data

MDPI

Article

A Robust Distributed Clustering of Large Data Sets on a Grid of Commodity Machines

Salah Taamneh [1,*,†], Mo'taz Al-Hami [2,†], Hani Bani-Salameh [3,†] and Alaa E. Abdallah [1,†]

[1] Department of Computer Science, The Hashemite University, Zarqa 13133, Jordan; aabdallah@hu.edu.jo
[2] Department of Computer Information Systems, The Hashemite University, Zarqa 13133, Jordan; motaz@hu.edu.jo
[3] Department of Software Engineering, The Hashemite University, Zarqa 13133, Jordan; hani@hu.edu.jo
* Correspondence: taamneh@hu.edu.jo
† These authors contributed equally to this work.

Abstract: Distributed clustering algorithms have proven to be effective in dramatically reducing execution time. However, distributed environments are characterized by a high rate of failure. Nodes can easily become unreachable. Furthermore, it is not guaranteed that messages are delivered to their destination. As a result, fault tolerance mechanisms are of paramount importance to achieve resiliency and guarantee continuous progress. In this paper, a fault-tolerant distributed k-means algorithm is proposed on a grid of commodity machines. Machines in such an environment are connected in a peer-to-peer fashion and managed by a gossip protocol with the actor model used as the concurrency model. The fact that no synchronization is needed makes it a good fit for parallel processing. Using the passive replication technique for the leader node and the active replication technique for the workers, the system exhibited robustness against failures. The results showed that the distributed k-means algorithm with no fault-tolerant mechanisms achieved up to a 34% improvement over the Hadoop-based k-means algorithm, while the robust one achieved up to a 12% improvement. The experiments also showed that the overhead, using such techniques, was negligible. Moreover, the results indicated that losing up to 10% of the messages had no real impact on the overall performance.

Keywords: k-means clustering; distributed k-means algorithm; actor model; active replication; passive replication; peer-to-peer network

Citation: Taamneh, S.; Al-Hami, M.; Bani-Salameh, H.; Abdallah, A.E. A Robust Distributed Clustering of Large Datasetson a Grid of Commodity Machines. *Data* **2021**, *6*, 73. https://doi.org/10.3390/data6070073

Academic Editor: Kassim S. Mwitondi

Received: 26 April 2021
Accepted: 14 June 2021
Published: 7 July 2021

1. Introduction

Commodity machines are everywhere and often available in large numbers. Such machines, if put together, represent tremendous computing power. Building a distributed system from these available resources for compute-intensive applications is considered a cost-effective choice, compared to hosting and maintaining a high-performance computer [1]. A grid of commodity machines consists of autonomous machines connected via a LAN network. These machines interact with each other to solve computational problems that cannot be solved individually. Computers in such systems are loosely coupled, heterogeneous, have no common physical clock, and have no shared memory; yet, they appear to users as a single coherent computer. The advantages of using a distributed system include (but are not limited to): increasing the performance/cost ratio, ensuring reliability, improving scalability, and sharing resources. Distributed systems with loosely coupled machines are not considered the best choice for fine-grained parallel programs, as the latency delay caused by the frequent communication over the network would significantly degrade the overall performance. Therefore, such systems are mainly used for running coarse-grained parallel programs where the communication/computation ratio is low [2]. Programs with coarse-grained parallelism are characterized by their low communication and synchronization overhead.

Iterative algorithms are a class of algorithms that use an initial set of values to generate a sequence of successive approximations to reach a solution [3]. Each of these algorithms has a termination criterion to determine when the algorithm stops. Examples of this class of algorithms include rank search, k-means clustering, and the Jacobi method. An iterative algorithm starts with initial values and performs some computations to generate a new set of values. After that, the convergence condition is tested: if the condition is met, the algorithm stops; otherwise, another round of computations begins. The steps of iterative algorithms are shown in Figure 1.

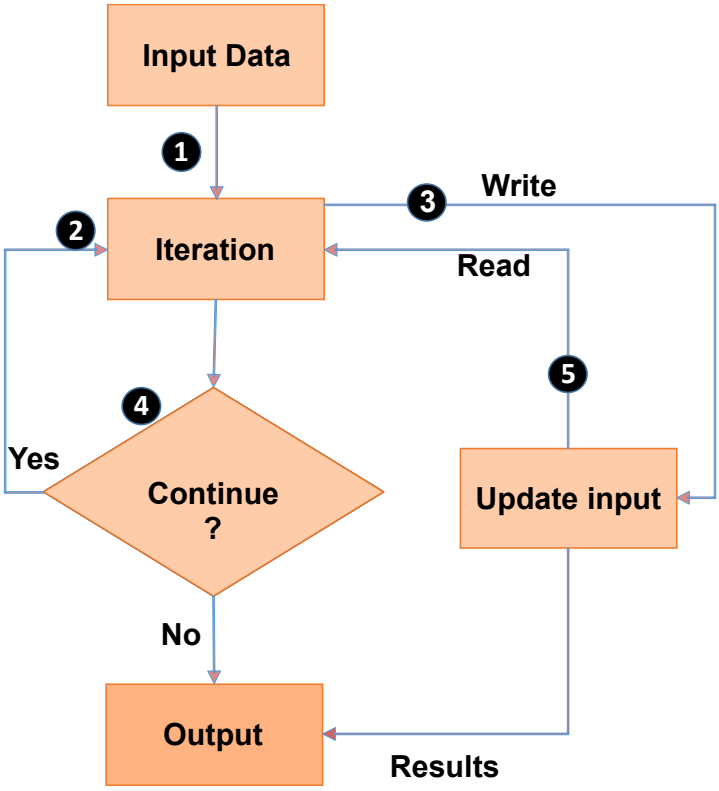

Figure 1. The flowchart of iterative algorithms.

Iterative algorithms are considered a good fit for Single-Instruction Multiple-Data (SIMD) parallel processing [4]. Several instances of the same task that is responsible for performing operations on a subset of the input data can be run in a parallel fashion, where each instance is assigned a range of data points. Once all instances have finished their work, they make their results available for a special kind of task. This task, often called the master, needs to gather these partial results, update the values of the last iteration, and decide whether to perform another round or not. Making data available from the workers to the master can generally be performed in two different ways: shared memory and message passing. The former has the disadvantage of requiring the access to the shared data to be synchronized. Therefore, the message-passing model is considered more efficient and scalable.

The k-means algorithm is an iterative algorithm that groups a set of data points into a given number of clusters [5]. Each data point in the data set is assigned to the closest cluster (centroid). The k-means algorithm follows a heuristic approach to find a good enough solution; finding the optimal solution is an NP-hard problem. A random set of points is first selected and used as the centroids. The algorithm then performs iterative (repetitive) calculations to optimize the positions of the centroids. The optimization process is stopped when either there are no significant changes to the centers' values or when the defined number of iterations has been reached. Algorithm 1 illustrates the steps of the sequential k-means algorithm.

In this paper, an attempt was made to harness the computing power of commodity machines for coarse-grained parallel algorithms, such as the k-means algorithm. The computation was distributed among several computational entities running on commodity machines connected over a decentralized peer-to-peer network. The model of the concurrent execution used in the proposed system was the actor model. The actor model consists of several computational entities called actors, which interact with each other via asynchronous message passing; therefore, no synchronization is needed [6]. The Akka framework was used to implement the presented distributed algorithm [7].

Algorithm 1: The sequential k-means algorithm.

Input: (k = number of clusters, $X = \{x_1, x_2, x_3, \ldots., x_n\}$ (a set of data points)
Output: set of k clusters $S = \{s_1, s_2, s_3, \ldots., s_k\}$

Method
Randomly select cluster centroids $C = \{c_1, c_2, c_3, \ldots., c_k\}$;
for *a given number of iterations,* **do**
    Calculate the distance between each data point and the cluster centers;
    Assign each data point x to the nearest cluster using $argmin_{c_i \in C} E_{dist}(c_i, x)^2$;
    Recalculate the new cluster centroids using $c_i = \frac{\sum_{x_i \in s_i} x_i}{|s_i|}$;
    if *no data point was reassigned* **then**
        stop.

The contributions of this work are as follows: (1) a distributed implementation of the k-means algorithm using the actor model, which was by design free of the race condition; (2) a distributed clustering method that stored the intermediate data in memory, thus reducing the overall time required; (3) a robust computing environment that was resilient to failures, such as node failures and lost messages.

2. Related Work

The k-means method has gained popularity due to its simplicity and efficiency compared to other clustering methods [5]. The literature is abundant with studies that have sought to enhance the performance of this method. One problem with the k-means algorithm is that the value of k, the number of desired clusters, must be known before running the algorithm. Therefore, several methods have been proposed for finding the optimal value of k. The elbow method was among the first widely used methods for determining the most appropriate number of clusters in a data set [8]. The idea behind this method is to plot the values of cost (i.e., within-cluster variance) as the value of k changes. The plot takes the shape of an arm, and the point at which the arm bends is used as the optimal k value. The silhouette coefficient is another method used to analyze the distance between the resulting clusters via visual representation to determine the best k-value, visually [9,10]. This method measures how similar a data point is to its own cluster (cohesion), compared to other clusters (separation). The value of the silhouette ranges between [1, −1], where a high value means that the data point is well matched to its own cluster and poorly matched to neighboring clusters. Another method, called gap statistics, compares the change within intra-cluster variation between observed data and reference data with a random uniform distribution [11]. The number of clusters is determined as the smallest value of k such that the gap statistic is within one standard deviation of the gap at k + 1.

Another set of studies focused on addressing the scalability of the k-means algorithm by proposing various distributed implementations of the sequential k-means algorithm. These implementations aim to exploit the computational power of commodity machines to achieve good speedup over the sequential versions. One set of studies proposed parallel k-means clustering algorithms using MPI, a message-passing model for distributed computing [12–15]. Typically, a master–slave Single Process Multiple Data (SPMD) approach is used in this model. The master process is responsible for broadcasting the centroids

along with a subset of the data set to all slaves. The slaves, in turn, assign a cluster for each data point and return the results. The master then computes the new centroids and continues until reaching the optimal solution. The results demonstrated that the MPI-based k-means algorithm achieves up to 1.7 times speedup against the sequential k-means algorithm [16]. However, MPI was found efficient only when the size of the data set is small or moderate [17]. Moreover, MPI does not provide any fault-tolerant constructs for users to handle failures [18]. Thus, the recovery procedure is activated only when the application is aborted.

The MapReduce parallel programming model has recently gained attention for running the k-means algorithm on a cluster of machines. Apache Hadoop is an open-source framework for running applications on clusters of commodity machines, using the MapReduce programming model [19,20]. Hadoop was the most widely used MapReduce-based framework for clustering large data sets in a distributed fashion [21–27]. The MapReduce programming model consists of two phases: map and reduce. Data are first divided into subsets of a specified size. These subsets are then distributed among various mappers running on several nodes. The clusters' centers are distributed to all mappers at the beginning of each map task. Mappers assign a cluster for each data point in a subset. The results generated by the mappers are stored in the Hadoop Distributed File System (HDFS). A combiner is then used to build a partial sum for each cluster. The reducers process these values to compute the new centers and store the results in the file system. These steps are repeated until the convergence criteria are met. A recent study conducted by [24] showed that the k-means algorithm based on the Hadoop framework with 3 slave nodes achieves up to 2 times speedup against the sequential k-means. However, the overhead accompanied by reading and writing data to the local storage at each iteration harms the overall performance of the Hadoop framework [28].

In an attempt to overcome the above-mentioned Hadoop drawback, the Apache Spark cluster computing framework was developed. Spark was designed to support a wider range of applications than MapReduce. In this framework, data are kept in memory using a distributed memory abstract called Resilient Distributed Data sets (RDDs). RDDs can restore the lost data, using lineages. Each transformation-like map or join causes a new RDD to be formed. The RDDs are immutable and thus easy to recover. The fact that spark stores data in memory has drawn researchers' attention to use Spark for k-means clustering [29–31]. Spark can process data stored in any Hadoop compatible data source, such as HDFS, Cassandra, HBase, etc. The typical Spark-based k-means algorithm starts with loading the data from HDFS into RDDs. Data are horizontally divided and distributed among the machines. Random instances are then selected as initial centroids from RDDs and broadcast to all nodes. After computing the distance between all data points and centroids, the aggregator operator is used for each node to reduce the amount of data to be transferred. The partial results are then collected from RDDs and sent to the central nodes, where the new centroids are computed. The same steps are then repeated until the stopping condition is met. It was proven that Spark is $2\times$ to $5\times$ faster than Hadoop for k-means clustering [25,32]. One of the main drawbacks of Spark is that it is not designed to deal with small files. Additionally, Spark relies on other platforms for file management.

In summary, the most widely used frameworks for running the k-means algorithm in a distributed manner are MPI, Hadoop, and Spark. Several studies showed that the MPI-based k-means algorithm achieves good speedup, compared to the sequential one. However, the efficiency of MPI-based parallel applications degrades when dealing with large data sets. Moreover, programming with MPI requires programmers to explicitly deal with the individual nodes' status and communication patterns [33]. Finally, failures in MPI are dealt with by using stop-and-restart checkpointing solutions [34]. The Apache Hadoop framework was designed to process large data sets on commodity hardware. Hadoop achieves reliability by replicating data across several nodes. It was found to achieve a significant speedup over the sequential k-means algorithm. However, the overhead accompanied by storing and retrieving data to/from the HDFS at each iteration degrades

the overall performance of the Hadoop framework [28]. Additionally, Hadoop is not suited for small files [35]. Spark is another cluster-computing framework for running parallel applications. In Spark, intermediate results are written in memory. This helps overcome the bottleneck of the Hadoop framework.

The actor model programming paradigm has been effectively used in a variety of distributed applications, ranging from small applications to big data processing [7,36–39]. The ability of this model to scale resiliently and tolerate faults were among the key reasons for adopting it for building distributed systems. The actor model also provides a higher-level abstraction than the shared state concurrency models, where each exchange of data has to be implemented using hand-crafted sends and receives. Moreover, with the actor model, there is no need for locks or any other synchronization techniques. This helps developers focus more on coordinating access to data rather than protecting all accesses to data using synchronization techniques.

In this paper, we propose a robust distributed k-means algorithm on a cluster of commodity machines connected in a peer-to-peer fashion. The actor model was used to implement the proposed algorithm. Several fault-tolerance techniques were utilized to make the algorithm resilient to failures, which are common in commodity machines. The Akka framework was used to build the distributed algorithm.

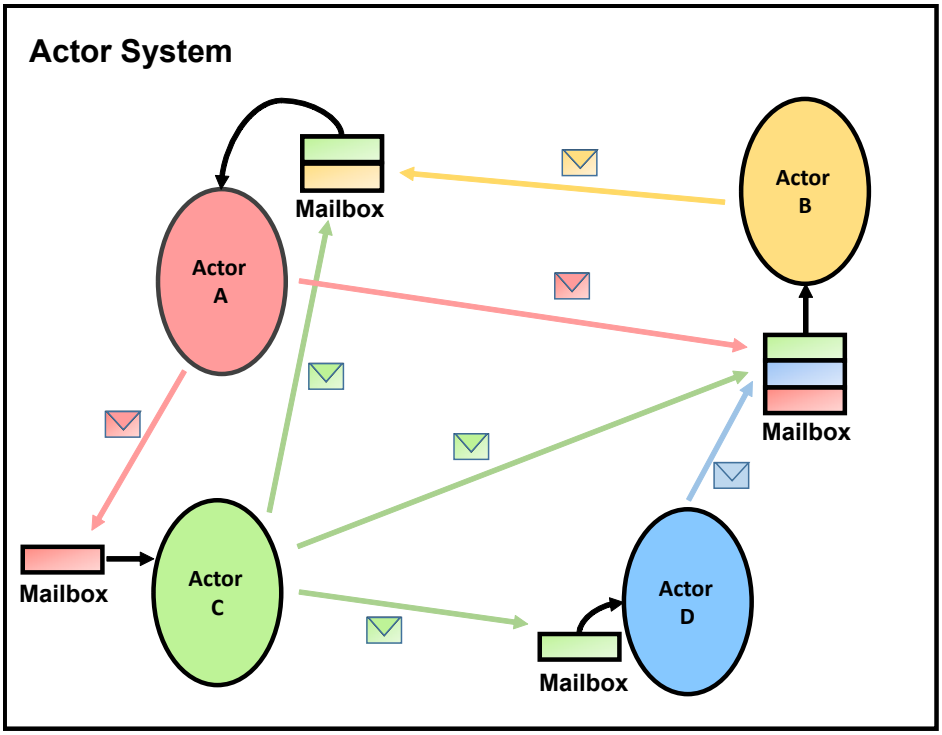

Figure 2. An example of 4 actors sending messages to each other.

3. Implementation

In this section, an implementation of a distributed k-means algorithm on a grid of commodity machines is presented. First, the principles of the actor model are explained in detail. Second, an actor model-based implementation of a distributed k-means algorithm with no-fault-tolerance mechanisms is described. The issues that this implementation suffers from are then discussed. Finally, a robust implementation that addresses the issues in the preliminary algorithm is presented.

3.1. The Actor System

The actor model consists of small units, called actors, communicating exclusively by exchanging messages [7]. Each actor consists of a state, behavior, and mailbox. Ideally, changes to the actor's internal state are made by the behavior, which is a reaction

to a received message in the mailbox. Processing messages in the actor's mailbox are done sequentially in a FIFO fashion, while sending messages to other actors is done asynchronously. That is, an actor sends a message and continues executing without blocking. Figure 2 provides an example of several actors communicating with each other via message passing.

Actors in the actor model are logically organized into a hierarchical structure called the actor system. An actor willing to split up the task into smaller pieces can create child actors and delegate the task to them. This actor is then responsible for supervising its children and reacting to their failures.

3.2. The Actor System on a Cluster of Nodes

The Akka cluster module enables developers to build an application that spans multiple nodes, where each node runs a part of the application [40]. Nodes in such a cluster are connected using a decentralized peer-to-peer topology. For the nodes to be part of the same cluster, they need to use the same name for the actor system running on them. Each actor in the cluster is uniquely identified using the following format: <akk.tcp://actorsystemname@ip:port/user/parent1/ partent2 ..... /actor-name>.

Each cluster should be configured to have one or more contact points called seed nodes. These nodes are used by new members to join the cluster. Nodes willing to join the cluster should include these seed nodes in their configurations. The actor system running on each node should be designed to be able to do the following:

1. Join the cluster. This is done by using the cluster extension, which, behind the scene, handles all the aspects of joining the cluster. This of course requires using the correct seed nodes in the configuration file.
2. Recognize the other parts of the actor system and the role assigned to each one of them. This is attained by subscribing to change notifications of the cluster membership. The subscribed nodes receive information about the joining nodes and the role assigned to each one.
3. Exchange messages with the targeted nodes transparently without necessarily knowing their physical locations. The actor system in each node is programmed to communicate with the remote actor system using its logical address. The physical address is automatically identified at runtime.

Cluster membership is disseminated using a gossip protocol in which the state of the cluster is gossiped randomly through the cluster, with preference given to members that have not yet seen the latest version. Messages are sent between nodes using Jackson, a Java-based library for serializing objects to the JSON format [41]. The Phi accrual failure detector is used for node failure detection [42]. Some important aspects of clusters in Akka are presented next.

- Joining the cluster: Each node in the cluster is identified using a <hostname:port:uid> tuple, where the uid is used to uniquely identify a specific actor system running on <hostname:port> machine. Some nodes must be set as seed nodes with known IP and port numbers. These nodes are used as contact points for new nodes to join the cluster. A node is introduced to the cluster by sending *JOIN* message to one of the cluster members. If the cluster consists only of seed nodes, this message is sent to one of them; if not, the message can be sent to any member of the cluster. Figure 3 provides an overview of a cluster with five nodes with two seed nodes.
- Assigning a leader: The role of the leader in a cluster is to add and remove nodes to/from the cluster. This role is given to the first node at which the cluster coverages to a globally consistent state, and can be given to different nodes between rounds of convergence.
- Membership Lifecycle: After sending a *JOIN* message, the state of the node is switched to Joining. Once all nodes have seen that a node is joining the cluster, the state of that node is changed to Up, and it can then participate in the cluster. If a node opts to leave the cluster, the state is changed to Leaving state. Once all nodes have seen that a node

is leaving, the state is switched to existing and eventually will be marked as removed. Each node in the cluster is monitored by several other nodes. When a node becomes unreachable, for whatever reason, the rest of the nodes in the cluster receive this information via the gossip protocol. The unreachable nodes need to become reachable again or be removed from the cluster, as a cluster with an unreachable node cannot add new nodes.

- Node roles: A specific role can be assigned to any node in the cluster. The cluster can then be configured to specify the required number of nodes with a certain role. In such a case, the status of the joining nodes will only be changed to Up when the required number of nodes have already joined the cluster.
- Failure detector: The nodes in a cluster monitor each other using heartbeating. Each node keeps a record of the heartbeat of each node in the cluster. Based on that history, the decision is made about whether a node is reachable or not. However, this does not lead to rapid failure of the node. When a node determines that another node is unreachable, it disseminates this information to other nodes in the cluster. Once all nodes receive this information (i.e., convergence is reached) that node is marked as unreachable. All nodes will keep on trying to connect with the unreachable node. If it remains disconnected for a relatively long period of time, it is marked as Down and removed from the cluster.

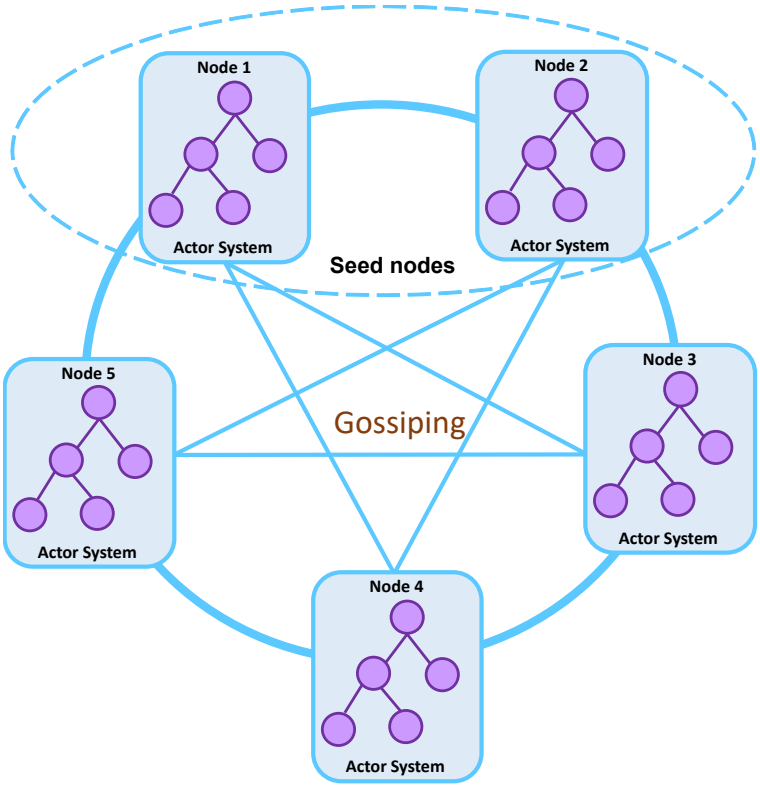

Figure 3. A peer-to-peer cluster of five nodes with two seed nodes.

3.3. The Preliminary Model

In this section, we present the preliminary implementation of the distributed k-means algorithm. This implementation serves as a proof of concept; therefore, no fault-tolerant mechanisms were used. As shown in Figure 4, the preliminary model consists of two types of nodes: master and worker. The master node is a singleton node, that is, the system can have only one node with that role, whereas several workers are allowed to function simultaneously. A separate code was developed for each role, using Akka and Scala. The master node is responsible for splitting and distributing the data among workers as well as collecting the results. At each iteration, the master node builds a new set of

centroids and checks whether the convergence criteria are met or not. Each worker is in charge of computing the nearest centroids for the potion of the data set assigned to it. Next, we explain in detail how each node is implemented.

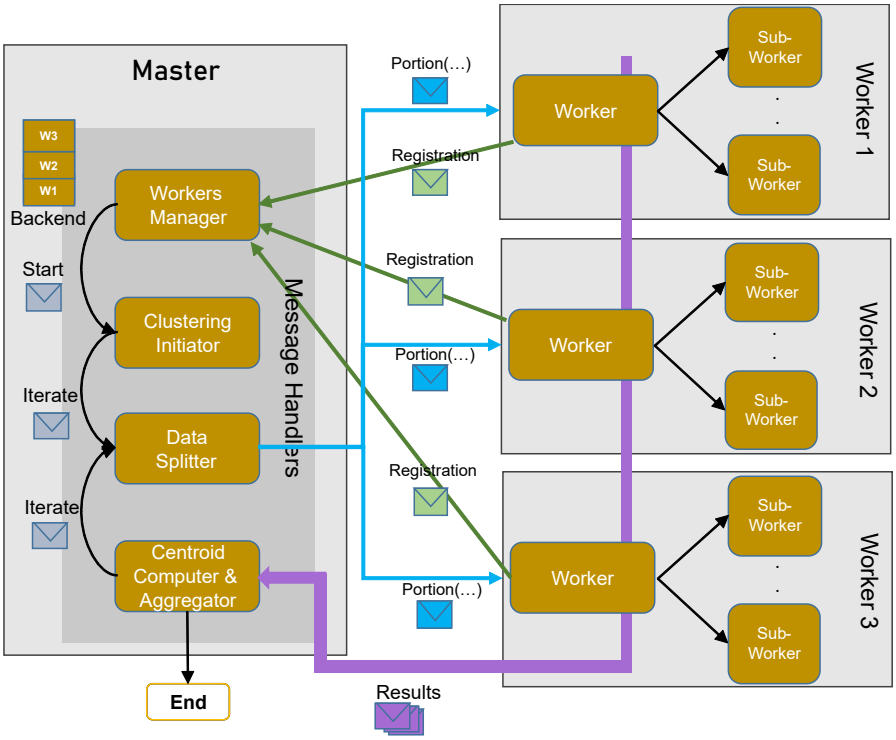

Figure 4. The architecture of the preliminary model.

3.3.1. The Master Node

The whole process starts at the master node. The minimum number of workers is first secured ahead of starting the clustering process. After that, the data set is logically split into portions, where each portion is assigned to a particular worker. At the first iteration, a random set of centroids is selected and distributed among workers. For the subsequent iterations, the centroids are computed using the results obtained from the workers. After constructing the new centroids, the convergence criteria are tested. If the criteria are met, the clustering process stops. Otherwise, it continues iterating. The centroids are computed using the following equation:

$$c_i = \frac{1}{|S_i|} \sum_{x \in S_i} x_i \tag{1}$$

where S_i is the set of all points assigned to the ith cluster.

The k-means algorithm aims to divide the data set into k clusters so as to minimize the within-cluster sum of squares:

$$argmin_s \sum_{x=1}^{k} \sum_{x \in S_i} ||x - \mu||^2 \tag{2}$$

The code running on the master node consists of an actor of the type master, created by an actor system with a "ClusterSystem" name. This name needs to be maintained for all nodes in the cluster to allow them to communicate with one another. When the master actor is launched, it waits until the required number of workers joins the cluster. Upon reaching the required number of workers, the state of all nodes is switched from Joining to Up.

The master node consists of four message handlers, each responsible for a particular task. Next, we provide a detailed description of each handler:

1. The workers' manager: This handler maintains a data structure of the type map, called backends, to keep track of all workers in the cluster. Upon receiving a WorkRegistration message from a worker, the master adds the address of that worker to the backend. On registering the minimum number of workers, the master actor sends a Start message to itself, a message that is handled by the clustering process initiator.
2. The clustering process initiator: This handler is only invoked once at the beginning of the clustering process. When a Start message is received, a random set of centroids is selected, and an Iterate message is sent to the data splitter handler.
3. The data splitter: This handler is executed at the beginning of each iteration. It is responsible for distributing data evenly among workers along with the current centroids, using a portion(..) message. At each iteration, the algorithm iterates over the backend map and splits the data among the registered workers.
4. The centroids computer and data aggregator: This handler is responsible for collecting the results from all workers. These results are sent by workers via WorkerDone messages. The data aggregator receives these messages from all workers and consequently constructs a new set of centroids. After that, the algorithm checks whether the convergence criteria were met or not. If not, An Iterate message is sent to the splitter. These steps are repeated until the optimal solution or the maximum number of iterations is reached.

3.3.2. The Worker Node

Workers join the cluster, using one of the seed nodes. When a worker becomes a member of a cluster, it immediately subscribes to the cluster membership events. Such a subscription enables workers to receive information about each other. For each node that joins the cluster, a worker receives a MemberUp(m) Message. Upon receiving this message, the role of that node is checked. If the role is Master, a WorkerRegistration(noOfCores) message is sent to the master. After sending this message, the worker waits for the master to send a Portion message, containing the portion of data assigned to it, along with the current centroids. Worker nodes are designed to make use of all available cores to accelerate the clustering process. A number of SubWorker actors equal to the number of available cores are then created by the worker. The data points received from the master are distributed evenly among the SubWorkers. For each data point, SubWorkers identify the closest cluster, using the Euclidean distance. The sum and count of all points in each cluster are also maintained. The data point x is assigned to a cluster based on the following:

$$argmin_{c_i \in C} dist(c_i, x)^2 \tag{3}$$

After completing the assigned task, a SubWorkerDone message is sent back to the worker, notifying it that a SubWorker has just finished its job.The worker actor serves as a task distributor and data aggregator for all SubWorkers. After receiving the results from all SubWorkers, they are aggregated and included in a WorkerDone message that is sent back to the master, marking the end of a worker's task. The master gathers this information from all workers and computes the new centroids.

In the preliminary model, it is assumed that the data set is stored in the local disk of each worker. Loading the data set into memory is initiated by the prestart() method. This ensures that the data will be available for processing once the portion message arrives from the master.

3.3.3. Issues with the Preliminary Model

The initial model works fine in failure-free scenarios. The achieved speedup was found to be very promising, specifically when using a large number of nodes. However, the model has some limitations, such as manual distribution of data and the lack of a mechanism for validating the results received. Moreover, such a distributed environment is prone to failure. Nodes may suddenly become unreachable or may simply crash. In addition, it is not guaranteed that messages will be delivered to their destination.

In this paper, we propose a robust distributed k-means algorithm that is resilient to failures. The issues addressed in the new implementation are summarized below:

- The master is a single point of failure in the system. If the master fails, the whole clustering process fails.
- If one worker is lagging, the whole process will be waiting for that worker to finish.
- If one worker fails, the last iteration needs to be repeated and the load needs to be distributed among the intact workers.
- Messages are not guaranteed to be delivered to their destination.
- Workers can temporarily become unreachable. This leads to losing the messages sent from the worker to master or vice versa.

In addition to the issues mentioned above, the initial model has the following restrictions:

- There is no result checking.
- Data splitting is done manually.

3.4. A Robust, Distributed k-Means Algorithm

To make the proposed environment resilient to failures, several techniques were incorporated into the initial model. First, the active replication technique was used to create multiple replicas of each worker. In this technique, the same commands are executed in each replica, thus overcoming the problem of having lagging workers or failed ones. Second, the passive replication technique was used to create multiple replicas of the master node. In this technique, the primary replica processes the request and then updates the state of the other replicas. If the primary node fails, one of the other replicas takes its place. Third, a message acknowledgment technique was used to ensure that messages were delivered to their final destination. Finally, the process of distributing data among workers was automated. The architecture of the proposed system is presented in Figure 5.

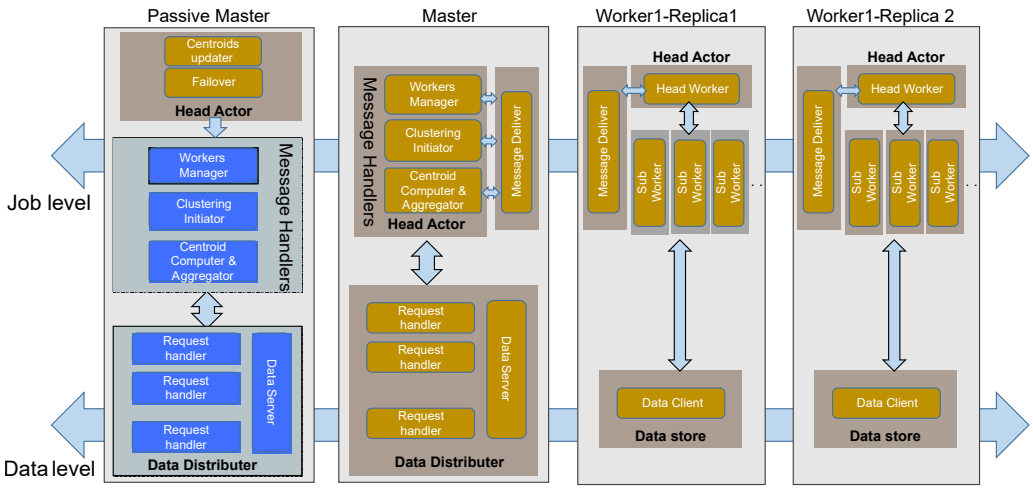

Figure 5. The architecture of the robust system.

3.5. Data Distribution

In the preliminary model, data are manually distributed among workers. This is a cumbersome, inefficient way to distribute data. In this paper, the data distribution process was automated. Data sets are initially stored in the local storage of the master node. As shown by the Algorithm 2, when the required number of workers join the cluster, an actor called the data distributor is created and, at the same time, a Start message is sent to the self. In response to this message, each worker is assigned an ID number (Algorithm 3-line 11). All replicas performing the same task are assigned the same ID, with each ID being associated with a specific range of data points based on the total number of workers (Algorithm 3). This information is stored in a map, called workerInfo, which the data

distributor uses to split data between workers. Finally, the assigned ID along with the current centroids are sent to each worker and all of its replicas.

The data distributor actor listens on a predetermined port number, which is known by all workers. When a worker receives an ID number, it creates an actor called Data Store that connects to the server, requesting its portion of the data. Data are then sent to the client as a raw byte stream. The client receives those data and stores them in a proper format to be used by the worker. Once the data are available, the worker starts working, sending the results back to the master.

Algorithm 2: Workers Manager Handler—Master Node

```
   Result: securing the minimum number of nodes
1  case MemberUp(m) =>
2  if m.hasRole("masterReplica") then
3      masterReplica = context.actorSelection(RootActorPath(m.address)/../
        "masterReplica");
4      masterReplica ! WatchMe
5  case WorkerRegistration(cores) =>
6  if !backends.contains(sender()) then
7      context.watch(sender()) ;
8      backends += sender() ->cores ;
9      if (backend.size() == noOfWorker*noOfReplica) AND !working then
10         create a data distributor actor;
11         self ! Start;
```

Algorithm 3: Clustering Initiator Handler—Master Node

```
   Result: Initiating the clustering process
1  case Start =>
2  var fromRow=0; var toRow = -1; var id =0 ;
3  val portion= noRow / totalCores ;
4  for (i =0; i<backends.size(); i++) do
5      fromRow = toRow + 1 ;
6      if i == numOfWorkers - 1 then
7          toRow = noRow - 1
8      else
9          toRow = fromRow + (portion * cores) - 1
10     workerInfo(id) = assignedData(fromRow, toRow) ;
11     for j =0; j<noOfReplica; j++ do
12         backends.get(i) ! ID(id, currentCetnroids) ;
13     id+=1;
14 for (j<- 0 until numOfWorkers) do
15     receivedWorker += j -> false
```

3.6. Failing and Lagging Workers

In the absence of any fault tolerance mechanisms, the failure of a worker leads to the loss of the entire computation performed before the failure. Additionally, if some workers are lagging behind the other workers, the whole process is affected because the k-means algorithm cannot proceed until it receives the results from all workers. Performing the same task on multiple nodes can make the system more resilient to failures. This is referred to as active replication. In this technique, a task is replicated on multiple nodes, and the same request is processed by all replicas. If one of the replicas fails, the other replicas can get the job done. Moreover, this technique reduces the impact of lagging workers on system performance. If one replica is lagging, the results can still be obtained from other replicas.

After specifying the number of replicas per worker, the master waits until the required number of nodes joins the cluster. Each worker is assigned a unique ID by the master. Replicas performing the same tasks are assigned the same ID. As illustrated by Algorithm 4, this ID is used by the master to distinguish between workers and to ignore the already-received messages from other replicas as well. Figure 6 provides an example of a replica that is lagging behind and trying to send outdated results to the master.

Algorithm 4: Data Aggregator Handler—Master Node

```
   Result: collecting results from workers and deciding when to stop
 1 case WorkerDone(itr, id, localSum, localCount)
   /* avoid receiving outdated results                                  */
 2 if itr == currentIteration then
       /* avoid receiving replicated results                            */
 3     if !received(id) then
 4         received(id) = true;
 5         sender() ! Akw(itr);
 6         globalSum += localSum ;
 7         globalCount += localCount ;
 8         workerDone +=1;
 9         if all workers sent their results then
10             newCentroids = globalSum / globalCount;
11             if stop criteria met then
12                 stop clustering process
13             else
14                 centroids = newCentroids;
15                 masterReplica ! Centers(centroids, round);
16                 self ! Iterate;
17     else
18         sender() ! Akw(itr)
19 else
20     sender() ! Akw(itr)
21 case Iterate ;
22 for ((worker, cores) <- backends ) do
23     worker ! begin(centroids);
```

Each message sent to the master is accompanied by the <worker ID, iteration No> tuple. The iteration number is important to handle the situation when a previously unreachable node or lagging node becomes available and starts sending outdated results. An additional benefit that we can get by performing the same task on multiple nodes is ensuring that the results received from workers are correct. By comparing the results received from several replicas, we can find out whether the results received are valid or not.

3.7. Master Node Failure

The master node is a single point of failure in the system. Losing such a node will definitely result in stopping the whole clustering process. Therefore, a backup plan needs to be put in place to deal with such a failure. Akka uses an accrual failure detector that dynamically adjusts to reflect the current network conditions [42]. Heartbeats are sent every second by default, which is configurable. The suspicion level of failure is represented by a value called ϕ. The value of ϕ is interpreted as follows: given some threshold Φ, and we consider that a node fails when $\phi > \Phi = 2$, then the probability that we make a mistake is 1%, 0.1% with $\Phi = 3$, 0.01% with $\Phi = 4$, and so on.

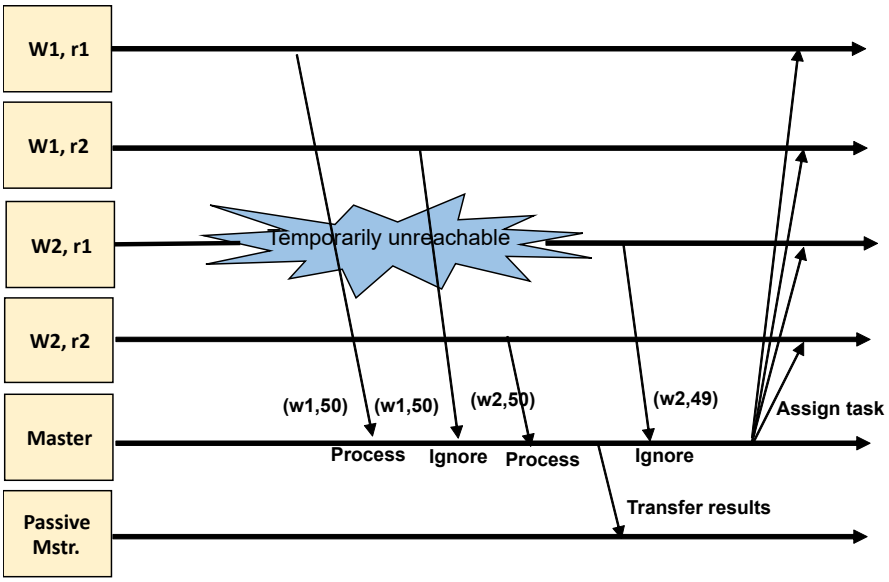

Figure 6. An example of a replica that is lagging behind and trying to send outdated results to the master.

The steps of estimating the value of ϕ are (1) storing the heartbeats' arrival times in a sampling window, (2) determining the distribution of inter-arrival times using the stored samples, and (3) computing the current value of ϕ using the distribution. The value of ϕ is calculated as follows:

$$\phi(t_{now}) = -log_{10}(P_{later}(t_{now} - T_{last})) \tag{4}$$

where T_{last} is the time when the last heartbeat was received, t_{now} is the current time, and P_{later} is the probability that a heartbeat will arrive more than t time units after the previous one. The estimation of the distribution of inter-arrival times assumes that inter-arrivals follow a normal distribution. The parameters of the distribution are estimated from the sampling window by determining the mean μ and the variance σ^2 of the samples. P_{later} is computed using the following formula:

$$P_{later} = \frac{1}{\sigma\sqrt{2\pi}} \int_{i=1}^{+\infty} e^{-\frac{(x-\mu)^2}{2\sigma^2}} dx \tag{5}$$

Figure 7 illustrates how the value of Phi(ϕ) increases as the time since the last heartbeat increases when the heartbeat interval = 1000 ms and standard deviation = 200 ms.

In this paper, the failover process is utilized to move the system from a state of failure back to its working state. In this process, a secondary copy of the master node is maintained. In the case of failure, that copy takes the responsibility of the master node. Replicating the master node can be done in two different ways:

- Active replication: The same operations are performed at each replica.
- Passive replication: Only the main master performs the operations and transfers the results to the other replicas.

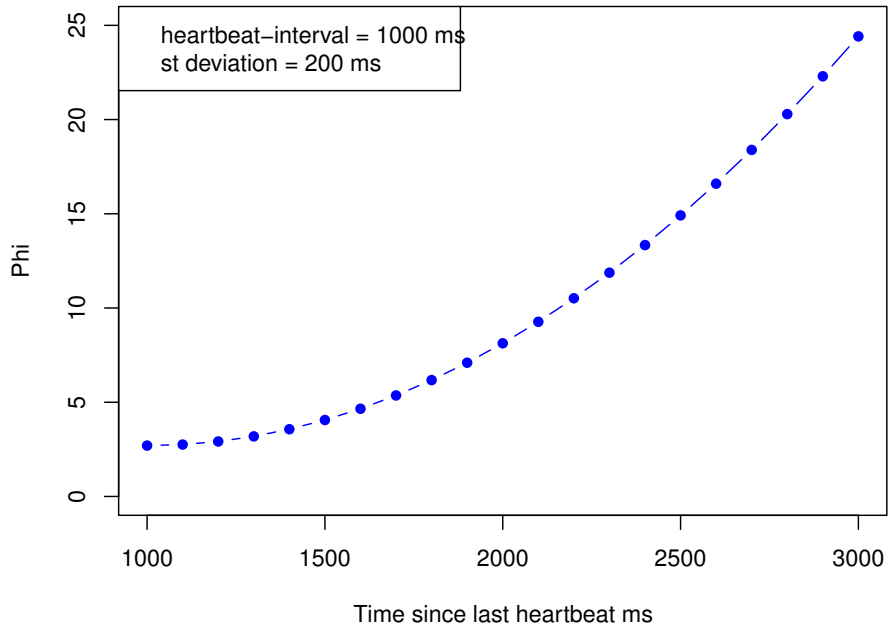

Figure 7. An example of how the values of Phi change as the time since the last heartbeat increases when the heartbeat-interval = 1000 ms and the standard deviation = 200 ms.

Passive replication was used in this paper, as it incurs less communication overhead in the cluster. The primary master collects the partial results from all workers, constructs the new centroids, and then transfers the new centroids to the passive replicas before proceeding with the next iteration. The implementation of the passive master node is presented in Algorithm 5. All workers should register themselves in the passive replicas as well as in the active ones. In failure-free scenarios, no real communication takes place between workers and the passive master. This, however, is not the case when the primary master fails. The process of switching to the secondary master consists of the following steps:

1. Determining that the master has failed: The secondary master should be able to detect the primary master failure and react accordingly. When the primary master receives a MemberUp message, it checks the role of that member; if the role is passiveMaster, the master keeps the address of that node to provide it with the updated centroids. A WatchMe message is also sent to the passive master. Upon receiving that message, the passive master starts watching the main master (Algorithm 5-line 3) using something called Deathwatch. Deathwatch uses the Akka cluster failure detector for nodes in the cluster. It detects JVM crashes and network failures as well as the graceful termination of watched nodes, and as a result, a Terminated() message is generated and sent to the passive replica.
2. Promoting a secondary master to primary: Upon receiving a Terminated() message, the passive master changes its internal state, using Become(master) to start behaving as a primary master (Algorithm 5 line 8).
3. Reconfiguring the system to communicate with the new master: A message is then sent from the new master to all workers, instructing them to start sending their results to the new master (Algorithm 5-lines 5–7).

Figure 8 provides the sequence diagram for a cluster with 2 replicas when the master node fails.

Algorithm 5: Passive Master

```
1  case WatchMe =>
2  master = sender() ;
3  context.watch(master);
4  case Terminated(a) =>
5  if a == master then
6      for ((worker, cores) <- backends ) do
7          worker ! UpdateMaster(currentRound);
8      context.become(becomeMaster) ;
9      self ! Iterate
10 case Centers(centroids, iter) =>
11 curretCenters = centroids ;
12 currentRound = iter ;
13 case WorkerRegistration(cores) =>
14 if !backends.contains(sender()) then
15     context.watch(sender()) ;
16     backends += sender() ->cores ;
```

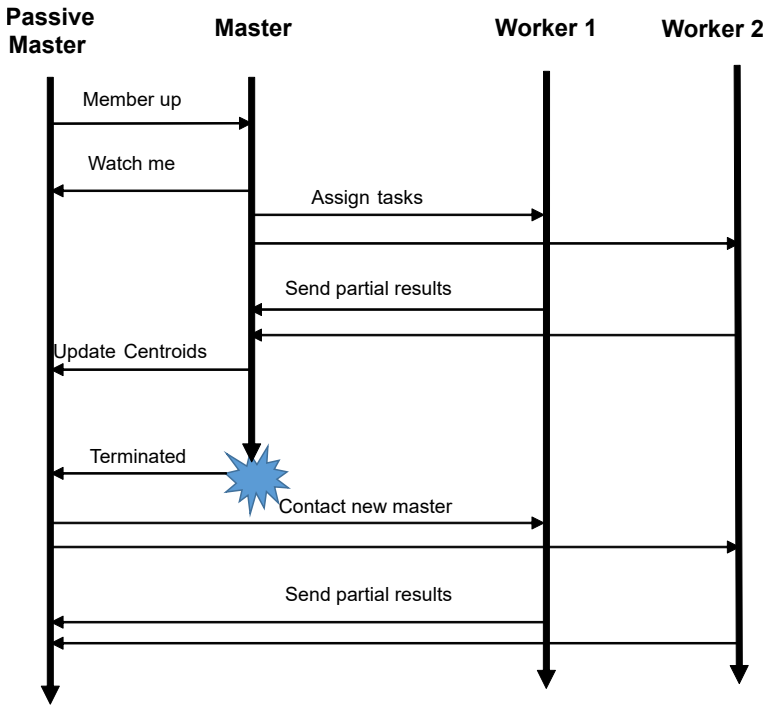

Figure 8. The sequence diagram for a system with 2 replicas when the master node fails.

3.8. Message Delivery

Although Akka guarantees that messages sent from one actor to another will not arrive out of order, there is no guarantee that messages will ever be delivered. The only way for a sender to know that a message was successfully delivered is via receiving an acknowledgment message from the receiver. To address this issue, a messageHandler actor is created for each message to be delivered. This actor forwards the received message to its destination and switches its behavior to Waiting for acknowledgment. At the same time, a message with the same information is scheduled to be sent to the actor itself after some time. Upon receiving this message, the messageHandler checks if the ack has been received or not; if not, the message is sent again. This way, the message is eventually delivered to its final destination.

4. Results

In this section, the performance and robustness of the proposed environment are evaluated. A cluster of commodity machines was first built and used to run the distributed k-means algorithm on two data sets, using different experimental setups. The results were then obtained and reported in this section.

4.1. Experimental Setup

A grid of tens of nodes was exploited to build a distributed environment on which the experiments were performed. These nodes ran a Windows 10 operating system and were connected using 100 Mbps wired Ethernet. Each node had an Intel (R) Core (TM) i7-6700 CPU, @ 3.4 GHz (8 CPUs) processor with 8 GB RAM. The experiments are based on two synthetic data sets that were created in order to evaluate the proposed environment. As shown in Table 1, the first data set is of size 0.5 G and contains 2.5 million observations, each represented by 68 attributes. The second data set is of size 1 G and contains 5 million observations, each represented by 80 attributes.

Table 1. The list of data sets used in the experiments.

Data Set	No. of Rows (millions)	No. of Features	Size
DS1	2.5	68	0.5 G
DS2	5.0	80	1 G

Several experiments were conducted to shed light on the performance of the proposed algorithm under different scenarios. The first set of experiments aimed to evaluate the performance of the preliminary model and robust model against the Hadoop-based k-means algorithm. The second set of experiments assessed the performance of the robust model when using different numbers of replicas, thus providing an insight into the overhead associated with using replication as a fault tolerance technique. The third set of experiments was designed to test the resiliency of the proposed algorithm when the master node fails. Finally, the fourth set of experiments measured the effect of lost messages on the overall performance.

4.2. Evaluation of the Preliminary Model

This experiment aimed to measure the improvement gained by using the preliminary and robust models as compared to the Apache Hadoop. The same nodes used for running the proposed algorithm were also used to run the Hadoop version of the k-means algorithm.

The preliminary, robust, and Hadoop-based k-means algorithms were applied on DS1 with k = 400, and k = 800, and on DS2 with k = 500, and k = 100, using 1, 2, 4, and 8 nodes. The results are presented in Figures 9 and 10. As indicated by the figures, the preliminary model achieved between 18% and 34% improvement over the Hadoop-based k-means, while the robust distributed k-means algorithm achieved up to a 12% improvement.

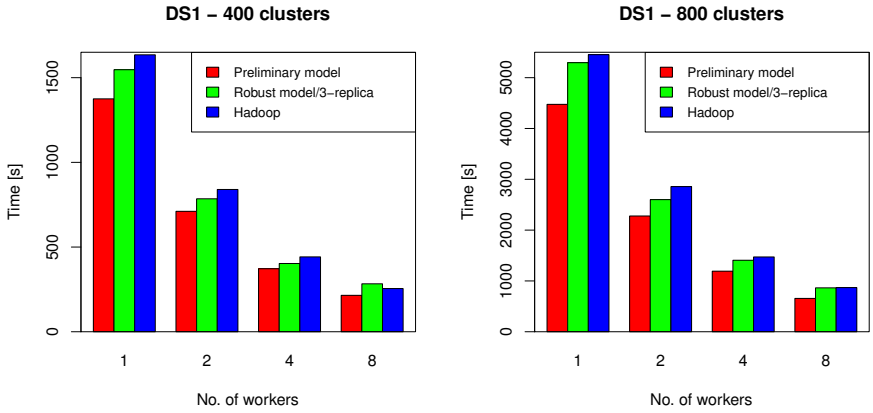

Figure 9. The execution time of running the Hadoop-based k-means algorithm, the proposed k-means algorithm with no fault tolerance, and the robust k-means algorithm on DS1, using one, two, four, and eight nodes, where k = 400 (**left**) and k = 800 (**right**).

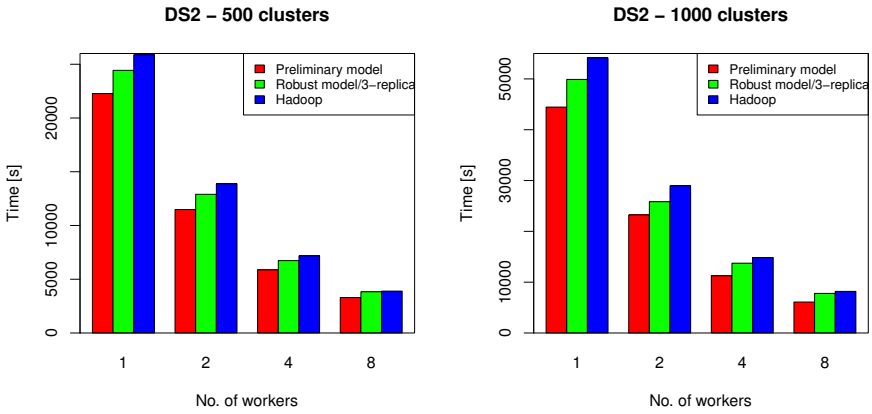

Figure 10. The execution time of running the Hadoop-based k-means algorithm, the proposed k-means algorithm with no fault tolerance, and the robust k-means algorithm on DS2 using one, two, four, and eight nodes, where k = 500 (**left**) and k = 1000 (**right**).

4.3. The Impact of Using Several Replicas on the Overall Performance

In order to measure the overhead introduced by the use of different numbers of replicas, the proposed algorithm was run on DS1 and DS2 data sets, using zero, one, two, and three replicas. The distributed k-means algorithm was run twice on DS1, one with k = 400 and the other with k = 800, where k is the number of clusters. The algorithm was also run on DS2 for k = 500 and k = 1000. Each run was repeated five times, using one, two, four, eight, and 16 workers. The execution times of the distributed k-means algorithm on DS1 and DS2 are shown in Figures 11 and 12, respectively. As indicated by the figures, the overhead was found to be insignificant, compared to the cost of restarting the clustering process from scratch in the event of failure when replication is not used. The figures also suggest that using one replica generally gave the best performance with an overhead ranging from 8–19% of the original execution time when the number of workers was equal to or greater than 4. Interestingly, the results showed that when the number of workers was relatively small (<4) the overhead of using more than one replica was indistinguishable from that of the single replica. Indeed, it was lower in some cases. This was due to the fact that assigning the same task to multiple nodes minimizes the possibility of having lagged workers, thus providing the best performance. This gain, however, diminishes when using more workers as the master node gets overwhelmed with messages received from multiple nodes, each with multiple replicas.

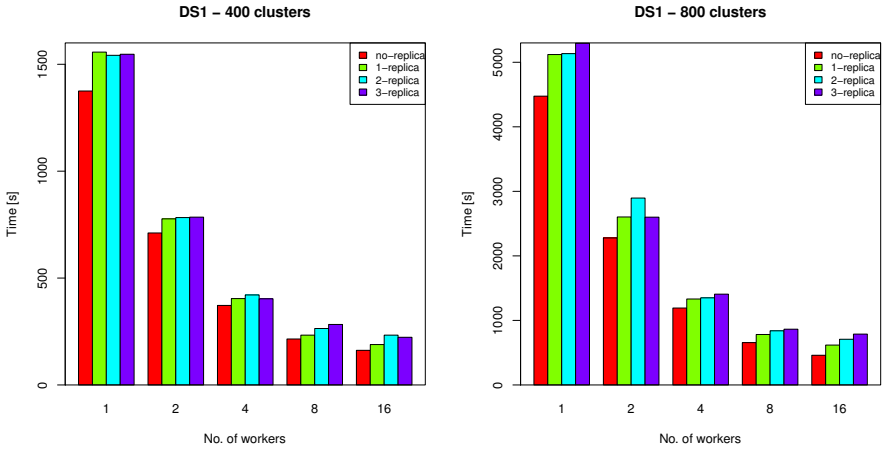

Figure 11. The execution time of running the distributed k-means algorithm on DS1 using one, two, four, eight and 16 workers with zero, one, two and three replicas, where k = 400 (**left**) and k = 800 (**right**).

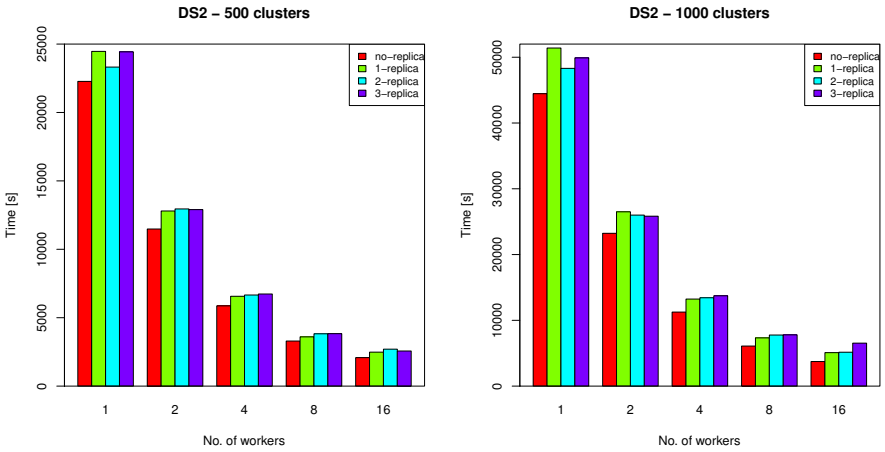

Figure 12. The execution time of running the distributed k-means algorithm on DS2 using one, two, four, eight and 16 workers with zero, one, two and three replicas, where k = 500 (**left**) and k = 1000 (**right**).

The speedup achieved when using replicas was also investigated. The speedup is defined as how much faster a problem runs on P processors, compared to one processor, and is computed using the following equation:

$$Speedup(P) = \frac{T_{total}(1)}{T_{total}(P)} \tag{6}$$

The speedups for DS1, and DS2 when using zero, one, two, and three replicas with different numbers of workers (i.e., 1, 2, 4, 8, and 16) are presented in Figures 13 and 14, respectively. As can be noted from the figures, the speedup achieved by using up to four workers was almost identical for all numbers of replicas, while when using more than four workers, the speedup decreased as the number of workers and replicas increased. This is mainly attributed to the high communication overhead introduced when using larger numbers of workers and replicas.

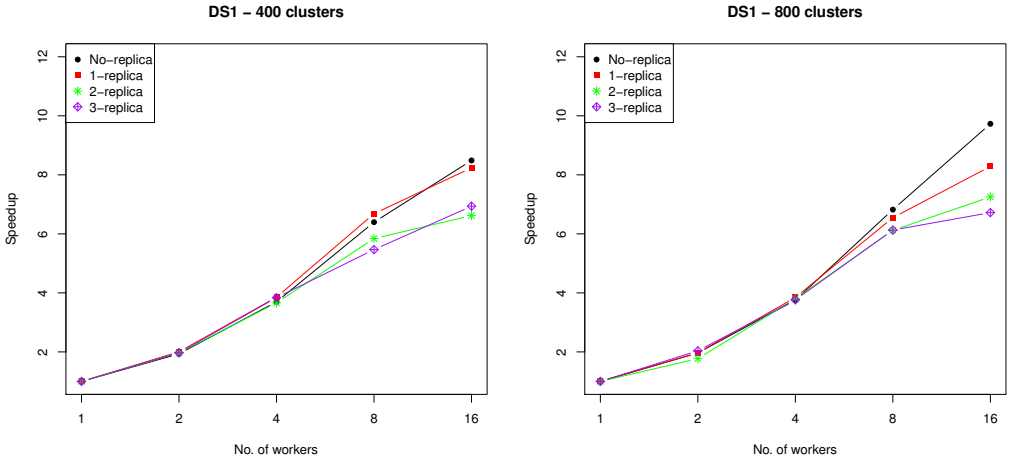

Figure 13. The speedup achieved by running the proposed algorithm on DS1, using 1, 2, 4, 8 and 16 workers with zero, one, two and three replicas, where k = 400 (**left**) and k = 800 (**right**).

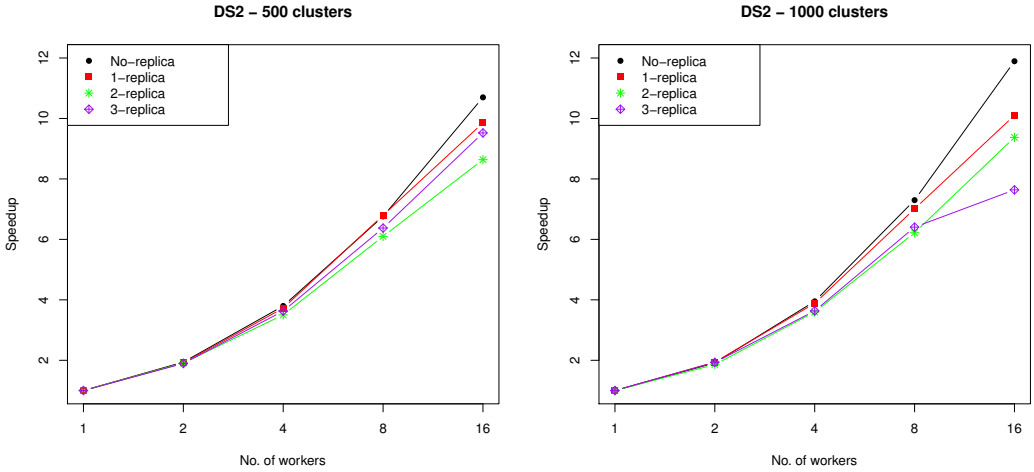

Figure 14. The speedup achieved by running the proposed algorithm on DS2 using 1, 2, 4, 8 and 16 workers with zero, one, two and three replicas, where k = 500 (**left**) and k = 1000 (**right**).

4.4. Master Failure

The master node is very critical to the entire system. If it fails, the failure must be quickly detected and a secondary copy must be upgraded to take charge of the primary copy. In this experiment, the primary master was forced to fail to gain insight into how this affected overall performance. The failure detector threshold was set to be 8 (i.e., the suspicious level is 0.000001%), which is suitable for most situations. Table 2 shows the execution time for DS1 with k = 800 using 4, 8, and 16 workers with one replica when the primary master node fails. The same conditions were used for DS2 with k = 1000 (see Table 3). The results showed that the time taken to detect the failure and resume the clustering process following a master failure was negligible.

Table 2. The execution time of the proposed algorithm on DS1 (k = 800) using 4, 8 and 16 workers with one replica when the master node fails.

Methods	4 Nodes	8 Nodes	16 Nodes
No master failure	1332	782	618
with master failure	1371	819	654

Table 3. The execution time of the proposed algorithm on DS2 (k = 1000) using 4, 8 and 16 workers with one replica when the master node fails.

Methods	4 Nodes	8 Nodes	16 Nodes
No master failure	13,233	7333	5088
with master failure	13,283	7361	5134

4.5. Undelivered Messages

As mentioned earlier, there is no guarantee that a message will be delivered to the node that it is supposed to be delivered to. Therefore, a delivery acknowledgment technique was used to ensure that a message will eventually be received by the receiver. In this experiment, the performance was measured when 10 %, 25%, and 50% of the messages sent from workers to the master were intentionally dropped. Figures 15 and 16 show the performance of the proposed algorithm on DS1 with k = 800 and on DS2 with k = 1000, respectively, using 4, 8, and 16 workers with one replica. As suggested by the figures, the algorithm can still deliver satisfactory performance, even when a large number of messages is lost. In most cases, a loss of up to 10% of messages has no real impact on the overall performance. This makes the proposed algorithm well-suited for unreliable environments, where message loss is not unusual.

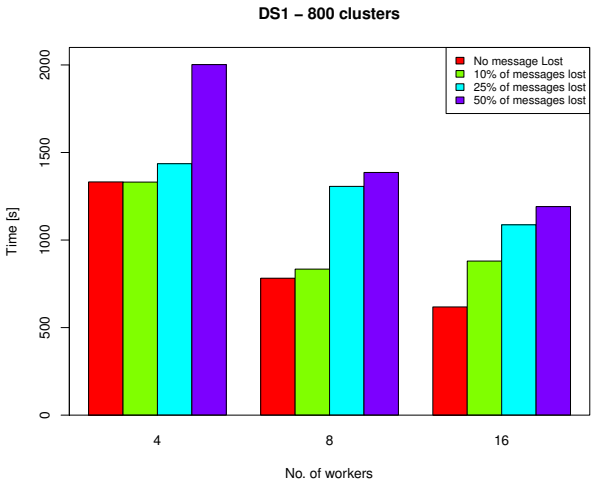

Figure 15. The performance of the proposed algorithm on DS1 (k = 800) using 4, 8 and 16 workers with one replica when 10%, 25% and 50% of messages are lost.

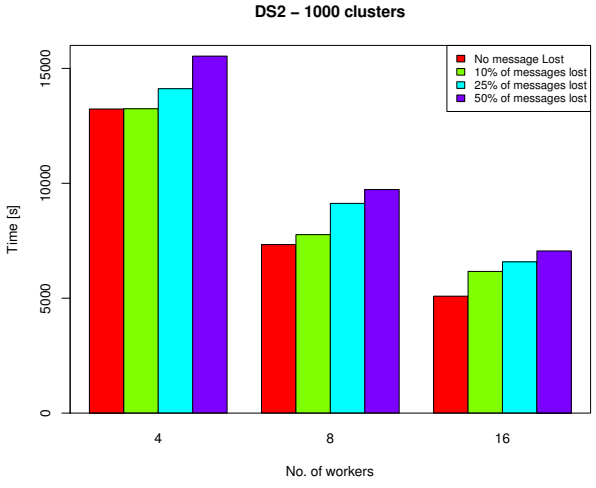

Figure 16. The performance of the proposed algorithm on DS2 (k = 1000) using 4, 8 and 16 workers with one replica when 10%, 25% and 50% of messages are lost.

5. Discussion

In this paper, a robust and distributed k-means algorithm was presented. The algorithm was implemented using the actor model and was designed to run on a network of commodity machines where failures are common and extremely difficult to avoid. Therefore, several techniques were employed to ensure continuous progress in the presence of failure. Active and passive replication techniques were used, as well as a technique for message delivery acknowledgment. The performance of the method was evaluated on two data sets.

The results showed that the distributed k-means algorithm with no fault-tolerant mechanism (i.e., the preliminary algorithm) achieved up to a 34% improvement over the Hadoop-based k-means, while the robust, distributed k-means algorithm achieved up to a 12% improvement. However, the preliminary algorithm is not tolerant to failures that are very common in a cluster of commodity machines. The results also revealed that the overhead introduced by using replicas was insignificant, especially when the alternative is to restart the entire clustering process from scratch. While the results suggest that using one replica is slightly better than using more replicas in most cases, the performance gained can still be satisfactory even when two or three replicas are used. Therefore, in environments where nodes are unreliable, using two or more replicas is highly recommended.

The experiments also highlighted the impact of master failure on overall performance. Such failure requires detecting the failure, promoting a secondary copy to be a primary one, and notifying workers about this change. It is evident from the results that the time needed to detect the failure and resume the clustering process is quite trivial.

The proposed system also addresses the issue of lost messages. Losing messages from/to the master is something intolerable for the presented distributed k-means algorithm, as it prevents it from making any progress. A message delivery technique was used to ensure that messages are delivered to their final destination. The experimental results indicate that losing up to 10% of messages has no significant impact on performance.

Finally, while the k-means algorithm was presented in the paper, the proposed methodology can easily be applied to all iterative algorithms.

Author Contributions: Conceptualization, S.T.; methodology, S.T. and M.A.-H.; implementation, S.T., H.B.-S. and M.A.-H.; experiments, S.T., A.E.A.; writing—original draft preparation, S.T., M.A.-H. and H.B.-S.; writing—review and editing, A.E.A. and S.T. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data presented in this study are available on request from the corresponding author.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Sedlmayr, M.; Würfl, T.; Maier, C.; Häberle, L.; Fasching, P.; Prokosch, H.U.; Christoph, J. Optimizing R with SparkR on a commodity cluster for biomedical research. *Comput. Methods Programs Biomed.* **2016**, *137*, 321–328. [CrossRef] [PubMed]
2. Balis, B.; Figiela, K.; Malawski, M.; Jopek, K. Leveraging workflows and clouds for a multi-frontal solver for finite element meshes. *Procedia Comput. Sci.* **2015**, *51*, 944–953. [CrossRef]
3. Mohebi, A.; Aghabozorgi, S.; Ying Wah, T.; Herawan, T.; Yahyapour, R. Iterative big data clustering algorithms: A review. *Softw. Pract. Exp.* **2016**, *46*, 107–129. [CrossRef]
4. Al Hasib, A.; Cebrian, J.M.; Natvig, L. A vectorized k-means algorithm for compressed datasets: Design and experimental analysis. *J. Supercomput.* **2018**, *74*, 2705–2728. [CrossRef]
5. Zhao, W.L.; Deng, C.H.; Ngo, C.W. k-means: A revisit. *Neurocomputing* **2018**, *291*, 195–206. [CrossRef]
6. Albert, E.; Flores-Montoya, A.; Genaim, S.; Martin-Martin, E. May-happen-in-parallel analysis for actor-based concurrency. *ACM Trans. Comput. Log. (TOCL)* **2015**, *17*, 1–39. [CrossRef]

7. Srirama, S.N.; Dick, F.M.S.; Adhikari, M. Akka framework based on the Actor model for executing distributed Fog Computing applications. *Future Gener. Comput. Syst.* **2021**, *117*, 439–452. [CrossRef]
8. Yuan, C.; Yang, H. Research on K-value selection method of k-means clustering algorithm. *J. Multidiscip. Sci. J.* **2019**, *2*, 16. [CrossRef]
9. Hasanzadeh-Mofrad, M.; Rezvanian, A. Learning automata clustering. *J. Comput. Sci.* **2018**, *24*, 379–388. [CrossRef]
10. Arsan, T.; Hameez, M.M.N. A clustering-based approach for improving the accuracy of UWB sensor-based indoor positioning system. *Mob. Inf. Syst.* **2019**. [CrossRef]
11. El-Mandouh, A.M.; Mahmoud, H.A.; Abd-Elmegid, L.A.; Haggag, M.H. Optimized k-means clustering model based on gap statistic. *Int. J. Adv. Comput. Sci. Appl.* **2019**, *10*, 183–188. [CrossRef]
12. Liang, F.; Lu, X. Accelerating iterative big data computing through mpi. *J. Comput. Sci. Technol.* **2015**, *30*, 283–294. [CrossRef]
13. Savvas, I.K.; Sofianidou, G.N. A novel near-parallel version of k-means algorithm for n-dimensional data objects using mpi. *Int. J. Grid Util. Comput.* **2016**, *7*, 80–91. [CrossRef]
14. Savvas, I.K.; Sofianidou, G.N. Parallelizing k-means algorithm for 1-d data using mpi. In Proceedings of the 2014 IEEE 23rd International WETICE Conference, Parma, Italy, 23–25 June 2014; pp. 179–184.
15. Nhita, F. Comparative Study between Parallel K-Means and Parallel K-Medoids with Message Passing Interface (MPI). *Int. J. Inf. Commun. Technol. (IJoICT)* **2016**, *2*, 27. [CrossRef]
16. Zhang, J.; Wu, G.; Hu, X.; Li, S.; Hao, S. A parallel clustering algorithm with mpi-mkmeans. *J. Comput.* **2013**, *8*, 1017. [CrossRef]
17. Sardar, T.H.; Ansari, Z. An analysis of MapReduce efficiency in document clustering using parallel k-means algorithm. *Future Comput. Inform. J.* **2018**, *3*, 200–209. [CrossRef]
18. Losada, N.; González, P.; Martín, M.J.; Bosilca, G.; Bouteiller, A.; Teranishi, K. Fault tolerance of MPI applications in exascale systems: The ULFM solution. *Future Gener. Comput. Syst.* **2020**, *106*, 467–481. [CrossRef]
19. Park, D.; Wang, J.; Kee, Y.S. In-storage computing for Hadoop MapReduce framework: Challenges and possibilities. *IEEE Trans. Comput.* **2016**. [CrossRef]
20. Bani-Salameh, H.; Al-Qawaqneh, M.; Taamneh, S. Investigating the Adoption of Big Data Management in Healthcare in Jordan. *Data* **2021**, *6*, 16. [CrossRef]
21. Sreedhar, C.; Kasiviswanath, N.; Reddy, P.C. Clustering large datasets using k-means modified inter and intra clustering (KM-I2C) in Hadoop. *J. Big Data* **2017**, *4*, 1–19. [CrossRef]
22. Ansari, Z.; Afzal, A.; Sardar, T.H. Data Categorization Using Hadoop MapReduce-Based Parallel K-Means Clustering. *J. Inst. Eng. (India) Ser. B* **2019**, *100*, 95–103. [CrossRef]
23. Vats, S.; Sagar, B. Performance evaluation of k-means clustering on Hadoop infrastructure. *J. Discret. Math. Sci. Cryptogr.* **2019**, *22*, 1349–1363. [CrossRef]
24. Lu, W. Improved k-means clustering algorithm for big data mining under Hadoop parallel framework. *J. Grid Comput.* **2019**, *18*, 239–250. [CrossRef]
25. Gopalani, S.; Arora, R. Comparing apache spark and map reduce with performance analysis using k-means. *Int. J. Comput. Appl.* **2015**, *113*, 8–11 . [CrossRef]
26. Issa, J. Performance characterization and analysis for Hadoop k-means iteration. *J. Cloud Comput.* **2016**, *5*, 1–15. [CrossRef]
27. Al-Hamil, M.; Maabreh, M.; Taamneh, S.; Pradeep, A.; Bani-Salameh, H. Apache Hadoop performance evaluation with resources monitoring tools, and parameters optimization: IOT emerging demand. *J. Theor. Appl. Inf. Technol.* **2021**, *99*, 2734–2750.
28. Won, H.; Nguyen, M.C.; Gil, M.S.; Moon, Y.S.; Whang, K.Y. Moving metadata from ad hoc files to database tables for robust, highly available, and scalable HDFS. *J. Supercomput.* **2017**, *73*, 2657–2681. [CrossRef]
29. Omrani, H.; Parmentier, B.; Helbich, M.; Pijanowski, B.The land transformation model-cluster framework: Applying k-means and the Spark computing environment for large scale land change analytics. *Environ. Model. Softw.* **2019**, *111*, 182–191. [CrossRef]
30. Haut, J.M.; Paoletti, M.; Plaza, J.; Plaza, A. Cloud implementation of the k-means algorithm for hyperspectral image analysis. *J. Supercomput.* **2017**, *73*, 514–529. [CrossRef]
31. Xing, Z.; Li, G. Intelligent Classification Method of Remote Sensing Image Based on Big Data in Spark Environment. *Int. J. Wirel. Inf. Netw.* **2019**, *26*, 183–192. [CrossRef]
32. Shi, J.; Qiu, Y.; Minhas, U.F.; Jiao, L.; Wang, C.; Reinwald, B.; Özcan, F. Clash of the titans: Mapreduce vs. spark for large scale data analytics. *Proc. VLDB Endow.* **2015**, *8*, 2110–2121. [CrossRef]
33. Abuín, J.M.; Lopes, N.; Ferreira, L.; Pena, T.F.; Schmidt, B. Big Data in metagenomics: Apache Spark vs MPI. *PLoS ONE* **2020**, *15*, e0239741. [CrossRef]
34. Losada, N.; Martín, M.J.; González, P. Assessing resilient versus stop-and-restart fault-tolerant solutions in MPI applications. *J. Supercomput.* **2017**, *73*, 316–329. [CrossRef]
35. Zhang, Y.; Zhu, Z.; Cui, H.; Dong, X.; Chen, H. Small files storing and computing optimization in Hadoop parallel rendering. *Concurr. Comput. Pract. Exp.* **2017**, *29*, e3847. [CrossRef]
36. Starzec, M.; Starzec, G.; Byrski, A.; Turek, W. Distributed ant colony optimization based on actor model. *Parallel Comput.* **2019**, *90*, 102573. [CrossRef]
37. Bagheri, M.; Sirjani, M.; Khamespanah, E.; Khakpour, N.; Akkaya, I.; Movaghar, A.; Lee, E.A. Coordinated actor model of self-adaptive track-based traffic control systems. *J. Syst. Softw.* **2018**, *143*, 116–139. [CrossRef]

38. Duan, J.; Yi, X.; Zhao, S.; Wu, C.; Cui, H.; Le, F. NFVactor: A resilient NFV system using the distributed actor model. *IEEE J. Sel. Areas Commun.* **2019**, *37*, 586–599. [CrossRef]
39. Taamneh, S.; Qawasmeh, A.; Aljammal, A.H. Parallel and fault-tolerant k-means clustering based on the actor model. *Multiagent Grid Syst.* **2020**, *16*, 379–396. [CrossRef]
40. Gupta, M. *Akka Essentials*; Packt Publishing Ltd.: Birmingham, UK, 2012.
41. Friesen, J. Processing JSON with Jackson. In *Java XML and JSON*; Springer: Dauphin, MB, Canada, 2019; pp. 323–403.
42. Hayashibara, N.; Defago, X.; Yared, R.; Katayama, T. The/spl phi/accrual failure detector. In Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, Florianopolis, Brazil, 18–20 October 2004; pp. 66–78.