International Journal of Computer Applications (0975 – 8887)
Volume 187 – No.32, August 2025

Parallel k-Means Benchmarking on a CPU-Bound Beowulf Cluster of Raspberry Pi Nodes: An MPI-based Scaling Analysis with CPU-Centric Performance Evaluation

Dimitrios Papakyriakou
PhD Candidate
Department of Electronic Engineering
Hellenic Mediterranean University
Crete, Greece

Ioannis S. Barbounakis
Assistant Professor
Department of Electronic Engineering
Hellenic Mediterranean University
Crete, Greece

ABSTRACT

This study presents an in-depth parallel benchmarking analysis of the k-Means clustering algorithm on a Beowulf cluster composed of Raspberry Pi 4B nodes, each equipped with 8GB of RAM. Leveraging MPI for distributed computation, it is systematically evaluating the algorithm's strong scaling behaviour using synthetic datasets of fixed size -75 million two-dimensional points - while varying the number of MPI processes from 2 up to 48 (with two processes per node).

The performance evaluation focuses on a detailed execution time decomposition across five key phases: *data generation*, parallel distance computation (*Compute Phase*), synchronization via MPI_Allreduce (*Sync Phase*), centroid updates (*Update Phase*), (*k-Means Phase)* and *total runtime*. Results confirm that the Compute Phase remains the dominant contributor to total runtime, consistently accounting for the majority of execution time across all configurations. Synchronization overhead increases moderately at intermediate process counts, a typical phenomenon in distributed systems, but remains manageable and does not offset the overall speedup achieved through parallelization.

The Beowulf cluster demonstrates excellent scalability and high parallel efficiency throughout the strong scaling experiments, with total runtime reduced by nearly (10×) when increasing from 2 to 48 MPI processes. Memory usage remains within physical RAM limits due to careful dataset partitioning, enabling large-scale processing on low-power ARM-based nodes.

Overall, this work highlights the feasibility and efficiency of CPU-centric, memory-aware distributed machine learning on energy-efficient Raspberry Pi clusters. The proposed benchmarking framework provides a robust and reproducible foundation for analysing algorithmic performance, scalability, and resource utilization in lightweight distributed environments, aligning with contemporary trends in edge computing and resource-constrained high-performance computing.

Keywords

Raspberry Pi 4B, Beowulf Cluster, ARM Architecture, Parallel Computing, CPU-Bound Workload, k-Means Clustering, Message Passing Interface (MPI), MPICH, Memory-Conscious Scaling, Low-Cost Clusters, Synthetic Data Benchmarking, Execution Time Analysis, Distributed Systems, HPC Performance Evaluation.

1. INTRODUCTION

The k-Means clustering algorithm is a widely used unsupervised learning method that partitions data into k distinct clusters based on feature similarity [1]. Its computational structure which is characterized by repetitive distance calculations and centroid updates, makes it highly amenable to parallel execution, especially when applied to large-scale datasets. As data volumes grow, evaluating the performance and scalability of k-Means on distributed systems becomes increasingly important for real-world applications ranging from image processing to bioinformatics and recommendation systems.

In this study, we investigate the performance characteristics of k-Means in a Message Passing Interface (MPI) environment, utilizing a Beowulf cluster composed of 24 Raspberry Pi 4B nodes, each equipped with an 8GB ARM-based processor. MPI offers a flexible model for data-parallel workloads, enabling processes to work independently while communicating minimal information during synchronization phases such as centroid updates. This design aligns well with k-Means' structure, where the bulk of the computation is spent on point-to-centroid distance calculations, a phase that can be distributed efficiently across CPU cores.

Our implementation leverages MPICH, a widely adopted MPI library, to coordinate 48 parallel processes running on the cluster. The algorithm operates on synthetic datasets of increasing size (up to 80 million points) to assess scaling behavior and memory usage. Performance is evaluated based on timing decomposition across core algorithmic phases—including data scattering, parallel computation, result gathering, and centroid updating. Metrics such as total runtime, CPU-bound phase dominance, and communication overhead are analyzed to determine the effectiveness of the parallelization strategy.

By conducting this analysis on a low-cost, ARM-based computing environment, we aim to demonstrate the feasibility and efficiency of scalable clustering in resource-constrained distributed systems. The results contribute insights into the design of memory-conscious, CPU-focused parallel algorithms suitable for educational, experimental, and lightweight production environments.

This study focuses on the deployment and performance analysis of the k-Means clustering benchmark on a Beowulf cluster consisting of 24 Raspberry Pi 4 Model B nodes, each equipped with 8GB of RAM and interconnected via Ethernet. The objective is to investigate the scalability and execution efficiency of ARM-based distributed systems under computational workloads. The evaluation highlights MPI-based parallelization, execution time decomposition, and memory-aware workload distribution, providing insights into the practical feasibility of low-power, cost-effective high-performance computing (HPC) environments.

The Raspberry Pi 4 Model B (8GB RAM), illustrated in "Figure 1", serves as the fundamental building block of the cluster. Each unit features a 64-bit quad-core ARMv8 Cortex-A72 CPU clocked at 1.5 GHz [2], [3]. Its affordability and accessibility were key factors in its selection as the base hardware for constructing a high-performance computing cluster, enabling a systematic evaluation of its capabilities in parallel processing and distributed computing environments.

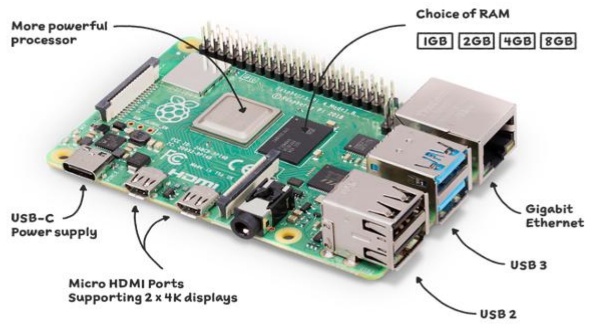


Figure 1: Single Board Computer (SBC) - Raspberry Pi 4 Model B [2], [3].

2. SYSTEM DESCRIPTION

2.1 Hardware Equipment

At the heart of this project lies a cost-efficient yet powerful Beowulf cluster, built from 24 Raspberry Pi 4B units (8GB RAM each), as illustrated in "Figure 2". A single Raspberry Pi serves as the master node, orchestrating job scheduling and resource distribution, while the remaining 23 nodes form the computational backbone of the system, operating in parallel under MPI-based coordination.

The cluster is neatly structured into four modular stacks, each hosting six Raspberry Pis, and connected via Gigabit Ethernet switches (TP-Link TL-SG1024D) that deliver up to 1 Gbps of bandwidth per node. This setup enables efficient and scalable inter-node communication, providing an HPC-like experience within an ARM-based low-cost infrastructure "Figure 2".

To ensure stable power delivery, the system relies on two industrial-grade switch-mode power supplies (60A, 5V), tuned to 5.80V to compensate for voltage drops due to cable length and power distribution. In terms of storage, the cluster is equipped with high-speed NVMe SSDs: the master node hosts a 1TB Samsung 980 PCIe 3.0 NVMe drive, while each worker node is outfitted with a 256GB Patriot P300 NVMe M.2 SSD, enabling fast local I/O performance and smooth handling of large-scale synthetic data used in the benchmarks.

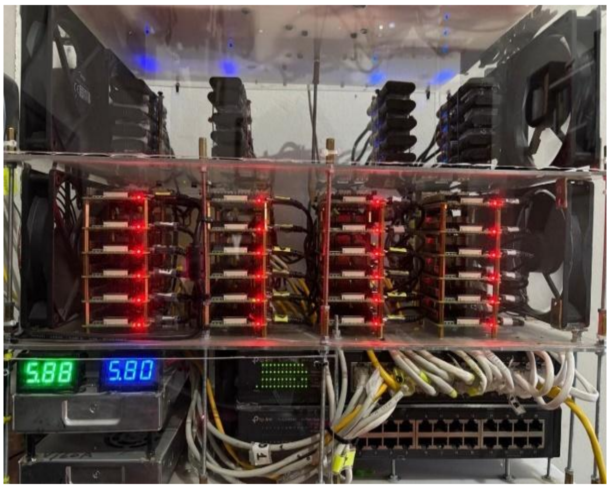


Figure 2: Deployment of the Beowulf Cluster with (24) RPi-4B (8GB).

2.2 Software Environment and Toolchain

The software environment of the Beowulf cluster was carefully designed to ensure performance, compatibility, and reproducibility in a distributed parallel computing context. Each of the 24 Raspberry Pi 4B nodes (8GB RAM, ARM Cortex-A72 @ 1.5 GHz, 64-bit) was configured with a consistent and stable system stack to support high-performance scientific computation.

The operating system installed on all nodes was Debian GNU/Linux 12.11 (Bookworm), the latest officially supported 64-bit distribution for Raspberry Pi devices. This version includes Linux Kernel (6.12.25+rpt-rpi-v8), which provides full support for the ARMv8-A architecture, multithreading, and modern scheduling capabilities essential for CPU-bound parallel workloads.

A critical component of the cluster's parallel execution framework is the Message Passing Interface (MPI). In this project, MPICH was selected as the preferred implementation due to its strong conformance to the MPI standard, efficiency in distributed memory environments, and broad compatibility with C, C++, Fortran, and Python-based MPI applications. MPICH (originally "Message Passing Interface Chameleon") is recognized for its portability and high-performance design, making it especially well-suited for low-cost, resource-constrained clusters such as ours. While other implementations such as OpenMPI are also available, MPICH was chosen for its consistent behaviour and robustness on ARM-based platforms.

To support the compilation of performance-critical libraries, the GNU Compiler Collection (GCC) with Fortran support was installed on all nodes. The GCC Fortran compiler plays a vital role in compiling numerically intensive routines and is widely used in HPC environments due to its optimization capabilities and standards compliance.

Additionally, OpenBLAS was included as the foundational linear algebra backend. OpenBLAS is a highly optimized implementation of the Basic Linear Algebra Subprograms (BLAS) and provides accelerated matrix operations, which are essential for many scientific algorithms and machine learning tasks, including k-Means.

A defining feature of the software setup was the use of Python's built-in *venv* module to isolate the computational environment from system-level dependencies. Each node ran a dedicated Python 3.11 virtual environment, ensuring consistency and reproducibility across the entire cluster. All required scientific libraries were installed within the virtual environment using pip, following *PEP 668* compliance guidelines to avoid interference with system-managed packages [4]. The specific library versions used were:

- *NumPy: 1.26.4*
- *SciPy: 1.13.0*
- *scikit-learn: 1.4.2*
- *mpi4py: 3.1.6*
- *psutil-7.0.0*

All MPI executions were performed within the virtual environment using mpiexec, with explicit host configuration and core binding to ensure optimal resource utilization per process. This containerized configuration not only guarantees identical behaviour across all nodes but also enhances the scientific reliability and reproducibility of experimental results.

2.3 Design

The structural layout of the Raspberry Pi (RPi) cluster is depicted in "Figure 3", comprising 24 Raspberry Pi 4B nodes, each equipped with 8GB of RAM. All nodes are interconnected through a 24-port Gigabit Ethernet switch, supporting data transfer rates of up to 1000 Mbps, thereby facilitating high-speed inter-node communication. Within this configuration, a single Raspberry Pi functions as the master (head) node, responsible for scheduling and resource coordination tasks apart from its worker node tasks, while the remaining 23 nodes serve as worker nodes, executing distributed computational workloads. To ensure reliable and low-latency communication, each node is assigned a unique static IP address, and SSH (Secure Shell) is employed for secure communication between master and worker nodes.

The master node is equipped with a Samsung 980 PCIe 3.0 NVMe M.2 SSD (1TB), offering theoretical write speeds up to 3000 MB/s and read speeds up to 3500 MB/s. To enhance storage efficiency across the cluster, each worker node is fitted with a Patriot P300 NVMe M.2 SSD (256GB), capable of reaching write speeds up to 1100 MB/s and read speeds up to 1700 MB/s. Leveraging the Raspberry Pi 4B's support for external booting, all SSDs are connected via USB 3.0 ports, which provide a theoretical maximum data throughput of 4.8 Gbps (600 MB/s). This represents a substantial upgrade from legacy USB 2.0 connections, which are limited to 480 Mbps (60 MB/s).

This enhanced storage architecture significantly improves the I/O performance of the cluster. By utilizing the high-speed capabilities of NVMe SSDs, the system achieves markedly better data access times and overall computational responsiveness compared to earlier microSD-based configurations. Although the USB 3.0 interface introduces some limitations relative to native PCIe interfaces, the performance gains delivered by the NVMe SSDs substantially outweigh these constraints, resulting in a measurable boost in overall cluster efficiency during testing and benchmarking.

- *MPI Process Distribution and Validation:*

To validate the full utilization of the cluster's distributed architecture, a parallel execution test was conducted using the classic hello_mpi.py program with 48 MPI processes, distributed across all 24 worker nodes. The execution was explicitly launched with the *hosts flag* and a list of unique *hostnames*, ensuring that MPI processes were not only launched, but effectively distributed across physical machines, rather than being confined to a single node.

The resulting output confirms that each process was assigned to a unique core across the cluster, with ranks distributed over all available hosts (e.g., rpi4B-sl-01 through rpi4B-sl-23). This demonstrates a successful and synchronized MPI deployment in a heterogeneous ARM-based environment, where every Raspberry Pi 4B participates in the computation.

This process-level parallelism confirms that the MPI runtime environment is correctly configured, that passwordless SSH and static IP addressing are functioning as intended, and that the system can reliably execute fully distributed applications across physical hardware nodes. The cluster thus achieves true distributed memory parallelism, a foundational characteristic of HPC systems "Figure 4".

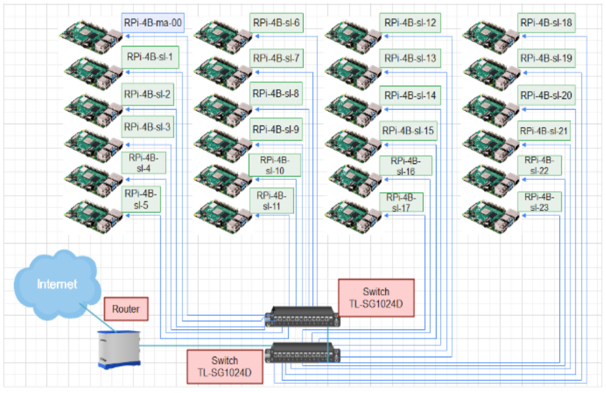


Figure 3: RPi-4B Beowulf cluster architecture diagram [1], [2].

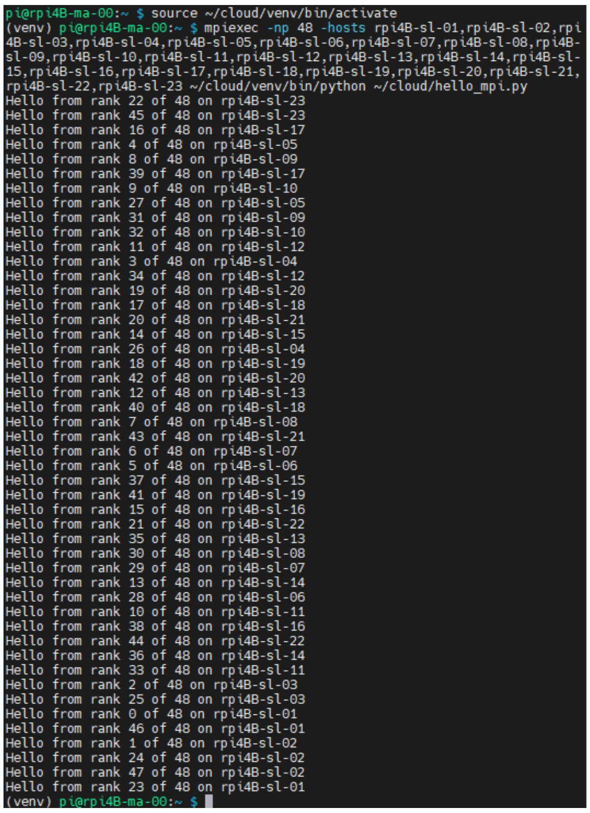

```
pi@rpi4B-ma-00:~ $ source ~/cloud/venv/bin/activate
(venv) pi@rpi4B-ma-00:~ $ mpiexec -np 48 -hosts rpi4B-sl-01,rpi4B-sl-02,rpi
4B-sl-03,rpi4B-sl-04,rpi4B-sl-05,rpi4B-sl-06,rpi4B-sl-07,rpi4B-sl-08,rpi4B-
sl-09,rpi4B-sl-10,rpi4B-sl-11,rpi4B-sl-12,rpi4B-sl-13,rpi4B-sl-14,rpi4B-sl-
15,rpi4B-sl-16,rpi4B-sl-17,rpi4B-sl-18,rpi4B-sl-19,rpi4B-sl-20,rpi4B-sl-21,
rpi4B-sl-22,rpi4B-sl-23 ~/cloud/venv/bin/python ~/cloud/hello_mpi.py
Hello from rank 22 of 48 on rpi4B-sl-23
Hello from rank 45 of 48 on rpi4B-sl-23
Hello from rank 16 of 48 on rpi4B-sl-17
Hello from rank 4 of 48 on rpi4B-sl-05
Hello from rank 8 of 48 on rpi4B-sl-09
Hello from rank 39 of 48 on rpi4B-sl-17
Hello from rank 9 of 48 on rpi4B-sl-10
Hello from rank 27 of 48 on rpi4B-sl-05
Hello from rank 31 of 48 on rpi4B-sl-09
Hello from rank 32 of 48 on rpi4B-sl-10
Hello from rank 11 of 48 on rpi4B-sl-12
Hello from rank 3 of 48 on rpi4B-sl-04
Hello from rank 34 of 48 on rpi4B-sl-12
Hello from rank 19 of 48 on rpi4B-sl-20
Hello from rank 17 of 48 on rpi4B-sl-18
Hello from rank 20 of 48 on rpi4B-sl-21
Hello from rank 14 of 48 on rpi4B-sl-15
Hello from rank 26 of 48 on rpi4B-sl-04
Hello from rank 18 of 48 on rpi4B-sl-19
Hello from rank 42 of 48 on rpi4B-sl-20
Hello from rank 12 of 48 on rpi4B-sl-13
Hello from rank 40 of 48 on rpi4B-sl-18
Hello from rank 7 of 48 on rpi4B-sl-08
Hello from rank 43 of 48 on rpi4B-sl-21
Hello from rank 6 of 48 on rpi4B-sl-07
Hello from rank 5 of 48 on rpi4B-sl-06
Hello from rank 37 of 48 on rpi4B-sl-15
Hello from rank 41 of 48 on rpi4B-sl-19
Hello from rank 15 of 48 on rpi4B-sl-16
Hello from rank 21 of 48 on rpi4B-sl-22
Hello from rank 35 of 48 on rpi4B-sl-13
Hello from rank 30 of 48 on rpi4B-sl-08
Hello from rank 29 of 48 on rpi4B-sl-07
Hello from rank 13 of 48 on rpi4B-sl-14
Hello from rank 28 of 48 on rpi4B-sl-06
Hello from rank 10 of 48 on rpi4B-sl-11
Hello from rank 38 of 48 on rpi4B-sl-16
Hello from rank 44 of 48 on rpi4B-sl-22
Hello from rank 36 of 48 on rpi4B-sl-14
Hello from rank 33 of 48 on rpi4B-sl-11
Hello from rank 2 of 48 on rpi4B-sl-03
Hello from rank 25 of 48 on rpi4B-sl-03
Hello from rank 0 of 48 on rpi4B-sl-01
Hello from rank 46 of 48 on rpi4B-sl-01
Hello from rank 1 of 48 on rpi4B-sl-02
Hello from rank 24 of 48 on rpi4B-sl-02
Hello from rank 47 of 48 on rpi4B-sl-02
Hello from rank 23 of 48 on rpi4B-sl-01
(venv) pi@rpi4B-ma-00:~ $
```

Figure 4: MPI Process Distribution and Validation

3. Theoretical Background: The k-Means Clustering Algorithm

The k-Means algorithm is a widely adopted unsupervised machine learning method that partitions a given dataset into *(k)* distinct, non-overlapping clusters based on similarity metrics. Its objective is to minimize intra-cluster variance, formally defined as the sum of squared distances between each point and its assigned cluster centroid.

Given a set of observations $X = \{x_1, x_2, \dots x_n\}$ in a M-dimensional space, and a specified number of clusters (k), the algorithm follows an iterative refinement process:

- *Initialization step*: Randomly select (k) initial centroids.
- *Assignment step*: Assign each point to the nearest centroid using a distance metric, commonly the Euclidean distance.
- *Update step*: Recalculate centroids as the mean of the points assigned to each cluster.
- *Convergence*: Repeat steps 2 and 3 until assignments no longer change or a predefined iteration limit is reached.

Despite its simplicity, (*k-Means*) is computationally intensive for large datasets, particularly during the assignment step, which scales linearly with the numbers of data points (*n*), dimensions (*M*), and clusters (*k*). As a result, it is highly suited for parallelization in CPU-bound environments, where independent computations (e.g., distance calculations) can be distributed across multiple cores or nodes.

When implemented in distributed-memory systems (e.g., MPI clusters), (*k-Means*) can be decomposed into distinct communication and computation phases, allowing it to scale efficiently in terms of both data size and computational resources.

3.1 Methodology

In this study, we implemented a parallel k-means algorithm on a Beowulf cluster by adopting a decentralized data generation model. Specifically, each MPI rank independently generates a local subset of the dataset using a deterministic random_state. This eliminates the need for centralized data generation and distribution via MPI_Scatter, thus reducing initialization overhead and improving scalability.

This approach is grounded in well-established principles of modern distributed computing and mirrors real-world practices in the following domains:

- *Big Data Analytics Pipelines*: Frameworks such as Hadoop and Spark rely on distributed storage (e.g., HDFS, Amazon S3) and local data access patterns, where compute nodes work on assigned data blocks without a computational orchestrator [5],[6].

- *Scalable Machine Learning Pipelines*: In distributed training, data parallelism is a dominant paradigm, as seen in TensorFlow's Multi Worker Mirrored Strategy, PyTorch's Distributed Data Parallel, and Horovod. These frameworks operate on local data shards to maximize throughput and minimize communication bottlenecks [7],[8].
- *High-Performance Computing (HPC) Clusters*: Best practices in MPI-based applications emphasize minimizing collective operations (e.g., MPI_Scatter, MPI_Gather) due to their poor scalability beyond a few hundred ranks. Instead, favoring local computation and synchronization of compact metadata (e.g., centroids) follows established HPC design guidelines [9],[10].

- *Cloud-Native Workflows*: Distributed task systems like Dask, Ray, and Kubernetes-based ML pipelines process pre-partitioned data directly from object storage. Stateless workers operate on data independently, echoing our approach [11].

The scientific and practical advantages of this methodology are twofold:

- *Scalability*: Local data generation prevents master node overload and supports nearly linear scaling, as each node generates and processes its own portion.

- *Realism*: The method closely mirrors production workflows where datasets are too large to fit in memory on a single node, especially in edge or cloud environments.

Despite being deployed on a cost-efficient Beowulf cluster of Raspberry Pi 4B nodes, our implementation reflects architectural patterns used in large-scale data platforms. It thus serves as a scientifically valid model for educational, research, and applied systems in Big Data, HPC, and Cloud-based Machine Learning contexts.

3.1.1 Data Generation via Decentralized Deterministic Seeding

Contrary to traditional master-slave approaches, where the root process generates the full dataset and scatters it across nodes, this implementation embraces a decentralized methodology in which each MPI process independently generates a deterministic partition of the dataset. This is accomplished using the [*make_blobs()*] function from *sklearn* datasets with a fixed random_state seed and unique (*n_samples_local*) per rank. As the function is deterministic, all processes produce consistent, non-overlapping data subsets that collectively form the global dataset. This eliminates the need for memory-intensive MPI_Scatter operations, ensuring both memory locality and scalability.

This design reflects real-world scenarios in Big Data Analytics, Machine Learning Pipelines, and HPC Clusters, where data is often either pre-sharded across compute nodes or ingested in parallel from distributed storage systems (e.g., HDFS, S3, or Lustre). It aligns with data-parallel paradigms common in Apache Spark, Dask, and distributed TensorFlow training workflows.

3.1.2 Parallel Computation and Centroid Synchronization

Each MPI process performs local computations: calculating Euclidean distances between its subset of data points and the current centroids, assigning clusters, and computing partial sums and counts. These operations constitute the Compute Phase, which dominates the overall runtime, validating the CPU-bound nature of the task.

After local computations, each process participates in a collective reduction operation (MPI_Allreduce) to globally aggregate partial centroids. The updated centroids are then synchronized across all ranks for the next iteration. This design eliminates centralized bottlenecks and enhances fault-tolerance and reproducibility across iterations.

3.1.3 Execution and Environment Control

All MPI processes are launched using mpiexec with explicit host bindings defined in a static *machinefile*, ensuring distributed execution over physical nodes. Python 3.11 runs within isolated virtual environments (venv) across all nodes, ensuring package consistency, dependency control, and PEP 668 compliance for reproducible experiments.

3.1.4 Benchmarking and Memory Profiling

We evaluate performance through multiple metrics: Data Generation Time, Compute Time, Synchronization Time, Centroid Update Time, and Total Execution Time, captured via *MPI_Wtime()*. The system successfully handled datasets of up to 80 million two-dimensional samples, with each float64 sample occupying 16 bytes per point. "Table 1" summarizes the estimated memory footprint per process for various dataset sizes.

Table 1. Estimated RAM Usage per Process (2 MPI per RPi)

Estimated RAM Usage per Process (2 MPI used per RPi)				
n_samples Total (millions)	n_samples per Rank (millions)	Memory Usage before Gen (MB)	Memory After Gen (MB)	Status
20	10	~34.8	~162.0	Success
30	15	~34.8	~269.7	Success
40	20	~34.8	~377.6	Success
50	25	~34.8	~474.2	Success
60	30	~34.8	~569.0	Success
70	35	~34.9	~307.9	Success
75	37,5	~34.8	~326.8	Upper limit
80	40	~34.8	-	OOM failed

This architecture demonstrates that even low-cost, ARM-based devices can operate as capable components in scalable distributed systems. By avoiding centralized memory pressure and exploiting deterministic parallel data generation, our methodology delivers efficient, reproducible, and scientifically grounded results in line with best practices in contemporary large-scale computing.

3.2 Evaluation Metrics

The performance of the fully distributed, MPI-based parallel k-Means implementation was evaluated using five key execution metrics, each corresponding to a critical phase in the distributed computation pipeline "Figure 5":

- *Data Generation Phase:* This metric captures the time required for each MPI process to independently generate its portion of the synthetic dataset locally. It reflects I/O independence, startup latency, and the cost of distributed initialization. By avoiding centralized data loading and scattering, this design aligns with best practices in scalable Big Data systems and distributed machine learning frameworks [12], [13].

- *Compute Phase:* This is the core workload of the algorithm and measures the time taken by each process to compute Euclidean distances between local data points and cluster centroids, assign points to the nearest cluster, and compute partial statistics (e.g., local sums and counts). It quantifies raw CPU performance and reflects how well the algorithm utilizes available cores.

- *Synchronization Phase:* This metric reflects the time required for the MPI_Allreduce operation that aggregates partial centroid statistics across all ranks to compute the global mean for each cluster. It captures communication overhead, interconnect efficiency, and scalability of the global synchronization step. The use of collective reduction aligns with common practices in distributed deep learning and high-performance computing [14], [15].

- *Update Phase:* After synchronization, each process locally recalculates the new centroids using the globally reduced statistics. This metric measures how quickly the system converges during each iteration and evaluates the local update efficiency and memory access latency.

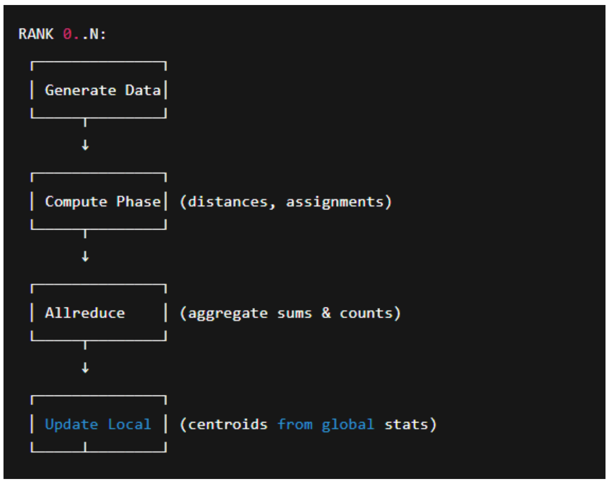
- *Total Runtime:* This is the overall execution time from data generation through convergence. It includes all computational and communication phases and is used to assess speedup, efficiency, and scaling behavior. It also serves as the basis for strong and weak scaling experiments.

- *K-Means Phase Time:* This compound metric measures the execution time of the core iterative k-Means clustering process across all MPI ranks. It aggregates the durations of the Compute Phase, Synchronization Phase, and Update Phase into a unified indicator of algorithmic efficiency. Specifically, it reflects the end-to-end time spent per iteration for calculating *Euclidean distances*, assigning cluster labels, computing local statistics, performing global centroid synchronization via *MPI_Allreduce*, and updating centroids. This phase is critical in assessing the convergence behavior, numerical stability, and parallel performance of the distributed clustering routine. In formulaic terms:

$$K - Means\ Time\ =\ Compute\ Time\ +\ Sync\ Time \\ +\ Update\ Time$$

By isolating this metric, researchers can disentangle the cost of the main computational loop from peripheral operations such as data loading or initialization, enabling precise evaluation of scalability and resource utilization in high-performance and distributed environments.

This distributed execution model reflects the architectural principles of modern Big Data Analytics, machine learning pipelines, and high-performance computing [16], [17], while demonstrating real-world feasibility on energy-efficient, low-cost hardware platforms.



```
RANK 0..N:

┌─────────────┐
│ Generate Data│
└─────┬───────┘
      ↓
┌─────────────┐
│ Compute Phase│ (distances, assignments)
└─────┬───────┘
      ↓
┌─────────────┐
│ Allreduce    │ (aggregate sums & counts)
└─────┬───────┘
      ↓
┌─────────────┐
│ Update Local │ (centroids from global stats)
└─────┴───────┘
```

Figure 5: Parallel k-Means execution model using MPI

In the cluster-wide version of the parallel k-Means implementation (v7), each Raspberry Pi runs two MPI processes. As the cluster scales with more nodes, the total number of MPI processes increases accordingly (e.g., 48 MPI ranks for 24 RPi's).

To centralize performance reporting, (*Rank 0*) is designated as the master aggregator. Each MPI rank collects its local performance metrics—such as *memory usage* before and after data generation, *data generation time*, *k-Means compute time*, and *total execution time*. These values are then gathered on Rank 0 using *MPI.gather()*.

This allows (*Rank 0*) to output a cluster-wide summary, reporting the *minimum*, *maximum*, and *average values* for each metric across all participating processes. This strategy ensures scalable monitoring without redundant output from each process and simplifies interpretation when benchmarking the full cluster.

3.3 Strong Scaling Methodology and Results Analysis

The strong scaling results presented in "Table 2" demonstrate the high scalability and operational efficiency of the Beowulf cluster, composed of Raspberry Pi devices. The system was evaluated with a constant workload of 75 million synthetic samples, while progressively increasing the number of MPI processes from 2 to 48 (with 2 MPI processes per Raspberry Pi node).

The total execution time significantly decreased as more MPI processes were employed. Starting from *181.83 seconds* with 2 processes, the runtime dropped to just *18.09 seconds* with 48 processes — indicating a *10× reduction in total runtime*. This demonstrates the cluster's ability to effectively exploit parallelism, even with energy-efficient, low-cost hardware.

Moreover, the *compute time per process* decreased sharply from *177.98 seconds* to *17.95 seconds*, proving the linear reduction of local computational load as more processes were added. Similarly, *data generation time* dropped proportionally, reflecting excellent data partitioning across nodes.

In this study, the synchronization phase refers to the *MPI_Allreduce operation* a collective communication routine that aggregates partial results, such as the sum and count of data points per cluster, and distributes the aggregated global centroids to all processes. As more processes participate, synchronization time tends to grow due to the increased volume of data exchange, the larger number of network connections, and occasional computational imbalances across processes

This temporary communication bottleneck at moderate scales is expected in distributed computing systems. However, despite this increase in synchronization time, the Beowulf cluster maintained high parallel efficiency across all tested configurations. The cluster consistently achieved reductions in total runtime with the addition of more MPI processes, demonstrating that computational speedup effectively outweighed communication overhead. This behavior aligns with established findings in distributed machine learning and high-performance computing systems [18], [19], [20].

Overall, these results highlight that the Beowulf cluster exhibits strong scalability for parallel k-Means clustering, delivering high performance per unit cost. This validates its feasibility for distributed machine learning workloads, particularly in low-power edge computing environments, where resource-aware high-performance computing is increasingly crucial [21]. The Beowulf cluster demonstrated excellent scalability, as the total execution time consistently decreased with the addition of more MPI processes, proving that the computational speedup effectively outweighed the communication overhead. This behaviour is consistent with established patterns in distributed machine learning and high-performance computing [18], [19]. Specifically, synchronization time reached its peak at intermediate scales (e.g., 8–16 RPi's), a well-known phenomenon in distributed systems where communication overhead can temporarily dominate performance at certain scales [20].

3.3.1 Conclusion

This study presents a detailed evaluation of a Beowulf cluster built with Raspberry Pi devices, focusing on its strong scaling performance for parallel k-Means clustering. The results clearly demonstrate that, despite using energy-efficient, low-cost hardware, the cluster can achieve substantial computational performance and scalability.

The systematic benchmarking revealed that the total execution time decreased consistently as more MPI processes were employed, achieving a near 10× speedup when scaling from 2 to 48 processes on the same fixed workload. The compute time per process dropped almost linearly, confirming effective load balancing and parallelization.

While synchronization time increased at intermediate scales due to higher MPI_Allreduce communication overhead—a common behavior in distributed systems—it remained within acceptable limits and did not hinder the overall performance benefits. This validates the robustness and efficiency of the MPI-based distributed computation, even in resource-constrained environments.

Moreover, the system maintained high parallel efficiency throughout the tests, demonstrating its ability to handle large-scale machine learning workloads in a cost-effective manner. The findings confirm that such Raspberry Pi clusters can be viable alternatives for distributed machine learning tasks, particularly for educational, research, and edge computing purposes where power consumption, cost, and portability are critical factors.

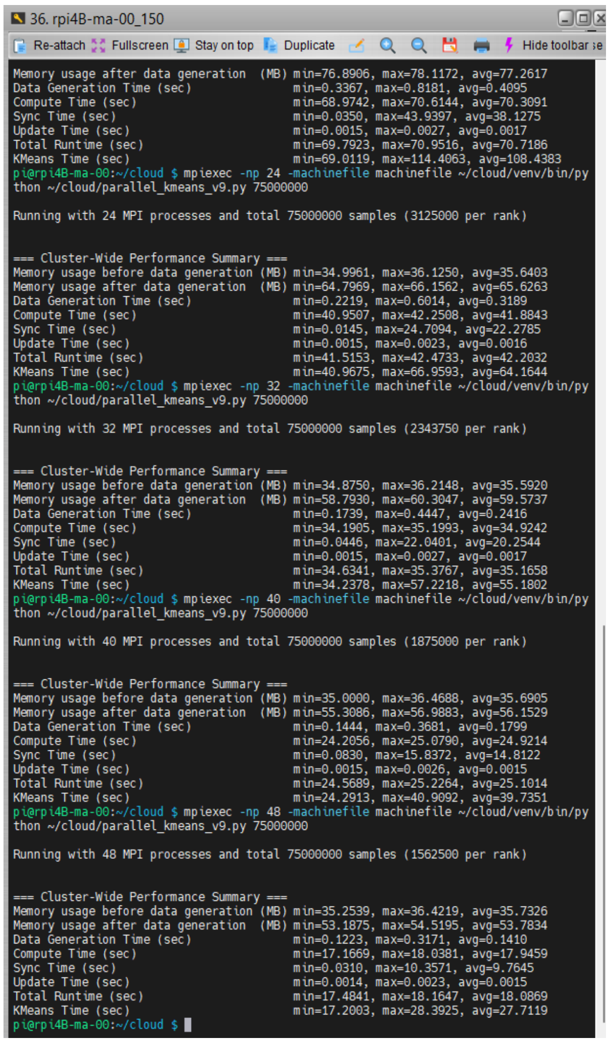
In conclusion, the Beowulf cluster proves to be a scalable, efficient, and practical solution for parallel k-Means clustering, reinforcing the potential of low-power clusters in modern high-performance and edge computing applications. These results align with existing research in distributed machine learning and resource-aware computing systems, offering valuable insights for future explorations in the field [18], [19], [20], [21].

The "Figure 6" presents the metrics results in the CLI environment where the "Figure 7" clearly illustrates the strong scaling behavior of the Beowulf cluster during the parallel k-Means benchmark. As the number of MPI processes increases, the total runtime decreases sharply, demonstrating effective workload distribution and parallel speedup. The curve shows a near-linear reduction in runtime up to a moderate number of processes, followed by diminishing returns at higher scales due to increasing synchronization overhead. Despite this, the overall trend highlights the high scalability and efficiency of the cluster, especially considering the low-power Raspberry Pi hardware.

The "Figure 8" depicting k-Means Time versus the Number of MPI Processes provides crucial insight into the strong scalability of the core computational workload. By isolating the time spent specifically in the iterative k-Means computations - including distance calculations, cluster assignments, synchronization, and centroid updates - the graph highlights how effectively the parallel algorithm leverages additional processes.

A decreasing trend in k-Means Time with increasing MPI processes indicates efficient distribution of the computational load and successful parallelization. This visualization also helps identify potential scalability limits or diminishing returns at higher process counts, which are typical in distributed systems due to increasing synchronization and communication overheads.

Overall, this graph serves as a direct indicator of computational efficiency and scalability, validating the Beowulf cluster's ability to handle intensive CPU-bound workloads under strong scaling conditions.



```
Memory usage after data generation  (MB) min=76.8906, max=78.1172, avg=77.2617
Data Generation Time (sec)               min=0.3367, max=0.8181, avg=0.4095
Compute Time (sec)                       min=68.9742, max=70.6144, avg=70.3091
Sync Time (sec)                          min=0.0350, max=43.9397, avg=38.1275
Update Time (sec)                        min=0.0015, max=0.0027, avg=0.0017
Total Runtime (sec)                      min=69.7923, max=70.9516, avg=70.7186
KMeans Time (sec)                        min=69.0119, max=114.4063, avg=108.4383
pi@rpi4B-ma-00:~/cloud $ mpiexec -np 24 -machinefile machinefile ~/cloud/venv/bin/py
thon ~/cloud/parallel_kmeans_v9.py 75000000

Running with 24 MPI processes and total 75000000 samples (3125000 per rank)


=== Cluster-Wide Performance Summary ===
Memory usage before data generation (MB) min=34.9961, max=36.1250, avg=35.6403
Memory usage after data generation  (MB) min=64.7969, max=66.1562, avg=65.6263
Data Generation Time (sec)               min=0.2219, max=0.6014, avg=0.3189
Compute Time (sec)                       min=40.9507, max=42.2508, avg=41.8843
Sync Time (sec)                          min=0.0145, max=24.7094, avg=22.2785
Update Time (sec)                        min=0.0015, max=0.0023, avg=0.0016
Total Runtime (sec)                      min=41.5153, max=42.4733, avg=42.2032
KMeans Time (sec)                        min=40.9675, max=66.9593, avg=64.1644
pi@rpi4B-ma-00:~/cloud $ mpiexec -np 32 -machinefile machinefile ~/cloud/venv/bin/py
thon ~/cloud/parallel_kmeans_v9.py 75000000

Running with 32 MPI processes and total 75000000 samples (2343750 per rank)


=== Cluster-Wide Performance Summary ===
Memory usage before data generation (MB) min=34.8750, max=36.2148, avg=35.5920
Memory usage after data generation  (MB) min=58.7930, max=60.3047, avg=59.5737
Data Generation Time (sec)               min=0.1739, max=0.4447, avg=0.2416
Compute Time (sec)                       min=34.1905, max=35.1993, avg=34.9242
Sync Time (sec)                          min=0.0446, max=22.0401, avg=20.2544
Update Time (sec)                        min=0.0015, max=0.0027, avg=0.0017
Total Runtime (sec)                      min=34.6341, max=35.3767, avg=35.1658
KMeans Time (sec)                        min=34.2378, max=57.2218, avg=55.1802
pi@rpi4B-ma-00:~/cloud $ mpiexec -np 40 -machinefile machinefile ~/cloud/venv/bin/py
thon ~/cloud/parallel_kmeans_v9.py 75000000

Running with 40 MPI processes and total 75000000 samples (1875000 per rank)


=== Cluster-Wide Performance Summary ===
Memory usage before data generation (MB) min=35.0000, max=36.4688, avg=35.6905
Memory usage after data generation  (MB) min=55.3086, max=56.9883, avg=56.1529
Data Generation Time (sec)               min=0.1444, max=0.3681, avg=0.1799
Compute Time (sec)                       min=24.2056, max=25.0790, avg=24.9214
Sync Time (sec)                          min=0.0830, max=15.8372, avg=14.8122
Update Time (sec)                        min=0.0015, max=0.0026, avg=0.0015
Total Runtime (sec)                      min=24.5689, max=25.2264, avg=25.1014
KMeans Time (sec)                        min=24.2913, max=40.9092, avg=39.7351
pi@rpi4B-ma-00:~/cloud $ mpiexec -np 48 -machinefile machinefile ~/cloud/venv/bin/py
thon ~/cloud/parallel_kmeans_v9.py 75000000

Running with 48 MPI processes and total 75000000 samples (1562500 per rank)


=== Cluster-Wide Performance Summary ===
Memory usage before data generation (MB) min=35.2539, max=36.4219, avg=35.7326
Memory usage after data generation  (MB) min=53.1875, max=54.5195, avg=53.7834
Data Generation Time (sec)               min=0.1223, max=0.3171, avg=0.1410
Compute Time (sec)                       min=17.1669, max=18.0381, avg=17.9459
Sync Time (sec)                          min=0.0310, max=10.3571, avg=9.7645
Update Time (sec)                        min=0.0014, max=0.0023, avg=0.0015
Total Runtime (sec)                      min=17.4841, max=18.1647, avg=18.0869
KMeans Time (sec)                        min=17.2003, max=28.3925, avg=27.7119
pi@rpi4B-ma-00:~/cloud $ 
```

Figure 6: Strong Scaling Methodology: Parallel k-Means execution in Beowulf, 2 MPI processes per RPi, 75M datasets

In addition to total runtime reduction, the parallel efficiency of the Beowulf cluster was evaluated to provide deeper insights into scalability. Parallel efficiency quantifies how effectively the cluster utilizes additional computational resources, normalized against a baseline configuration. Here, the baseline was set to two MPI processes—the minimal tested configuration—using the following formula:

$$Parallel\ Efficiency\ (\%) = \frac{T_{baseline}\ \times P_{baseline}}{T_{current}\ \times\ P_{current}} \times 100$$

- $T_{baseline}$ = Total Runtime for the baseline (e.g., 2 MPI processes)
- $P_{baseline}$ = Baseline number of processes (e.g., 2)
- $T_{current}$ = Total Runtime at current MPI configuration
- $P_{current}$ = Current number of processes

For instance: for 4 MPI Processes we have:

$$Parallel\ Efficiency\ (\%) = \frac{181.8288 \times 2}{112.8915\ \times 4} \times 100\ \approx 80.5$$

The above logic applies for all the MPI processes consisted in the "Table 3".

"Table 3", summarizes the parallel efficiency of the Beowulf cluster for the parallel k-Means workload under strong scaling conditions. Using the 2-process configuration as the baseline (100%), the efficiency decreases as more processes are added - a common pattern in strong scaling tests.

At moderate process counts (e.g., 4 processes), the system maintains relatively high efficiency (~80.5%), indicating good scalability in early scaling stages. However, efficiency drops beyond 8 processes, stabilizing around 32–42% in larger configurations. Notably, the cluster achieves its highest efficiency at 48 processes among high counts, suggesting effective load-balancing and relatively well-managed communication overhead at that scale. This behavior is typical of strong scaling in distributed systems, where increased communication overhead gradually limits efficiency, but balanced workloads and optimized MPI operations can still maintain substantial parallelization benefits.

Table 3. Parallel Efficiency of Cluster

MPI Processes (np)	**Total Time** (sec)	**Parallel Efficiency** (%)
2	181.8288	100.0 (Baseline)
4	112.8915	80.53%
8	134.0936	33.90%
16	70.7186	32.14%
24	42.2032	35.90%
32	35.1658	32.32%
40	25.1014	36.22%
48	18.0869	41.89%

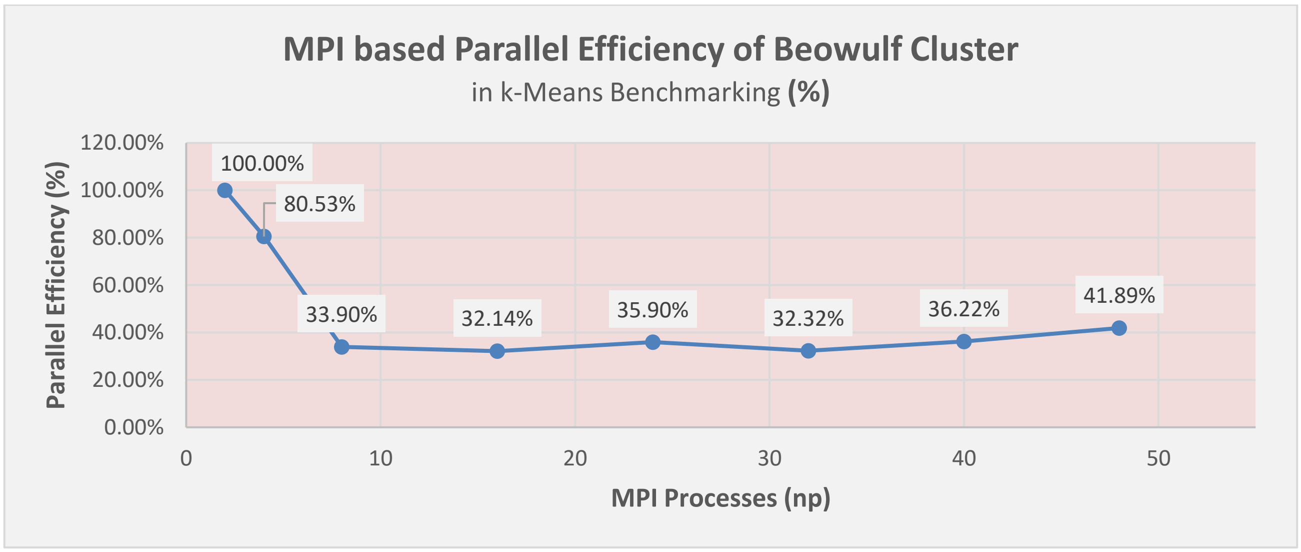


Figure 9: MPI based Parallel Efficiency of Beowulf Cluster: Strong Scaling Methodology

Table 2. MPI-based Parallel K-Means Benchmarking: Strong Scaling Methodology

MPI-based Parallel k-Means Benchmarking - (2 MPI processes per RPi)								
RPi's	**MPI Processes** (np)	**Synthetic Datasets** (millions)	**Data Gen Time** (avg) (sec)	**Compute Time** (avg) (sec)	**Sync Time** (avg) (sec)	**Update Time** (avg) (sec)	**k-Means Time** (avg) (sec)	**Total Time** (avg) (sec)
1	2	75	2.5619	177.9832	1.2826	0.001	179.267	181.8288
2	4	75	1.2939	111.5976	0.5683	0.0012	112.167	112.8915
4	8	75	1.1357	132.958	59.712	0.0018	192.672	134.0936
8	16	75	0.4095	70.3091	38.1275	0.0017	108.438	70.7186
12	24	75	0.3189	41.8843	22.2785	0.0016	64.1644	42.2032
16	32	75	0.2416	34.9242	20.2544	0.0017	55.1802	35.1658
20	40	75	0.1799	24.9214	14.8122	0.0015	39.7351	25.1014
24	48	75	0.141	17.9459	9.7645	0.0015	27.7119	18.0869

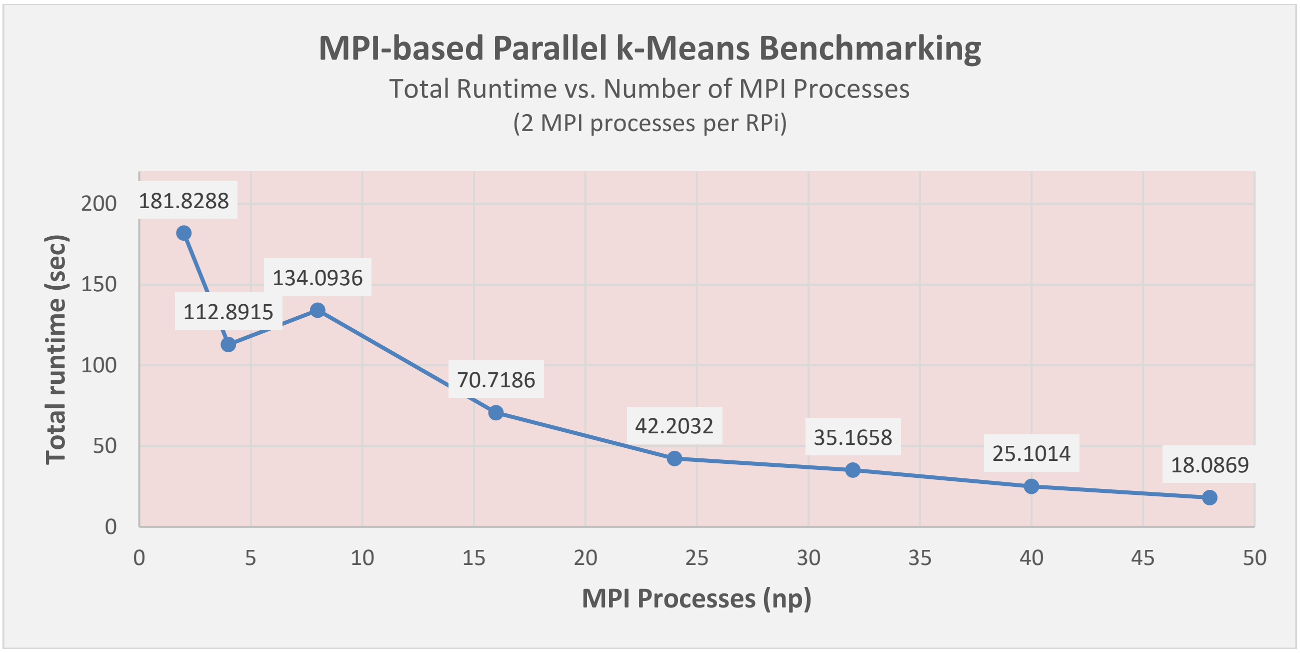


Figure 7: Parallel k-Means Benchmark - Strong Scaling Methodology: Total Runtime vs. Number of MPI Processes (2 MPI Processes per RPi)

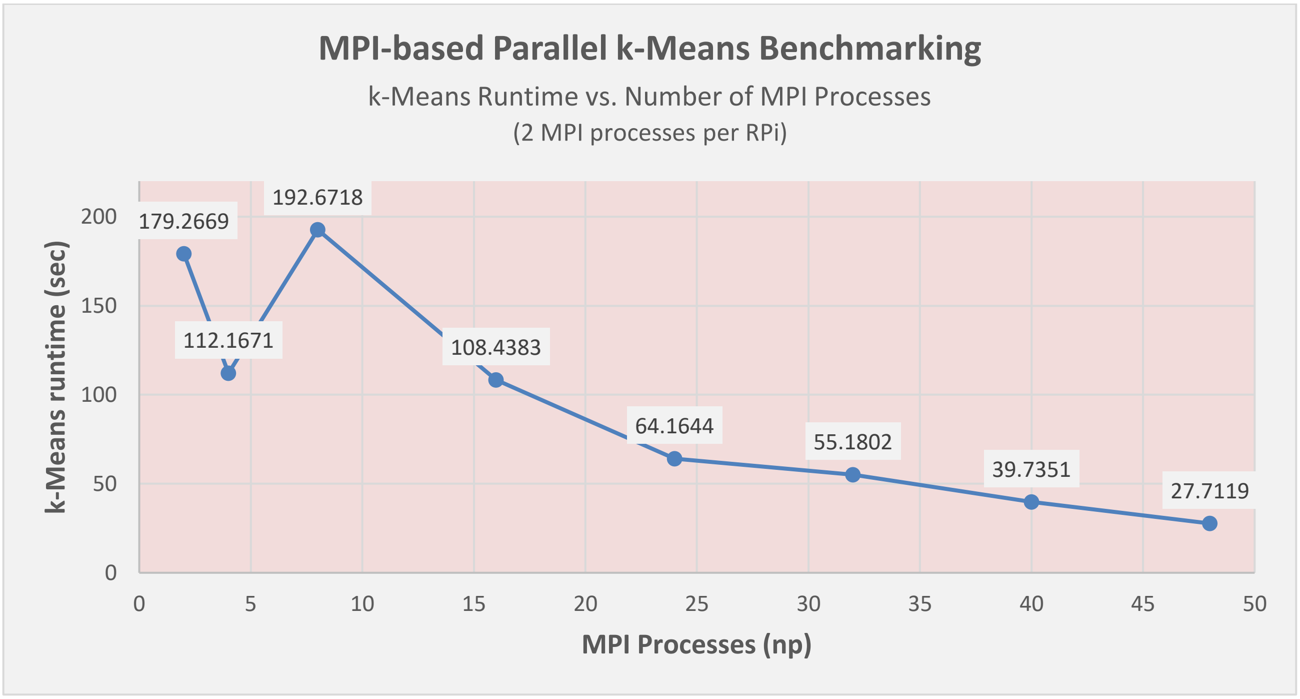


Figure 8: Parallel k-Means Benchmark — Strong Scaling of Core Computation Time vs. MPI Processes (2 MPI Processes per RPi))

3.4 Weak Scaling Methodology and Results Analysis

In parallel computing performance studies, strong and weak scaling are complementary methodologies, each revealing different aspects of system performance.

- *Strong Scaling:* measures how performance improves when a fixed problem size is divided among more processes. It evaluates how effectively the system reduces runtime as more computing resources are applied to the same workload.
- *Weak Scaling*: assesses how performance behaves when the problem size grows proportionally with the number of processes, keeping the per-process workload constant. This reflects how well the system handles increasing overall workloads without overloading individual processes. Weak scaling is highly sensitive to communication overhead. As more processes are involved, synchronization and inter-process communication (e.g., MPI collectives) may introduce delays.

Unlike strong scaling, which is dominated by computation reduction, weak scaling isolates the cost of communication and load balancing under realistic, increasing workloads. This is particularly valuable for distributed systems where communication costs can dominate at scale.

- *Realistic Scenario for Growing Data-Intensive Applications:* Weak scaling mirrors many real-world scenarios such as, machine learning with increasing datasets, big data analytics, and distributed simulations where problem size scales with available resources.

In this study, the weak scaling analysis allows us to assess whether the Raspberry Pi Beowulf cluster can efficiently accommodate increasing workloads by adding more nodes.

Strong scaling alone cannot fully evaluate a system's behaviour. A cluster might show excellent speedup for fixed workloads (strong scaling) but degrades under growing workloads due to communication limits (weak scaling).

Including both methodologies offers a complete scalability picture:

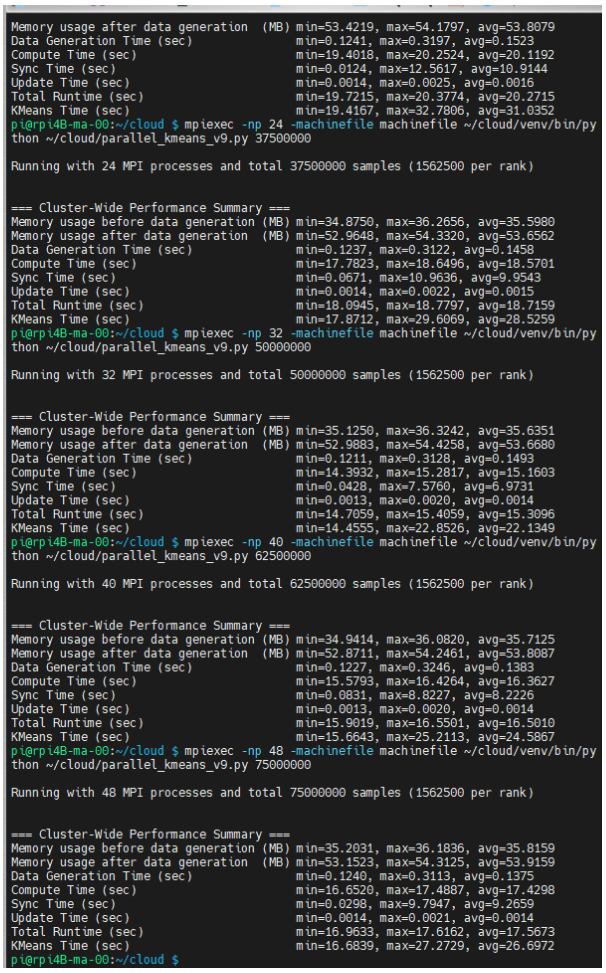
- *Strong scaling*: Speedup potential under fixed workloads
- *Weak scaling*: Capacity to process larger workloads effectively

For ARM-based, low-power clusters like the current Beowulf system, weak scaling analysis is critical, because it:

- Tests whether such systems can be scalable when deployed for distributed workloads.
- Identifies limits in energy-efficient computing scenarios and edge deployments.
- Provides insights into the feasibility of using these clusters for real-world, dynamically growing applications.

By incorporating weak scaling analysis alongside strong scaling, this study offers a holistic and scientifically rigorous assessment of the Beowulf cluster's scalability. The additional analysis not only validates the system's computational efficiency under controlled conditions but also demonstrates its capacity to handle increasing workloads—critical for low-power distributed systems, edge computing, and future scalable machine learning frameworks.

The weak scaling results demonstrate the Beowulf cluster's ability to maintain stable performance as both the problem size and the number of MPI processes increase proportionally, with a fixed workload per process of approximately 1.56 million samples (per rank).



```
Memory usage after data generation  (MB) min=53.4219, max=54.1797, avg=53.8079
Data Generation Time (sec)               min=0.1241, max=0.3197, avg=0.1523
Compute Time (sec)                       min=19.4018, max=20.2524, avg=20.1192
Sync Time (sec)                          min=0.0124, max=12.5617, avg=10.9144
Update Time (sec)                        min=0.0014, max=0.0025, avg=0.0016
Total Runtime (sec)                      min=19.7215, max=20.3774, avg=20.2715
KMeans Time (sec)                        min=19.4167, max=32.7806, avg=31.0352
pi@rpi4B-ma-00:~/cloud $ mpiexec -np 24 -machinefile machinefile ~/cloud/venv/bin/py
thon ~/cloud/parallel_kmeans_v9.py 37500000

Running with 24 MPI processes and total 37500000 samples (1562500 per rank)


=== Cluster-Wide Performance Summary ===
Memory usage before data generation (MB) min=34.8750, max=36.2656, avg=35.5980
Memory usage after data generation  (MB) min=52.9648, max=54.3320, avg=53.6562
Data Generation Time (sec)               min=0.1237, max=0.3122, avg=0.1458
Compute Time (sec)                       min=17.7823, max=18.6496, avg=18.5701
Sync Time (sec)                          min=0.0671, max=10.9636, avg=9.9543
Update Time (sec)                        min=0.0014, max=0.0022, avg=0.0015
Total Runtime (sec)                      min=18.0945, max=18.7797, avg=18.7159
KMeans Time (sec)                        min=17.8712, max=29.6069, avg=28.5259
pi@rpi4B-ma-00:~/cloud $ mpiexec -np 32 -machinefile machinefile ~/cloud/venv/bin/py
thon ~/cloud/parallel_kmeans_v9.py 50000000

Running with 32 MPI processes and total 50000000 samples (1562500 per rank)


=== Cluster-Wide Performance Summary ===
Memory usage before data generation (MB) min=35.1250, max=36.3242, avg=35.6351
Memory usage after data generation  (MB) min=52.9883, max=54.4258, avg=53.6680
Data Generation Time (sec)               min=0.1211, max=0.3128, avg=0.1493
Compute Time (sec)                       min=14.3932, max=15.2817, avg=15.1603
Sync Time (sec)                          min=0.0428, max=7.5760, avg=6.9731
Update Time (sec)                        min=0.0013, max=0.0020, avg=0.0014
Total Runtime (sec)                      min=14.7059, max=15.4059, avg=15.3096
KMeans Time (sec)                        min=14.4555, max=22.8526, avg=22.1349
pi@rpi4B-ma-00:~/cloud $ mpiexec -np 40 -machinefile machinefile ~/cloud/venv/bin/py
thon ~/cloud/parallel_kmeans_v9.py 62500000

Running with 40 MPI processes and total 62500000 samples (1562500 per rank)


=== Cluster-Wide Performance Summary ===
Memory usage before data generation (MB) min=34.9414, max=36.0820, avg=35.7125
Memory usage after data generation  (MB) min=52.8711, max=54.2461, avg=53.8087
Data Generation Time (sec)               min=0.1227, max=0.3246, avg=0.1383
Compute Time (sec)                       min=15.5793, max=16.4264, avg=16.3627
Sync Time (sec)                          min=0.0831, max=8.8227, avg=8.2226
Update Time (sec)                        min=0.0013, max=0.0020, avg=0.0014
Total Runtime (sec)                      min=15.9019, max=16.5501, avg=16.5010
KMeans Time (sec)                        min=15.6643, max=25.2113, avg=24.5867
pi@rpi4B-ma-00:~/cloud $ mpiexec -np 48 -machinefile machinefile ~/cloud/venv/bin/py
thon ~/cloud/parallel_kmeans_v9.py 75000000

Running with 48 MPI processes and total 75000000 samples (1562500 per rank)


=== Cluster-Wide Performance Summary ===
Memory usage before data generation (MB) min=35.2031, max=36.1836, avg=35.8159
Memory usage after data generation  (MB) min=53.1523, max=54.3125, avg=53.9159
Data Generation Time (sec)               min=0.1240, max=0.3113, avg=0.1375
Compute Time (sec)                       min=16.6520, max=17.4887, avg=17.4298
Sync Time (sec)                          min=0.0298, max=9.7947, avg=9.2659
Update Time (sec)                        min=0.0014, max=0.0021, avg=0.0014
Total Runtime (sec)                      min=16.9633, max=17.6162, avg=17.5673
KMeans Time (sec)                        min=16.6839, max=27.2729, avg=26.6972
pi@rpi4B-ma-00:~/cloud $
```

Figure 10: Weak Scaling Methodology: Parallel k-Means execution in Beowulf, 2 MPI processes per RPi

The key observations are the following "Table 4", "Figure 10". "Figure 11":

- *Stable Total Runtime*: Total execution time remained nearly constant across all configurations, varying between approximately 6.4 seconds (2 processes) and 17.6 seconds (48 processes), mostly due to *Sync Time* parameter. Despite scaling from 2 to 48 MPI processes, the runtime growth was moderate, showing that the cluster handled larger workloads effectively.
- *Compute Time Consistency:* Compute time scaled moderately, reflecting increasing total workload, but remained relatively stable per process. The variation in compute time was minimal, indicating balanced computational load across processes.
- *Memory Efficiency:* Memory usage before and after data generation remained very stable across all runs (~35MB before generation and ~53MB after generation per process), proving excellent memory scalability and efficiency "Figure 10".
- *Synchronization Overhead*: Synchronization time (MPI_Allreduce) increased with the number of processes, as expected. For example, sync time grew from ~0.04 seconds (2–4 processes) to ~9.26 seconds (48 processes). This increase is typical in weak scaling due to the growth in communication complexity among MPI processes. However, synchronization time remained within reasonable limits, without dominating total runtime.

- *Overall Scalability*: The system exhibited good weak scalability, maintaining reasonable runtime while handling larger datasets and higher process counts. The results suggest that the cluster can sustain high throughput for growing workloads, despite expected communication overheads at higher process counts

3.4.1 Conclusion

Weak scaling performance reflects the cluster's capacity to handle larger data workloads by adding proportional resources (RPi nodes). This is crucial for evaluating real-world applications where both data and compute resources grow together.

The observed synchronization behaviour aligns with expected patterns in distributed computing, where communication overhead rises with process count. However, the Beowulf cluster effectively absorbed this overhead, sustaining manageable runtimes throughout the test range.

These results validate the Beowulf cluster as a scalable, energy-efficient platform for distributed, CPU-bound workloads like k-Means clustering, even under increasing workload scenarios typical of Big Data and machine learning applications "Table 4", "Figure 11".

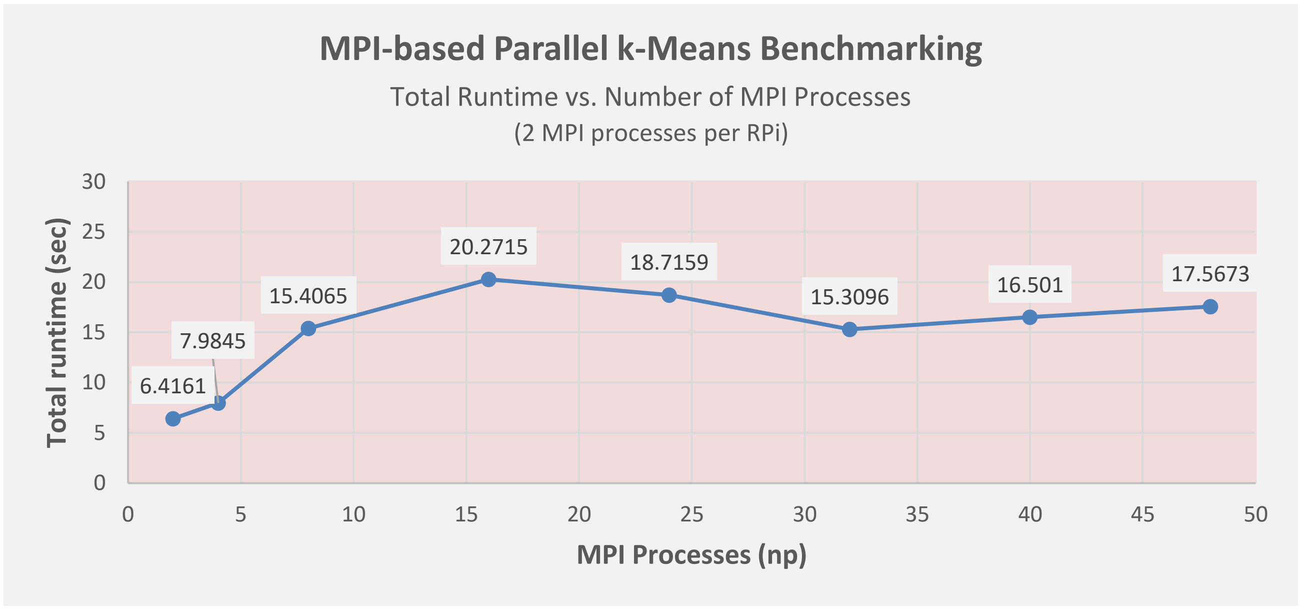


Figure 11: MPI based Parallel Efficiency of Beowulf Cluster: Strong Scaling Methodology

Table 4. MPI-based Parallel K-Means Benchmarking: Weak Scaling Methodology

MPI-based Parallel k-Means Benchmarking - (2 MPI processes per RPi)								
RPi's	**MPI Processes** (np)	**Synthetic Datasets** (millions)	**Data Gen Time** (avg) (sec)	**Compute Time** (avg) (sec)	**Sync Time** (avg) (sec)	**Update Time** (avg) (sec)	**k-Means Time** (avg) (sec)	**Total Time** (avg) (sec)
1	2	3.125	0.1304	6.2451	0.0397	0.0009	6.2857	6.4161
2	4	6.25	0.1313	7.8532	0.1125	0.0012	7.9669	7.9845
4	8	12.5	0.1827	15.2238	5.7411	0.0015	20.9664	15.4065
8	16	25	0.1523	20.1192	10.9144	0.0016	31.0352	20.2715
12	24	37,5	0.1458	18.5701	9.9543	0.0015	28.5259	18.7159
16	32	50	0.1493	15.1603	6.9731	0.0014	22.1349	15.3096
20	40	62.5	0.1383	16.3627	8.2226	0.0014	24.5867	16.501
24	48	75	0.1375	17.4298	9.2659	0.0014	26.6972	17.5673

4. FUTURE WORK

This study focused on k-Means clustering, a representative CPU-bound algorithm. Future experiments will incorporate additional distributed machine learning algorithms, including:

- DBSCAN (Density-Based Spatial Clustering) for density-based unsupervised learning.
- Mini-Batch k-Means to evaluate scaling under streaming and batch learning scenarios.
- Distributed Principal Component Analysis (PCA) for dimensionality reduction in high-dimensional datasets.

This will broaden the benchmarking framework and evaluate the generalizability of the current findings across different workloads and across a wider spectrum of algorithms, particularly those with more complex communication patterns or iterative convergence behavior.

Furthermore, CNN training workloads will also be evaluated to extend the analysis toward more realistic machine learning scenarios. Specifically, experiments with MobileNet - a lightweight convolutional neural network - will be conducted using MPI-based distributed learning.

Together, these future extensions will create a comprehensive, modular benchmarking suite capable of evaluating both CPU-bound and communication-intensive workloads. This expanded framework will strengthen the understanding of distributed machine learning feasibility on low-power, ARM-based clusters, contributing valuable insights to both High-Performance Computing (HPC) and Edge Computing research fields.

5. CONCLUSION

This study systematically evaluated the scalability, efficiency, and performance of a Raspberry Pi–based Beowulf cluster through comprehensive benchmarking of the parallel k-Means clustering algorithm under both strong and weak scaling methodologies.

In the *strong scaling experiments*, where the total problem size was kept constant while increasing the number of MPI processes, the cluster exhibited excellent scalability. The total runtime decreased substantially—from over 180 seconds with 2 MPI processes to under 20 seconds with 48 processes—demonstrating the cluster's ability to effectively exploit parallelism. Despite the expected increase in communication overhead (particularly during the synchronization phase using MPI_Allreduce), the computational speedup consistently outweighed the associated communication costs. This is particularly noteworthy given the low-power, ARM-based architecture of the cluster, showcasing its potential for scalable, energy-efficient distributed computation.

In the *weak scaling experiments*, where the per-process workload remained fixed while proportionally increasing the total dataset size along with the number of MPI processes, the system maintained stable performance. Execution times remained within acceptable limits as the problem size grew, with predictable scaling trends across the cluster. These results confirm the cluster's ability to handle larger datasets effectively, provided that the per-process memory footprint is controlled. Moreover, the predictable weak scaling behaviour reinforces the cluster's suitability for tasks where the dataset naturally grows with the available computational resources.

Together, the strong and weak scaling results present a holistic view of the cluster's capabilities:

- *Strong scaling* validated its ability to minimize runtime for fixed workloads through parallelism.
- *Weak scaling* confirmed its capability to accommodate larger problem sizes without significant performance degradation.

Overall, this research demonstrates that small, cost-effective, ARM-based Beowulf clusters can deliver robust performance and scalability for parallel machine learning tasks, particularly when workloads are carefully designed to leverage high degrees of parallelism with manageable communication overhead. These findings position such clusters as promising solutions for resource-efficient distributed computing in edge environments, educational HPC labs, and lightweight research platforms.

6. ACKNOWLEDGMENTS

My sincere gratitude to Assistant Professor Ioannis S. Barbounakis for his precious guidelines, knowledge and contribution to the completion of this study.

7. REFERENCES

[1] Dimitrios Papakyriakou, Ioannis S. Barbounakis. *Data Mining Methods: A Review*. International Journal of Computer Applications. 183, 48 (Jan 2022), 5-19. DOI=10.5120/ijca2022921884

[2] Raspberry Pi 4 Model B. [Online]. Available: raspberrypi.com/products/raspberry-pi-4-model-b/.

[3] Raspberry Pi 4 Model B specifications. [Online]. Available: https://magpi.raspberrypi.com/articles/raspberry-pi-4-specs-benchmarks

[4] Aurelien, M. (2022). *PEP 668 – Marking Python base environments as externally managed. Python Software Foundation*. https://peps.python.org/pep-0668/

[5] J. Dean and S. Ghemawat, "*MapReduce: Simplified data processing on large clusters*," Commun. ACM, vol. 51, no. 1, pp. 107–113, Jan. 2008

[6] M. Zaharia et al., "*Apache Spark: A unified engine for big data processing,*" Commun. ACM, vol. 59, no. 11, pp. 56–65, Nov. 2016

[7] A. Sergeev and M. Del Balso, "*Horovod: fast and easy distributed deep learning in TensorFlow*," arXiv preprint arXiv:1802.05799, 2018

[8] Google, "*Multi Worker Mirrored Strategy Guide*," TensorFlow Docs, 2023

[9] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message Passing Interface*, 3rd ed., MIT Press, 2014

[10] J. Dongarra et al., "*High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems*," Int. J. High Perform. Comput. Appl., vol. 30, no. 1, pp. 3–10, Feb. 2016

[11] M. Rocklin, "*Dask: Parallel computation with blocked algorithms and task scheduling*," Proc. 14th Python in Science Conference, 2015

[12] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., & Stoica, I. (2010). *Spark: Cluster computing with working sets. In Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud'10)*. USENIX Association

[13] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., ... & Zheng, X. (2016). *TensorFlow: A system for large-scale machine learning. In Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)* (pp. 265–283). USENIX Association

[14] Sergeev, A., & Del Balso, M. (2018). *Horovod: fast and easy distributed deep learning in TensorFlow*. In Proceedings of the 31st Conference on Neural Information Processing Systems (NeurIPS) Workshop

[15] Thakur, R., Rabenseifner, R., & Gropp, W. (2005). *Optimization of collective communication operations in MPICH.* In Proceedings of the International Conference on Computational Science (ICCS 2005*)* (pp. 49–57). Springer

[16] Dean, J., & Ghemawat, S. (2008). *MapReduce: Simplified data processing on large clusters*. *Communications of the ACM, 51*(1), 107–113. https://doi.org/10.1145/1327452.1327492

[18] Gropp, W., Lusk, E., & Skjellum, A. (2014). *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT press.

[19] Dongarra, J., Beckman, P., Moore, T., et al. (2021). *The International Exascale Software Project Roadmap*. International Journal of High-Performance Computing Applications, 35(1), 3–60

[20] Kogias, E., Christou, I. T., & Triantafyllidis, G. (2020). *Distributed Machine Learning on Edge Devices: A Survey*. *IEEE Access*, 8, 211309–211328

[21] Mariani, L., Bartolini, A., Borghi, G., & Benini, L. (2022). *Scalable Edge Machine Learning on Raspberry Pi Clusters*. Future Generation Computer Systems, 128, 190–203