processing into two key phases:

- **Map Phase:** Input data is divided into independent chunks and processed in parallel. Each mapper transforms input key-value pairs into a set of intermediate key-value pairs.
- **Reduce Phase:** The intermediate data is grouped by key, and reducers process each group to generate the final output.

MapReduce provides:

- **Scalability:** Easily handles petabytes of data across many machines.
- **Fault tolerance:** Automatically handles failures by rerunning failed tasks.
- **Simplicity:** Developers can focus on writing the `map()` and `reduce()` logic, while the framework handles the distribution, parallelization, and fault recovery.

3.4 Massively Parallel Computing (MPC) Model

The Massively Parallel Computing (MPC) model is a theoretical framework designed to analyze and develop algorithms for large-scale distributed systems that process massive datasets in parallel. It captures the essence of systems like MapReduce, Hadoop, and Spark by abstracting the hardware details and focusing on communication complexity and memory constraints.

In the MPC model:

- The computation is performed by many machines (or processors) working in parallel.
- Each machine has sublinear memory, typically much smaller than the input size.
- The computation proceeds in synchronous rounds, where each round consists of local computation followed by a global communication phase.
- The goal is to minimize the number of communication rounds, as this often dominates the running time in distributed settings.

4. QUICKSORT-INSPIRED SORTING ALGORITHM IN THE MPC MODEL

4.1 Context and Model

This algorithm is designed for the Massively Parallel Computation (MPC) model, which reflects modern cloud computing systems. In the MPC model:

- There are multiple machines (nodes), each with memory bounded by W words.
- The input size is N, and the number of machines M and memory W per machine satisfy $M \times W \in \Theta(N)$.
- Communication occurs in synchronous rounds, with each machine allowed to send/receive up to W words per round.
- The efficiency of an algorithm is measured in terms of the number of communication rounds.

4.2 Goal

Sort N elements distributed across machines, and compute the rank of each element in the sorted sequence, while keeping communication minimal and local computation efficient.

4.3 Algorithm Overview

This algorithm mimics the behavior of Quicksort but adapts it to the MPC constraints using multiple pivots to divide the global input into manageable buckets. Each bucket can then be sorted locally by a machine.

4.4 Detailed Steps

Step 1: Pivot Selection

- Each machine M_i has a local dataset N_i of size n_i.
- Each element in N_i is independently selected as a pivot with probability p.
- Expected number of local pivots: $n_i \times p$.

Step 2: Pivot Distribution

- Each machine sends its selected pivots to all other machines.
- Now, every machine knows all pivots selected in the system.

Step 3: Local Rank Computation

- For each pivot q, a machine computes LocalRank(q, i): the number of elements in its local data smaller than q.

Step 4: Global Pivot Selection

- All machines send their local ranks to a designated coordinator machine M (say M_1).
- M uses this information to:

  - Compute global ranks for all pivots.
  - Choose a subset of N^p pivots ensuring that every segment of size N^δ has at least one pivot (proved via a lemma).
- The selected pivots $Q = \{q_1, q_2, ..., q_{N^{1-\delta}}\}$ are broadcast to all machines.

Step 5: Bucketing Elements

- Using the received pivots Q, each machine partitions its data into $|Q| + 1$ buckets, where bucket j contains elements between q_{j-1} and q_j.

Step 6: Redistributing Buckets

- Each machine sends bucket j's elements to machine M_j.
- Machines are assigned buckets such that machine M_j receives all elements in the interval $(q_{j-1}, q_j]$.

Step 7: Local Sorting and Rank Assignment

- Each machine sorts the received bucket locally.
- It assigns global ranks to the sorted elements, beginning from the known global rank of the lower pivot q_{j-1}.

4.5 Communication Analysis

Let:

$$m = \frac{N}{W} \quad \text{(number of machines)}$$

Each machine sends/receives $O(n_i \times p) = O(W \times p)$ pivots.

Total words received per machine: $O(m \times W \times p) = O(N \times p)$.

To ensure that each machine does not exceed its memory W, choose $p = \frac{W}{N}$.

Thus, each communication round involves $O(W)$ words per machine, satisfying the model constraint.

The algorithm uses $O(1)$ communication rounds, making it very efficient under the MPC model.

4.6 Correctness Guarantee (Lemma)

The algorithm uses a lemma to ensure pivot quality:

With high probability, every segment of N^δ ranks in the sorted input contains at least one selected pivot.

This ensures that no bucket is too large to handle locally.

4.7 Observations and Design Principles

- No single machine ever sees the entire input, complying with the memory limit.
- Global structure is inferred using local statistics (local ranks).
- Efficient pivot selection is central to the algorithm's performance.
- Reflects key distributed algorithm design goals: local work, global coordination, sublinear resource usage.

5. OBSERVATION AND ANALYSIS: MAPREDUCE

5.1 Objective

To analyze how the number of communication rounds required for sorting using a MapReduce-based Quicksort algorithm varies with the number of parallel processes (mappers and reducers).

5.2 Experimental Setup

- **Input Size (NUM_ELEMENTS)**: 10,000 integers
- **Number of Processes (NUM_MAPPERS)**: Varied from 1 to 100
- **Data Distribution**: 50% of integers in range [1, 200], remaining 50% in range [201, 1000]
- **Pivot Selection**: Each mapper selects a random pivot from its local data
- **Communication Measurement**:
  - **Phase 1**: Pivot collection
  - **Phase 2**: Data bucketing and redistribution
- Measured by total number of key-value emissions across all mappers

5.3 Results Summary

5.4 Graph Analysis

The resulting graph shows a rapid decline in communication rounds as the number of processes increases from 1 to 20, followed by a gradual flattening of the curve beyond 20 processes.

This behavior approximates an inverse logarithmic or hyperbolic decay, reflecting that:

- **The Graph remains the same for cases where integers are distributed evenly and the case where integers are unevenly distributed.**
- Initial increases in parallelism ($1 \rightarrow 20$) offer significant gains in communication efficiency.
- Beyond 20 processes, the improvement is marginal, with diminishing returns due to overhead and coordination costs.

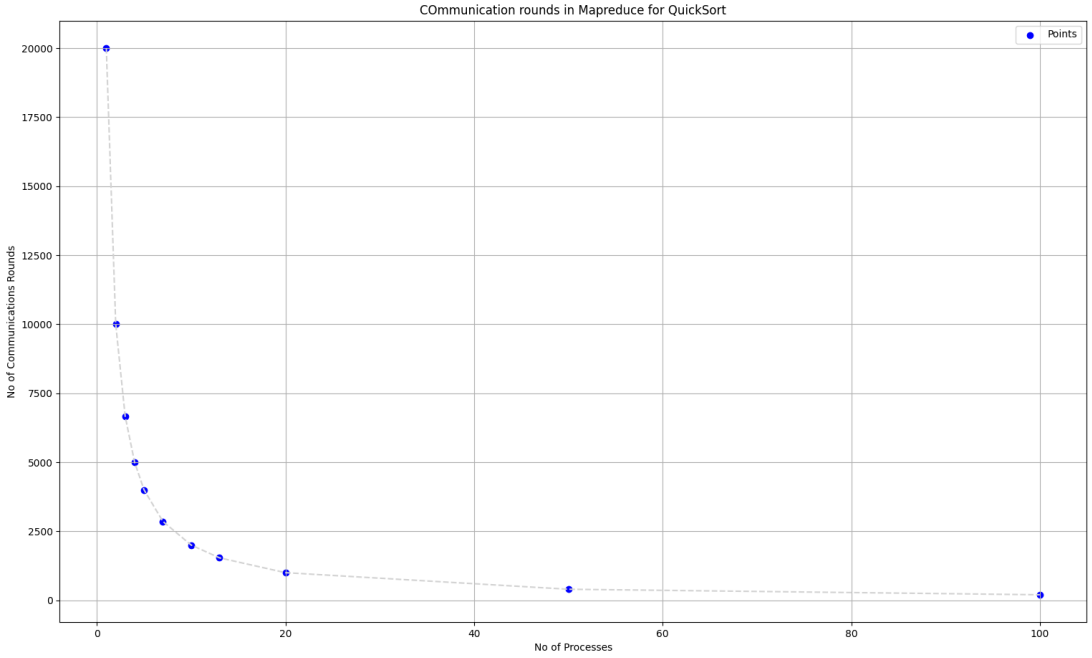


Figure 1: Communication rounds vs Number of Processes

5.5 Reasons Behind the Trend

1. Parallel Data Splitting
As the number of processes increases:

- Each process handles a smaller subset of the data.
- *Data per mapper* $= \frac{\text{Total Elements}}{\text{No. of Mappers}}$
- This reduces the number of key-value pairs emitted per mapper, thereby reducing total communication.

2. Pivot-Based Bucket Partitioning

- Each mapper emits one pivot (Job 1), and sends values to buckets (Job 2) based on globally selected pivots.
- As mappers increase, bucket distribution becomes finer.
- This makes sorting more localized and reduces inter-process shuffling.

3. Diminishing Returns

- After ≥ 20 processes, adding more:
  - **The graph is Hyperbolic around the origin in the (+ , +) quadrant of the graph**

    - Reduces per-mapper workload, but
    - Increases overhead from task setup, coordination, and I/O.
  - The fixed input size ($N = 10,000$) starts dominating, leading to a saturation in performance gains.

5.6 Insights and Implications

- **Optimal Range:** For this workload, the optimal number of processes is between 10 and 20.
- **Scalability:** MapReduce Quicksort scales well initially, but scalability is sublinear (logarithmic rather than linear).
- **Data Distribution Impact:** The mixed-range distribution simulates skewed datasets. Pivot sampling strategy still handles partitioning reasonably well.

6. OBSERVATION AND ANALYSIS: ASYNCHRONOUS MPI COMMUNICATION

1. **Local Data Generation**
   Each process generates W random integers locally using `generateData()`.
2. **Pivot Selection**
   Each process randomly selects one pivot from its local data.
3. **Asynchronous Gathering of Pivots**
   All processes send their selected pivot to process 0 using `MPI_Igather`.
   Process 0 waits for all pivots, then sorts them.
4. **Asynchronous Broadcast of Global Pivots**
   Process 0 broadcasts the sorted global pivots to all processes using `MPI_Ibcast`.
5. **Local Rank Computation**
   Each process computes the local rank of each pivot — the number of local elements less than each pivot.
6. **Asynchronous Reduction to Root**
   All local ranks are asynchronously reduced to a global rank vector at process 0 using `MPI_Ireduce`.
7. **Data Bucketing**
   Each process partitions its local data into $P + 1$ buckets based on the pivot values. Each bucket corresponds to a range destined for a specific process.
8. **Asynchronous Sending of Buckets**
   Each process sends the appropriate bucket to each target process using non-blocking `MPI_Isend`.
   Bucket size is sent first, followed by the actual data.
9. **Asynchronous Receiving of Buckets**
   Each process receives bucket sizes and data from other processes using `MPI_Recv`

and `MPI_Irecv`.
All received data is merged into a local array `received_data`.

10. **Local Sorting**
    Each process sorts its merged `received_data` locally.

11. **Asynchronous Gathering of Sorted Data**
    Process 0 collects sorted segments from all processes.
    Each process sends its size and data using `MPI_Isend`.
    Root combines all segments to obtain the fully sorted array.

12. **Finalization**
    `MPI_Finalize()` is called to cleanly end the MPI environment.

Outcome:
The data is globally sorted using a parallel approach based on random sampling, pivot partitioning, and asynchronous MPI communication.

7. ISSUES FACED AND ADVANTAGES OF ASYNCHRONOUS MPI COMMUNICATION

7.1 Problem Encountered with Synchronous Communication

In the initial implementation using synchronous MPI primitives (`MPI_Send`, `MPI_Recv`), the system encountered segmentation faults due to unmatched send/receive operations. This occurred because not all processes had data to send to every other process—some buckets were empty—yet receive calls were made unconditionally.

These dangling `MPI_Recv` calls resulted in:

- Blocking behavior waiting for messages that were never sent,
- Undefined behavior and memory corruption,
- Process crashes and segmentation faults.

Such issues are particularly prevalent in sorting algorithms involving data-dependent communication patterns, where only a subset of processes may communicate based on the data distribution.

7.2 Benefits of Asynchronous Communication

Switching to non-blocking MPI communication primitives—`MPI_Isend`, `MPI_Irecv`, `MPI_Igather`, `MPI_Ibcast`, and `MPI_Ireduce`—resolved these problems and brought several key advantages:

- **Overlapping Computation and Communication:** Non-blocking operations allow computation to proceed while communication is pending, enhancing performance.
- **Controlled Communication Timing:** All `MPI_Isend` operations can be issued before corresponding `MPI_Irecv` calls, preventing premature or unmatched receives.

- **Deadlock Avoidance:** Processes do not block indefinitely, eliminating circular wait conditions and race hazards.
- **Graceful Handling of Empty Buckets:** Zero-sized messages are detected and skipped or sent conditionally, preventing communication mismatches.

These improvements not only stabilized the program but also improved its efficiency and scalability.

7.3 Key Lessons and Takeaways

- **Asynchronous communication is essential** for algorithms with unpredictable communication patterns, such as parallel sorting.
- Ensure that for every `MPI_Recv`, there is a matching `MPI_Send`; if this cannot be guaranteed, prefer `MPI_Irecv` with size checks.
- Use `MPI_Wait` or `MPI_Waitall` judiciously to manage and complete communication events.
- Asynchronous design not only enhances fault tolerance but also enables better performance through reduced synchronization overhead.

Transition taken: The transition from synchronous to asynchronous MPI communication made the parallel sorting algorithm robust, deadlock-free, and communication-efficient.

8. CONCLUSION AND COMPARISON: MPI VS. MAPREDUCE IN PARALLEL SORTING

8.1 Conclusion

MPI-Based Sorting

The MPI-based implementation provided a highly efficient and fine-grained control over parallel sorting through direct message-passing. Key takeaways include:

- Asynchronous communication using `MPI_Isend` and `MPI_Irecv` enabled overlap of computation and communication, resulting in robust and performant execution.
- The algorithm scaled well with increasing number of processes up to a saturation point, beyond which the communication overhead offset the benefits.
- Randomized pivot selection, local rank computation, and bucket redistribution were effectively managed through collective operations like `MPI_Igather`, `MPI_Ibcast`, and `MPI_Ireduce`.

MapReduce-Based Quicksort

The MapReduce-based version abstracted the complexity of communication and provided a fault-tolerant, data-parallel framework suitable for large-scale distributed systems. Key observations include:

Aspect	MPI	MapReduce
Programming Model	Low-level message passing with explicit control over communication and synchronization.	High-level abstraction with map and reduce functions; hides communication details.
Communication Control	Full control (e.g., `MPI_Isend`, `MPI_Irecv`); complex but flexible.	Implicit communication through shuffling and partitioning phases.
Fault Tolerance	Limited; failure of one process can halt the system.	Built-in fault tolerance; failed tasks are automatically retried.
Scalability	Scales well on tightly coupled systems (e.g., HPC clusters).	Highly scalable on commodity hardware and distributed cloud environments.
Overhead	Low overhead but requires manual memory and sync management.	Higher overhead due to framework orchestration and disk I/O.
Suitability for Sorting	Excellent for fine-grained, iterative, and latency-sensitive sorting tasks.	Suitable for large, batch-oriented, and fault-tolerant sorting jobs.
Ease of Use	Steep learning curve; error-prone in complex scenarios.	Easier for data scientists and engineers with high-level interfaces.

Table 1: Comparison between MPI and MapReduce for Parallel Sorting

- Communication rounds significantly decreased as the number of mappers increased, exhibiting logarithmic scalability.
- Pivot sampling and bucket partitioning in the map phase enabled a reasonably balanced distribution despite skewed input data.
- Although initial gains from parallelism were strong, performance gains flattened beyond a certain number of processes due to coordination and I/O overhead.

8.2 Comparison: MPI vs. MapReduce

8.3 Final Insight

MPI is ideal for high-performance computing environments requiring tight control, low latency, and synchronous or asynchronous parallelism. In contrast, MapReduce is preferred for large-scale data processing over distributed systems, emphasizing fault tolerance and scalability at the cost of communication overhead. The choice depends on application scale, environment constraints, and performance goals.

ACKNOWLEDGMENT

We, Shantanu Pathak and Rutuja Deshpande, would like to express our sincere gratitude to our Professor Kishore Kothapalli for his invaluable guidance, insightful feedback, and constant encouragement throughout this project. His expertise in parallel and distributed systems greatly enriched our understanding and helped shape the direction of our work.

The complete source code and documentation are available on GitHub:
MPI and MapReduce