Multiagent and Grid Systems – An International Journal 16 (2020) 379–396
DOI 10.3233/MGS-200336
IOS Press

Parallel and fault-tolerant k-means clustering based on the actor model

Salah Taamneh*, Ahmad Qawasmeh and Ashraf H. Aljammal
Department of Computer Science and Applications, The Hashemite University, Zarqa, Jordan

Received 13 February 2020
Accepted 8 September 2020

Abstract. K-means algorithm is a well-known unsupervised machine learning tool that aims at splitting a given dataset into a fixed number of clusters via iterative refinement approach. Running such an algorithm on today's datasets that are characterized by its high multidimensionality and huge size requires using fault-tolerance mechanisms to mitigate the impact of possible failures. In this paper, we propose an actor-based implementation of k-means algorithm. The algorithm was made fault-tolerant by periodically saving the centroids into a stable storage during the failure-free execution, and restarting from the last saved centroids upon a failure. This was implemented in two different ways: optimistic checkpointing (blocking) and pessimistic checkpointing (non-blocking). The actor-based k-means algorithm was evaluated on a machine with eight cores. The experiments showed that the proposed algorithm scales very well as the number of workers increases, and can be up to $\sim$ 2x faster than a Java-thread-based implementation of k-means algorithm. The results also showed that the optimistic algorithm outperformed the pessimistic one, specifically, in the presence of competing I/O operations. Several failures were forced to occur during the execution to evaluate the performance of the fault-tolerant implementations. The experiments showed that the average amount of lost work ranged from 3–6%.

Keywords: Parallel k-means, actor-model, checkpointing

1. Introduction

Clustering is a well-known machine learning technique that is applied in a variety of fields such as: image processing, computer vision, astronomy, pattern recognition, and bioinformatics. Clustering is an unsupervised method that aims at grouping data points into a set of classes called clusters, so that data points belong to the same cluster are more similar to each other than points in the other clusters. K-means is a partitioning-based clustering algorithm that aims at grouping n data points into k clusters, so that each data point belongs to the cluster whose mean has the least squared Euclidean distance (i.e., to the cluster with the nearest mean). Finding the optimal solution in such a problem is computationally difficult (NP-hard). Instead, efficient heuristic algorithms are used to converge quickly to a local optimum. The algorithm, in its simplest form, starts with a first group of randomly selected centroids, which are used as the beginning points for every cluster, and then performs iterative (repetitive) calculations to optimize the positions of the centroids. It stops optimizing the clusters when either there are no significant changes to the values of the centers, or when the defined number of iterations have been reached.

*Corresponding author: Salah Taamneh, Department of Computer Science and Applications, The Hashemite University, Zarqa, Jordan. Tel.: +962 790107961; Fax: +962 53826625; E-mail: taamneh@hu.edu.jo.

With people relying more and more on information technology in all aspects of their lives, the world's data are growing exponentially. It is sometime useful for companies and individuals to cluster their data as it helps identify interning patterns. Clustering today's data using the sequential clustering algorithms is not practical due the long execution time needed by such algorithms. As a result, more efficient methods of clustering need to be developed to cope up with the growing amount of data. Clustering data in a distributed and parallel fashion can greatly reduce the time needed to finish the job. However, even with this improvement, some huge datasets still need long time to be clustered. Running long jobs in a distributed, parallel fashion open the doors for failures along the way. Such failures are very costly as usually require the whole process to be restarted from the beginning. Therefore, using fault-tolerance mechanisms in such an environment is of a paramount importance.

In this paper we use the actor model to develop a parallel k-means algorithm that is tolerant to failures. The actor model is a model of concurrent computation that consists of several primitive units of computations, called actors, working concurrently. Actors, in this model, interact with each other using message passing. The fact that his model does not require any explicit locking makes it an appealing choice for a wide range of parallel applications. Fault-tolerance in this algorithm is achieved by using a well-known fault-tolerance technique, called checkpointing, with the actor model. In checkpointing, an actor's sate is periodically stored into a stable storage during the failure-free execution, and restoring the system back to a consistent state after a failure. The rest of the paper is organized as follows. The literature review is given in Section 2. In Section 3, the actor-based k-means algorithm is discussed in detail. The fault-tolerant actor-based k-means implementations are provided in Section 4. The performance evaluation is discussed in Section 5. Section 6 discusses related work. Finally, the conclusion and future work is outlined in Section 7.

2. Literature review

Various parallel k-means implementations have been proposed in the literature. These implementations fall under one of the following categories: shared memory-based, message passing-based, cluster computing framework-based, and GPU computing-based. In the following subsections, we provide a review of literature on each of these categories. These subsections were followed by a subsection about the applications of the actor model in the literature.

2.1. Shared memory-based k-means

Several studies implemented k-means algorithm using a shared memory model, called OpenMP (Open Specifications for Multiprocessing) [9,18,19]. In such a model, the part of the k-means algorithm that is responsible for assigning a cluster for each data point is placed in a parallel region, followed by a sequential code for computing the new centroids, and checking whether the convergence criteria is met or not. Such a region allows several threads to operate in parallel. Compared to Message Passing Interface (MPI) and Compute Unified Device Architecture (CUDA), the results showed that for small datasets OpenMP achieves the best results, while for large datasets CUDA is better. The results also showed OpenMP always outperforms OpenCL (Open Computing Language) model.

2.2. Message passing-based k-means

K-means algorithm has also been implemented using the message passing model. A set of studies proposed parallel k-means clustering algorithms using MPI, a message passing model for distributed computing [3,8,16–18]. 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 datasets 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 continue until reaching the optimal solution. The results demonstrated that implementing k-means clustering using MPI achieves a good speed up and scales well when the number of nodes or size of datasets increases.

2.3. Cluster computing framework-based k-means

Several studies have recently exploited the MapReduce parallel programming model to run the k-means algorithm in a parallel fashion [1,4,10–12,15]. These studies use Hadoop framework to facilitate running applications, which involve huge amount of data, in parallel on a large number of nodes. The MapReduce programming model consists of two phases: map and reduce. In this model, the input data is split into independent partitions which are processed by various mappers running on several nodes. The outputs of the mappers are then sorted, shuffled, and used as inputs to the reduce tasks. The input and the output of the job is stored in the file system. In the MapReduce-based k-means algorithm, the dataset is split and distributed to all mappers. Additionally, a global set of centroids are constructed and made available to all mappers before each map task. Mappers then assign a cluster for each data point. After that, a combiner is used to construct a partial sum for each cluster. These intermediate data are then distributed among the reducers. The reducers then compute the new centroids and store the results to a file-system. The same steps are repeated until the convergence criterion is met. Although Hadoop is naturally tolerant to failures as data are stored to disk after every operation, the fact that iterative algorithms have to run as a sequence of jobs, each of which read data from stable storage and write it back to stable storage incur significant overhead.

As a result of Hadoop's limitation, Apache spark cluster computing framework was developed. In spark, both input data and intermediate data can be stored in memory using distributed memory abstract called resilient distributed datasets (RDDs). This greatly reduce the cost of reading and writing from the file system. As a result, several studies have used Spark to run k-mean clustering in parallel fashion [12–15]. It has been proven that spark is 2x to 5x faster than Hadoop for k-mean clustering [12,15,24]. Sparks achieved fault tolerance via logging the transformations (lineage) used to build a dataset rather than the data itself. Lost partitions can be reconstructed by replaying the transformations applied to data. Although lineage can always be used to recover RDDs after a failure, such recovery may be time-consuming for RDDs with long lineage chains. Thus, it can be helpful to checkpoint some RDDs to stable storage.

2.4. GPU computing-based k-means

Graphic Processing Units' (GPUs) power has also been harnessed to achieve improvement in clustering algorithms. Several studies used OpenCL framework to run k-mean clustering on GPUs, CPUs, and other processor in parallel fashion [19,20,23]. The results suggested that the GPUs are in general faster than the CPUs for k-means clustering. Moreover, the results indicated that running the map task on the GPU and the reduce task on the CPU is not optimal for k-means clustering. Another set of studies used CUDA platform for k-means clustering [6,9,21,22]. The results showed that some CUDA-enabled GPU are significantly faster than the CPUs in k-means clustering. Additionally, it was found that shifting the task of data points assignment and centroids calculation to the GPU in parallel can achieve a considerable speedup.

2.5. The applications of the actor model in the literature

The actor model programming paradigm has been effectively used in a variety of applications ranging from small application to big data processing. The actor model has been recently applied in the field of smart grids. A co-simulation framework for analyzing properties of custom smart grids solutions was developed using Akka by [25]. The actor model was using in mapping every physical component in the smart grid to corresponding Information and Communications Technology (ICT) components. The results showed that the actor model can help efficiently run large-scale co-simulation experiments to analyze scalability and emergent properties of new solutions. Another study used the actor model to build a fault-tolerant distributed computing architecture for implementation a decentralized voltage stability monitoring and control application in the smart grid [26]. The built system was found to be efficient and scalable.

The actor model has been exploited to build a distributed actors framework for big data analysis [27]. The ability of this model to scale resiliently and tolerate faults were among the key reasons for choosing it to build such a framework. The system was designed to be used for composing and executing big data analytic in batch, streaming, or interactive workflow. Using the actor framework helped provide support for more dynamic workflows with automatic load balancing. A new distributed paradigm for cloud-based crowd sensing and IoT data collection and processing was proposed by [28]. This paradigm was built using the actor model to support some missing features in the available solutions such as multi-cloud and device multi-tenancy. Multi-clouds feature allows several cloud providers to communicate with the actor system, while multi-tenancy allows executing actors from different users on shared devices. The actor model was also exploited in building a sematic search engine using cohesion network analysis [29]. In this engine, SOLR platform was used for data storage and full text search, and akka was used for parallel and distributed processing. The results found to be promising, and the system is scalable and able to provide the search results in low response time. Akka actor platform was used for proposing a fine-grained level approach for managing execution of multiscale applications [30]. The results showed that using Akka as a scheduler for computational tasks (iterations) is a promising approach. A fault-tolerant system for managing a massive sets of Service Level Agreement (SLAs) using the actor system was presented by [31]. With the help of the actor system, the resources scheduling, outsourcing, and negotiation were separated into autonomous layers that can be combined into a parallelized, effective and efficient management structure. The results showed that the actor model is considered a good approach for the automated management of the SLA life cycle, especially negotiation and provisioning.

In this paper, we propose an actor-base implementation of the k-means algorithm. In this method, no synchronization techniques are needed as the parallel units interact with each other using message passing. Moreover, a fault tolerant technique, namely checkpointing, was used to make the proposed algorithm resilient to failures. This technique was implemented in two different ways: optimistic checkpointing, and pessimistic checkpointing. To the best of our knowledge, there has been no study addressing failures in k-means clustering that is implemented using the actor model. Additionally, this is the first work that compares between the performance of optimistic and pessimistic logging protocols in the context of k-means clustering.

3. An actor-based k-means algorithm

The actor model is a model of concurrent computation that provides a higher level of abstraction for developing concurrent and distributed systems. In this model, there is no need to deal with explicit locking and thread management, which makes it easy to build correct concurrent and parallel systems. An actor is the primitive unit of computation that receives and passes messages to other actors as well as performs computation based on the messages it receives. Sending a message does not transfer the thread of execution from the sender to the receiver. Instead, an actor sends a message and continues executing the remainder of the code without blocking. Unlike calling methods, passing messages does not require the receiver to return a value. By sending a message, the sender delegates a task to another actor. Actors respond to messages in the same way objects respond to methods invoked on them. The difference is that instead of multiple threads accessing the same object at the same time and leaving the internal state inconsistent, actors execute independently from the senders of a message, and they respond to incoming messages sequentially, one at a time. While the received messages by an actor are processed sequentially, different actors work concurrently, processing as many messages simultaneously as the hardware can support.

Akka is an Actor-based framework for building concurrent, distributed, message-driven applications on the Java Virtual Machine (JVM) for Java and Scala programming languages. Actors, in Akka, are organized into a tree-like hierarchy, i.e. an actor that creates another actor becomes the parent of that new actor. This is very similar to how operating systems organize processes into a tree. Actors are very light-weight in memory consumption and management overhead, so it's possible to spawn even millions in a single box Another important aspect of the actor model in Akka is that actors can be running locally or in another node, helping developers build systems that scale out, using multiple nodes. In this paper, Akka was used to develop a parallel k-means algorithm that can significantly reduce the amount of execution time needed to cluster datasets.

The sequential k-means algorithm begins with selecting random data points to be the initial centroids. Each data point is then assigned to the nearest centroids using the Euclidean distance. The mean of each cluster is then computed, and the newly computed means becomes the current centroids. These steps are repeated until either there are no significant changes to the values of the centers, or the defined number of iterations have been reached. The steps of the sequential k-means clustering is illustrated in Algorithm 1.

Algorithm 1 The steps of the sequential k-means clustering

```
 1: Input: $k$ (number of clusters)
 2:       $X = \{x_1, x_2, x_3, \ldots, x_n\}$ (a set of data points)
 3: Output: a set of $k$ clusters $S = \{s_1, s_2, s_3, \ldots, s_k\}$
 4: Methods:
 5: Randomly select cluster centroids $C = \{c_1, c_2, c_3, \ldots, c_k\}$
 6: for a given number of iterations do
 7:     Calculate the distance between each data point and cluster centers.
 8:     Assign each data point $x$ to the nearest cluster based on the
 9:         $argmin_{c_i \in C} E_{dist}(c_i, x)^2$
10:     Recalculate the new cluster centroids using:
11:         $c_i = \frac{\sum_{x_i \in s_i} x_i}{|S_i|}$
12:     If no data point was reassigned, then stop.
13: end for
```

The actor-based algorithm of the k-means clustering consists of one Master actor and several Workers. These actors are organized into a tree-like hierarchy where the master is the parent of the workers. The flow diagram of the parallel k-means is given in Fig. 1. The master is responsible for its children creation, failures, and stopping. The whole clustering process starts when an instance of master actor is created. The constructor used to create the master actor takes the following parameters: *dataset*, *number of clusters*, *maximum number of iterations*, and *the number of workers*. After creating the master actor, the Start

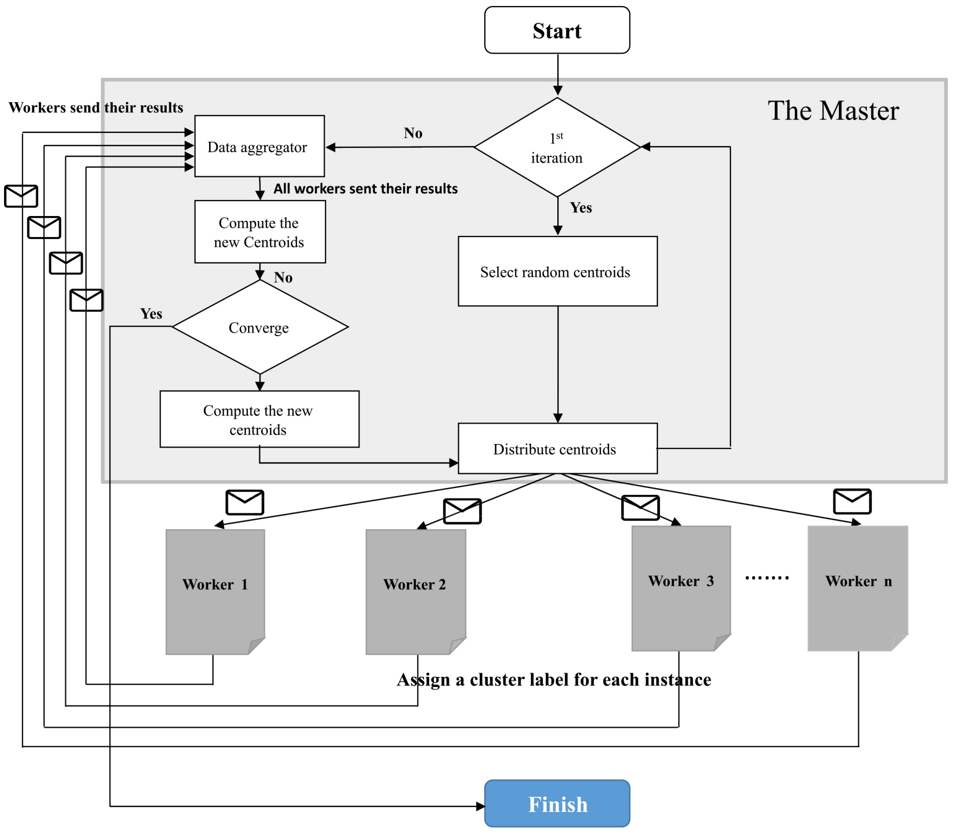


Fig. 1. The flow diagram for parallel k-means clustering based on the actor model.

message is sent to it. In response to this message, the master creates the required number of workers, randomly selects k centroids, and sends Iterate message to itself. Upon receiving this message, the master splits the data among the workers, and sends both the data and centroids to the workers. At this point, the master will be waiting for workers to finish their tasks and send back their results. On receiving the results from all workers, the master actor constructs the new centroids based on the results received from workers, and test if the convergence criteria is met or not. If not, the master sends Iterate message to itself again. The same steps are then repeated again until the algorithm reaches the maximum number of iterations or an optimal solution is found. The pseudo code of the master actor is shown in Algorithm 2.

Each worker receives its share of observations along the current centroids from the master. Using Euclidean distance formula, workers determine the closet cluster for each observation and at the same time maintain, for each cluster, the summation and count of all observations it contains. After completing the assigned task, a WorkerDone message is sent to the master, notifying it that a worker has just finished its job. This message also contains the summations and counts of all data points belonging to each cluster. The master aggregates this information from all works, and computes the new centroids out of it. The pseudo code of the worker actor is illustrated in Algorithm 3.

4. Fault-tolerant actor-based k-means algorithms

Although the proposed parallel algorithm can be extended to run it on a cluster of machines (i.e.,

Algorithm 2 The implementation of the master actor

```
1: Input: $w$: number of worker
2:       $k$: number of clusters
3:       $X = \{x_1, x_2, x_3, \ldots, x_n\}$ (a set of data points each with $f$ features)
4: Output: a set of $k$ clusters $S = \{s_1, s_2, s_3, \ldots, s_k\}$
5: Methods: Receive
6: case Start =>
7:    centroids = select random data points
8:    create $w$ workers
9:    self ! Iterate // send Iterate message to self
10: case Iterate =>
11:    for $i = 1$ to $w$ do
12:       $w_i$ ! portion (...) // send centroids along with the assigned data to all workers
13:    end for
14: case WorkerDone (localSum, localCount) =>
15:    for $i = 1$ to $k$ do
16:       for $j = 1$ to $f$ do
17:          globalSum[$i$][$j$] + = localSum[$i$][$j$]
18:       end for
19:       globalCount[$i$] + = localCount[$i$]
20:    end for
21:    if all workers are done then
22:       round = round + 1
23:       for $i = 1$ to $k$ do
24:          for $j = 1$ to $f$ do
25:             newCentroids[$i$][$j$] = globalSum[$i$][$j$]/globalCount[$i$]
26:          end for
27:       end for
28:       build the new clusters
29:       if clustering process is done then
30:          stop
31:       else
32:          centroids = newCentroids
33:          self ! Iterate
34:       end if
35:    end if
```

Algorithm 3 The implementation of the worker actor

```
1: Methods: Receive
2: case portion (frows, trows, iteration, centroids) =>
3:    for $i$ = frows to trows do
4:       val cluster = FindClosetCluster ($x_i$)
5:       for $j = 1$ to $f$ do
6:          localCentroids[cluster][$j$] = localCentriods[cluster][$j$] + $x_i$[$j$]
7:       end for
8:       counts[cluster] = counts[cluster] + 1
9:    end for
10: master ! WorkerDone (localCentroids, counts)
```

several JVMs), we focus here on running the algorithm on a single JVM. In long running jobs such as clustering huge data, the chances for a JVM crash increase for several reasons. Such crashes lead to losing the whole computations, and therefore restarting the clustering process from the beginning. This cannot be tolerated when the data that needs to be clustered is huge and takes a considerable amount of time to be finished. The impact of such failures can be minimized by periodically storing the state of the process into a stable storage. This process is called Checkpointing. Once the algorithm is restarted, this information can be read from where it was saved and the algorithm can resume working from the last saved state. This process is called rollback recovery.

Checkpointing and rollback recovery are widely used techniques for achieving fault-tolerance in distributed systems. The fault tolerance achieved by periodically saving a snapshot of the process's state to a stable storage during the failure-free execution (i.e., checkpointing), and restoring the system back to a consistent state after a failure (rollback recovery). This results in greatly reducing the amount of lost work. One challenge of using checkpointing is determining how often a checkpoint should be taken. Saving too many checkpoints imposes overhead due to its time requirement, while saving few checkpoints may increase the amount of lost work. The cost of checkpointing depend heavily on the level at which the checkpoints are taken. There are four majors levels at which the checkpoint can be taken 1) hardware-level: additional hardware is incorporated into the processor to save state, 2) kernel-level: in this level the operating system will be responsible for checkpointing the running processes, 3) user-level: a library linked to a program will be responsible for saving checkpoints. This is transparent to the user and does not require any changes to the program 4) Application-level: the programmer is responsible for writing the code of checkpointing. The main disadvantage of the first three approaches is that nearly the entire process state is saved in order to recover the applications. Moreover, the saved checkpoints in these approaches are machine-dependent, which means that they cannot be run on a different architecture. Application-level Checkpointing, on the other hand, is considered more efficient as the programmer determines which part of data should be saved and which are not, resulting in smaller size checkpoints. Additionally, the application-level checkpoints are portable and can be used to restart the application on any machine. In this paper, the application-level Checkpointing is used to develop two fault-tolerant parallel versions of the k-means algorithm using the actor model.

There are several stateful data in our proposed algorithm. In the master actor, the centroids are stateful data that are updated after each iteration. Additionally, the results received from workers are also stored in stateful data structures called globalSum, and globalCounts. After all workers finish their tasks, these data types are used to construct the new centroids. In workers, the intermediate summations and counts are also stateful data that are updated after finding the nearest centroids for each data point. Storing all these data into an external storage will guarantee that nothing will be lost when a failure happens. However, the amount of introduced overhead will significantly affect the performance of the algorithm. Ideally, a good solution should minimize the amount of lost work when a failure happens while at the same time keeps the overhead as low as possible during the failure-free execution.

The approach taken in this paper is to store only the centroids obtained after each iteration (i.e., Checkpointing). In this case, only few iterations might be lost and the clustering process can resume working from the last persisted iteration. we used two different techniques for checkpointing the master's internal state: pessimistic logging and optimistic logging. Pessimistic logging, often referred to as synchronic, waits for acknowledgment that the data was logged to a stable storage before updating the internal state of the process. It assumes that a failure can happen before the logging is complete. This protocol incurs overhead during failure-free execution as the whole application will be blocking until the acknowledgment is received. However, it guarantees that the system will always be restored to a stable state in case of a failure with the minimum amount of lost work. On the other hand, optimistic logging stores data asynchronously to a stable storage assuming that the logging will be complete before a failure may occur. Hence, it does not block the application. While this protocol incurs less overhead during failure-free execution, it may lead to a complicated recovery process with a large amount of lost computation. In this paper, we provide implementations for both techniques. The performance of each implementation is presented in the experimental results section.

Akka persistence component enables programmers either to persist all events or changes to an actor's state (i.e., event sourcing) or persist the actual state of the actor (checkpointing). In event sourcing, the changes to an actor's state are appended to an external storage without mutating the pervious events. The recovery is achieved by replaying the persisted events to the actor, enabling it rebuild its state. In Checkpointing snapshots of an actor's state are periodically persisted. In this case, the internal state is rebuilt starting from the last saved snapshot which can drastically reduce recovery times. However, snapshots are costly in terms of space and time.

In this paper, only checkpointing is used for two reasons: first, the size of the persisted state (i.e., centroids) is small and fixed, and does not increase as the clustering algorithm progresses. Second, rebuilding an actor's state from a snapshot is faster than rebuilding it from persisted events. The checkpoints are stored to a local levelDB journal, an on-disk, key-value store built by Google. Each persistent actor should be assigned an identifier that does not change from one running to another. This number is used to identify which data belongs to which actor in the store. An increasing sequence number is also assigned for each snapshot. During recovery, the latest saved checkpoint is automatically offered to the persistent actor via SnapshotOffer message. The persisted actor should implement the *receiveRecover ( )* method which receives the persisted data upon restarting the actor.

4.1. Fault-tolerant actor-based k-means algorithm using optimistic checkpointing

In optimistic checkpointing, the master actor saves the centroids asynchronously and continues with the clustering process assuming that the centroids will eventually be saved. This implementation, however,

Algorithm 4 The implementation of the master actor using optimistic checkpointing

```
 1: Input: $w$: number of worker
 2:        $k$: number of clusters
 3:        $X = \{x_1, x_2, x_3, \ldots, x_n\}$ (a set of data points each with $f$ features)
 4: Output: a set of $k$ clusters $S = \{s_1, s_2, s_3, \ldots, s_k\}$
 5: Methods: receiveRecover
 6: case SnapshotOffer (metadata, snapshot) =>
 7:    round = metadata.sequenceNr.toInt + 1
 8:    centroids = snapshot
 9: Methods: Receive
10: case Start =>
11:    if round == 0 then
12:       centroids = select random data points
13:    end if
14:    create $w$ workers
15:    self ! Iterate // send Iterate message to self
16: case Iterate =>
17:    .
18:    .
19: case WorkerDone (localSum, localCount) =>
20:    .
21:    .
22:    if clustering process is done then
23:       stop
24:    else
25:       persistAsycn (saveSnapshot (newCentroids))
26:       centroids = newCentroids
27:       self ! Iterate
28:    end if
29:    .
```

does not guarantee that the clustering process will be resumed from the iteration that preceded the iteration at which the failure happened. The pseudo code of the master actor in the fault-tolerant actor-based k-means algorithm that uses optimistic checkpointing is presented in Algorithm 4. After computing the new centroids, k-means tests if the difference between the current and new centroids is less than a predetermined threshold or not. If not, the new centroids are persisted using *persistAsynch ( )* method, and these centroids become the current centroids. This method saves the snapshot asynchronously and allow the clustering process to continue while the checkpoint is still being persisted. When a failure happens, the master actor will be restarted, and again the Start message will be sent to it. Before processing any messages, a snapshotOffer message consisting of the last saved checkpoint and its sequence number is sent to the *recieveRecovere ( )* method. This method uses the received centroids as the current centroids and the sequence number as the iteration number. The clustering algorithm can then start working normally by distributing the data and centroids among the workers. In order to prevent the algorithm from selecting random centroids, and thus losing the persisted centroids, a check statement was inserted to test if the iteration is 0 or greater than zero. In the latter case, the algorithm will not override the centroids.

4.2. Fault-tolerant actor-based k-means algorithm using pessimistic checkpointing

Using pessimistic checkpointing, the centroids are synchronously persisted by the master actor. This means that the master actor will not be able to process any messages until receiving an acknowledgment

Algorithm 5 The implementation of the master actor using pessimistic checkpointing

```
1: Input: $w$: number of worker
2:     $k$: number of clusters
3:     $X = \{x_1, x_2, x_3, \ldots, x_n\}$ (a set of data points each with $f$ features)
4: Output: a set of $k$ clusters $S = \{s_1, s_2, s_3, \ldots, s_k\}$
5: Methods: receiveRecover
6: case SnapshotOffer (metadata, snapshot) =>
7:   round = metadata.sequenceNr.toInt + 1
8:   centroids = snapshot
9: Methods: Receive
10: case Start =>
11:   if round == 0 then
12:     centroids = select random data points
13:   end if
14:   create $w$ workers
15:   self ! Iterate // send Iterate message to self
16: case Iterate =>
17:   .
18:   .
19: case WorkerDone (localSum, localCount) =>
20:   .
21:   .
22:   if clustering process is done then
23:     stop
24:   else
25:     persist (saveSnapshot (newCentroids)) { $e$ =>
26:     centroids = newCentroids
27:     self ! Iterate
28:     }
29:   end if
30:   .
```

that the centroids were successfully stored. The pseudo code of the master actor in the fault-tolerant actor-based k-means algorithm that uses pessimistic checkpointing is presented in Algorithm 5. In this implementation, if a failure happens during iteration x, the clustering process will resume working from iteration x-1. In this implementation, the snapshots are persisted using *persist ( )* method. This method was used in a way that prevents the algorithm from making any progress until receiving an acknowledgment that the snapshot was persisted. The method takes two argument: the data that need to be persisted as first argument, and an event handler as second argument. The event handler is executed for successful persisted snapshot. Once a snapshot has been saved, the event handler is executed. As a result, the current centroids will be replaced by the new centroids, and an *Iterate* message is sent to resume the clustering process. The recovery process in the optimistic and pessimistic implementation is similar.

5. Experiments

5.1. Setting

We have evaluated our proposed algorithm on two datasets. The first dataset (DS1) consists of $\sim$ 1 million records, each with 5 features, while the second dataset (DS2) consists of 2.5 million records, each with 65 features. All experiments were run on a machine with an Intel (R) Core (TM) i7-6700 CPU, @ 3.4 GHz (8 CPUs) RAM 8 GB, Windows 10 $\times$ 64. Each run was repeated 10 time, and the average execution time was reported.

The performance of the parallel k-means algorithm is affected by three main factors or inputs: the size of the dataset, number of parallel tasks (worker/threads), and the number of clusters. We report the results of five sets of experiments in this section. The first three sets were designed to investigate the performance of our methods under different values of dataset size, number of workers, and number of clusters. The fourth set of experiments compares between the performance the fault-tolerant k-means that uses optimistic checkpointing and the one that uses pessimistic checkpointing. In the fifth set of experiments, the cost of resuming the clustering process following a failure in the fault-tolerant k-means implementations was evaluated.

In order to validate the effectiveness of the proposed algorithm, we have compared our results with a Java thread-based implementation of the k-means algorithm. In such an implementation, the task of identifying the nearest cluster is distributed among several working threads. Updating the central centroids with the local ones requires the thread to impose a lock on such data to prevent the occurring of the race condition problem as a result of several threads trying to update the same data at the same time. This is completely different than the approach taken by the actor model where data in the program are exchanged via message-passing.

5.2. Test scenarios and results

5.2.1. Evaluating the performance of the proposed algorithm on different datasets sizes

In the first set of experiments, the performance of the actor-base k-means algorithm was evaluated on different sizes of datasets using different numbers of workers and a fixed number of clusters. the proposed algorithm was used to split DS1 and DS2 into 80 and 320 clusters, respectively, at different number of workers (i.e., 1, 2, 4, and 8).

The speedup achieved by the actor-based k-means algorithm on both datasets is illustrated in Fig. 2. As the figure suggests, the proposed actor-based k-means algorithm scales very well as the number of

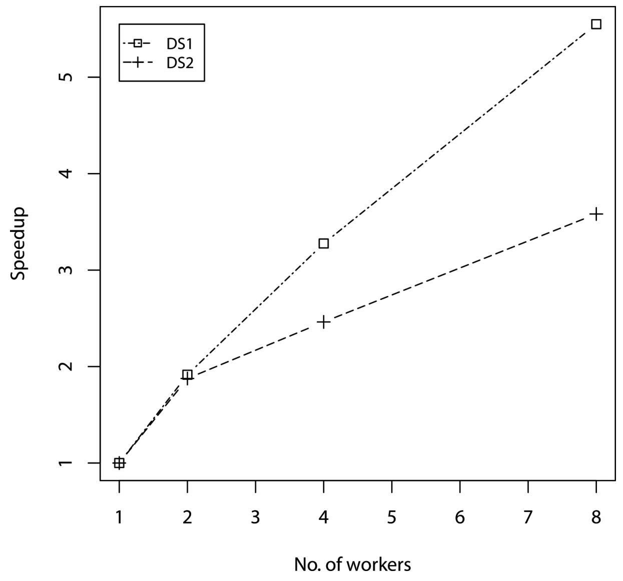


Fig. 2. The speedup achieved by the actor-based k-means using 2, 4, and 8 actors.

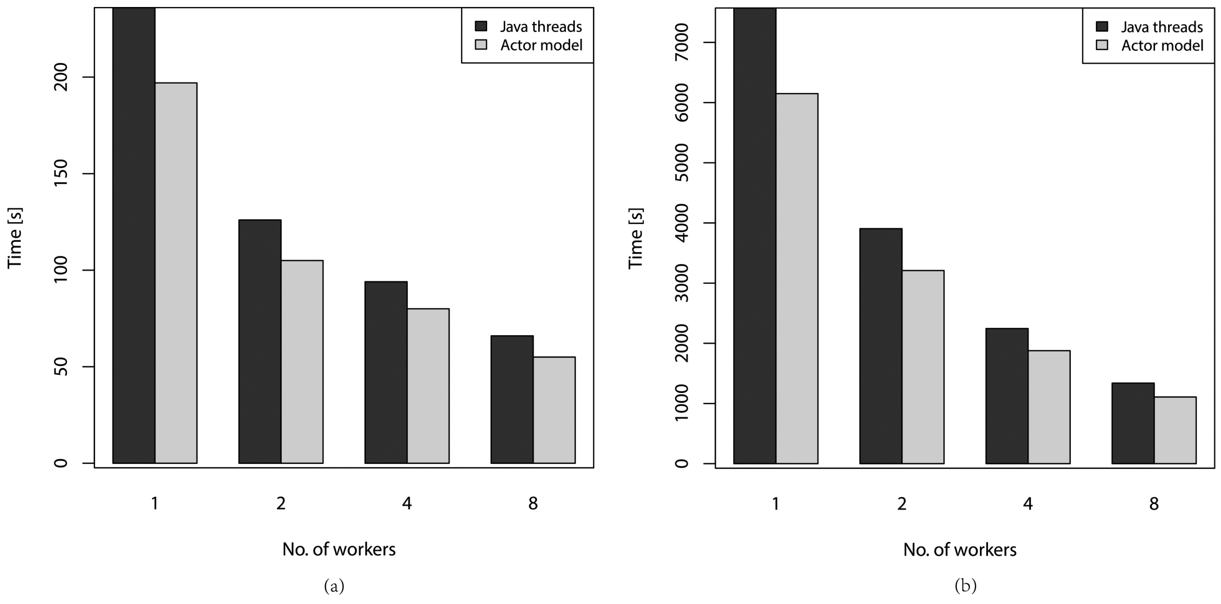
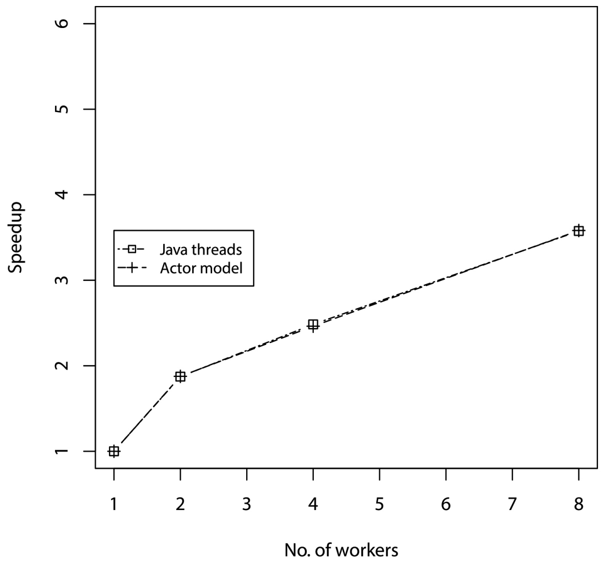
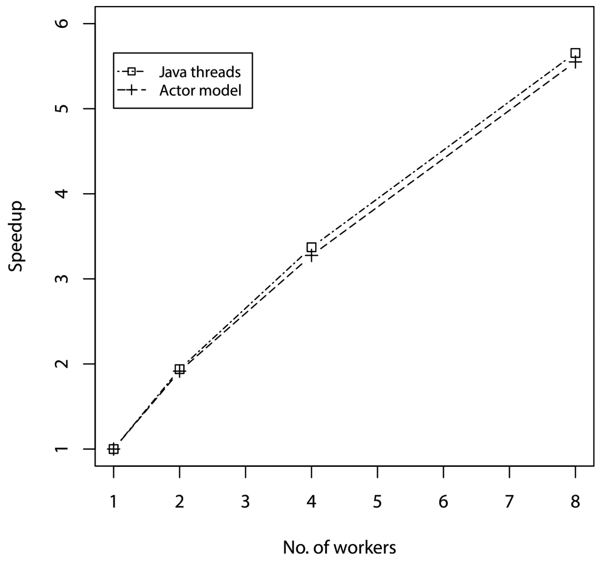


Fig. 3. a. The performance of the actor-based and Java-based k-means algorithms using 1, 2, 4, and 8 workers, respectively, in splitting DS1 into 80 clusters. b. The performance of the actor-based and Java-based k-means algorithms using 1, 2, 4, and 8 workers, respectively, in splitting DS2 into 320 clusters.

workers and size of the dataset increase. It can also be noted that the algorithm performed better, in terms of speedup, on the larger dataset as compared to on the smaller dataset, especially when large number of workers are used. This is due to the fact that the overhead incurred by managing a relatively large number of workers in small datasets is non-negligible compared to the time needed to finish the clustering process.



(a)



(b)

Fig. 4. a. The speedup achieved by the actor-based and Java-based k-means algorithms using 1, 2, 4, and 8 workers, respectively, in splitting DS1 into 80 clusters. b. The speedup achieved by the actor-based and Java-based k-means algorithms using 1, 2, 4, and 8 workers, respectively, in splitting DS1 into 80 clusters.

5.2.2. Evaluating the performance of the proposed algorithm against a Java thread-based k-means

In order to evaluate the effectiveness of the proposed algorithm, the java thread-based algorithm was run on DS1 and DS2 using the same values for number of clusters and number of workers as in the first set of experiments. The results were then compared with those obtained by the actor-based algorithm. As can be noted from Fig. 3, the performance of our proposed algorithm outperforms the java-thread-based algorithm for both DS1 and DS2 at all numbers of workers. The reason behind this difference is that updating the master's internal state by workers requires locking the targeted data to avoid the race condition problems, which is not the case with the actor model version. In terms of speed up, both algorithm achieved almost identical speedup as shown in Fig. 4.

5.2.3. Evaluating the performance of the proposed algorithm for large numbers of clusters

In the third set of experiments, we evaluated the impact of increasing the number of clusters on the performance of our method by applying it on DS2 to split it into 1000, 2000, and 4000 clusters using 8 workers. The results obtained were compared with those obtained by the java-based algorithm.

As illustrated in Fig. 5, the results of comparing the proposed algorithm and java threads-based algorithm at different numbers of clusters suggested that as the number of clusters increases, the performance difference becomes more pronounced when using the actor model. In some cases, our method was $\sim$ 2x faster than the java-based algorithm. This is again due to the fact that as the number of clusters increases, the amount of time spent on synchronization becomes more significant in the java thread-based k-means.

5.2.4. Evaluating the performance of the proposed fault-tolerant k-means algorithms during failure free scenarios

In order to get insight into the overhead accompanied by persisting the intermediate centroids in the fault-tolerant implementations of k-means, the performance of the original actor-based algorithm was

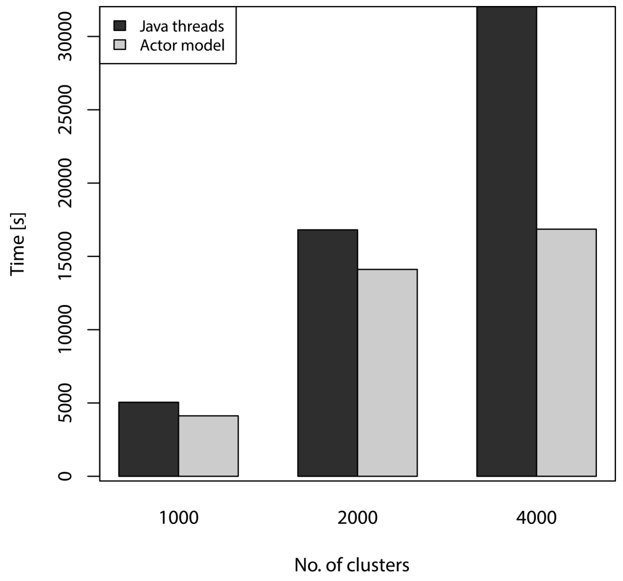


Fig. 5. The performance of the actor-based and Java-based k-means algorithms using 8 workers in splitting DS1 into 1000, 2000, and 4000 clusters, respectively.

compared with the performance of the algorithm that uses optimistic checkpointing and the algorithm that uses pessimistic checkpointing during the failure-free execution. The comparison was first made in the absence of frequent I/O operations in the system, and then made in the presence of frequent I/O operations requested by a process designed for that task. This process performs $\sim$ 70 MB read/write per seconds. The performance of each version in clustering DS2 into 80 and 160 clusters, respectively, using 8 workers in the absence and presence of frequent I/O operations was investigated. The number of iterations was predetermined to be 500 iterations to split the data into 80 clusters, and 800 iterations to split the data into 160 clusters.

The results of evaluating the fault-tolerant k-means algorithms during failure free scenarios revealed that while the overhead caused by saving the intermediate centroids was very insignificant using the optimistic checkpointing, it was a bit more significant using the pessimistic checkpointing (Table 1). This is due to the fact that in pessimistic checkpointing the algorithm cannot make any progress until receiving an acknowledgement that the centroids were persisted. This is not the case in the optimistic checkpointing as the algorithm keeps progressing without waiting for the centroids to be stored. The difference becomes even more obvious in the presence of competing I/O operations, as the algorithm that uses pessimistic checkpointing will take longer time to persist the centroids, which increases the amount of blocking time.

5.2.5. *Evaluating the performance of the proposed fault-tolerant k-means algorithms following failures*

Finally, the cost of resuming the clustering process following a failure in the fault-tolerant k-means implementations was evaluated. Both implementations were used to cluster the second dataset into 500 clusters, and both of them were forced to fail at the end of the 70th iteration. The failure causes the Java Virtual Machine (JVM) to terminate immediately. This was then followed by a manual restarting to the clustering algorithm. The amount of lost work was then measured by comparing the execution time during the failure-free execution with the total execution time when a failure occurred.

When a failure occurs, the results showed that the amount of lost work is very trivial compared to restarting the algorithm from scratch when no fault tolerance mechanism is used. Moreover, the k-means

Table 1
The performance of the original, optimistic, and pessimistic algorithm in clustering two datasets into 80 and 160 clusters in the presence/absence of competing I/O operations

Dataset	Number of clusters	No of iterations	Algorithm	Execution time (s) – without I/O	Execution time (s) – with I/O
DS1	80 cluster	500	Original	55	77
		500	Optimistic	58	79
		500	Pessimistic	82	193
	160 clusters	800	Original	119	173
		800	Optimistic	125	174
		800	Pessimistic	157	351
DS2	80 clusters	500	Original	963	1350
		500	Optimistic	969	1353
		500	Pessimistic	1002	1419
	160 clusters	800	Original	3030	4344
		800	Optimistic	3039	4431
		800	Pessimistic	3095	4583

algorithm that uses optimistic checkpointing slightly outperformed the algorithm that uses pessimistic checkpointing in terms of recovery from a failure. The percentage of lost work was 6% using an optimistic checkpointing, and 3% using pessimistic checkpointing. This was due to the fact that when a failure happens in the optimistic algorithm several iterations might have not been persisted yet. This of course increases the amount of lost work.

6. Related work

Several implementations of the k-means algorithm have been reported in the literature. OpenMP, MPI, Hadoop, Spark, CUDA, and OpenCL were among the most widely used models for implementing K-means. The OpenMP model is mainly used for loop parallelization. It does not suit all kinds of parallel applications. This is due to the fact that data corruption is possible when more than one thread attempt to update the same data. Therefore, synchronization techniques need to be used to prevent such a problem. This could negatively affect the application performance.

Although using MPI for K-means clustering requires a great deal of efforts to go from serial to parallel, it can achieve a good speed up and scales well when the number of nodes or size of datasets increases. However, the performance is highly affected by the communication network between nodes. Additionally, the resources needed for running the MPI-based algorithms cannot easily be obtained.

Apache Hadoop has also been used for running k-means algorithm using the MapReduce model. While Hadoop is naturally tolerant to failures as data are stored to disk after every operation, the fact that iterative algorithms have to run as a sequence of jobs, each of which read data from stable storage and write it back to stable storage incur significant overhead. Spark came to fill this gap by storing both input data and intermediate data in memory using distributed memory abstract called resilient distributed datasets (RDDs). Both Hadoop and Spark, however, requires building a dedicated cluster or grid of computers, which makes it unfavorable or costly for those seeking for easy and less time-consuming solutions.

Several GPU-based models have been used for implementing the k-means algorithm such as: CUDA and OpenCL. One main problem with these models is that the data transfer from CPU to GPU and from GPU to CPU incurs a significant overhead. Additionally, CUDA was designed to work only with CUDA-enabled graphic cards. Finally, some synchronization techniques might need to be used to avoid the race conditions.

In this work, an actor-base implementation of the k-means algorithm was proposed that overcomes the limitations of most of the previous solutions in the sense that no synchronization techniques are needed, and in the sense that little work is needed to set up the running environment. Moreover, a fault tolerant technique, namely checkpointing, was used to make the proposed algorithm resilient to failures. This technique was implemented in two different ways: optimistic checkpointing, and pessimistic checkpointing. To the best of our knowledge, there has been no study addressing failures in k-means clustering that is implemented using the actor model. Additionally, this is the first work that compares between the performance of optimistic and pessimistic logging protocols in the context of k-means clustering.

7. Conclusion

An implementation of an actor-based k-means algorithm was presented in this paper. The proposed implementation aims at reducing the execution time by splitting the task among several actors working concurrently. Actors interact with each other via messages, thus no synchronization is needed. The results showed that proposed algorithm achieve a good scalability as the number of workers and size of the dataset increases. Compared to a java thread-based implementation of the k-means algorithm, the actor-based version of the k-means algorithm was found to achieve better speedup that in some cases reaches up to $\sim$ 2x.

The original actor-based algorithm was adjusted it to make resilient to failures. Such resiliency is achieved by periodically saving the intermediate centroids into a stable storage during the failure-free execution, and restarting from the last saved centroids upon a failure to reduce the amount of lost work. Storing the intermediate centroids was implemented in two different ways: optimistic checkpointing (blocking) and pessimistic checkpointing (non-blocking). In the former, the application does not wait for acknowledgment when logging the centroids assuming that they will eventually be logged, while the latter blocks until receiving an acknowledgment that the messages were logged before proceeding with the computation. The results indicated that while the overhead accompanied by saving the intermediate centroids was very insignificant using the optimistic checkpointing, it was a bit more significant using the pessimistic checkpointing. This is due to the wasted time waiting for acknowledgments. In case of failure, the results revealed that the amount of lost work is very trivial compared to restarting the algorithm from scratch when no fault tolerance mechanism is used. Moreover, the k-means algorithm that uses optimistic checkpointing slightly outperformed the algorithm that uses pessimistic checkpointing in terms of recovery from a failure.

While the algorithms presented in this paper were designed to be run on a single machine, they can be easily extended to run them on a cluster of machines. This requires the system to deal with different failure scenarios. One of these scenarios is when one of worker nodes fails or becomes unreachable. The system should be able to detect the failure as soon as it happens and direct the load of the failed node to a new node or distribute it among the running workers. The system should also take into account that the node where the master is running on may fail. The whole cluster will be affected from such a major failure. Therefore, a new node should immediately be assigned to be the new master. Several issues need also to be addressed in this system such as, data distribution, remote lunching of works, data replication, etc.

Finally, although the focus was on the k-means algorithm, the approach taken in this paper can be applied to any iterative algorithm that aims to find successive approximation in sequence to reach a solution.

References

[1] W. Zhao, H. Ma and Q. He, Parallel k-means clustering based on mapreduce, in: *IEEE International Conference on Cloud Computing*, Springer, Berlin, Heidelberg, 2009, pp. 674–679.

[2] K. Stoffel and A. Belkoniene, Parallel k/h-means clustering for large data sets, in: *European Conference on Parallel Processing*, Springer, Berlin, Heidelberg, 1999, pp. 1451–1454.

[3] S. Kantabutra and A. Couch, Parallel k-means clustering algorithm on NOWs, *NECTEC Technical Journal* **1** (2000), 243–247.

[4] Z. Lv, Y. Hu, H. Zhong, J. Wu, B. Li and H. Zhao, Parallel k-means clustering of remote sensing images based on mapreduce, in: *International Conference on Web Information Systems and Mining*, Springer, Berlin, Heidelberg, 2010, pp. 162–170.

[5] Y. Zhang, Z. Xiong, J. Mao and L. Ou, The study of parallel k-means algorithm, *2006 6th World Congress on Intelligent Control and Automation* **2** (2006), 5868–5871.

[6] R. Farivar, D. Rebolledo, E. Chan and R.H. Campbell, A parallel implementation of k-means clustering on GPUs, *Pdpta* **13** (2008), 212–312.

[7] T. Kwok, K. Smith, S. Lozano and D. Taniar, Parallel fuzzy c-means clustering for large data sets, in: *European Conference on Parallel Processing*, Springer, Berlin, Heidelberg, 2002, pp. 365–374.

[8] J. Zhang, G. Wu, X. Hu, S. Li and S. Hao, A parallel k-means clustering algorithm with mpi, in: *Fourth International Symposium on Parallel Architectures, Algorithms and Programming*, IEEE, Tianjin, China, 2011, pp. 60–64.

[9] J. Bhimani, M. Leeser and N. Mi, Accelerating k-means clustering with parallel implementations and GPU computing, in: *2015 IEEE High Performance Extreme Computing Conference (HPEC)*, Waltham, MA USA, 2015, pp. 1–6.

[10] C.T. Chu, S.K. Kim, Y.A. Lin, Y. Yu, G. Bradski, K. Olukotun and A.Y. Ng, Map-reduce for machine learning on multicore in: *Advances in Neural Information Processing Systems*, Vancouver, Canada, 2007, pp. 281–288.

[11] P.P. Anchalia, A.K. Koundinya and N.K. Srinath, Mapreduce design of k-means clustering algorithm in: *2013 International Conference on Information Science and Applications (ICISA)*, IEEE, Pattaya, Thailand, 2013, pp. 1–5.

[12] S. Gopalani and A. Rohan, Comparing apache spark and map reduce with performance analysis using k-means, *International Journal of Computer Applications* **113** (2015), 8–11.

[13] B. Wang, J. Yin, Q. Hua, Z. Wu and J. Cao, Parallelizing k-means-based clustering on spark in: *2016 International Conference on Advanced Cloud and Big Data (CBD)*, IEEE, Chengdu, China, 2016, pp. 31–36.

[14] K. Wang and M.M.H. Khan, Performance prediction for apache spark platform, in: *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, IEEE, New York, NY, USA, 2015, pp. 166–173.

[15] J. Shi, Y. Qiu, U.F. Minhas, L. Jiao, C. Wang, B. Reinwald and F. Özcan, Clash of the titans: mapreduce vs. spark for large scale data analytics, *Proceedings of the VLDB Endowment* **8** (2015), 2110–2121.

[16] I.K. Savvas and G.N. Sofianidou, A novel near-parallel version of k-means algorithm for n-dimensional data objects using mpi, *International Journal of Grid and Utility Computing* **7** (2016), 80–91.

[17] I.K. Savvas and G.N. Sofianidou Parallelizing k-means algorithm for 1-d data using mpi., in: *IEEE 23rd International WETICE Conference*, IEEE, Parma, Italy, 2014, pp. 179–184.

[18] S. Mohanavalli, S.M. Jaisakthi and C. Aravindan, Strategies for parallelizing kmeans data clustering algorithm in: *International Conference on Advances in Information Technology and Mobile Communication*, Springer, Berlin, Heidelberg, 2011, pp. 427–430.

[19] J. Shen, J. Fang, H. Sips and A.L. Varbanescu, Performance gaps between OpenMP and OpenCL for multi-core CPUs, in: *41st International Conference on Parallel Processing Workshops*, IEEE, Pittsburgh, PA, USA, 2012, pp. 116–125.

[20] B. Dhanasekaran and N. Rubin, A new method for GPU based irregular reductions and its application to k-means clustering, in: *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, ACM, New York, NY, USA, 2011, pp. 1–8.

[21] B. Hong-Tao, H. Li-li, O. Dan-tong, L. Zhan-shan and L. He, K-means on commodity GPUs with CUDA, in: *2009 WRI World Congress on Computer Science and Information Engineering*, **3**, 2009, pp. 651–655.

[22] Y. Li, K. Zhao, X. Chu and J. Liu, Speeding up k-means algorithm by gpus, *Journal of Computer and System Sciences* **79** (2013), 216–229.

[23] S.A. Shalom, M. Dash and M. Tue, Efficient k-means clustering using accelerated graphics processors, in: *International Conference on Data Warehousing and Knowledge Discovery*, Springer, Berlin, Heidelberg, 2008, pp. 166–175.

[24] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M.J. Franklin, S. Shenker and I. Stoica, Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing, in: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, USENIX Association, Berkeley, CA, USA, 2012, pp. 15–28.

[25] Bytschkow, Denis, Martin Zellner and Markus Duchon, Combining SCADA, CIM, GridLab-D and AKKA for smart grid

co-simulation, in: *2015 IEEE Power & Energy Society Innovative Smart Grid Technologies Conference (ISGT)*, IEEE, 2015, pp. 1–5.

[26] H. Lee, S. Niddodi, A. Srivastava and D. Bakken, Decentralized voltage stability monitoring and control in the smart grid using distributed computing architecture, in: *2016 IEEE Industry Applications Society Annual Meeting*, Portland, OR, USA, 2016, pp. 1–9.

[27] S. Mohindra, D. Hook, A. Prout, A.H. Sanh, A. Tran and C. Yee, Big data analysis using distributed actors framework, in: *Proc. of the 2013 IEEE High Performance Extreme Computing Conference (HPEC)*, Waltham, MA, USA, 2013, pp. 1–5.

[28] D.D. Sanchez, R.S. Sherratt, P. Arias, F. Almenarez and A. Marin, Enabling actor model for crowd sensing and IoT, in: *2015 International Symposium on Consumer Electronics (ISCE)*, IEEE, Madrid, Spain, 2015, pp. 1–2.

[29] I.D. Chelcioiu, D. Corlatescu, I.C. Paraschiv, M. Dascalu and S. Trausan-Matu, Semantic Meta-search Using Cohesion Network Analysis, in: *International Conference on Artificial Intelligence: Methodology, Systems, and Applications*, Varna, Bulgaria, 2018, pp. 207–217.

[30] K. Rycerz and M. Bubak, Using Akka actors for managing iterations in multiscale applications, in: *International Conference on Parallel Processing and Applied Mathematics*, Bialystock, Poland, 2015, pp. 332–341.

[31] K. Lu, R. Yahyapour, P. Wieder, E. Yaqub, M. Abdullah, B. Schloer and C. Kotsokalis, Fault-tolerant service level agreement lifecycle management in clouds using actor system, *Future Generation Computer Systems* **54** (2016), 247–259.

Author's Bios

Salah Taamneh is currently an Assistant Professor at the Department of Computer Science and its Applications, Hashemite University, Zarqa, Jordan. He received the B.S. degree in computer science from Jordan University of Science and Technology, Irbid, Jordan, in 2005, the M.S. degree in computer science from Prairie View A&M University, Prairie View, Texas, in 2011 and the Ph.D. degree in computer science from University of Houston, Houston, Texas, USA, in 2016. His current research interests include parallel and distributed computing, machine learning and human-computer interaction.

Ahmad Qawasmeh is a native of Jordan where he studied Computer Engineering. He obtained his MS degree in Computer Science in 2010 and completed his PhD on performance analysis support for HPC applications in Computer Science from the University of Houston, USA in 2015. His research interests include parallel and high performance computing, performance analysis tools, and machine learning. He joined the Hashemite University, Jordan in 2016 as an assistant professor in the Department of Computer Science.

Ashraf H. Aljammal is currently an Associate Professor at the Department of Computer Science and its Applications, Hashemite University, Zarqa, Jordan. He received the B.S. degree in computer science from Albalqa' Applied University, Al-Salt, Jordan, in 2006, the master's degree from Science University of Malaysia, USM, Malaysia, in 2007, and the PhD degree from Science University of Malaysia, USM, Malaysian, in 2011. His research interests include network monitoring, network security, cloud.