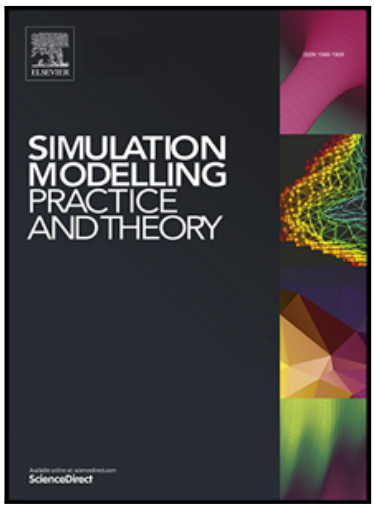

Parallel Random Swap: An Efficient and Reliable Clustering Algorithm in Java

Libero Nigro , Franco Cicirelli , Pasi Fränti

PII: S1569-190X(22)00181-2
DOI: https://doi.org/10.1016/j.simpat.2022.102712
Reference: SIMPAT 102712

To appear in: *Simulation Modelling Practice and Theory*

Received date: 30 September 2022
Revised date: 30 November 2022
Accepted date: 16 December 2022

Please cite this article as: Libero Nigro , Franco Cicirelli , Pasi Fränti , Parallel Random Swap: An Efficient and Reliable Clustering Algorithm in Java, *Simulation Modelling Practice and Theory* (2022), doi: https://doi.org/10.1016/j.simpat.2022.102712

Parallel Random Swap: An Efficient and Reliable Clustering Algorithm in Java

Libero Nigro[1], Franco Cicirelli[2], Pasi Fränti[3]
[1]DIMES - Department of Informatics Modelling Electronics and Systems Science
University of Calabria, 87036 Rende, Italy
Email: l.nigro@unical.it
[2]CNR - National Research Council of Italy - Institute for High Performance Computing and Networking (ICAR) - 87036 Rende, Italy
Email: f.cicirelli@icar.cnr.it
[3]School of Computing, Machine Learning Group
University of Eastern Finland
P.O.Box 111, 80101 Joensuu, Finland
Email: franti@cs.uef.fi

Highlights

- Original parallel support in Java of the random swap clustering algorithm.
- Applying the tool to 15 benchmark datasets, with two very challenging to solve.
- Demonstrating accurate clustering and significant execution speedups.

***Abstract*—Solving large-scale clustering problems requires an efficient algorithm that can also be implemented in parallel. K-means would be suitable, but it can lead to an inaccurate clustering result. To overcome this problem, we present a parallel version of the random swap clustering algorithm. It combines the scalability of k-means with the high clustering accuracy of random swap. The algorithm is implemented in Java in two ways. The first implementation uses Java parallel streams and lambda expressions. The solution exploits a built-in multi-threaded organization capable of offering competitive speedup. The second implementation is achieved on top of the Theatre actor system which ensures better scalability and high-performance computing through fine-grain resource control. The two implementations are then applied to standard benchmark datasets, with a varying population size and distribution of managed records, dimensionality of data points and the number of clusters. The experimental results confirm that high-quality clustering can be obtained together with a very good execution efficiency. Our Java code is publicly available at: https://github.com/uef-machine-learning.**

***Keywords*—Clustering problem, K-Means, Random swap, Parallelism, Java, Streams, Lambda Expressions, Actors, Message-passing, Multi-core machines.**

I. INTRODUCTION

The clustering problem occurs in many application areas such as physics, bioinformatics, image segmentation, machine learning, medicine, and artificial intelligence. It can be stated as follows. There are N data points $\{x_i\}$ (records or data vectors), here assumed to have numerical attributes, $x_i \in R^D$, which are to be partitioned into $K \ll N$ clusters, in a such a way that points within the same cluster are similar and points in different clusters are dissimilar. Each cluster is represented by its central point (centroid or prototype). It has been shown that finding the optimal solution to the problem is NP-hard, and only relatively small problem sizes can be solved optimally [1]. Heuristic algorithms are therefore necessary to generate sub-optimal solutions efficiently.

K-means is a well-known clustering algorithm which partitions data points according to Euclidean nearest centroid by minimizing the *Sum of Squared Error* (SSE) objective function (a measure of internal variance in clusters). Although more sophisticated clustering algorithms have been defined [2-5], K-means is often used due to its simplicity and efficiency.

The properties of K-means have been thoroughly investigated in [6]. It has been noted that its behaviour is strongly influenced by the initialization method used to assign the initial values to centroids [7-8]. Such methods can be random or deterministic, can favour density-based choices, can avoid centroids which are too close to one another or which coincide with outliers, but the space and time requirements of a specific method can prohibit its practical application to large datasets. Nevertheless, regardless of the initialization method, the natural behaviour of K-means is to get stuck around a local sub-optimal solution.

To overcome the limitations of K-means, the random swap technique was proposed in [4] together with a formal characterization of its properties. Its design qualifies for independence from any specific initialization method, for its volition to search for a clustering solution from a global point of view, and then to refine the solution locally by K-means. It has been verified that with a reasonable number of iterations, random swap is capable to find a solution very close to the optimal one with high probability.

Most operations of random swap can be executed in parallel. Its scalability can therefore be potentially improved while preserving clustering accuracy. The idea of a parallel implementation using Java streams [9] was introduced in a conference paper [10]. In this paper, we extend these preliminary results with the following new contributions:

1) We provide a more complete description of the random swap operation and the quality indexes used to check the accuracy of a clustering solution.
2) We present two original reference implementations in Java. The first one is based on Java streams and lambda expressions [9]. The second one depends on the efficient Theatre actor system [11] which enables better exploitation of computing resources.
3) We perform a more extensive evaluation using 13 standard benchmark datasets and two more challenging datasets.
4) We compare the time efficiency and clustering accuracy of the two realized Java implementations on the chosen datasets.

The paper is structured as follows. Section II reviews some clustering algorithms that improve K-means and motivates the recourse to parallel solutions to cope with increased running time. The parallel support offered by Java streams and the Theatre actor system is also clarified. Section III describes the modus operandi of random swap and its relationship with K-means. The applied clustering quality indexes are also presented. Section IV describes the two proposed implementations in Java. Section V illustrates the properties of 15 benchmark datasets chosen to

check the behaviour of Parallel Random Swap. Section VI reports the experimental results gathered on the benchmark datasets, which confirm very good execution performance and accurate clustering. The higher time efficiency provided by Theatre actors is demonstrated. Section VII, finally, presents some conclusions and draws directions for on-going and future work.

II. CLUSTERING ALGORITHMS

To improve K-means (KM) clustering, different algorithms have been defined in the literature, among which: Repeated K-means, K-means++, Agglomerative Clustering, Global K-means, Random Swap, Recombinator K-means, Genetic Algorithm, Density Peaks Clustering, Swarm-Intelligence-based K-means.

Due to KM attitude to remain stuck around a local sub-optimal solution, Repeated KM (RKM) runs the basic KM a number of times and selects the best solution among those. The more it is repeated, the higher is the chance to find a good solution.

K-means++ (KM++) [12] initializes centroids probabilistically. The first centroid is randomly chosen in the data space. Then, any other data point x_j is selected as next centroid with probability:

$$p(x_j) = \frac{D(x_j)^2}{\sum_{i=1}^{N} D(x_i)^2} \quad (1)$$

where $D(x_j)$ denotes the minimal distance of x_j to existing centroids. The initialization strategy distributes more evenly the centroids in the data space and tends to choose centroids far away from each other. Since the stochastic behavior, also K-means++ must be repeated a certain number of times.

Agglomerative clustering (AC) generates the solution by a sequence of cluster merge operations (bottom-up approach) [13-14]. At each step, the pair of clusters is merged which increases the objective function cost least.

Global K-means (GKM) [3] rests on a top-down approach. At each step, it considers every data point as a potential location for a new cluster, applies some k-means iterations (e.g., 10) and selects the candidate solution that decreases the objective function value most.

Random swap (RS) [4], which is the core algorithm in this paper, constructs a clustering solution by a sequence of centroid swaps so as to decrease the objective function cost, and locally fine-tuning the solution by a few KM iterations (e.g., 2).

Recombinator K-means (RecKM) [15] applies K-means multiple times to produce new dataset of the obtained centroids. K-means is then run for this new set with the idea to capture true cluster centroids often missed by standard K-means in low-density areas.

Genetic algorithm (GA) [2] maintains a set of solutions. It generates new candidate solutions by AC-based crossover, which are fine-tuned by two iterations of KM.

Density Peaks Clustering (DPC) [5,16] goes beyond the hypothesis of spherical clusters assumed by classical algorithms like K-Means. It detects cluster centers as points in dense areas, surrounded by points with lower density. Two parameters are used: density and delta. First, the density of points is calculated. Then the distance to the nearest point with a higher density is evaluated (delta). Centroids are selected as points with high delta and density. After that, clusters are formed by merging the remaining points to the nearest higher-density point.

In [4], RS clustering was compared to K-means and its variants. It emerged that KM, RKM, KM++ fail to find the correct solution in more than 50% of the cases. Better algorithms are AC, RS, GKM and GA that successfully detected the correct solution in all the experimented cases. The most affecting factors are the cluster overlap, number of clusters, and cluster size balanced [4].

The downside of using better algorithms is their slower running time, which motivates the need to develop parallel implementations. More efficient variants have also been proposed for some methods like the one in [16] which improves density peaks clustering.

Swarm intelligence has also been applied for clustering [17]. Examples include particle swarm optimization (PSO) [18], artificial bee colony (ABC) [19] and ant colony optimization (ACO) [20]. Particle swarm optimization, exploited in [21] in combination with K-means for image classification, would be particularly suitable for parallel processing [22].

Parallel solutions

Different parallel solutions have been reported in the literature, e.g. tailored to an efficient support of KM, either on a distributed multi-computer context, or on a shared-memory multiprocessor. Examples of the first case include message-passing solutions based on MPI [23], with a master/slave organization [24], or based on the functional framework MapReduce [25-26]. Notable examples of the second case are the experiments conducted by using OpenMP [27] or based on the GPU architecture [28].

Distributed solutions can manage very large datasets decomposed on the local memory of various computing nodes. Shared memory solutions split the dataset into subblocks to be operated in parallel by multiple computing units (cores).

A general framework for big data and parallel clustering algorithms is Apache Spark [29]. Spark characterizes by its Resilient Distributed Datasets (RDD) abstraction for storing data in memory. RDD can be partitioned on a cluster of master/worker nodes, to enable fast and efficient map/reduce operations. Both partition, hierarchical and density-based [30] clustering algorithms have been experimented.

Solution 1: Java Streams

The first parallel solution in this paper will be based on Java data streams, lambda expressions and functional programming style [9], which were introduced since the Java 8 version. We chose this solution because it hides a ready-to-exploit lock-free multi-threaded programming layer which (by some cautions) could also be used by not expert parallel programmers.

Streams are *views* (not copies) of collections (such as lists and arrays) of objects, which make it possible to express a fluent style of operations. Each operation works on a stream, selects and transforms every object according to a lambda expression related to functional interfaces such as Predicate, Consumer and Function [9]. It typically returns a new stream, ready to be handled by the next operation. In a fluent code segment, only the execution of the terminal operation triggers the execution of the intermediate operations.

What makes the streaming approach very appealing is its apparent simplicity by which the data streams can be processed in parallel. In this way one can benefit from the underlying multi-core architecture with shared memory of a modern commodity machine, through the fork/join mechanism and associated splitting of data into segments processed in parallel, extracting results from segments and finally combining the results.

However, some subtle aspects can make the recourse to parallel streams risky or useless. Lambda expressions should never refer to external shared data which can introduce race conditions and data inconsistency. In addition, only properly sized streams can deliver good performance.

We will exploit Java streams to provide an efficient support to the parallel execution of random swap algorithm, so as to deliver good execution performance and careful clustering. The development owes and generalizes preliminary work about parallel K-means described in [31].

Solution 2: Theatre actor system

The second implementation of parallel random swap is based on the Theatre actor system [11]. It is a lock-free *message-based* software architecture capable of ensuring high-performance computing on a multi-core machine, through explicit resource control. A Theatre system is a federation of computing nodes (theatres) which are mapped onto threads and then cores.

One theatre (that with ID 0) hosts a *time server* which is responsible for managing a global notion of time or it can be simply devoted to checking application termination. Within the same theatre, light-weight threadless actors execute according to a *cooperative concurrency*: only one message at a time.

Message interleaving is the source of logical concurrency and ensures shared data can be safely accessed by local actors. An actor encapsulates an internal data status which can only be modified by an exposed message interface. Responding to a message is encoded in a *message server* method, that is a normal Java method equipped with the @Msgsrv annotation, which can admit parameters but always returns void. A basic operation is the non-blocking *send* operation which creates (and schedules) a message.

A theatre rests on a *reflective control layer* which, transparently, regulates message scheduling and dispatching. The control layer can be purely concurrent, or it can manage a time notion, real-time or simulated time, see [11] for more details. True *parallelism* occurs in message processing in actors executing on distinct physical theatres. Sharing data among actors belonging to separate theatres should be avoided or managed by application-tailored mechanisms, e.g., by using a precedence constraint rule [11] to forbid data races without using locks or semaphores.

The concurrent control layer guarantees that messages sent by a source actor to a given destination actor, either local or remote, are eventually received and processed in the sending order. Non-determinism in the message order occurs when messages are sent simultaneously by multiple source actors executing onto different theatres toward a given remote destination actor. The time server of a concurrent Theatre system detects the termination condition when all the control layers have an empty queue of messages, and no message is in transit across theatres.

A Theatre-based solution typically splits the dataset into subblocks (regions) which are assigned to manager actors running on independent theatres/cores, possibly organized according to a master/worker architecture (see later in this paper). This paper uses two versions of Theatre: the parallel one and a standalone flat version, which implicitly uses one theatre and does not pay for the use of parallel concerns such as transport layer and time server.

III. RANDOM SWAP

Random swap algorithm [4] was designed to solve the cluster structure by a sequence of centroid swaps (global search), and by fine-tuning the result by K-means (local search). The algorithm significantly improves K-means because it almost never gets stuck in a sub-optimal

local solution. The results in [6] showed that it reaches the correct global allocation of the clusters with all benchmark datasets.

K-means operation

K-means aims at minimizing the *Sum of Squared Error* (SSE) objective function. Let $\{C_1, C_2, \dots, C_K\}$ be the K clusters, and $\{\mu_1, \mu_2, \dots, \mu_K\}$ the corresponding representative centroid points. The SSE is defined as:

$$SSE = \sum_{i=1}^{N} \|x_i - nc(x_i)\|^2 \tag{2}$$

where $nc(x_i)$ is the nearest centroid μ_j to x_i according to Euclidean distance, that is:

$$nc(x_i) = \mu_j, \text{where } j \text{ is: } \underset{1 \le j \le K}{arg\,min} \ \|x_i - \mu_j\|^2 \tag{3}$$

It is sometimes preferable to use the normalized mean SSE, indicated as $nMSE$, defined as: $nMSE = SSE/(N * D)$. Starting from an initialization of centroids, K-means iterates the two steps shown in Alg. 1 a maximum number of iterations or until convergence is sensed (the new centroids are equal to the previous ones).

Algorithm 1. Basic steps of K-means.

1. *Partition* data points $\{x_i\}$ into clusters according to $nc(x_i)$;
2. *Update* centroids as the mean point of each cluster:

$$\mu_j = \frac{1}{|C_j|} \sum_{x_i \in C_j} x_i, \ \forall j \text{ in } [1, K]$$

Random swap operation

The steps of the Random Swap algorithm are shown in Alg. 2. The *modus operandi* of the algorithm is the *Swap* step 3, which defines a new global configuration. The new centroids are then locally fine-tuned, and corresponding cluster boundaries better defined by K-means at step 4. Steps from 3 to 5 are repeated for a fixed number of iterations (T) and these steps constitute the most processing time.

Algorithm 2. Steps of random swap algorithm.

1. *Centroids initialization*. Define initial centroids as K randomly selected points.
2. *Initial partition*. Partition data points according to the initial centroids.

Repeat T times:

3. *Swap*. A centroid is randomly selected, and replaced by a randomly chosen data point in the data space:

$$c_s \leftarrow x_i \,, s = rand(1, K), i = rand(1, N)$$

4. *K-means*. A few iterations of K-means (e.g. 2) are executed.
5. *Test*. Check if the new solution (new centroids and associated partition) has a lower $nMSE$ cost than the previous solution. If so, it is accepted and made as the current solution. Otherwise, previous centroids and the corresponding data partition are

restored.

6. *Final tuning*. The last defined solution is improved by a final execution of K-means, which is iterated until convergence or after a maximum number of iterations are executed.

Evaluation of clustering accuracy

Random swap iterations are controlled by the $nMSE$ value, which monotonically decreases as the swap iterations proceed. This quantity guides the search. The clustering accuracy can also be evaluated by another internal index like the Silhouette index (see later in this section). External quality indexes can be used for testing purposes if ground truth (GT) centroids or ground truth partitions (GP) are known. Ground truth partitions are specified by furnishing the initial clusters, that is the cluster label each point is partitioned to. Some of such indexes are described below.

Centroid Index (CI)

CI was introduced in [32] to measure how well the centroid locations match the ground truth centroids. It can be used with synthetic datasets constructed by some specific distribution of points around pre-defined centroids (prototypes). The CI value of a solution can be understood intuitively in Fig. 1, where the clustering solution admits 4 real clusters not having any centroid and 3 with more than one centroid. The CI value is the greater of these numbers.

Formally, the CI value of two centroid solutions C_1 and C_2 can be computed by mapping C_1 onto C_2 and vice versa and counting dissimilarities. In particular, first each centroid of C_1 is mapped onto its nearest centroid in C_2 according to Euclidean distance. Then the number of "orphans" in C_2 is counted as the elements of C_2 where no item of C_1 was mapped onto. In a similar way, centroids of C_2 are subsequently mapped onto the centroids of C_1 and the resultant number of "orphans" in C_1 is counted. Finally:

$$CI(C_1, C_2) = \max(\, orphans(C_1 \rightarrow C_2), orphans(C_2 \rightarrow C_1)\,) \quad (4)$$

In the case the bijection evaluates to 0, the solution C_2 has the same distribution of the centroids globally as C_1, and has therefore high probability to be correct with respect to C_1.

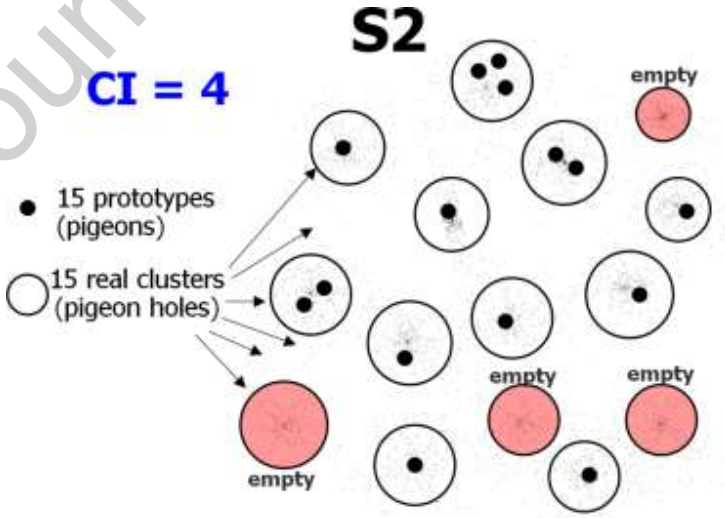

Figure 1. An intuitive interpretation of the Centroid Index (CI) [4].

Two cases of accepted swap iterations are shown in Fig. 2. The one that decreases both $nMSE$ and CI is considered successful even though the algorithm accepts both of them.

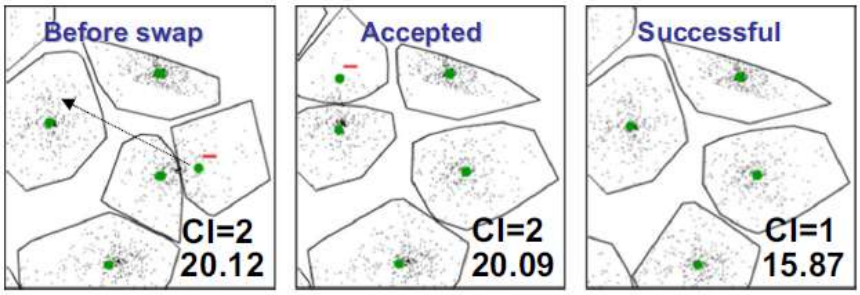

Figure 2. A successful swap iteration [4].

Generalized Centroid Index (GCI)

The centroid index was generalized in [33] for it to be applied to set-based partitions. Two set-based partitions PA_1 and PA_2 are compared. For example, a partition $PA_1[j]$ can be mapped onto the partition of PA_2 which shares most points with $PA_1[j]$. In another case, $PA_1[j]$ can be mapped onto the partition of PA_2 according to minimal Jaccard distance:

$$JDistance(pa_1, pa_2) = 1 - \frac{|pa_1 \cap pa_2|}{|pa_1 \cup pa_2|} \tag{5}$$

In the Jaccard distance, the number of shared points is normalized against the total number of points in both the partitions. This work adopts the Jaccard distance as the criterion for comparing two set-based partitions. Similarly to CI, the number of orphans in the two mapping directions is established. The generalized centroid index (GCI) is defined as follows:

$$GCI(PA_1, PA_2) = \max(\, orphans(PA_1 \rightarrow PA_2), orphans(PA_2 \rightarrow PA_1)\,). \tag{6}$$

Silhouette Index (SI)

It is a classic measure of clustering accuracy [34]. For each point x two quantities are computed: a_x and b_x (see Fig. 3). The component a_x is an intra-cluster measure, that is the average distance of x from all the other points in the same cluster. The component b_x is the minimum average distance of x from all the points in other clusters. The Silhouette value associated with the point x is calculated as:

$$S_x = \frac{b_x - a_x}{\max(b_x, a_x)} \tag{7}$$

The Silhouette index is the average of the S_x values:

$$SI = \frac{1}{N}\sum_{i=1}^{N} S_{x_i} \tag{8}$$

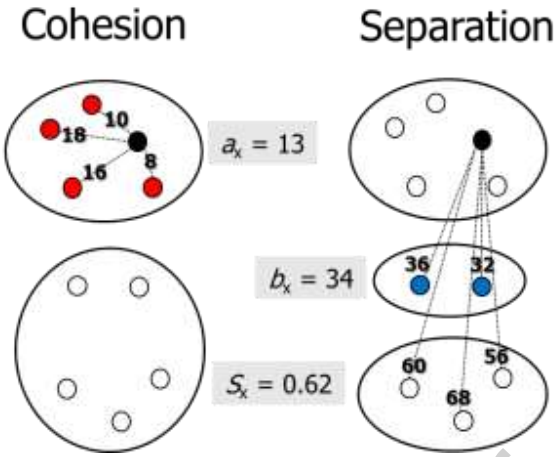

Figure 3. Intuitive illustration of the Silhouette Index (SI).

The SI value is in the range [-1,1]. A value 1 indicates well-separated clusters (reduced overlap). A value close to 0 mirrors high overlap of clusters. A value toward -1 indicates incorrect clustering.

IV. ENABLING PARALLEL RANDOM SWAP IN JAVA

Different operations of the random swap algorithm shown in Alg. 2 can be executed in parallel. The initial partition (step 2), the partition and update steps of K-means (steps 4 and 6) and the restore partition of step 5, can purposely be carried out in parallel. Further operations that can benefit of parallelism include the computation of clustering indexes like the Silhouette (SI) which has a cost of $O(N^2)$ because it requires the calculation of all pairwise distances. The corresponding sequential wall-clock time can then be a problem for large datasets. All operations mentioned above will be carried out in parallel.

Stream-based Parallel Random Swap

The following briefly outlines the developed Java classes supporting parallel/sequential random swap. As in [31], all the parameters which drive simulations, data structures, basic algorithms for I/O, for centroids initialization (implemented methods include [8]: Random, Kmeans++, Maxmin, Kaufman, ROBIN and DKmeans++), and for computing clustering indexes, are collected in a G (global) class. Here the user can specify the dataset external name to load, the value of N (dataset size), D (number of coordinates of points), K (number of clusters), T (number of swap iterations), and so forth.

The dataset and centroids are mapped onto native arrays of $DataPoint$ objects, from which streams are derived. $DataPoint$ exposes methods for point arithmetic (e.g., addition and mean) and various kinds of distance notions (including Euclidean, Manhattan, and Hamming). A data point also holds cid, which is the index of the centroid it was partitioned to.

Algorithm 3. Random swap program in Java based on streams.

```
public static void main( String[] args ) throws IOException{
    initialize();
    start=System.currentTimeMillis();
    partition();
    previous_cost=nMSE();
```

```
    step=1;
    for( ; step<=T; ++step ) {
        if( accepted ) save_prototypes();
        make_swap();
        k_means(5);
        current_cost=nMSE();
        if( current_cost<previous_cost ) {
            accepted=true; previous_cost=current_cost;
            rec_accept(); //bookkeeping
        }
        else {
            accepted=false;
            restore_partition();
            restore_centroids();
            rec_refuse(); //bookkeeping
        }
    }//for( step... )
    k_means();
    end=System.currentTimeMillis();
    output();
}//main
```

Alg. 3 draws the stream-based random swap program, where helper methods hide basic and bookkeeping (recording data gathered during a simulation) operations and contribute to keeping the Java code closer to the pseudo-code in Alg. 2.

The $initialize()$ method loads the dataset and ground truth centroids and partitions, if available, from the file system. The method also initializes centroids by a random selection of K points in the dataset (random() method of the G class). The real-time required by the T swap iterations is annotated at the end of each run.

Alg. 4 shows the partition operation which, depending on the value of the $PARALLEL$ parameter, can be executed in parallel or sequentially.

Algorithm 4. The partition operation.

```
Stream<DataPoint> p_stream=Stream.of( dataset );
if( PARALLEL ) p_stream=p_stream.parallel();
p_stream
    .map( p -> {
        double md=Double.MAX_VALUE;
        for( int k=0; k<K; ++k ) {
            double d=p.distance( centroids[k] );
            if( d<md ) { md=d; p.setCID(k); }
        }
        return p; } )
    .forEach( p->{} );
```

A basic design issue emerges from Alg. 4. The map operation receives a lambda expression (Function value) whose parameter p is a point, whose cid is set to the index of the nearest centroid. No shared data gets modified: each individual point modifies itself. The terminal $forEach$ operation serves just to trigger the mapping operation which can proceed in parallel. The partition operation is also used within the $kmeans(times)$ method which receives the

number of iterations to accomplish. However, in this case, at the first iteration, the cid of each point gets preliminarily saved.

Alg. 5 depicts an excerpt of the update centroids phase of $kmeans()$ method. Individual centroids can be updated in parallel. It is worth noting that traversing once the dataset points as done in Alg. 4, was avoided because it would cause race conditions tied to multiple data points that share the same centroid.

Algorithm 5. An excerpt of the update centroids step of K-means.

```
//update centroids
Stream<DataPoint> c_stream=Stream.of( centroids );
if( PARALLEL ) c_stream=c_stream.parallel();
c_stream
    .map( c -> {
        for( int i=0; i<N; ++i ) {
            if( dataset[i].getCID()==c.getCID() ) c.add(dataset[i]) ;
        }
        c.mean();
        return c; } )
    .forEach( c->{} );
```

The invocation of $kmeans()$ at the end of a run (see Alg. 3), has no parameters because it gets iterated at maximum T times or as early as convergence is reached.

The $restore\_partition()$ method, similarly to the partition operation in Alg. 4, processes all the points of the dataset so as to restore previously saved cid (partition). All of this can naturally be accomplished in parallel.

It is worth noting that the adopted program organization purposely executes in parallel specific macro-actions of a swap iteration, like partition and update steps of K-means or restoring a previous partition. Consequently, a swap iteration path starts sequentially, making it possible to execute a swap using a single Random object (the Java Random class is thread-safe) without thread contention which would degrade the execution performance.

Theatre-based Parallel Random Swap

This second implementation exploits the Theatre actor system [11] developed for high-performance computing. The solution rests on a Master/Worker organization (see also [31]) plus the initial configurator (main program) which bootstraps the system and launches the execution. The configurator creates P theatres, where P is the number of (physical+virtual) available cores. Each Theatre instance has a unique ID in $[0..P-1]$ and it is provided of an instance of the transport layer (used to receive externally generated messages directed to actors of this theatre) and control layer (here parallel $PConcurrent$). By convention, theatre 0 also receives an instance of $PCTimeServer$ which controls the termination condition (complete message exhaustion all over the system).

After that, the configurator continues by creating one instance of the Master and $P-1$ instances of the Worker actor class, which are moved respectively to theatre 0 and theatres from 1 to $P-1$. Master and Worker classes are heirs of a basic $Manager$ class which defines common messages. Each manager actor is initialized by sending to it an $init()$ message which carries the identity of the manager (0 for the Master, $1..P-1$ for the Workers), a subblock of the dataset (region) and the identity of the Master for workers, and the array of all the manager references for the Master. The configurator finishes its task by activating all the theatres.

Alg. 6 reports the class Worker. Each worker manages a contiguous region (segment) of the dataset transmitted through the $init()$ message. The $partition()$ message server receives the (absolute) current centroids and assigns each point of the local region to the nearest centroid. Finally, the worker sends a $done()$ message to the master. The $kmeans(...)$ message server receives the reference centroids (accessed by *copy semantics*, that is, since the array is shared by all workers, no one will update it) and realizes *one* iteration (assignment and update) of K-means.

Obviously, a worker can only evaluate a partial view of new centroids, according to the local region. The partial array is sent back to the Master which will assemble all the received partial centroids to generate the effective new centroids. $kmean()$ also receives a Boolean save parameter which can ask to save previous partitioning (cluster ID or CID) of region points. Upon such a saved information will operate the $restore\_partition()$ message which is requested by Master following an unproductive swap iteration.

The master actor which embodies the random swap logic through a finite state machine, has more complex behavior. In the following we will highlight only the major aspects.

First of all, the master includes the worker's behavior. It therefore implements the $partition()$, $kmeans()$ and $restore\_partition()$ messages, see Alg. 6.

The master enters the PARTITION state at the end of the $init()$ message, after having sent in broadcast a $partition()$ message to all the workers and to itself. The $init()$ message also initializes the local variable $accepted$ to true. In the same way, following an unaccepted swap, the master broadcasts a $restore\_partition()$ message to all the workers (and to itself), and enters the RESTORE_PARTITION state. In both cases, the master will wait for $P\ done()$ messages.

Algorithm 6. The Worker actor class.

```
public class Worker extends Manager{
    private DataPoint[] region, partial;
    private int ID;
    private Master m;
    @Msgsrv
    public void init( Integer ID, DataPoint[] region, Master m ) {
        this.ID=ID;
        this.region=region; //a sub-block of the whole dataset
        this.m=m;
        this.partial=new DataPoint[K];
        for( int i=0; i<K; ++i )  this.partial[i]=new DataPoint();
    }//init
    @Msgsrv
    public void partition( Integer id, DataPoint[] centroids ) {
        for( int i=0; i<region.length; ++i ) {
            double minD=Double.MAX_VALUE,d=0;
            for( int j=0; j<K; ++j ) {
                d=region[i].distance( centroids[j] );
                if( d<minD ) { minD=d; region[i].setCID(j); }
            }
        }
        m.send( "done" );
    }//partition
    @Msgsrv
    public void kmeans( Boolean save, DataPoint[] centroids ) {
```

```
        //partition local data points according to received centroids
        for( int i=0; i<region.length; ++i ) {
            if( save ) region[i].saveCID();
            double minD=Double.MAX_VALUE,d=0;
            for( int j=0; j<K; ++j ) {
                d=region[i].distance( centroids[j] );
                if( d<minD ) { minD=d; region[i].setCID(j); }
            }
        }
        for( int k=0; k<K; ++k ) partial[k].reset();
        //updates partial centroids
        for( int i=0; i<region.length; ++i ) {
            int cid=region[i].getCID();
            partial[cid].add( region[i] );
        }
        m.send( "done", ID, partial );
    }//kmeans
    @Msgsrv
    public void restore_partition() {
        for( int i=0; i<region.length; ++i ) region[i].restoreCID();
        m.send( "done" );
    }//restore_partition
}//Worker
```

In the PARTITION state, the master will then initialize the previous cost and sends to itself a $swap()$ message thus starting the first swap iteration and entering the KMEANS state. Following a RESTORE_PARTITION, first centroids are restored from a copied value, then either a new swap iteration is started by a new $swap()$ message, or the algorithm terminates if the maximum number of steps (T) was reached.

Responding to a $swap()$ message causes the master to make a new swap. If the last swap was successful (i.e., the $accepted$ state variable is true) first a copy of the current centroids is created, then the current cost is made as the new previous cost. After that a swap is accomplished followed by a K-means execution. A local variable $times$ is used which indicates to the master how many K-means iterations remain to execute. When $times$ reaches 0 (or convergence is sensed in the last iteration of K-means), current K-means is finished.

Two kinds of $done()$ messages exist. The first one, without arguments, follows the PARTITION or a RESTORE_PARTITION state. The second type of $done()$, used in KMEANS state, carries as an argument a partial evaluation of centroids proposed by a worker (or by the master itself). When P of such $done()$ messages are received, the master first determines the new centroids, then sends to itself a $prosecute()$ message to decide what to do.

The $prosecute()$ message first checks the acceptable or not acceptable status of the just concluded swap iteration. In the case of a successful swap, the termination condition of random swap is sensed and if it was reached, an exhaustive step of K-means is launched by setting $times$ to T and broadcasting the $kmeans()$ message. For an unproductive swap, $prosecute()$ puts $accepted$ to false, moves to the RESTORE_PARTITION state and broadcasts the $restore\_partition()$ message.

At the end of RESTORE_PARTITION (P $done()$ messages arrived) the $done()$ message server checks termination and, in case it was reached, a last exhaustive K-means step is executed. The $prosecute()$ message which follows K-means and recognizes termination, does not self-send a new $swap()$ message thus causing the whole application to terminate.

The above-outlined actor-based parallel random swap was also stripped down (only a simplified version of the Master actor is kept) so as to use the standalone version of Theatre for sequential execution comparisons.

V. EXPERIMENTAL SETUP

Correctness of the developed parallel random swap implementations was verified by applying them to 15 benchmark datasets [6,35], of which two are very challenging to solve. For all the datasets either the ground truth (GT) centroids or ground true partitions are publicly available. The datasets characterize for the number and shape of clusters and point distributions used to construct the dataset.

The A_1, A_2 and A_3 datasets (Fig. 4) contain equal size spherical clusters with an increasing number of clusters. The sets A_1 and A_2 are merely subsets of A_3 as follows: $A_1 \subset A_2 \subset A_3$.

The S sets (Fig. 5) contain Gaussian clusters with varying degree of overlap.

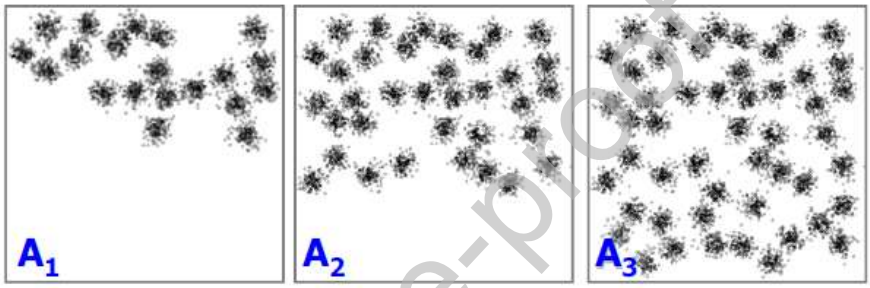

Figure 4. The A1, A2 and A3 datasets [35].

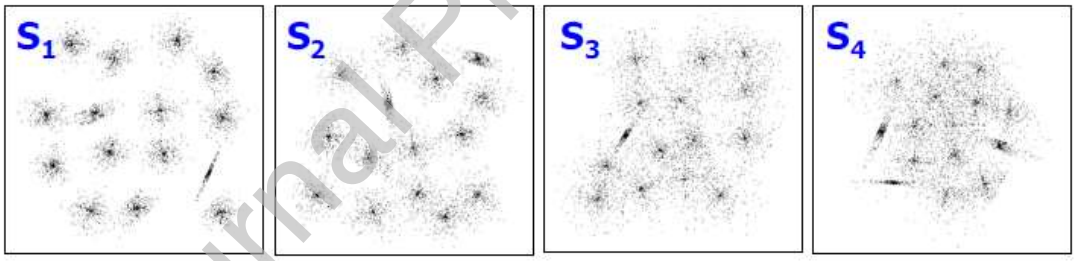

Figure 5. The S1, S2, S3 and S4 datasets [35].

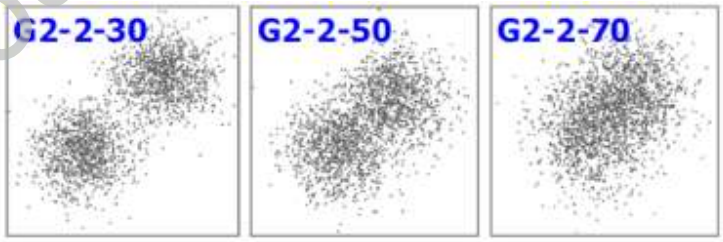

Figure 6. Examples of G2 datasets [35].

The G2 sets (Fig. 6) contain 2048 points distributed according to two Gaussian clusters at fixed locations. Overlap was created by varying the standard deviation from 10 to 100. A particular dataset is named $G2 - dim - std$ accordingly. The $G2 - 1024 - 100$ dataset was selected for the experiments.

The DIM sets (Fig. 7) contain well-separated clusters in high-dimensional space. For the experiments $DIM32$ was chosen: 32 dimensions for each one of the 1024 points. Points are randomly distributed among clusters by Gaussian distribution.

The Unbalance dataset (Fig. 8) has eight clusters organized in two well-separated groups. The first three clusters are dense with 2000 points each. The remaining five clusters contain 100 points each.

Figure 7. The DIM32 dataset [35].

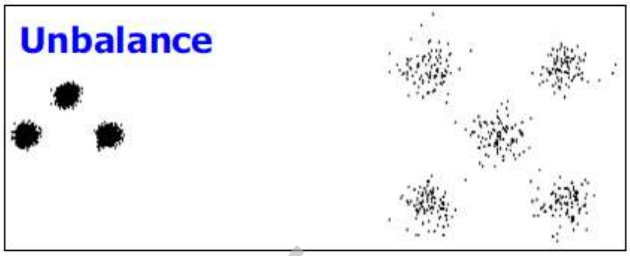

Figure 8. The Unbalance dataset [35].

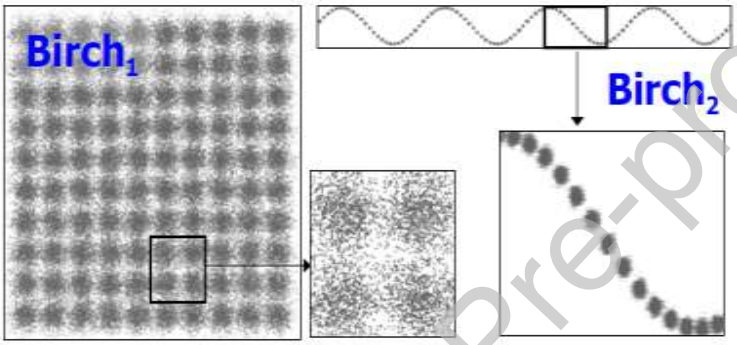

Figure 9. The Birch1 and Birch2 datasets [35].

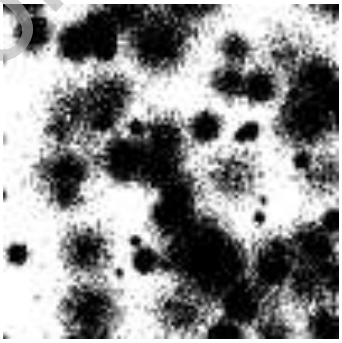

Figure 10. The Birch3 dataset [35].

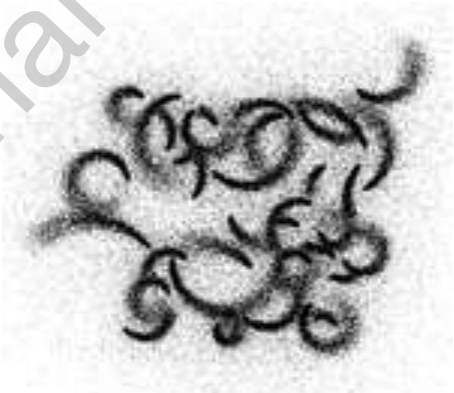

Figure 11. The Worms_2d dataset [35].

The Birch1 and Birch2 sets of Fig. 9 contain spherical clusters organized on a 10×10 grid (Birch1), or on a sine curve (Birch2). Birch3 is more challenging because it is composed of random-sized clusters located in random positions.

We also selected the two worms artificial datasets discussed in [16]. Worms_2D contains 35 worm-shaped clusters in 2-dimensional space, and Worms_64D contains 25 clusters in 64-dimensional space. The worm shapes start from a random position and move to a random direction. At each step, points are drawn from a Gaussian distribution whose variance increases

gradually with each step. The direction of movement is continually altered to an orthogonal direction. In the 64D case, the orthogonal direction is randomly selected at each step.

Parameters of the datasets are collected in Table 1. They have been used in [6] for an in-depth analysis of K-means properties, and in [4] for comparing the clustering accuracy of random swap against K-means and its variants. We use the datasets for checking the computational efficiency and verify that the results of the proposed parallel random swap are equal to the original sequential algorithm.

Table 1. Parameters of the benchmark datasets [35].

dataset	N	D	K
A1, A2, A3	3000,5250,7500	2	20,35,50
S1, S2, S3, S4	5000	2	15
G2 − 1024 − 100	2048	2	2
DIM32	1024	32	16
UNBALANCE	6500	2	8
BIRCH1, BIRCH2, BIRCH3	100000	2	100
WORMS_2D	105600	2	35
WORMS_64D	105000	64	25

VI. RESULTS

The achieved Parallel random swap tools were applied to the benchmark datasets described in Section V. Although only 2 iterations of K-means per swap were suggested in [4] to refine a local solution at each swap iteration, we increased this to 5 motivated by the parallel support. In addition, not originally used in [4], we also refine the final solution after the swap iterations, by executing K-means exhaustively until convergence (step 6 of Alg. 2). The maximum number of swap iterations and the maximum number of steps of the final invocation of K-means were fixed, by default, to $T = 5000$. Only for the complex Birch3 and Worms_2d datasets, T=10^6 was used.

All the execution experiments were carried out on a Win10 Pro, Dell XPS 8940, Intel i7-10700 (8 physical cores with the support of 16 threads via hyperthreading), CPU@2.90 GHz, 32GB Ram, Java 17.

Experimental setup

Preliminary runs were devoted to assess the correctness of the developed random swap programs, in particular that the programs deliver the same result both in sequential and in parallel mode on all the benchmark datasets.

Table 2 summarizes a snapshot of the output generated by one run of Birch1. Steps 2 and 3 are examples of successful steps. Unsuccessful steps ($nMSE$ did not improve) are omitted in the table. Sometimes $nMSE$ improved but CI did not. Such changes are also accepted by the algorithm although they are not considered successful steps like those that failed to improve $nMSE$.

Table 2. Debugging example of Birch1.

Step	nMSE	CI
1	6.622308508119364E8	13
2	6.05899752096259E8	9
3	5.74568446939268E8	8
5	5.73673847218781E8	8
14	5.629913152219414E8	7
19	5.385070562725195E8	6
20	5.3324325081301624E8	5

27	5.213528313807281E8	4
32	5.052130404197854E8	3
100	4.815435483122903E8	1
237	4.7379832580636716E8	0
255	4.664072159088196E8	0
265	4.6386566539892936E8	0
330	4.6386462201360106E8	0
706	4.638641013410574E8	0
1968	4.638640964116055E8	0
2071	4.638640668751212E8	0
2559	4.638640333827571E8	0
4346	4.638640290995247E8	0
5000	4.638640290995247E8	0

The results in Table 2 confirm the monotonic decrease of $nMSE$. Starting from step 237, CI stabilizes to value 0 indicating that the centroids are roughly at their correct positions. Further improvement in $nMSE$ still happens until step 4346 after which all remaining swaps are unproductive as $nMSE$ did not improve further.

Computational efficiency

To check the computational efficiency, five runs of the stream-based random swap program were executed on the Birch1 dataset (see Table 1), both in sequential and parallel mode. The overall wall-clock time required for completing the $T = 5000$ swap iterations (see Alg. 3) was measured. The two times are referred to as RS^S Sequential Elapsed Time (RS^S-SET) and RS^S Parallel Elapsed Time (RS^S-PET). After that, the Average SET (aSET) and the Average PET (aPET) were calculated. The speedup was evaluated as the ratio of aSET/aPET.

Similarly, the SI-SET and SI-PET were measured by computing the Silhouette index (SI) both in the sequential and parallel mode. The average execution times, namely aSI-SET and aSI-PET, were calculated, and the corresponding speedup evaluated as the ratio aSI-SET/aSI-PET. The recorded execution times are summarized in Table 3.

Table 3. Birch1 execution times on the stream-based random swap (RS^S) and Silhouette Index (SI) (P=16 threads).

#run	RS^S-SET (ms)	RS^S-PET (ms)	SI-SET (ms)	SI-PET (ms)
1	1255537	182846	91264	6457
2	1243723	187559	89098	6592
3	1237939	194194	90270	6587
4	1246666	192066	89113	6452
5	1237789	196093	89662	6442

From Table 3, we derive an average value $aSET^S$=1244331 ms, and $aPET^S$=190552 ms. This results in a speedup of 6.53. The average times for SI emerged to be: $aSI\text{-}SET^S$=89881 ms and $aSI\text{-}PET^S$=6506 ms, with a speedup of 13.82.

For completeness, Table 4 reports the measured execution times of Birch1 using the Theatre-based implementation of random swap (RS^T), in sequential (one single, standalone theatre is used) and parallel mode ($P = 16$ theatres are spawned).

From the results in Table 4 an average value $aSET^T$=464035 ms, and an average value $aPET^T$=33612 ms were derived, with a speedup of 13.81. A detailed comparison between Tables 3 and 4 reveals that the Theatre actor-based version of RS outperforms the stream-based version

both in the sequential and the parallel mode. In particular, a *relative* speedup, that is the ratio between aPET^S and aPET^T emerges to be 5.67.

Table 4. Birch1 execution times on the Theatre-based random swap (RS^T) (P=16 threads).

#run	RS^T-SET (ms)	RS^T-PET (ms)
1	460768	31241
2	465323	33547
3	463477	34039
4	464521	35617
5	466087	33617

Better performance of the Theatre-based version mirrors better resource exploitation with respect to the stream implementation. In the stream-based version, each time a parallel operation, like a partition/assignment or a centroid update of K-means or restoring the previous partition as a consequence of an unsuccessful swap iteration, requires to be carried on the dataset, a new stream has to be created upon which the fork/join mechanism spawns new threads for processing chunks of the data points. In the Theatre organization, theatres/threads are initially configured to run on different cores. Each theatre is initialized with the *region* of the dataset to be managed, which remains unchanged for the duration of the simulation. Actors Master and Workers can process, in parallel, operations simply by accepting and executing messages. In this way, the costs of dynamically creating and dismissing threads of the stream-based program, are completely avoided.

Clustering accuracy

The quality of clustering solutions generated by parallel random swap was estimated by 10 runs, each of $T = 5000$ swap iterations, executed in parallel mode, for each dataset in Table 1. A clustering solution consists of the final centroids and the index of the clusters in which the data points were assigned. The accuracy of the solution is captured by $nMSE$, Silhouette index (SI), and Centroid index (CI) calculated between the found solution and the ground truth centroids.

Table 5 reports the resultant quality values for the 10 runs of Birch1.

Table 5. Resulting quality values for the 10 runs of Birch1.

#run	nMSE	SI	CI
1	463863839.562	0.46	0
2	463863818.344	0.46	0
3	463864339.411	0.46	0
4	463864208.385	0.46	0
5	463863830.393	0.46	0
6	463864007.823	0.46	0
7	463864253.632	0.46	0
8	463864127.842	0.46	0
9	463864050.860	0.46	0
10	463863850.588	0.40	0

As seen from Table 5, every solution for Birch1 has value $CI = 0$ which guarantees the correctness of the cluster-level structure of the found solution. The corresponding Silhouette index is always 0.46. The $nMSE$ values also reduce to a common result of 4.64E8 with only minor variation.

The achieved results, $CI = 0$ and $nMSE = 4.64E8$, comply exactly with the same results documented in [32] and in [4].

Table 6 contains the recorded results for the 15 benchmark datasets discussed in Section V, which were studied using $T = 5000$ swap iterations. The data of each row are the averages of 10 independent runs.

As a notable property, both the stream-based and Theatre actor-based variants were capable of finding the correct solution for all datasets except for the Birch3 and Worms_2d cases (see Table 6). For Worms_64d they always obtained value $CI = 0$ ($GCI = 0$) which is the same result as reported in [16]. An average value of $GCI = 7.5$ is reported for the Worms_2d in [16] by using the fast and powerful density peaks clustering algorithm based on a kNN graph construction.

The accuracy of parallel random swap tools was also compared to the Repeated K-Means [31] under random (RKM^R) and K-means++ (RKM^{++}) centroids initialization (see Section II). K-means was repeated 100 times. For simplicity, the minimum nMSE value, the average CI/GCI value (avCI/GCI), the average relative CI/GCI value (avRel − CI/GCI, that is, the average CI/GCI divided by the number of clusters K), the success − rate, i.e. the number of times CI/GCI was found equal to 0 divided by the number of runs, and the average number of iterations (avIT) K-means executed until convergence, were monitored. Tables 7 and 8 report the results separately for the two cases RKM^R and RKM^{++}. For simplicity, all the $nMSE$ values in the Tables from 6 to 8, are reported by dividing them by a scale factor: 10^6 for $A1$ to $A3$, 10^9 for $S1$ to $S4$, 10^0 for $G2-1024-100$ and $D32$, 10^7 for $UNBALANCE$, 10^8 for $BIRCH1$ and $BIRCH3$, 10^6 for $BIRCH2$ and $WORMS\_2D$ and $WORMS\_64D$.

Table 6. Clustering accuracy of the selected benchmark datasets.

dataset	nMSE	SI	CI/GCI
A1	2.02	0.60	0
A2	1.93	0.60	0
A3	1.93	0.60	0
S1	0.89	0.71	0
S2	1.33	0.63	0
S3	1.69	0.49	0
S4	1.57	0.48	0
G2 − 1024 − 100	9982	0.18	0
DIM32	7.10	0.95	0
UNBALANCE	1.65	0.86	0
BIRCH1	4.64	0.46	0
BIRCH2	2.28	0.74	0
BIRCH3	1.86	0.52	13.7
WORMS_2D	0.02	0.36	7.7
WORMS_64D	2.13	0.09	0

Table 7. Results of applying 100 repetitions of RKM^R to the 15 benchmark datasets.

RKM^R	nMSE	avCI/GCI	avRel − CI/GCI	success − rate	avIT
A1	2.35	2.42	12.1%	0%	24.05
A2	2.21	4.56	13.0%	0%	26.29
A3	2.41	6.66	13.3%	0%	27.69
S1	1.32	1.97	13.1%	0%	18.75
S2	1.33	1.41	9.4%	1%	24.3
S3	1.69	1.24	8.3%	11%	29.44
S4	1.57	0.99	6.6%	20%	37.07
G2 − 1024 − 100	9982	0.0	0.0%	100%	2.59
DIM32	142.40	3.63	22.7%	0%	4.78
UNBALANCE	6.49	3.83	47.9%	0%	34.54
BIRCH1	5.02	6.51	6.5%	0%	115.16

BIRCH2	5.49	17.03	17.0%	0%	47.11
BIRCH3	2.03	19.69	19.7%	0%	128.78
WORMS_2D	0.02	27.05	77.3%	0%	126.03
WORMS_64D	2.15	20.08	80.3%	0%	88.14

Table 8. Results of applying 100 repetitions of RKM^{++} to the 15 benchmark datasets.

RKM^{++}	nMSE	avCI/GCI	avRel − CI/GCI	success − rate	avIT
A1	2.02	1.41	7.1%	6%	16.43
A2	2.11	2.84	8.1%	0%	18.88
A3	2.07	4.15	8.3%	0%	22.73
S1	0.89	0.97	6.5%	21%	14.24
S2	1.33	1.01	6.7%	21%	19.82
S3	1.69	1.01	6.7%	21%	23.97
S4	1.57	0.79	5.3%	30%	33.49
G2 − 1024 − 100	9982	0.0	0.0%	100%	2.39
DIM32	7.10	0.14	0.9%	86%	2.18
UNBALANCE	16.5	0.57	7.1%	49%	11.39
BIRCH1	4.88	4.78	4.8%	0%	93.31
BIRCH2	3.36	6.96	7.0%	0%	43.48
BIRCH3	1.90	17.03	17.0%	0%	99.06
WORMS_2D	0.02	27.5	78.6%	0%	110.52
WORMS_64D	2.18	19.83	79.3%	0%	77.63

From the Tables 6, 7 and 8 it clearly emerges that Parallel Random Swap is superior to RKM^{R} and RKM^{++}, and, as expected, RKM^{++} also outperforms RKM^{R} for the better initialization technique which chooses centroids far away each other. Only the $G2 - 1024 - 100$ dataset was correctly solved in all the 100 repetitions of both RKM^{R} and RKM^{++}. This was anticipated in [6] as a consequence of the positive impact of overlap (which is significant in the chosen G2 dataset) on the K-means behavior.

The unsatisfactory clustering results registered by using repeated K-Means algorithms to Birch and Worms datasets should be noted.

Tables 9 and 10 report an observed execution trace respectively for Worms_2d and Worms_64d datasets, when $T = 5000$ swap iterations was used. For simplicity, steps where the GCI value repeats are omitted.

Table 9. An execution trace for Worms_2d

Step	nMSE	GCI
1	31617.126664532883	12
2	25526.79312268663	10
3	23607.109779022692	10
4	23436.676790899786	9
5	21625.758079806976	7
6	20677.503100491762	8
8	20229.59512684583	7
9	19992.72890435367	8
15	19888.94688278889	9
16	19685.0593379738	8
28	19159.778478917495	7
29	19044.685310944787	6
48	18647.38655185947	9
76	18456.228546408256	8
121	18365.40531029041	7
211	18231.97969573148	8
266	18144.940517231156	7

Table 10. An execution trace for Worms_64d

Step	nMSE	GCI
1	2320435.5335298977	13
7	2258728.9430443444	11
12	2244208.0184919285	10
19	2231982.2069647466	9
26	2229550.809941683	9
29	2219612.544481429	8
47	2207176.936861572	7
49	2198411.3557836837	6
68	2181701.4026172264	5
74	2175031.860439957	4
113	2167974.855536757	3
225	2155946.7481358326	2
234	2144827.541439402	1
1105	2139296.362707094	1
1181	2138956.2773386287	0
4049	2132424.422130961	0
5000	2132208.9876207393	0

308	18061.5029279617	9
342	18056.11609784498	7
390	17971.577874260052	8
519	17824.253756818613	9
726	17702.16609296501	8
921	17690.568528220727	9
1710	17642.91790804943	8
1990	17561.070543492715	7
2262	17516.81140096279	8
5000	17494.584121060423	8

From the Table 9 it emerges that, for worms_2d, although the $nMSE$ value monotonically diminishes, the corresponding GCI index fluctuates and sometimes increases even if $nMSE$ decreases. All of this indicates that Worms_2d is an example of a dataset where "good" clustering does not follow the optimization of the $nMSE$ cost.

A different situation occurs for Worms_64d (see Table 10). Here, a decrease in the $nMSE$ value never causes an increase in the GCI value. In addition, the final value of GCI is 0 as expected (see also Table 5). What can explain the different behavior of Worms_2d and Worms_64d are the consequence of multi-dimensionality. While higher dimensions are seemingly more demanding, the clusters actually become more separated when more dimensions are added.

The Birch3 dataset is another well-known complex case where, as for the Worms_2d, "good" clustering does not follow the $nMSE$ optimization.

Two further simulation experiments were executed using the parallel Theatre-based version of random swap applied to the Worms_2d and Birch3 datasets with $T = 10^6$ swap iterations. Table 11 collects the emerged results that confirm the clustering solutions generated by parallel random swap are in the same line of accuracy of other powerful clustering algorithms [16,36].

Table 11. Clustering accuracy of Birch3 and Worms_2d datasets using $T=10^6$ with Theatre-based Parallel Random Swap.

dataset	nMSE	SI	CI/GCI	PET (ms)
BIRCH3	1.87E8	0.520	12	7340912
WORMS_2D	1.75E4	0.36	7	5322889

VII. CONCLUSIONS AND FUTURE WORK

This paper proposes Parallel Random Swap [4], an efficient and reliable clustering algorithm on multi-core machines. It leverages the lock-free concurrency support which Java natively offers when dealing with parallel streams of non-trivial datasets. A second implementation of the algorithm uses the Theatre actor-based framework [11,31,37] which can deliver higher execution performance by exploiting better resource control. The two realizations were applied to a set of benchmark datasets and the results confirmed that the original clustering accuracy can be achieved but with significantly better time efficiency.

Our future work consists of the following.

First, to identify possible heuristics for early termination of the method to avoid an overwhelming number of swap iterations (T). Preliminary experiments suggest that when a certain number of consecutive accepted iterations do not change the reported *local* CI-value between the current and previous solution, followed by a few hundred unproductive iterations, it serves as evidence that the search can be terminated and the last found solution is the correct

one with high probability. More investigation, though, is deemed necessary using both synthetic and real-world datasets.

Second, to extend the Theatre-based approach proposed in this paper toward a better support of big data, e.g., through an implementation based on Spark [29].

Third, to experiment with Java parallelism in other clustering algorithms, e.g., based on density peaks [5]. Some preliminary work [38-39] has been directed to improving specifically K-means with a centroids initialization which depends on the identification of density peaks. A k-nearest neighbors (kNN) approach [16] is used to predict the hyperball radius in D-dimensional space, which is then used to estimate point densities. Density peaks are finally exploited to select initial centroids using a technique derived from Density K-means++ [8,40].

REFERENCES

1 P. Fränti, O. Virmajoki. Optimal clustering by merge-based branch-and-bound. Applied Computing and Intelligence, 2(1):63–82, 2022.

2 P. Fränti, Genetic algorithm with deterministic crossover for vector quantization. Pattern Recognit Lett., 21(1):61–8, 2000.

3 A. Likas, N. Vlassis, JJ. Verbeek. The global k-means clustering algorithm. Pattern Recognition, 36:451–461, 2003.

4 P. Fränti. Efficiency of random swap algorithm. J. Big Data, 5(1):1-29, 2018.

5 A. Rodriguez, A. Laio. Clustering by fast search and find of density peaks. Science, 344(6191):14.92–14.96, 2014.

6 P. Fränti, S. Sieranoja. K-means properties on six clustering benchmark datasets. Applied Intelligence, 48(12):4743-4759, 2018.

7 P. Fränti, S. Sieranoja. How much can k-means be improved by using better initialization and repeats? Pattern Recognition, 93:95-112, 2019.

8 A. Vouros, S. Langdell, M. Croucher, E. Vasilaki. An empirical comparison between stochastic and deterministic centroid initialization for K-means variations. Machine Learning, 110:1975–2003, 2021.

9 R.G. Urma, M. Fusco, A. Mycroft. Modern Java in Action. Manning, Shelter Island, 2019.

10 L. Nigro, F. Cicirelli, P. Fränti. Efficient and reliable clustering by parallel random swap algorithm. In Proc. of IEEE/ACM 26th Int. Symp. on Distributed Simulation and Real Time Applications (DS-RT 2022). IEEE, 2022.

11 L. Nigro. Parallel Theatre: An actor framework in Java for high performance computing. Simulation Modelling Practice and Theory, 106, 102189, 2021.

12 D. Arthur , S. Vassilvitskii. K-means++ : the advantages of careful seeding. ACM-SIAM Symp. on Discrete Algorithms (SODA'07), January 2007.

13 T. Kurita. An efficient agglomerative clustering algorithm using a heap. Pattern Recognition, 24:205–209, 1991.

14 P. Fränti. T. Kaukoranta, D-F. Shen, K-S. Chang. Fast and memory efficient implementation of the exact PNN. IEEE Trans. Image Process., 9(5):773–7, 2000.

15 C. Baldassi. Recombinator K-means: An evolutionary algorithm that exploits K-means++ for recombination. IEEE Transactions on Evolutionary Computation, 20(1), 2022.

16 S. Sieranoja, P. Fränti. Fast and general density peaks clustering. Pattern Recognition Letters, 128:551-558, 2019.

17 E. Figueiredo, M. Macedo, H.V. Siqueira, C.J. Santana Jr, A. Gokhale, & C.J. Bastos-

Filho. Swarm intelligence for clustering - A systematic review with new perspectives on data mining. Engineering Applications of Artificial Intelligence, 82:313-329, 2019.

18 D.W. Van der Merwe, A.P. Engelbrecht. Data clustering using particle swarm optimization. In The 2003 Congress on Evolutionary Computatio (CEC'03), 1:215-220, IEEE, 2003.

19 E. Hancer, C. Ozturk, D. Karaboga. Artificial bee colony based image clustering method. In 2012 IEEE Congress on Evolutionary Computation, pp. 1-5, IEEE, 2012.

20 S. Saatchi, C.C. Hung. Hybridization of the ant colony optimization with the k-means algorithm for clustering. In Scandinavian Conference on Image Analysis, pp. 511-520. Springer, 2005.

21 C.C. Hung, H. Purnawan. A hybrid rough k-means algorithm and particle swarm optimization for image classification. In Mexican International Conference on Artificial Intelligence, pp. 585-593. Springer, 2008.

22 I. Aljarah, S.A. Ludwig. Parallel particle swarm optimization clustering algorithm based on MapReduce methodology, 2012 Fourh World Congress on Nature and Biologically Inspired Computing (NaBIC), pp. 104-111, IEEE, 2012.

23 J. Zhang, G. Wu, X. Hu, S. Li, S. Hao. A parallel k-means clustering algorithm with MPI. In IEEE Fourth International Symposium on Parallel Architectures, Algorithms and Programming, pp. 60-64, 2011.

24 S. Kantabutra, A.L. Couch. Parallel K-means clustering algorithm on NOWs. NECTEC Technical Journal. 1(6):243-247, 2000.

25 W. Zhao, H. Ma, Q. He. Parallel k-means clustering based on MapReduce. In IEEE International Conference on Cloud Computing, pp. 674-679, Springer, 2009.

26 T.H. Sardar, A. Ansari. An analysis of MapReduce efficiency in document clustering using parallel K-means algorithm. Future Computing and Informatics Journal, 3(2):200-209, 2018.

27 D.S.B. Naik, S.D. Kumar, S.V. Ramakrishna. Parallel processing of enhanced K-Means using OpenMP. In IEEE International Conference on Computational Intelligence and Computing Research, pp. 1-4, 2013.

28 S. Cuomo, V. De Angelis, G. Farina, L. Marcellino, G. Toraldo. A GPU-accelerated parallel K-means algorithm. Computers & Electrical Engineering, 75:262-274, 2019.

29 W. Xiao, J. Hu. A survey of parallel clustering algorithms based on spark. Scientific Programming, 2020.

30 M. Ghaffari, S. Lattanzi, S. Mitrović. Improved parallel algorithms for density-based network clustering. In International Conference on Machine Learning, PMLR, pp. 2201-2210, 2019.

31 L. Nigro. Performance of parallel K-means algorithms in Java. Algorithms, 15(4), 117, 2022.

32 P. Fränti, M. Rezaei, Q. Zhao. Centroid index: cluster level similarity measure. Pattern Recognition, 47(9):3034-3045, 2014.

33 P. Fränti and M. Rezaei. Generalized centroid index to different clustering models. Joint Int. Workshop on Structural, Syntactic, and Statistical Pattern Recognition (S+SSPR 2016), Merida, Mexico, LNCS 10029, 285-296, November 2016.

34 P.J. Rousseeuw. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. Journal of Computational and Applied Mathematics, 20:53-65, 1987.

35 Benchmark datasets, http://cs.uef.fi/sipu/datasets/, accessed on July 2022.
36 P. Fränti, R. Sane and J. Piironen. Nested K-means clustering. Unpublished manuscript.
37 F. Cicirelli, L. Nigro. Analyzing Stochastic Reward Nets by model checking and parallel simulation. Simulation Modelling Practice and Theory, 116, 102467, 2022.
38 L. Nigro, F. Cicirelli. Improving K-means by an agglomerative method and density peaks. 3rd Congress on Intelligent Systems (CIS 2022), Springer LNNS, 2022.
39 L. Nigro, F. Cicirelli. Fast and accurate K-means clustering based on density peaks. Int. Conf. on Advances in Data-driven Computing and Intelligent Systems (ADCIS 2022), Springer LNNS, 2022.
40 N. Nidheesh, K.A. Nazeer, P.M. Ameer. An enhanced deterministic K-Means clustering algorithm for cancer subtype prediction from gene expression data. Computers in biology and medicine, 91:213-221, 2017.