Scalability Benchmark: Multi-Node Sorting on Hadoop and Spark

Chi Zhang
Department of Mathematics
Indiana University
Bloomington, IN
czh4@iu.edu

Krishna Priya
Department of ISE
Indiana University
Bloomington, IN
fkrishna@iu.edu

Sanjana Reddy Nenturi
Department of ISE
Indiana University
Bloomington, IN
snenturi@iu.edu

Sai Shriya Surla
Department of Computer Science
Indiana University
Bloomington, IN
ssurla@iu.edu

***Abstract*—This project presents an extensive benchmarking study of four cloud computing environments, Hadoop and Spark deployed in Docker containers and on Jetstream2 cloud infrastructure, using the TeraSort benchmark, a shuffle-heavy sorting task. By systematically varying the number of worker nodes and analyzing performance, we identify the conditions under which each platform is most suitable. Contrary to prior findings by Samadi *et al.* [1], who reported Spark's general superiority over Hadoop, our results demonstrate that in Dockerized environments, Hadoop exhibits more stable and efficient performance than Spark for shuffle-heavy workloads. This challenges the common intuition that Spark consistently outperforms Hadoop. Notably, Hadoop in Docker achieved the best overall performance for our benchmark. A comprehensive analysis, along with refined metrics such as speedup, parallel efficiency, and communication penalty, is provided to contextualize these findings and guide practical deployment decisions.**

***Index Terms*—Distributed Computing, Apache Spark, Apache Hadoop, TeraSort, TeraGen, Scalability, Cloud Computing, Jetstream2, Big Data Benchmarking, Shuffle Bottlenecks**

I. Introduction

The exponential growth in digital data has made large-scale data processing vital for modern computing systems. As organizations increasingly rely on data-driven decision-making, the ability to efficiently process massive datasets across distributed infrastructures has become critical. Distributed computing frameworks such as Apache Hadoop and Apache Spark have emerged as foundational technologies for scalable data processing, leveraging clusters of commodity hardware to enable parallelism and fault tolerance.

While both Hadoop and Spark are widely adopted and extensively benchmarked for a variety of workloads, comparatively less attention has been devoted to understanding how these frameworks perform across heterogeneous deployment environments, particularly when transitioning from containerized setups (e.g., Docker) to cloud-native infrastructures such as Jetstream2. This is a critical gap, as real-world software development pipelines often involve developing, testing, and benchmarking distributed systems in local, containerized environments, followed by deployment in production-scale cloud platforms. Performance disparities between these two contexts can significantly affect resource provisioning, scalability planning, performance tuning, and overall cost efficiency.

In this project, we perform a comparative benchmarking study of Hadoop and Spark using the TeraSort benchmark across Hadoop and Spark on Docker and Jetstream2. We assess scalability by varying the number of worker nodes and analyze performance to reveal how deployment choices impact efficiency and bottlenecks. Our contributions are summarized in Table I

II. Background and Motivation

Sorting is a foundational operation in distributed data processing, which supports tasks such as indexing, query optimization, and data integration. Among large-scale frameworks, Apache Hadoop and Apache Spark have become the actual platforms for executing such workloads in parallel environments. While Hadoop, built on the MapReduce paradigm [2], emphasizes fault tolerance and robustness in batch processing, Spark [3] offers faster performance through in-memory execution, particularly benefiting iterative and shuffle-heavy jobs.

Although previous benchmarking studies have compared Hadoop and Spark across a range of workloads, including classification and text processing [1], the performance of these systems in sorting tasks - especially under shuffle-intensive conditions - remains less thoroughly examined. Sorting benchmarks like TeraSort stress both I/O and network subsystems, making them well-suited for identifying infrastructure-level bottlenecks.

This project focuses on systematically evaluating Hadoop and Spark on the TeraSort benchmark across two infrastructure setups: containerized clusters (Docker) and cloud native instances (Jetstream2). By analyzing various performance metrics and scalability metrics, our objective is to provide targeted insights into performance scalability behavior and practical limitations that arise in real deployments.

III. Related Work

Performance benchmarking of distributed data processing frameworks such as Apache Hadoop and Apache Spark has been an area of study in recent years. A number of comparative evaluations have been conducted across a variety of workloads. However, few studies isolate the performance of sorting workloads, particularly under different infrastructure contexts such as Docker-based containers and cloud-native

TABLE I: Summary of Contributions and Impact

Contribution	Description and Impact
Comprehensive Benchmarking with Diverse Metrics	We benchmarked Hadoop and Spark in four environments (Docker and Jetstream2) using the shuffle-heavy TeraSort benchmark. Metrics included wall time, shuffle throughput, I/O throughput, aggregated resource utilization, memory utilization, and derived metrics (speedup, parallel efficiency, communication penalty), offering a detailed analysis of scalability and infrastructure effects.
Breaking Naive Intuition	While prior studies [1] found Spark generally superior, we observed that in Dockerized environments, Hadoop outperformed Spark in stability and scalability for shuffle-heavy workloads, challenging the assumption that Spark is always the better choice.
Practical Deployment Recommendations	We recommend Hadoop in Docker for small-scale local sorting due to its stable performance and low communication penalty, and Spark on Jetstream2 for batch jobs despite some cloud contention. These findings guide deployment choices based on workload and infrastructure. We also recommend systems in Table II for shuffle–heavy tasks at small scale.

clusters. Samadi et al. [1] evaluated multiple HiBench jobs and confirmed Spark's general speed advantage, but noted its higher memory demands. However, the study did not specifically dissect the shuffle-intensive nature of sorting workloads or examine performance in diverse deployment contexts. Liu [4] specifically examined WordCount and TeraSort, offering baseline comparisons but limited discussion of scaling dynamics or infrastructure impact. Liu's work primarily focused on fixed-node configurations, leaving questions about elasticity and cluster efficiency unanswered. Pavlo et al. [5] offered an early comparison of parallel data processing systems using both MapReduce and parallel DBMSs. Although their study predated modern Spark implementations, it identified I/O bottlenecks and shuffle communication. In contrast to these studies, our work takes a closer look specifically at TeraSort performance, comparing Hadoop and Spark in two different environments—Docker containers and the Jetstream2 cloud. Unlike earlier research, we focus on how factors like infrastructure setup, number of nodes, shuffle stage behavior, and execution time affect performance when running the same sorting task. By carefully controlling these conditions, our study helps explain how the choice of deployment environment can impact the efficiency of distributed sorting. This offers useful insights for developers and engineers who build and manage large-scale data systems in both cloud and local environments.

IV. Design

This project outlines the performance evaluation of Hadoop MapReduce and Apache Spark with the TeraSort benchmark to test how execution time and resource usage change when using more nodes in the cluster. The data was generated using TeraGen and TeraSort was implemented upon it. Various metrics were considered for comparison of Hadoop MapReduce and Spark. The full code and data are available on GitHub.[1]

A. Tools, Frameworks and Environments

Our benchmarking setup includes Apache Hadoop, Apache Spark, the TeraSort benchmark, and the TeraGen data generator, deployed across two cloud computing environments: Docker and Jetstream2. Table II provides a detailed overview of all components, configurations, and benchmarking environments used in this study.

[1]https://github.com/chizhang24/ECCFinalProject

B. Experiment Design

The experimental workflow (Fig. 1) consists of the following steps:

1) **Dataset Preparation:** We use TeraGen to generate a uniform 1GB dataset (10 million records), which serves as input for all TeraSort experiments.
2) **Cluster Setup:** Both Docker and Jetstream2 clusters are configured for Hadoop and Spark, with 1–6 worker nodes to examine scalability.
3) **Execution:**
   - *Hadoop:* TeraSort jobs are executed via the built-in Hadoop examples JAR, reading and writing data to HDFS.
   - *Spark:* PySpark TeraSort is run using `sortBy()`, submitted via `spark-submit` with YARN resource controls.

   Both frameworks are benchmarked under identical conditions across Docker and Jetstream2.
4) **Benchmarking:** Each test is repeated three times to compute stable average wall times. Derived metrics—speedup, parallel efficiency, and communication penalty—are calculated for deeper scalability analysis.
5) **Analysis:** We compare performance based on wall time, shuffle and I/O throughput, CPU/memory utilization, and derived metrics to identify bottlenecks and evaluate framework efficiency for shuffle-heavy workloads.

V. Evaluation and Results

To evaluate the performance and scalability of Apache Hadoop and Apache Spark for shuffle–heavy tasks on both Jetstream 2 and Docker, we use the following evaluation metrics. These metrics are chosen because they show the computation and resource behavior of the systems when the data size and node number change. The goal is to see how the 4 platforms behave in real–world tasks.

However, due to the complexity of cloud computing infrastructures, evaluation metrics like shuffle throughput and I/O throughput show underperformance in all 4 platforms. To have deeper understanding on how infrastructure masks evaluation metrics, we also introduce some refined *derived metrics*.

TABLE II: Overview of Sorting Algorithms, Cloud Environments, Benchmarking Setups, and Experiment Design

Category	**Component**	**Description and Configuration**
Sorting Algorithms	Apache Hadoop	MapReduce-based batch framework. TeraSort jobs executed via Hadoop's built-in examples JAR; results stored in HDFS.
	Apache Spark	In-memory distributed framework using RDDs. TeraSort implemented in PySpark with `sortBy()`; submitted via `spark-submit` over YARN.
Data Generation and Benchmark	TeraGen	Hadoop tool generating a 1GB synthetic dataset (10 million records); input to TeraSort in all experiments.
	TeraSort	Benchmark stressing shuffle and sort phases; tests I/O and network performance of frameworks.
Cloud Computing Environments	Docker	Container orchestration platform for local deployment of Hadoop and Spark clusters (1–6 workers); supports reproducible experiments.
	Jetstream2	Cloud platform (scientific computing) with 1 master + 1–6 worker nodes (Ubuntu VMs). Hadoop and Spark deployed on YARN-managed clusters; HDFS stores data and results.
Benchmarking Environments	Hadoop in Docker	Hadoop cluster (1–6 workers) in Docker; TeraSort benchmark run with data in HDFS.
	Spark in Docker	Spark cluster (1–6 workers) in Docker; PySpark TeraSort with HDFS input/output.
	Hadoop on Jetstream2	Cloud Hadoop cluster (1–6 workers); TeraSort executed with 2 map tasks for data splits; results stored in HDFS.
	Spark on Jetstream2	Cloud Spark cluster (1–6 workers); PySpark TeraSort run via YARN; output written to HDFS.
Experiment Components	Scaling Tests	Number of worker nodes varied from 1 to 6 across all environments to assess scalability.
	Execution and Repeats	Each experiment run 3 times; average wall time calculated for stability.
Metrics and Analysis	Primary Metrics	Wall time, shuffle throughput, I/O throughput, CPU and memory utilization recorded.
	Derived Metrics	Speedup, parallel efficiency, and communication penalty calculated to diagnose scalability bottlenecks.

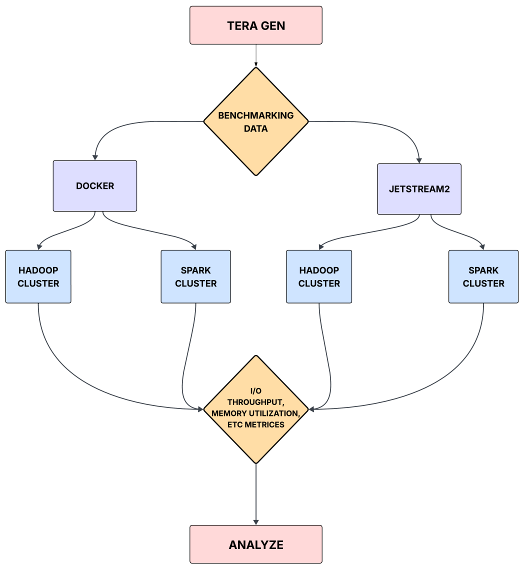


Fig. 1: Overview

A. Evaluation Metrics on Performance

Wall Time (↓)

We define our first metric, **Wall Time** by

$$T_{\text{total}} = T_{\text{start}}^{\text{job}} - T_{\text{end}}^{\text{job}}. \tag{1}$$

The wall time is the total elapsed time from the moment when the job is submitted to the moment when it is completed, *i.e.*, the total time duration of the TeraSort job. The wall time is a direct indicator of the overall job performance, and captures overheads in computation, communication and I/O.

Shuffle Throughput (↑)

Data shuffling is heavy in TeraSort, so we use **shuffle througput** [6] to characterize the performance of the 4 platforms in data shuffling.

We define shuffle throughput by

$$\text{Shuffle Throughput} = \frac{D_{\text{shuffle}}}{T_{\text{shuffle}}}, \tag{2}$$

where D_{shuffle} is the total volume of data shuffled (in MB or GB), and T_{shuffle} is the total time spent in the shuffle phase.

Shuffle throughput measures how efficiently the network resources are utilized, and how fast the data is redistributed across nodes after map tasks.

I/O Throughput (↑)

Distributed workloads are frequetly I/O–bound, especially in batch processing systems [7]. So we choose *I/O–Throughput* to monitor the disk performance in shuffle–heavy jobs.

I/O–Throughput is defined by

$$\text{I/O–Throughput} = \frac{D_{\text{I/O}}}{T_{\text{I/O}}},$$

where $D_{\text{I/O}}$ is the total amount of data read or written to disk (in MB/GB), and $T_{\text{I/O}}$ is the total time spent on I/O operations.

I/O–Throughput tracks how fast system reads from and writes to storage. In Apache Hadoop, which heavily relies on Hadoop Distributed File System (HDFS), disk performance is a major factor in overall performance. In Apache Spark, although data is in–memory in nature, disk I/O is still important whence data does not fit into memory.

Aggregated Resource Utilization ($\downarrow$)

To understand the true computational cost to complete the job, and optimize cost and performance, we choose **Aggregated Resource Utilization** to measure the total CPU effort across all nodes.

Aggregated Resource Utilization is defined by

$$\text{AggResUtil} = \sum_{i=1}^{N} C_i \times T_i,$$

where C_i is the number of vritual CPU cores (vCores) allocated to worker i, and T_i is the total time when worker i was active, while N is the total number of workers in the job.

The Aggregated Resource Utilization also describes parallelism quality, *e.g.*, high GPU resource usage with elongated wall time means inefficient parallel computation.

Memory Utilization ($\downarrow$)

Balancing memory and computation is vital in efficient large–scale data processing [8]. We then use **Memory Utilization** as a metric, to understand whether memory is a limiting factor in shuffle–heavy jobs.

Memory utilization is defined as

$$\text{MemUtil} = \frac{1}{N}\sum_{i=1}^{N} \text{RAM}_i,$$

where RAM_i is the memory (in MB/GB) used by the ith worker, and N is the total number of workers.

In our task, especially in Apache Spark, the execution is in–memory, thus Memory Utilization is closely related to performance. Although Apache Hadoop is more disk–based, memory is still crucial for buffering. In both cases, Memory Utilization describes memory pressure in tasks and is an indicator of job failure or unexpected shutdown.

B. Derived Metrics on Scalability

While the metrics defined above: Wall Time, Shuffle Troughput, I/O Throughput, Aggregated Resource Utilization, and Memory Utilization offer insights into system performance and resource behavior, but they do not fully reveal scaling efficiency or the nuanced trade-offs arising as system size increases.

To understand how well additional resources are utilized and see deeper bottlenecks, we need the derived metrics below.

Speedup ($\uparrow$)

As a core metric in parallel computing [9], **speedup** shows directly the gaining of adding more resources in parallel computing, and has become a standard in parallel computing.

Speedup is defined by

$$\text{Speedup}(N) = \frac{T_{\text{wall}}(1)}{T_{\text{wall}}(N)} \tag{3}$$

where $T_{\text{wall}}(1)$ is the wall time using 1 worker, while $T_{\text{wall}}(N)$ is the wall time using N workers in the system. The wall time is defined as in Eq. (1). The speedup for ideal scaling is linear.

Parallel Efficiency ($\uparrow$) To measure how effectively each additional worker contributes to the job, we use the **parallel efficiency** defined below

$$\text{Paral. Efficiency} = \frac{\text{Speedup}(N)}{N},$$

where $\text{Speedup}(N)$ is the speedup when there are N workers in the system, as defined in Eq. 3.

Parallel efficiency highlights scaling inefficiencies that may not be visible in speed-up alone. The ideal parallel efficiency is 1.

Communication Penalty ($\downarrow$)

To quantify performance degradation caused by network congestion or inefficiencies during the shuffle phase, we use **communication penalty**, defined by

$$\text{CommPenalty}(N) = T_{\text{wall}} \times (1 - \frac{\text{Shuffle Throughput (N)}}{\max_k \text{Shuffle Throughput } (k)})$$

where T_{wall} and Shuffle Throughput (N) are defined in Eq. 1 and Eq. 2, and $\max_k$ Shuffle Throughput (k) is the highest shuffle throughput across all k from 1 to N.

C. Results

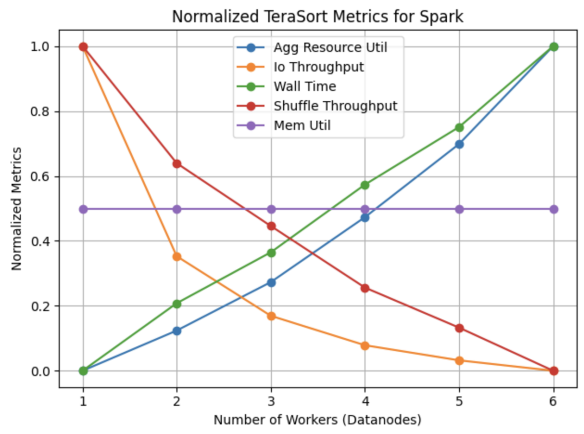
We benchmarked Spark and Hadoop clusters across two environments: Docker (container-based local setup) and Jetstream 2 (cloud-based virtual machines). For each framework, we measured wall time, shuffle throughput, I/O–throughput, aggregated resource utilization, and memory utilization. Each configuration, ranging from 1 to 6 workers, was executed once. The reported metrics reflect the outcomes of these single-run experiments.

1) Spark Performance on Docker and Jetstream 2: Spark showed consistent behavior across both environments. In the Docker setup, the wall time decreased steadily as the number of workers increased, suggesting effective distribution of tasks. On Jetstream 2, the wall time stayed mostly stable across all configurations, indicating that Spark handled resource variability in the cloud well.

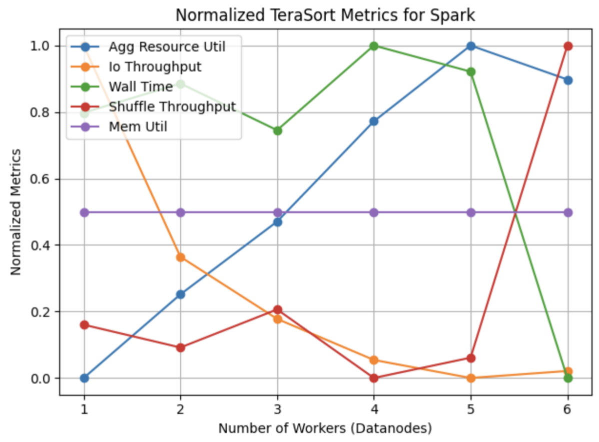
In terms of shuffle throughput, Spark experienced a slight drop on Docker as the number of workers increased, likely due to communication overhead. On Jetstream 2, this metric remained more stable, possibly due to balanced network performance across virtual machines.

I/O throughput showed a downward trend in both environments as more workers were added. This could be due to increased fragmentation or parallel I/O overhead during the sorting phase.

Aggregated resource utilization increased as the number of workers increased, showing that Spark scaled its use of available compute effectively. Memory usage remained constant across all configurations since the executor memory was fixed throughout the runs.

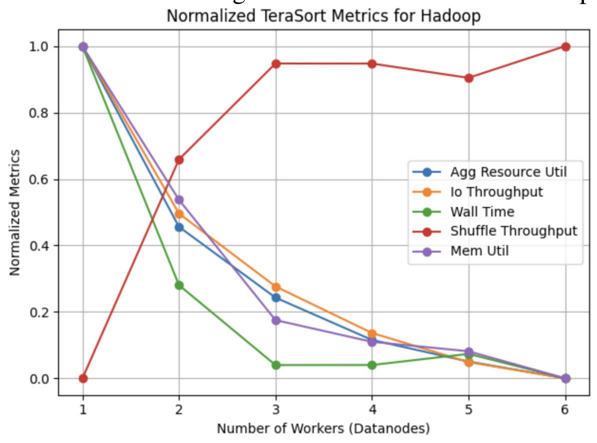


(a) Spark metrics on Docker (normalized)

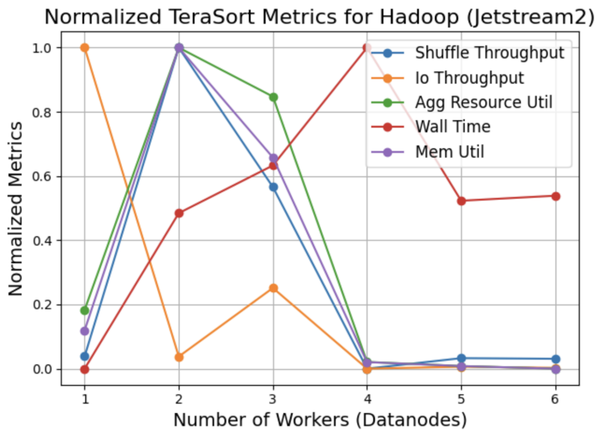


(b) Spark metrics on Jetstream 2 (normalized)

Fig. 2: Performance metrics for Spark across Docker and Jetstream 2 environments.



(a) Hadoop metrics on Docker (normalized)



(b) Hadoop metrics on Jetstream 2 (normalized)

Fig. 3: Performance metrics for Hadoop across Docker and Jetstream 2 environments.

2) Hadoop Performance on Docker and Jetstream 2: Hadoop showed improvements in wall time up to 3 workers, but further scaling provided limited benefits. In some cases, especially on Jetstream 2, wall time even increased beyond 3 workers. This instability is likely caused by overhead in HDFS and job scheduling when more workers are added in a virtualized environment.

On Docker, shuffle throughput improved steadily with the number of workers, suggesting better parallelism. However, on Jetstream 2, the values were inconsistent and noisy. This may be due to unpredictable performance of network and disk resources on virtual machines.

I/O throughput decreased in both environments with more workers, and the drop was sharper on Jetstream 2. This suggests that resource sharing in the cloud introduced additional latency or I/O contention.

Aggregated resource utilization dropped in Docker as the cluster grew, while Jetstream 2 showed more irregular patterns. These results indicate that Hadoop did not scale efficiently under changing system conditions.

Memory usage dropped on Docker with higher worker counts due to smaller task slices, while on Jetstream 2 it fluctuated, possibly due to speculative execution or uneven workload allocation.

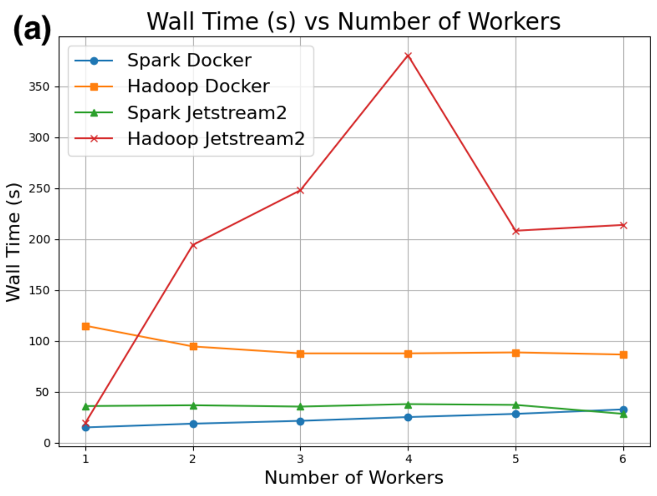
D. Deeper Analysis

We present the trends of wall time along with the three derived metrics introduced in Sec. V-B, plotted against the number of workers. This analysis aims to provide deeper insight into both the scalability characteristics and the infrastructure-induced bottlenecks observed across all four systems: Spark in Docker, Hadoop in Docker, Spark on Jetstream2, and Hadoop on Jetstream2.

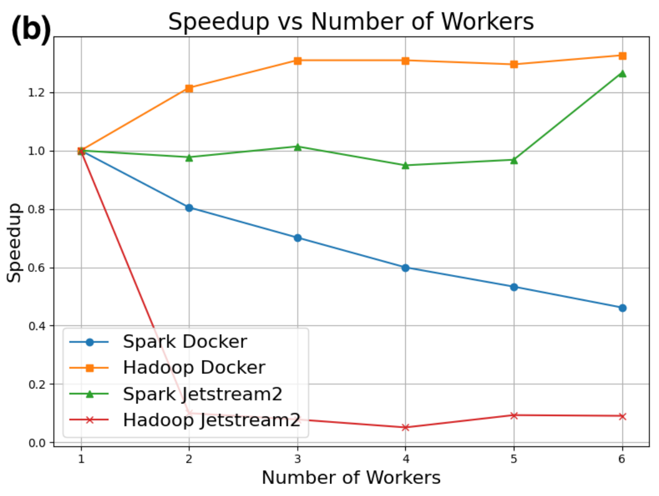
Fig. 4 illustrates how these derived metrics evolve as the cluster scales, highlighting key patterns and deviations that emerge with increasing parallelism.

In Fig. 4(a), we see that adding more workers does not guarantee faster performance. Only Hadoop in Docker shows minor reduction in Wall time as more workers are added, while in other 3 systems, the wall time is elongated as more workers are added. We also note that Hadoop on Jetstream 2 is very fragile, likely due to cloud resource contention. Fig. 4(a) shows clearly that infrastructure bottlenecks hinder real–world scaling in cloud computing.

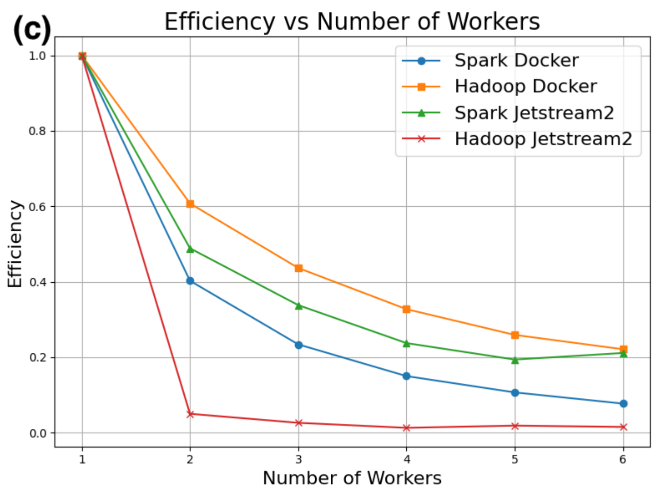
In Fig. 4(b), we see that Spark in Docker and Hadoop on Jetstream 2 show decay in speedup as more workers are added, while Hadoop in Docker and Spark on Jetsteam 2 gain minor



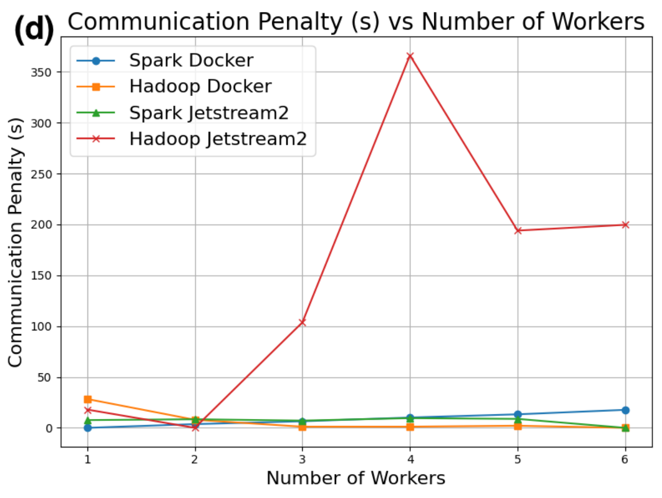
(a) Wall Time



(b) Speedup



(c) Efficiency



(d) Communication Penalty

Fig. 4: Derived Metrics for Different Numbers of Workers across All 4 Platforms: (a) Wall Time, (b) Speedup, (c) Efficiency, and (d) Communication Penalty

Platform	Behavior	Major Bottleneck	Conclusion
Hadoop Docker	Stable wall time, low Comm-Penalty	I/O Throughput	Best for small-scale sorting on local clusters
Spark Jetstream2	Flat wall time, low Comm-Penalty	Cloud contention	Acceptable for batch jobs
Spark Docker	Wall time decays, Comm-Penalty grows	Network shuffle congestion	Good for simple deployment
Hadoop Jetstream2	Highly unstable	Disk and network congestion	Not recommended without heavy tuning

TABLE III: Summary of platform behavior, bottlenecks, and recommended use cases

speedup. The Spark in Docker fails to scale up due to its container nature, while Hadoop on Jetstream 2 fail to scale up due to cloud contention and congestion. From this example, we see that scability in real world is far beyond ideal linear scaling law, and infrastructure bottlenects usually mask scalability.

In Fig. 4(c), we see that all four systems experience sharp efficiency loss when more workers are added, while ideal parallel efficiency is 1. Among all 4 systems, Hadoop in Docker drops moderately in efficiency, which is the relatively most stable one among all systems, showing that Hadoop in Docker has better resource use and less resource contention due to the containerized nature and disk–focused I/O logic. This again shows that in real–world cloud computing, scaling systems up without solving bottlenecks could possibly lead to resource waste.

In Fig. 4(d), we see that Hadoop in Docker has low communication penalty, but still suffers from disk I/O. Spark on Jetstream 2 handles shuffle well at moderate scale, while Spark in Docker is penalized with more workers. Hadoop on Jetstream 2 is extremely volatile and network–bound, as the communication penalty explodes at 4 workers.

In summary, we have shown that all 4 systems could not achieve perfect scaling in real–world cloud computing environments, and suffer severe infrastructure–bound bottlenecks. However, among the 4 systems, we conclude that Hadoop in Docker is most suitable for shuffle–heavy task (TeraSort) at small sacale (1GB data).

VI. Conclusion

This study provided a comprehensive benchmarking and analysis of Hadoop and Spark deployed in Docker containers and on Jetstream2—using the shuffle-heavy TeraSort benchmark.

The analysis summarizes the key observations and recommendations for the evaluated experiments. Hadoop Docker demonstrated stable wall time and low communication penalty, making it the best option for small-scale sorting tasks on local clusters, despite I/O throughput being its main limitation. Spark on Jetstream2 maintained flat wall time and low communication penalty but was affected by cloud contention, making it suitable for batch jobs. Spark on Docker exhibited degrading wall time and increasing communication penalty due to network shuffle congestion, though it remains a useful choice for simple and lightweight deployments. Finally, Hadoop on Jetstream2 was highly unstable, suffering from disk and network congestion, and is therefore not recommended without extensive tuning. The results serve as a practical summary for selecting the most appropriate platform based on specific workload and infrastructure requirements.

Importantly, while Samadi *et al.* [1] concluded that Spark generally outperforms Hadoop, our results reveal that in Dockerized environments, Hadoop outperforms Spark in stability and overall suitability for shuffle-heavy workloads. This challenges the naive intuition that Spark is universally superior and underscores the need to consider deployment context carefully. Beyond the specific system recommendations in Table II, our work highlights broader implications: deployment environment has a significant impact on distributed framework performance, and platform selection should be guided not only by framework design but also by infrastructure characteristics and workload nature.

VII. Discussion

A. Challenges

While this study provided valuable insights into the performance and scalability of Apache Hadoop and Apache Spark on shuffle-heavy workloads, several challenges were encountered during the experiments. **The experimentation on Jetstream required avoiding misconfiguration of hostnames.** Another recurring issue was **resource constraint errors, particularly while configuring YARN container memory and core limits.** If these values were misaligned with the Spark or Hadoop requirements, jobs failed at launch. We encountered **safe mode lock errors, causing HDFS write operations to be blocked until the cluster stabilised.** On the Spark side, **managing executor memory and avoiding memory overhead errors required trial-and-error tuning.**

B. Future Work

Future research can take this finding in a variety of directions. To begin, **studies may be carried out on bigger datasets (e.g., 10GB–1TB)** to better understand how Hadoop and Spark grow in truly large-scale situations. **Incorporating container orchestration technologies like Docker Swarm or Kubernetes** would allow for a more realistic evaluation of containerized distributed systems. Additionally, **tweaking Spark and Hadoop configurations, such as executor memory, shuffle parallelism, and compression settings**, could give insights into improving performance and cost effectiveness.

Migrating from Docker-based simulations to real distributed computing platforms or production clusters would help eliminate the virtualization overhead and provide more accurate performance insights. **A comparative examination of alternative cloud platforms, such as AWS or Azure**, would also aid in determining cost-performance tradeoffs in various production scenarios. Finally, **increasing the collection of performance indicators to include energy consumption and environmental effect** may give a more comprehensive knowledge of big data processing frameworks' sustainability.

Data and Code Availability

All experimental data, benchmarking scripts, and analysis code supporting the findings of this report are publicly available at:

- GitHub Repository: https://github.com/chizhang24/ECCFinalProject

The repository includes:

- Docker compose files for cluster setup,
- Hadoop env files for both Docker and Jetstream 2
- Benchmarking scripts for Hadoop and Spark TeraSort,
- Raw and processed result data for all 4 systems,
- Analysis notebooks and plotting scripts.

Author Contributions

- **Chi Zhang**: Conducted experimentations and analysis on Docker-based clusters, including benchmarking Hadoop and Spark performance locally.
- **Krishna Priya**: Led the experimentations and setup of Hadoop and Spark clusters on Jetstream2, and handled job execution in the cloud environment.
- **Sanjana Reddy Nenturi**: Analyzed Hadoop and Spark performance on Jetstream2, and helped extract and interpret key metrics from raw logs.

- **Sai Shriya Surla**: Carried out deeper analysis including speedup, parallel efficiency, and communication penalty metrics across all platforms.

References

[1] Y. Samadi, M. Zbakh, and C. Tadonki, “Comparative study between Hadoop and Spark based on Hibench benchmarks,” in *2016 2nd International Conference on Cloud Computing Technologies and Applications (CloudTech)*, May 2016, pp. 267–275. [Online]. Available: https://ieeexplore.ieee.org/document/7847709

[2] J. Dean and S. Ghemawat, “MapReduce: simplified data processing on large clusters,” *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008. [Online]. Available: https://dl.acm.org/doi/10.1145/1327452.1327492

[3] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, “Spark: Cluster Computing with Working Sets.”

[4] Y. Liu, “An empirical comparison between MapReduce and Spark : a thesis presented in partial fulfilment of the requirements for the degree of Master of Science in Information Sciences at Massey University, Auckland, New Zealand,” Ph.D. dissertation, Massey University, 2019. [Online]. Available: http://hdl.handle.net/10179/15304

[5] A. Pavlo, D. Ascaris, C. Bowers, R. Chandramouli, and M. Rittinger, “A comparison of the performance of nosql and relational databases for olap workloads,” *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 2034–2044, 2009. [Online]. Available: https://www.vldb.org/pvldb/2/vldb09-88.pdf

[6] “IOPS,” Mar. 2025, page Version ID: 1283261230. [Online]. Available: https://en.wikipedia.org/w/index.php?title=IOPS&oldid=1283261230

[7] M. Chen, S. Mao, Y. Zhang, and V. C. Leung, *Big Data: Related Technologies, Challenges and Future Prospects*, ser. SpringerBriefs in Computer Science. Cham: Springer International Publishing, 2014. [Online]. Available: https://link.springer.com/10.1007/978-3-319-06245-7

[8] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, “Heracles: improving resource efficiency at scale,” in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. Portland Oregon: ACM, Jun. 2015, pp. 450–462. [Online]. Available: https://dl.acm.org/doi/10.1145/2749469.2749475

[9] G. M. Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” in *Proceedings of the April 18-20, 1967, spring joint computer conference*, ser. AFIPS ’67 (Spring). New York, NY, USA: Association for Computing Machinery, Apr. 1967, pp. 483–485. [Online]. Available: https://dl.acm.org/doi/10.1145/1465482.1465560