IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received February 24, 2019, accepted March 19, 2019, date of current version April 12, 2019.

*Digital Object Identifier 10.1109/ACCESS.2019.2907885*

# A Hybrid MPI/OpenMP Parallelization of *K*-Means Algorithms Accelerated Using the Triangle Inequality

**WOJCIECH KWEDLO, (Member, IEEE), AND PAWEL J. CZOCHANSKI**
Faculty of Computer Science, Bialystok University of Technology, 15-351 Bialystok, Poland

Corresponding author: Wojciech Kwedlo (w.kwedlo@pb.edu.pl)

This work was supported in part by the Bialystok University of Technology under Grant S/WI/2/2018, funded by the Polish Ministry of Science and Higher Education.

**ABSTRACT** The standard formulation of the *K*-means clustering (Lloyd's method) performs many unnecessary distance calculations. In this paper, we focus on four approaches that use the triangle inequality to avoid unnecessary distance calculations. These approaches are Drake's, Elkan's, Annulus, and Yinyang algorithms. We propose a hybrid MPI/OpenMP parallelization of these algorithms in which the dataset and the corresponding data structures storing bounds on distances are evenly divided among MPI processes. Then, in the assignment step of a *K*-means iteration, each MPI process computes the assignment of its portion of data using OpenMP threads. In the update step of the iteration, the cluster centroids are computed using a hierarchical all-reduce operation. In the computational experiments, we compared the strong scalability of these four algorithms with the scalability of Lloyd's algorithm, parallelized using the same approach. The results indicate that all four algorithms maintain an advantage in computing time over Lloyd's algorithm. A comparison with two software packages, whose sources are publicly available, in the same computing environment, shows that our implementations are more efficient.

**INDEX TERMS** Clustering, *K*-means, triangle inequality, MPI, OpenMP, hybrid parallelization.

## I. INTRODUCTION

Clustering [1], [2] is an unsupervised classification technique that is widely applied in many diverse areas, such as biology, social science, and image processing. The aim of clustering can be defined as dividing a set of objects into $K$ disjoint groups, called clusters, in such a way that objects within one cluster are very similar, whereas objects in different clusters are very distinct. In this work, it is assumed that the number of clusters $K$ is known a priori.

The clustering problem can be formulated as the problem of searching for a $K$-partition of the data which minimizes a certain criterion function. The sum of squared error (SSE), which is defined as the sum of squared distances (e.g., Euclidean) from each data item to the centroid of its cluster, is one of the most common criterion functions. The clustering problem with the SSE criterion based on the Euclidean distance is known to be NP-hard [3]. For this reason, heuristic methods, for instance those based on an iterative "hill climbing" approach, must be employed. These methods do not guarantee finding the optimal solution. Instead, they stop searching after a local convergence in which no improvements in the neighborhood of the current solution can be found.

The *K*-means algorithm [4], [5], arguably the most popular clustering method, belongs to a group of such hill climbing approaches. It is an iterative refinement algorithm which, given an initial clustering solution (i.e., a set of cluster centroids), produces a sequence of solutions with decreasing values of the SSE criterion. An iteration of the algorithm consists of two steps: an assignment step and an update step. In the assignment step, each data item is assigned to the cluster with the closest centroid. In the update step, cluster centroids are recalculated based on the assignment of data items. The algorithm is usually stopped when the cluster membership stabilizes.

Interest in clustering algorithms has been renewed recently due to the dramatic growth of volumes of data ("big data")

The associate editor coordinating the review of this manuscript and approving it for publication was Bora Onat.

available from services and resources, such as social networks (e.g., Facebook and Twitter), cloud storage systems, and sensor networks. These huge volumes of data present an opportunity for useful analysis but create their own problems related to storage, processing, and analytical operations [6]. Data clustering is a potential method for overcoming some of these problems by producing clusters/summaries of the entire dataset in a compact and informative form. In this application of iterative clustering methods, sequential implementations cannot get the job done in an acceptable amount of time. Moreover, data are unlikely to fit in the memory of a single computer. For this reason, some form of distributed and/or parallel processing is necessary.

Computer clusters of shared memory nodes are the most widespread parallel architecture. A single node of a cluster usually consists of several multi-core chips sharing memory, with non-uniform memory access (NUMA). The nodes are interconnected via a high-speed network. For such architecture, it is natural to consider a hybrid programming model, which employs a shared memory paradigm (e.g., OpenMP [7]) for parallelization within a node, and a message-passing paradigm (e.g., MPI [8]) for communication across nodes. Although this model lacks some features (e.g., fault tolerance) of higher-level approaches, like MapReduce [9] or Spark [10], it fits well with the hierarchical architecture of a computer cluster, which gives it the possibility of attaining the highest performance [11], [12].

The main contribution of this work is the design, implementation, and experimental evaluation of hybrid parallel MPI/OpenMP versions of five variants of the $K$-means algorithm: the well-known Lloyd's algorithm [4] and newer Drake's algorithm [13], [14], as well as Elkan's [15], Annulus [14], [16], and Yinyang [17] algorithms. Whereas Lloyd's algorithm is a brute-force method in which each iteration must compute the distance between each cluster centroid and each data item, the latter four algorithms use the triangle inequality to skip some distance calculations. This approach can substantially reduce computing time. However, it can pose problems in terms of parallelization because of the possibility of load imbalance. This paper tries to answer the question of whether the superiority of these four algorithms, as demonstrated for their single node (computer) implementations (e.g., [16], [17]), can be maintained in a parallel implementation in a cluster system consisting of dozens of multi-core nodes. To our knowledge, such multi-node parallelization of the triangle inequality accelerated methods have never been attempted before.

The rest of the paper is organized as follows. The next section presents research related to our work. Section III describes the five variants of the $K$-means algorithm investigated in the paper. Section IV presents the method of parallelization of these algorithms. Section V shows the results of computational experiments demonstrating the scalability of the algorithms on a computer cluster. The last section concludes the paper.

## II. RELATED WORK

The $K$-means algorithm is very popular and widely used. In [18], it is nominated as one of the top 10 data mining methods. The batch version (Lloyd's method), which is considered in this paper, was invented by Lloyd in 1957, but was published only in 1982 [4]. Independently, it was proposed in [19].

There are several lines of research on improving the efficiency of the $K$-means algorithm. Since both the number of iterations and the quality of the final solution depend on the initial clusters, some researchers have tried to devise new initialization methods to improve effectiveness (final SSE) and/or efficiency (number of iterations). A standard mean of initialization is to choose random data objects, using the uniform distribution, as cluster centroids [5]. $K$-means++ [20], a popular initialization method, initializes cluster centroids by objects chosen with the probability proportional to the squared shortest distance to the already initialized centroids. It has been demonstrated [20] that initialization by $K$-means++ improves both effectiveness and efficiency. A major limitation of $K$-means++ is its sequential nature. However, a variant of the method, called $K$-means$\|$, which can be efficiently parallelized, was proposed [21]. A brief description of other initialization methods and a thorough experimental comparison with respect to efficiency and effectiveness can be found in [22].

Another line of research is related to the development of fast approximate algorithms. Most of the computation time for iterations of Lloyd's algorithm is spent searching for the nearest centroid of each data item. The time complexity of this step can be significantly reduced by replacing an exact nearest neighbor search with an approximate nearest neighbor search using data structures like a forest of randomized KD-trees [23] or a trinary-projection tree [24]. In [25], a method which computes only distances from a cluster centroid to the points belonging to the cluster closure (neighborhood) was proposed. These cluster closures can be efficiently constructed using multiple random partition trees.

The problem with approximation methods is that they produce clustering results different from Lloyd's algorithm, and the quality of these results can be worse than the quality of Lloyd's algorithm results. For this reason, many researchers have focused on exact algorithms, which give identical results as Lloyd's method. Some have proposed [26], [27] using a KD-tree structure to organize the clustered data. The KD-tree is a binary tree constructed by recursively partitioning the data using axis-aligned hyperplanes. By using the tree to restructure the assignment step of the $K$-means algorithm, many distance calculations can be skipped. This approach works well for low-dimensional data; for dimensions greater than about 20, traversing the tree becomes too expensive [26].

In [28], it was noticed that some cluster centroids, called static centroids, do not move in consecutive iterations. If the update step moves the assigned centroid closer to a point, all static centroids cannot be closer to this point than its

assigned centroid. Thus, the calculations of distance to the static centroids can be avoided.

All four accelerated algorithms considered in this paper belong to a group of methods that maintain an upper bound on the distance to the currently assigned cluster centroid and lower bounds on distances to some other cluster centroids. The bounds can be efficiently updated using the triangle inequality. This line of research was started with Elkan's algorithm [15]. Apart from using $K$ lower bounds (one for each centroid), his method in each iteration computes a matrix of pairwise distances between cluster centroids. Using the triangle inequality, lower and upper bounds, and the matrix, the algorithm is able to omit many unnecessary distance calculations, generating significant speedup. However, maintaining the bounds and the matrix requires additional $O(KN)$ memory, where $N$ is the number of clustered items and $O(K^2)$ time, which may not be feasible for large $K$. For small dimensional feature spaces, the gains from the avoided distance calculations may be offset by the additional effort required for maintaining the bounds and the matrix. As a result, Elkan's algorithm may, although rarely, run slower than Lloyd's method [17].

Hamerly's algorithm [29] is a simplification of Elkan's method. It uses only one lower bound on the distance to the second-closest centroid. Instead of maintaining the matrix of inter-centroid distances, the algorithm maintains, for each cluster centroid, the distance to the other closest centroid. Annulus algorithm [14], [16] improves on Hamerly's method by considering, in the inner loop of the assignment step, only the centroids in an annular region centered by the origin. For low-dimensional feature spaces, these two algorithms yield a shorter computing time than Elkan's method. Their important advantage is much lower memory complexity. However, in high-dimensional spaces, their single lower bound is not efficient enough in comparison to the $K$ bounds of Elkan's method, and the algorithms may run slower.

Drake's algorithm [13], [14] combines the strengths of Elkan's and Hamerly's/Annulus methods by maintaining $1 < B < K$ lower bounds to the $B$ next-closest centroids, excluding the currently assigned closest centroid. The computational experiments [16] indicate that the algorithm performs very well for the medium-dimensional feature spaces. However, Annulus and Elkan's algorithms can be more efficient in low and high dimensions, respectively.

The recently proposed Yinyang algorithm [17] also maintains $1 < B < K$ lower bounds on distances to groups of centroids. These groups are formed at the start of the algorithm by using the $K$-means method to divide $K$ initial centroids into $B$ clusters. The $B$ lower bounds of the Yinyang algorithm are, similar to Drake's method, a compromise between the $K$ lower bounds of Elkan's algorithm and the single lower bound of Hamerly's and Annulus algorithms. Computational experiments have demonstrated [17] a consistent speedup of the Yinyang algorithm over Elkan's and Drake's methods. In [30], a new method for obtaining the lower bounds for the Yinyang algorithm using Euclidean distance was proposed.

MPI [8] and OpenMP [7] are de facto open standards for parallel programming in a distributed and shared memory model, respectively.

The idea of parallelizing $K$-means algorithms is not new. However, research efforts in this direction mostly either focus on Lloyd's algorithm or are limited to single-node parallelization using threads. A parallel version of Lloyd's algorithm for distributed systems implemented using MPI was described in [31]. A hybrid parallelization of Lloyd's method based on MPI and OpenMP was proposed in [32]. Contrary to our method, that approach employed a reduction algorithm with a linear computational complexity depending on the number of cores. More recent implementations of Lloyd's algorithm were developed for architectures used in modern supercomputers. In [33] a version for systems using a high bandwidth scratchpad DRAM, which is physically bonded to a die containing compute cores, was proposed. An example of such architecture was discontinued Intel Knights Landing many-core processor. In [34] a parallel version of Lloyd's algorithm for shared memory systems, which efficiently utilized both caches of a multi-core machine and available Single Instruction Multiple Data (SIMD) units, was proposed. The algorithm was tested on a system with two quad-core Intel CPUs giving a significant speedup over a naive implementation. [35] and [36] describe two implementations of Lloyd's algorithm for the SW26010 many-core processor used in Sunway TaihuLight supercomputer (at the time of writing this paper it was third on the Top500 supercomputer list [37]). While the previous work focuses on fine-tuned kernel running on a single processor, the latter discusses the implementation on thousands of nodes of TaihuLight.

Very few authors have tried to combine exact accelerated algorithms with multi-node parallelization. In [38], a parallel formulation of a KD-tree-based algorithm was proposed. The algorithm, which uses static partitioning of the tree, was implemented using MPI bindings for Java. The method was later extended with dynamic load balancing [39]. According to our knowledge, a hybrid parallelization of triangle inequality-accelerated exact $K$-means algorithms based on a message-passing model for inter-node communication and a shared memory model for intra-node communication has not yet been proposed.

There are several papers that have compared different triangle inequality-based accelerated $K$-means algorithms. However, in all the cases, these comparisons were limited to single-node parallelization using threads or higher-level primitives. The original implementation of the Yinyang algorithm [17] was compared to Elkan's and Drake's methods. All three algorithms were parallelized using a high-level Graphlab package [40] for graph-structured algorithms. This package was implemented in C++ using POSIX threads.

In [16], a comparison of seven triangle inequality-based algorithms, along with their parallelization based on POSIX threads, was described. This work included experimental evaluation of parallel efficiency and speedup on a single 12-core system. A more recent review [41] involved a

comparison of triangle inequality-based acceleration methods, including the Yinyang algorithm and its simplified version, implemented using C++11 threads on a four-core system. This paper also proposed a modified bound update rule, which in most cases reduces the runtime of the $K$-means. Another contribution of [41] was a demonstration that optimized implementations of triangle inequality-based $K$-means using low-level mechanisms, like threads, may outperform implementations using high-level counterparts by a large margin, which was shown by an example of a fine-tuned version of the Yinyang $K$-means compared to the original implementation [17] in Graphlab.

Another research direction is related to implementations of $K$-means algorithms on parallel accelerators, e.g., Graphics Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs). GPU versions of Lloyd's algorithm include [42] and [43]. Among several realizations on FPGAs, [44] and [45] are the most related to our work. The former describes implementation of a simplified version of Elkan's algorithm [15], which does not use inter-centroid distances. The latter presents an implementation of the KD-tree based filtering algorithm [26] for low-dimensional data. One disadvantage of accelerators is that available memory resources are usually much lower than host memory capacity, which is a limiting factor in applications to big data clustering. For instance, both the aforementioned FPGA implementations use on-chip memory to store auxiliary data structures (upper and lower bounds or a KD-tree). On-board memory available to a GPU accelerator usually has lower capacity than the host memory. To tackle this problem, some GPU implementations try to process data in batches [43]. In such cases a host-accelerator link can be a bottleneck.

Practically every machine learning library includes a $K$-means method. Among them, MLPACK [46] includes single-threaded versions of Elkan's, Hamerly's, and KD-tree-based algorithms. A version of Lloyd's method, with some optimizations for sparse data, is part of the MLlib machine learning library [47] for the popular Spark big-data processing engine [10]. Compared to our approach, implementations of distributed machine learning algorithms in Spark offer two important advantages: fault tolerance and no requirement to store the dataset and the bounds in memory. However, their performance can be of one order of magnitude slower than the performance of implementations on HPC platforms using MPI/OpenMP [48].

## III. FIVE VARIANTS OF THE $K$-MEANS ALGORITHM

We assume that clustered items are represented by $N$ feature vectors $x(1), x(2), \ldots, x(N)$, $x(i) \in \mathbb{R}^M$, where $M$ is the dimension of the feature space. We also assume that cluster assignment of feature vectors is stored in an array $a$, where $a(i) \in \{1, 2, \ldots, K\}$ for $i = 1, \ldots, N$.

The SSE criterion is defined by:

$$\mathrm{SSE} = \sum_{i=1}^{N} \mathrm{d}^2(x(i), c(a(i))), \tag{1}$$

**Data:** Initial cluster centroids $c(1), c(2), \ldots, c(K)$ and feature vectors $x(1), x(2), \ldots, x(N)$

1 **repeat**
2  {assignment step}
3  **for** $i \leftarrow 1$ **to** $N$ **do**
4  $a(i) \leftarrow \arg\min_j \mathrm{d}(x(i), c(j))$
5  {update step}
6  **for** $j \leftarrow 1$ **to** $K$ **do** $y(j) \leftarrow \mathbf{0}$, $z(j) \leftarrow 0$
7  **for** $i \leftarrow 1$ **to** $N$ **do**
8  $y(a(i)) \leftarrow y(a(i)) + x(i)$
9  $z(a(i)) \leftarrow z(a(i)) + 1$
10  **for** $j \leftarrow 1$ **to** $K$ **do** $c(j) \leftarrow y(j)/z(j)$
11 **until** TerminationCondition

**Result:** Final cluster centroids $c(1), c(2), \ldots, c(K)$ and cluster assignment $a(1), a(2), \ldots, a(N)$

**FIGURE 1.** Pseudo-code of Lloyd's algorithm.

where $\mathrm{d}(x, y)$ is a distance (e.g., Euclidean) and $c(j) \in \mathbb{R}^M$ is the centroid of the $j$-th cluster for $j = 1, \ldots, K$.

### A. LLOYD'S ALGORITHM

Lloyd's algorithm, commonly referred to as the $K$-means algorithm, is the most popular technique for local minimization of (1). The method, shown in Fig. 1, consists of the alternate application of two steps: an assignment step and an update step. In the former step (lines 3-4), each feature vector $x(i)$ is assigned to the cluster represented by the closest centroid. In the latter step (lines 6-10), the cluster centroids are computed as sample means by accumulating the so-called sufficient statistics: sums of feature vectors assigned to each cluster in $y(1), y(2), \ldots, y(K)$, and counts of feature vectors assigned to each cluster in $z(1), z(2), \ldots, z(K)$ (lines 6-9). Next, for each cluster, its sum is divided by the count, giving new centroid coordinates (line 10).

For the sake of simplicity in Fig. 1, we have omitted the case of an empty cluster, i.e., a cluster centroid which is not the closest centroid to any of the feature vectors. In such a case, the corresponding $z(j)$ will be equal to zero, and a division by zero will happen in line 10 of Fig. 1. There are several possible methods for dealing with empty clusters (e.g., choose as a new centroid the feature vector, which contributes most to SSE). In our implementation, we used the simplest approach, which replaces the centroid of the empty cluster with a random feature vector. This method was chosen because it did not impede parallelization.

It can be shown [49] that both assignment and update steps reduce the SSE, and thus the algorithm converges to a local minimum. The iterations of the algorithm are terminated when the cluster membership vector a stabilizes (which is equivalent to SSE ceasing to improve), although alternative stopping criteria based on minimal relative improvement of SSE or a maximal number of iterations can be employed.

The computational complexity of the assignment step of Lloyd's algorithm is $O(NKM)$ for $K$ clusters and $N$ feature vectors in $M$ dimensions. Because usually $K \ll N$,

the computational complexity of the update step is $O(NM)$. The complexity of the whole algorithm is $O(wNKM)$, where $w$ is the number of iterations.

This paper is mostly concerned with applications of the triangle inequality to optimize the costlier assignment step. There is, however, one possible optimization of the update step: If we keep the sufficient statistics $y$ and $z$ between consecutive $K$-means iterations, summing over all feature vectors (lines 7-9 of Fig. 1) is not necessary. Instead, we can update $y$ and $z$ for feature vectors, which changed the cluster membership [17], [29]. By doing so, we incur the cost of one vector addition and one subtraction for each feature vector that changed membership. Thus, this optimization is beneficial if less than one-half of feature vectors change cluster membership in an iteration. Following the authors of the original papers [4], [13], [15]–[17], we implemented this optimization in the Annulus and Yinyang algorithms; our implementations of Lloyd's, Elkan's, and Drake's algorithms did not use it.

### B. ELKAN'S ALGORITHM

The triangle inequality states that if we have three vectors in $v_1, v_2, v_3 \in \mathbb{R}^M$, then:

$$\mathrm{d}(v_1, v_3) \leq \mathrm{d}(v_1, v_2) + \mathrm{d}(v_2, v_3). \tag{2}$$

Elkan's algorithm [15] uses this inequality in multiple ways to avoid distance calculations. In addition to the variables $x, a, c$ maintained by Lloyd's method, the algorithm maintains, for each feature vector $x(i)$, an upper bound $u(i)$ and $K$ lower bounds $l(i,k)$; $u(i)$ bounds the distance between $x(i)$ and the centroid of its assigned cluster $c(a(i))$, whereas for each $k \in \{1, \ldots K\}$, $l(i,k)$ bounds the distance between $x(i)$ and $c(k)$. At each stage of the algorithm, the bounds must satisfy the following conditions:

$$u(i) \geq \mathrm{d}(x(i), c(a(i))), \tag{3}$$

and

$$l(i,k) \leq \mathrm{d}(x(i), c(k)). \tag{4}$$

If $u(i) \leq l(i,k)$, then from (3) and (4) we get $\mathrm{d}(x(i), c(a(i))) \leq \mathrm{d}(x(i), c(k))$, which means that the centroid $c(k)$ cannot be closer to $x(i)$ than the centroid of the currently assigned cluster $c(a(i))$. Thus, the calculation of the distance from $x(i)$ to $c(k)$ is not necessary.

Maintaining the bounds, Elkan's algorithm must account for the drift (movement) of centroids. Denoted by $\delta(k)$, the drift of a centroid $c(k)$ is the result of the update step. After the update step, lower and upper bounds are recalculated using the following rules:

$$l(i,k) \leftarrow l(i,k) - \delta(k), \quad u(i) \leftarrow u(i) + \delta(a(i)). \tag{5}$$

It is easy to show [15], using the triangle inequality, that the above update rules ensure that conditions (3) and (4) hold in each iteration.

Elkan's algorithm also uses the triangle inequality by exploiting a lemma provided by Phillips [50]. The lemma

```
Data: Initial cluster centroids c(1), c(2), ..., c(K) and feature
      vectors x(1), x(2), ..., x(N)
1  {Initialization of bounds}
2  for i ← 1 to N do
3      a(i) ← 1, u(i) ← ∞
4      for k ← 1 to K do l(i,k) ← 0
5  repeat
6      Compute inter-centroid distance matrix C
7      for k ← 1 to K do s(k) ← min_{j≠k} C(j,k)/2
8      for i ← 1 to N do
9          if u(i) > s(a(i)) then
10             r ← true
11             for k ← 1 to K do
12                 z ← max(l(i,k), C(a(i),k)/2)
13                 if k = a(i) or u(i) ≤ z then continue
14                 if r then
15                     u(i) ← d(x(i), c(a(i)))
16                     r ← false
17                     if u(i) ≤ z then continue
18                 l(i,k) ← d(x(i), c(k))
19                 if l(i,k) < u(i) then
20                     a(i) ← k
21                     u(i) ← l(i,k)
22     c' ← c, c ← UpdateStep
23     for k ← 1 to K do δ(k) = d(c(k), c'(k))
24     for i ← 1 to N do
25         u(i) ← u(i) + δ(a(i))
26         for k ← 1 to K do l(i,k) ← l(i,k) − δ(k)
27 until TerminationCondition
   Result: Final cluster centroids c(1), c(2), ..., c(K) and cluster
           assignment a(1), a(2), ..., a(N)
```

**FIGURE 2.** Pseudo-code of Elkan's algorithm.

states that for each centroid $c(k)$, if $\mathrm{d}(c(a(i)), x(i)) \leq \frac{1}{2}\mathrm{d}(c(k), c(a(i)))$ then $\mathrm{d}(x(i), c(a(i))) \leq \mathrm{d}(x(i), c(k))$. In this case, the calculation of the distance to $c(k)$ is not necessary. To efficiently use Phillips' lemma, the algorithm at the beginning of each iteration pre-computes an inter-centroid distance matrix $C(k,j) = \mathrm{d}(c(k), c(j))$ for each $k \neq j$. Moreover, the algorithm tries to avoid computing distance where possible by exploiting the fact that $\mathrm{d}(x(i), c(a(i))) \leq u(i)$. Thus, if $u(i) \leq \frac{1}{2}d(c(k), c(a(i)))$, then the premise of Phillips' lemma is satisfied and we do not have to compute the distance to $c(k)$.

The whole algorithm, based on a re-formulation of Elkan's ideas in [16], is presented in Fig. 2. At the beginning, the lower and upper bounds must be initialized (lines 2-4). Following [16], we take as $a(i)$ an index of an arbitrary centroid and ensure that in the first iteration, all the lower bounds overlap the corresponding upper bound. This initialization guarantees that tight lower and upper bounds in the first assignment step will be obtained.

Before the assignment step, the algorithm pre-computes the inter-centroid distance matrix $C$ (line 6) and, for each $k = 1 \ldots, K$, s(k), one-half the shortest distance from centroid $c(k)$ to any other centroid (line 7). In the assignment step (lines 8-21), Phillips' lemma is used in two tests: the outer test (line 9) and the inner test (line 13). If $u(i) \leq s(k)$, then any

other centroid cannot be closer to the feature vector $x(i)$ than $c(a(i))$. Thus, the algorithm skips the inner loop, iterating over all the centroids (lines 11-21). In the inner loop, the algorithm uses Phillips' lemma and lower bounds in an inner test to skip the calculation of distances where possible. It also uses the following optimization: If the inner test fails and $u(i)$ is not an exact distance between $x(i)$ and $c(a(i))$, then it may be beneficial to recompute $u(i)$ as the exact distance (i.e., tighten the upper bound) and check the test again (line 17). To avoid multiple calculations of this distance, the algorithm uses the logical variable $r$. If $r =$ **false**, then the upper bound is tight.

After the assignment step, Elkan's algorithm performs the update step (line 22) and recalculates the bounds (lines 23-26) according to (5).

A drawback of Elkan's algorithm is memory complexity. Apart from $O(NM)$ storage required for the learning set, the algorithm requires $O(NK)$ storage for the bounds. In situations where $K \gg M$, this may be prohibitive.

### C. THE ANNULUS ALGORITHM

The Annulus algorithm [14], [16] is an extension of the earlier Hamerly's method [29], which in turn is a simplification of Elkan's approach. The upper bound $u(i)$ is defined exactly the same as in Elkan's algorithm. However, instead of using $K$ lower bounds, a single lower bound $l(i)$ on the distance to the second-closest cluster centroid is employed; $l(i)$ must satisfy the following condition:

$$l(i) \leq \min_{k \neq a(i)} \mathrm{d}(c(k), x(i)). \tag{6}$$

If $u(i) \leq l(i)$, then $a(i)$ cannot change, which allows the algorithm to skip the innermost loop iterating over the centroids. The Annulus method also uses Phillips' lemma [50] to skip the innermost loop by pre-computing in $s(k)$ one-half the distance from $c(k)$ to its nearest other centroid. When the bounds overlap and Philips' lemma check fails, the innermost loop cannot be avoided. The original Hamerly's algorithm [29] in this case performs the standard iteration over all the centroids. However, the Annulus method in this situation considers only centroids in an annular region centered by the origin of the coordinate system (Fig. 3). If

$$|\mathrm{d}(x(i), O) - \mathrm{d}(c(j), O| \geq \mathrm{d}(x(i), c(a(i)), \tag{7}$$

where $O$ is the origin, then it is easy to show [16] that the centroid $c(j)$ cannot possibly be closer to $x(i)$ than $c((a(i))$. Thus, the calculation of the distance from $x(i)$ to $c(j)$ is not necessary. The pseudo-code of the Annulus method is shown in Fig. 4. A new variable $b(i) \in \{1, \ldots, K\}$ is required, which represents the index of the second-nearest centroid to $x(i)$.

The assignment step starts with a simultaneous test for overlap of two bounds and Philips' condition (lines 7-8). If this check fails, $u(i)$ is made tight (line 9) and the test is repeated (line 10). If the test fails again, the inner loop searching the annulus is entered. Because the search must obtain not only new $a(i)$ (lines 17-18) but also new $b(i)$ (line 21), the radius of the annulus (lines 11-12) must be wider [16] than the one depicted in Fig. 3.

The key to higher efficiency with the algorithm in comparison with Hamerly's approach is the rapid determination of centroids belonging to the annulus. The inequality in

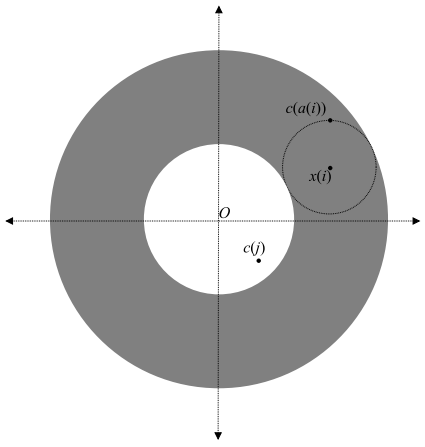


**FIGURE 3.** The annular region (gray ring) limiting the search for a centroid closer to $x(i)$ than $c(a(i))$ [16]. Since $c(j)$ is outside the annulus, it need not be considered during the search.

**Data:** Initial cluster centroids $c(1), c(2), \ldots, c(K)$ and feature vectors $x(1), x(2), \ldots, x(N)$

1 **for** $i \leftarrow 1$ **to** $N$ **do**
2  $a(i) \leftarrow 1$, $b(i) \leftarrow 1$
3  $u(i) \leftarrow \infty$, $l(i) \leftarrow 0$
4 **repeat**
5  **for** $k \leftarrow 1$ **to** $K$ **do** $s(k) \leftarrow \min_{j \neq k} \mathrm{d}(c(j), c(k))/2$
6  **for** $i \leftarrow 1$ **to** $N$ **do**
7  $z \leftarrow \max(l(i), s(a(i)))$
8  **if** $u(i) \leq z$ **then continue**
9  $u(i) = \mathrm{d}(c(a(i)), x(i))$
10  **if** $u(i) \leq z$ **then continue**
11  $l(i) \leftarrow \mathrm{d}(x(i), c(b(i)))$
12  $r \leftarrow \max(l(i), u(i))$
13  $J \leftarrow \{k : |\mathrm{d}(x(i), O) - \mathrm{d}(c(k), O)| < r\}$
14  **forall** $k \in J$ **do**
15  **if** $\mathrm{d}(x(i), c(k)) < u(i)$ **then**
16  {New nearest centroid}
17  $l(i) \leftarrow u(i)$, $u(i) \leftarrow \mathrm{d}(x(i), c(k))$
18  $b(i) \leftarrow a(i)$, $a(i) \leftarrow j$
19  **else if** $\mathrm{d}(x(i), c(k)) < l(i)$ **then**
20  {New second-nearest centroid}
21  $b(i) \leftarrow j$, $l(i) \leftarrow \mathrm{d}(x(i), c(k))$
22  $c' \leftarrow c$, $c \leftarrow$ **UpdateStep**
23  **for** $k \leftarrow 1$ **to** $K$ **do** $\delta(k) = \mathrm{d}(c(k), c'(k))$
24  $\delta' = \max_k \delta(k)$
25  **for** $i \leftarrow 1$ **to** $N$ **do**
26  $u(i) \leftarrow u(i) + \delta(a(i))$
27  $l(i) \leftarrow l(i) + \delta'$
28 **until TerminationCondition**

**Result:** Final cluster centroids $c(1), c(2), \ldots, c(K)$ and cluster assignment $a(1), a(2), \ldots, a(N)$

**FIGURE 4.** Pseudo-code of the Annulus algorithm.

line 13 of Fig. 4 can be transformed to the equivalent form:

$$\mathrm{d}(c(i), O) \in (\mathrm{d}(x(i), O) - r, \mathrm{d}(x(i), O) + r)\,. \qquad (8)$$

If we pre-compute the distance between each feature vector and the origin (once per $K$-means run) and sort all the centroids according to their distance from the origin (once per $K$-means iteration), then the set of centroids satisfying (8) can be obtained quickly by performing two binary searches over the sorted list of centroids.

After the update step, the algorithm recalculates the bounds (lines 23-27). The update rule for $u(i)$ is the same as in Elkan's method. The lower bound $l(i)$ is reduced by the maximum centroid drift. This update rule ensures the correctness of $l(i)$ in the next iteration [16].

### D. DRAKE'S ALGORITHM

Drake's algorithm [13], [14], [16] attempts to combine the advantages of the Elkan's and Annulus methods. For each feature vector $x(i)$, it uses one upper bound $u(i)$ on the distance to the closest centroid $c(a(i))$ and $1 < B < K$ lower bounds $l(i, k)$, where $k = 1, \ldots, B$. The algorithm relies only on bounds in eliminating distance calculations, abandoning the use of Philips' lemma, which requires $O(MK^2)$ additional computational burden for computing inter-centroid distances.

The strict interpretation of lower bounds is as follows: $l(i, k)$, for $k = 1, \ldots, B-1$, is a lower bound on the distance between $x(i)$ and $k$-th closest centroid, excluding $c(a(i))$; $l(i, B)$ represents a lower bound on the distance to $K - B - 1$ furthest centroids. At each stage of the algorithm, the lower bounds $l(i, 1) \ldots l(i, B)$ are maintained in increasing order. Apart from using $l(i, k)$ the algorithm has to track, in a variable $b(i, k) \in \{1, \ldots K\}$ index of $k$-th closest centroid, excluding $c(a(i))$. The pseudo-code of the method is shown in Fig. 5.

```
   Data: Number of bounds B, initial cluster centroids
         c(1), c(2), ..., c(K) and feature vectors
         x(1), x(2), ..., x(N)
 1 for i ← 1 to N do
 2     a(i) ← 1, u(i) ← ∞
 3     for k ← 1 to B do l(i,k) ← 0, b(i,k) ← 1
 4 repeat
 5     for i ← 1 to N do
 6         if u(i) ≤ l(i,1) then continue
 7         else if u(i) > l(i,B) then
 8             for k ← 1 to K do L[k] ← (k, d(x(i), c(k)))
 9             PartialSort (L,B)
10             (a(i), u(i)) ← L[1]
11             for k ← 1 to B do (b(i,k), l(i,k)) ← L[k+1]
12         else
13             j ← min_{k:u(i)≤l(i,k)} k
14             L[1] ← (a(i), d(x(i), c(a(i))))
15             for k ← 1 to j − 1 do
                   L[k+1] ← (b(i,k), d(x(i), c(b(i,k))))
16             Sort (L)
17             (a(i), u(i)) ← L[1]
18             for k ← 1 to j − 1 do
                   (b(i,k), l(i,k)) ← L[k+1]
19     c' ← c, c ← UpdateStep
20     for k ← 1 to K do δ(k) = d(c(k), c'(k))
21     δ' = max_k δ(k)
22     for i ← 1 to N do
23         u(i) ← u(i) + δ(a(i))
24         l(i,B) ← l(i,B) − δ'
25         for k ← B − 1 downto 1 do
26             l(i,k) ← l(i,k) − δ(b(i,k))
27             l(i,k) ← min(l(i,k), l(i,k+1))
28 until TerminationCondition
   Result: Final cluster centroids c(1), c(2), ..., c(K) and cluster
           assignment a(1), a(2), ..., a(N)
```

**FIGURE 5.** Pseudo-code of Drake's algorithm.

In the assignment step (lines 5-18), the algorithm, similar to the Elkan's and Annulus methods, first checks whether the upper bound $u(i)$ is not greater than the smallest lower bound $l(i, 1)$. In this case, $a(i)$ cannot change and all distance calculations from $x(i)$ to the centroids can be avoided. If this test fails, then the algorithm checks (line 7) whether $u(i)$ is greater than the maximum lower bound $l(i, B)$. In this case, the bounds failed to eliminate any distance calculations. The algorithm then computes distances from $x(i)$ to all the centroids (line 8). Next, it partially sorts, by increasing distance, a list L of pairs consisting of centroid indices and distances (line 9). After the sorting $a(i)$ becomes the index of the closest centroid and $u(i)$ the distance to this centroid (line 10). The indices of next $B$ closest centroids and the corresponding distances to $x(i)$ are stored in $b(i, 1), \ldots, b(i, B)$ and $l(i, 1), \ldots, l(i, B)$, respectively (line 11).

Lines 13-18 of Fig. 5 represent a middle case between the two extremes discussed above, in which the upper bound $u(i)$ is smaller than or equal to $j$-th (where $j > 1$) lower bound $l(i, j)$. In this situation, a new centroid closest to $x(i)$ could be any of the following: $c(a(i)), c(b(i, 1)), \ldots, c(b(i, j-1))$. Thus, the algorithm must calculate (lines 14-15) and sort (line 16) the list of distances to the centroids in question, obtaining new indices $a(i), b(i, 1), \ldots, b(i, j-1)$ and the corresponding tight bounds $u(i), l(i, 1), \ldots, l(i, j-1)$ (lines 17-18).

The upper bound $u(i)$ is updated (line 23) in the same way as in the Elkan's and Annulus algorithms. In the update rules for lower bounds (lines 24-27) there are two compromises that increase efficiency at the cost of reduced tightness and the option to perform more distance calculations in the next $K$-means iteration. First, $l(i, B)$ is reduced (line 24) by the maximum drift of all the centroids. It could be reduced by the maximum drift of $K - B - 1$ furthest centroids, but doing so would require tracking the identities of these centroids. Secondly, if the lower bound update (line 26) violates the requirement that the lower bounds stay in increased order, then the algorithm replaces $l(i, k)$ by $l(i, k+1)$, thereby ensuring the order of the bounds at the cost of their tightness. Apparently, this strategy is more efficient than sorting $B$ lower bounds and the corresponding centroid indices.

$B$ is a parameter of Drake's method that may be kept constant or controlled adaptively using the following

strategy [13], [14], [16]. Initially, $B = K/4$. During each iteration of $K$-means, the algorithm computes the maximum, across all $i$, of the value of $j$ obtained in line 10 of Fig. 5. $B$ in a subsequent iteration is reduced to that maximum, subject to limit $B = K/8$.

### E. THE YINYANG ALGORITHM

The Yinyang algorithm [17] seeks an efficient compromise between the $K$ lower bounds of Elkan's method and one lower bound of Hamerly's and Annulus approaches. Similar to Drake's algorithm, it does not employ Philips' lemma and uses one upper bound $u(i)$ and $1 < B < K$ lower bounds $l(i,k)$, where $1 \leq k \leq B$. However, the lower bounds are defined entirely differently than in Drake's method. At the beginning of the $K$-means run, cluster centroids are partitioned into $B$ groups. In [17], it was proposed to set $B = K/10$ and obtain this $B$-partition by using the $K$-means algorithm, taking centroids $c(1), \ldots, c(K)$ as the data. The B-partition is denoted by $\Pi = \{\pi(1), \ldots, \pi(B)\}$, where $\pi(k)$ represents the centroid indices belonging to the $k$-th group. $\Pi$ must satisfy the following two conditions:

$$\bigcup_k \pi(k) = \{1, \ldots, K\} \quad \text{and } \forall_{j \neq k} \pi(j) \cap \pi(k) = \emptyset. \tag{9}$$

$l(i,k)$ is a lower bound on the distance between $x(i)$ and the closest centroid from the $k$-th group, excluding the currently assigned $c(a(i))$:

$$l(i,k) \leq \min_{j \in \pi(k) \wedge j \neq a(i)} \mathrm{d}(c(j), x(i)). \tag{10}$$

If $u(i) \leq l(i,k)$, then the calculations of distances to all the centroids from the $j$-th group can be avoided. However, if $u(i) > l(i,k)$, then the Yinyang algorithm makes one final attempt to avoid distance calculations to some centroids from the $k$-th group via a local filtering step. For each $j \in \pi(k)$, the algorithm tests the following condition:

$$l^*(i,k) < l'(i,k) - \delta(j), \tag{11}$$

where $l^*(i,k)$ is the exact distance to closest centroid found thus far from the $k$-th group, excluding $c(a(i))$, $l'(i,k)$ is the value of the lower bound before the update step, and $\delta(j)$ is the distance moved by $c(j)$ as a result of the update step. In [17], it is shown, using the triangle inequality, that in this case, $c(j)$ cannot be either the new closest centroid or the closest centroid from the $k$-th group, excluding $c(a(i))$, which is needed to tightly compute the lower bound $l(i,k)$. Thus, the calculation of the distance from $x(i)$ to $c(j)$ can be omitted.

The pseudo-code of Yinyang $K$-means, shown in Fig. 6, is structured differently [17] from the three earlier triangle inequality-based methods. After the partition of the initial centroids into $B$ groups (line 1), a single assignment step from Lloyd's algorithm (lines 2-5) is performed that allows tight upper (line 4) and lower bounds (line 5) to be obtained. Next, the algorithm enters the main loop of $K$-means (lines 6-22), which starts with the update step (line 7). After the update step, the algorithm computes the drift of both each centroid

```
Data: Number of bounds B, initial cluster centroids
      c(1), c(2), ..., c(K) and feature vectors
      x(1), x(2), ..., x(N)
1  Π ← Partition((c(1), ..., c(K)), B)
2  for i ← 1 to N do
3      a(i) ← arg min_k d(x(i), c(k))
4      u(i) ← d(x(i), c(a(i)))
5      for k = 1 to B do l(i,k) ← min_{j:j∈π(k)\a(i)} d(x(i), c(j))
6  repeat
7      c' ← c, c ← UpdateStep
8      for j ← 1 to K do δ(j) ← d(c(j), c'(j))
9      for k ← 1 to B do Δ(k) ← max_{j∈π(k)} δ(j)
10     for i ← 1 to N do
11         u(i) ← u(i) + δ(a(i))
12         a' ← a(i)
13         for k ← 1 to B do
14             l'(k) ← l(i,k)
15             l(i,k) ← l(i,k) − Δ(k)
16         L ← min_k l(i,k)
17         if u(i) ≤ L then continue
18         u(i) ← d(x(i), c(a(i)))
19         if u(i) ≤ L then continue
20         Π̂ ← {k : l(i,k) < u(i)}
21         LocalFiltering
22 until TerminationCondition
   Result: Final cluster centroids c(1), c(2), ..., c(K) and cluster
           assignment a(1), a(2), ..., a(N)
```

**FIGURE 6.** Pseudo-code of the Yinyang algorithm.

```
1  forall k ∈ Π̂ do
2      l(i,k) ← ∞
3      forall j ∈ π(k) do
4          if a' == j then continue
5          if l(i,k) < l'(k) − δ(j) then continue
6          if d(x(i), c(j)) < u(i) then
7              m ← g(a(i))
8              l(i,m) ← u(i)
9              u(i) ← d(x(i), c(j))
10             a(i) ← j
11         else if d(x(i), c(j)) < l(i,k) then
               l(i,k) ← d(x(i), c(j))
```

**FIGURE 7.** Local filtering in the Yinyang algorithm.

(line 8) and each group (line 9). Next, it iterates over all feature vectors (lines 10-21), first updating the upper (line 11) and lower bounds (lines 13-15), and second recalculating, if necessary, the assignments $a$ (lines 16-21).

In the assignment step the algorithm first checks (lines 16–17), whether the upper bound is less or equal than minimum of lower bounds. If this condition holds, none of the centroids can be closer to $x(i)$ than the currently assigned $c(a(i))$. All the distance calculations for $x(i)$ can be thus avoided. If this test fails, the algorithm, similarly to Elkan's and Annulus methods, tightens the upper bound (line 18) and repeats the test (line 19). If the test fails again, then some distance calculation have to be performed. Yinyang $K$-means performs the local filtering (Fig. 7), considering only the groups which failed the group filtering test (line 20 of Fig. 6).

In the local filtering stage, the algorithm searches for a new assignment $a(i)$ while also computing tightly the lower bound for each group in question. For each centroid $c(j)$, different from the previously assigned $c(a')$ (the distance to $c(a')$ was already calculated in line 18 of Fig. 6), which fails the local filtering condition (line 5), the algorithm computes distance, checking whether $c(j)$ becomes the closest centroid found thus far (lines 7-10) or the closest centroid found thus far from the $k$-th group, excluding $c(a(i))$ (line 11).

An interesting question arises as to whether relying on a single partitioning of centroids before the beginning of the $K$-means run reduces the efficiency of the Yinyang algorithm. Obtaining a high efficiency of bounds is dependent on compact $B$ groups being formed by the centroids. Thus, if the centroids in a group drift in different directions, this efficiency can be greatly reduced. In [17], the authors stated, without giving any experimental details, that adding a regrouping of centroids to Yinyang $K$-means does not significantly improve its performance. Our own experiments [51] seem to confirm this.

## IV. PARALLELIZATION OF $K$-MEANS ALGORITHMS

### A. MPI AND OPENMP

MPI is the de facto standard for parallel programming in a message-passing model. The MPI specification [8] defines a set of library calls for the Fortran and C/C++ languages. An MPI application consists of processes running in parallel in separate memory address spaces. Data transfer (communication) between address spaces of different processes occurs when they exchange messages. In MPI, there are two categories of communication operations: point-to-point operations, which involve two processes; and collective operations, which involve groups of processes.

Among the collective communication operations, an all-reduce operation [52] plays an important role in our implementation. In this operation, each process contributes a vector of numbers. The result of the operation is obtained by applying an associative reduction operator (the sum in our case) to the vectors. The result is returned to all the contributing processes.

OpenMP [7] is the de facto standard for parallel programming for shared memory architectures. An OpenMP program is executed by a group of cooperating threads. OpenMP is implemented via a set of compiler directives and library calls for the C/C++ and Fortran languages. The directives instruct the compiler to manage and synchronize the threads and balance the work. OpenMP is supported by many open source and proprietary C/C++ and Fortran compilers.

Thread creation and management in OpenMP follows the fork-join pattern, in which a program starts executing in a single (master) thread. The master thread forks additional threads when it encounters a parallel region. When all threads in the region finish executing their specified work, the program performs the thread-join operation and resumes sequential execution by the master thread. OpenMP defines several work-sharing constructs for partitioning the computation in a parallel region. In particular, a loop construct makes it possible to distribute the execution of iterations of a `for` loop among the team of OpenMP threads.

### B. PARALLEL ALGORITHMS DESIGN

From the programmer's standpoint, the simplest method for programming multi-core computer clusters is to ignore the hierarchical memory structure and use a pure message-passing model (i.e., pure MPI). This strategy, which places an MPI process in each core of a parallel system, has some justification because most MPI implementations are optimized to use shared memory in intra-node communication. A pure MPI model does not require changing the existing MPI codes or support from the MPI library for multi-threading. However, all communication between processes within a node add an MPI layer overhead. In contrast, a hybrid model combining shared memory and message-passing programming better matches the hierarchical architecture of computer clusters and may thus offer some benefits in terms of performance.

Our implementation uses a hybrid master-only parallelization [12] in which MPI calls are performed by the master thread outside OpenMP parallel regions. The MPI library must support multi-threading on the simplest `MPI_THREAD_FUNNELED` level [8]. Practically all the MPI libraries have such support.

Our design, similar to the pure MPI approach to parallelizing Lloyd's algorithm [31], exploits the fact that all calculations performed by the algorithms in the assignment step are inherently data-parallel and can be executed independently for each feature vector given some global data. These global data are the centroids $c(1), c(2), \ldots, c(K)$ and, for the Elkan's and Annulus algorithms, the inter-centroid distance matrix $C$ and the vector of the shortest distances $s$ (the Annulus algorithm does not need to store $C$). The dataset $X$, the assignment $a$, and the bounds $l$ and $u$ are statically partitioned by $n$ MPI processes $P_1, P_2, \ldots, P_n$. Each MPI process is responsible for computing the assignment for approximately $N/n$ feature vectors using the corresponding bounds (except Lloyd's algorithm, which does not use bounds). To enable this parallel computation, each process maintains a local copy of the centroids as well as the inter-centroid distance matrix $C$ and the vector of shortest distances $s$ (where applicable). A similar decomposition of data is used in the update step, where each process computes partial sufficient statistics only for its partition of feature vectors.

Each MPI process executes in $k$ threads in each OpenMP parallel region. Let us denote by $t_{j,1}, t_{j,2}, \ldots, t_{j,k}$ OpenMP threads of process $P_j$. The decomposition scheme is hierarchically applied to the partition of data allocated to $P_j$. In the assignment step, each thread computes the cluster assignment for a subset of feature vectors allocated to $P_j$. Similarly, in the update step, each thread is responsible for the computation of partial sufficient statistics for a subset of vectors allocated to $P_j$. OpenMP facilitates implementation of this decomposition with the parallel loop

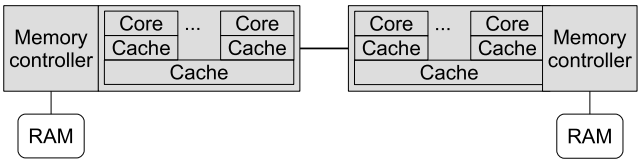


**FIGURE 8.** An example of a computer cluster node with two NUMA domains. Each domain consists of a memory controller, to which several cores are connected. The cores in a domain share the last level of cache memory.

construct [7]. All the loops in Figs. 1, 2, 4–6 with the range of iterations $[1, \ldots, N]$ are parallelized using this two-level approach.

Our implementation has a compile-time parameter that allows us to choose between OpenMP static and guided scheduling [7] of iterations of all loops parallelized using this hierarchical method. In static scheduling, the portion of loop iterations assigned to a thread is known before the execution of the loop. Each thread is responsible for computing $a$ (in the assignment step), for computing the partial sufficient statistics (in the update step), and for updating the bounds $l$ and $u$ (e.g., in lines 22-27 of Fig. 5) for approximately $N/(nk)$ feature vectors. The assignment of data to threads does not change during the $K$-means algorithm run. Static scheduling may be beneficial if the cluster nodes have a NUMA architecture.

Fig. 8 shows an example of such architecture as commonly used in nodes of modern computer clusters. The physical memory is divided into two NUMA domains. The main memory accesses performed by a core may be serviced by a local or remote memory controller. Requests to a remote controller have a larger latency than requests to a local one. Parallel applications should try to improve the percentage of local memory accesses.

Most operating systems default to a first-touch policy in which a memory page is allocated from the NUMA domain from which it was first accessed. In static scheduling, each OpenMP thread accesses the same memory pages in successive $K$-means iterations, which thereby optimizes locality. However, static scheduling may be susceptible to load imbalance, wherein the amount of work performed by each thread is not approximately equal. While Lloyd's algorithm should have perfect load balance, four accelerated algorithms may suffer from load imbalance due to the differing effectiveness of bounds in different threads.

Guided scheduling is performed during the execution of a loop. Iterations are distributed to threads in chunks of exponentially decreasing size. When a thread finishes the execution of a chunk of iterations, it requests another chunk from the internal work queue until there are no more iterations to work on. Guided scheduling does not optimize memory access locality since the set of iterations assigned to a thread (and the memory regions accessed) is non-deterministic. It also has higher overhead than the static scheduling. Its main advantage is the capability to handle poorly balanced and unpredictable workloads.

It is important to note that in guided thread scheduling, dynamic load balancing is performed on a subset of approximately $N/n$ iterations allocated to the MPI process. This subset remains constant during the algorithm's run, i.e., there is no dynamic load balancing between processes.

Fig. 9 illustrates our approach on an example of Drake's algorithm. The update step, corresponding to lines 6-10 in Fig. 1 and line 19 in Fig. 5, begins by computing partial sufficient statistics (the vector sums and counts) by OpenMP threads. Each thread computes the partial sufficient statistics for a subset of feature vectors assigned to the process. In order to compute new centroid coordinates, the partial sufficient statistics obtained by the threads must be summed and collected by each process using the all-reduce operation. This operation is implemented hierarchically: First, by using a minimum-spanning tree reduction algorithm [52], the master thread of each MPI process obtains a sum of partial sufficient statistics computed by the threads of the process. Next, the `MPI_Allreduce` MPI call [8], performed by the master threads, computes the sufficient statistics and sends them to each process. New centroid coordinates are computed from the sufficient statistics (line 10 in Fig. 1) by each process in a sequential manner.

### C. COMPUTATION OF INTER-CENTROID DISTANCES FOR THE ELKAN'S AND ANNULUS ALGORITHMS

Before the assignment step, the Elkan's and Annulus algorithms must compute the distances between all pairs of centroids (lines 6-7 in Fig. 2 and line 5 in Fig. 4). Both algorithms need these distances to obtain the vector of shortest distances $s$. Additionally, Elkan's algorithm requires the storage of the distances in matrix $C$.

The computation of inter-centroid distances is parallelized using a data decomposition as shown in Fig. 10. A process $P_j$ is responsible for the computation of a continuous block of approximately $K/n$ rows of $C$. Each thread of $P_j$ computes the distances for a subset of approximately $K/k$ columns of the block. In Elkan's algorithm, matrix $C$ is reconstructed by an `MPI_Allgatherv` call [8], which collects blocks of rows from the processes and distributes the combined matrix to all of them. The vector $s$ is then computed by each process from $C$ using an OpenMP parallelized loop.

In the Annulus algorithm, each process searches for elements of $s$ corresponding to the rows assigned to the process. The centroid distances are computed by the threads using the decomposition shown in Fig. 10. Each thread finds the shortest distance for its subset of elements in a given row. The row's shortest distance is then obtained by the master thread using the OpenMP reduce clause [7]. These steps are repeated for each row assigned to the process. Next, the vector $s$ is reconstructed from the partial results and distributed to each process by an `MPI_Allgatherv` operation.

## V. EXPERIMENTAL RESULTS AND DISCUSSION

In this section, the experimental results of five compared $K$-means algorithms running on a multi-node computer

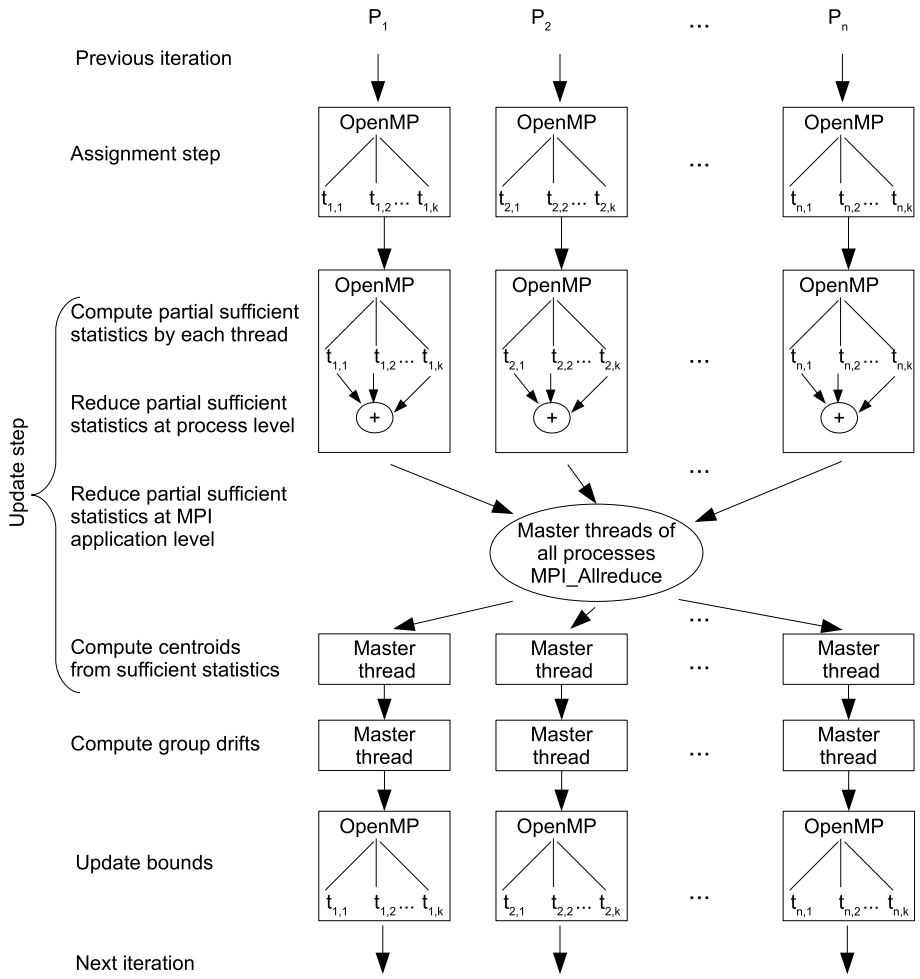


**FIGURE 9.** Hybrid MPI/OpenMP parallelization of Drake's algorithm. Lloyd's algorithm does not compute the group drifts nor update the bounds. The Elkan's and Annulus algorithms additionally compute in parallel the inter-centroid distance matrix before the assignment step.

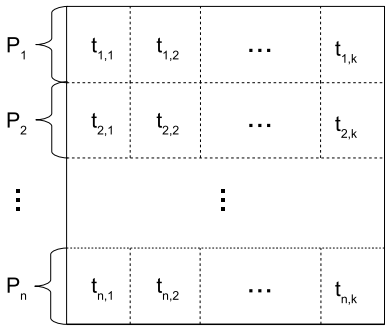


**FIGURE 10.** Decomposition of the matrix of inter-centroid distances $C$.

cluster are presented. The experiments had several objectives. The first objective was the selection of the most efficient OpenMP loop scheduling policy for each of the investigated algorithms. The second objective was the evaluation of strong scalability of the algorithms. The third objective was the selection of the fastest algorithm running in a multi-node setup. The last objective was the comparison of our hierarchical MPI/OpenMP parallelization with the pure MPI parallelization.

### A. TESTING PLATFORM

All the computational experiments reported in this paper were conducted on a Tryton supercomputer located at the Academic Computer Centre in Gdansk, Poland. Each node of the supercomputer has two 12-core Intel Xeon E5-2670 v3 (2.3 GHz) processors with 128 or 256 GiB DDR4 RAM. The nodes are interconnected by the Infiniband FDR 56 Gb/s network in a fat tree topology. The algorithms were implemented from scratch in C++ and compiled by an Intel C++ compiler (icpc version 17.0.1) using the following set of optimization options: `-Ofast -ipo -xCORE-AVX2` [53]. The programs[1] were linked with the OpenMPI 1.10.1 library [54]. The library was compiled by an Intel compiler and implements the MPI-3 standard.

[1]The source code of our implementation is available at https://bitbucket.org/wkwedlo/hybrid-triangle-kmeans/src

### B. RUNTIME MEASUREMENT METHODOLOGY

The scalability results reported in this section were based on a wall-clock execution time measurement of runs of $K$-means algorithms, excluding the time needed to load the dataset from the file system to the memory. The algorithms were terminated when the relative improvement of SSE (1) between two consecutive iterations was lower than $10^{-6}$. The execution time measurements were performed using nodes exclusively allocated to our application by a batch system (SLURM 16.05.07) during normal operation of the computer cluster. In all the experiments with the hybrid MPI/OpenMP version, all 24 cores of each allocated node were utilized by 24 OpenMP threads. The CPU affinity of the OpenMP threads was ensured by setting the `KMP_AFFINITY` environment variable [53] to `compact`.

Execution time in a parallel system is non-deterministic [55]. The sources of non-determinism include system noise (e.g., interrupts, process scheduling, network traffic), the application (e.g., load balancing), or even interference and resource contention from other applications running on the same system [56]. To minimize the influence of this non-determinism, we repeated each experiment five times. The medians of five execution times were used in efficiency and speedup calculations.

For each investigated $K$-means algorithm, the experiments were performed for a number of clusters $K \in \{64, 256, 1024, 4096\}$. First, using the $K$-means‖ method [21], we generated one initial solution for each value of $K$. The solution was later used to initialize all $K$-means algorithms running for the given value of $K$. Thus, for each value of $K$, all the algorithms had to execute the same number of $K$-means iterations.

### C. DATASET

All the clustering experiments were performed on a random sample of a tiny images dataset,[2] introduced in [57]. The original dataset consists of 79,302,017 32x32 color images collected from the internet. Each image is labeled with one of 75,062 English nouns. The labeling information was not utilized during clustering. We used a version of the data in which each image is represented by a global GIST descriptor [58]. A GIST descriptor of an image is a 384-dimensional feature vector representing the texture within localized grid cells. To fulfill the memory requirements of Elkan's, Yinyang, and Drake's algorithms running on a single node, we used a random sample consisting of 19,826,778 (approximately one-quarter) of the vectors.

It is important to note that the vectors in the dataset were sorted by the labels. Our sampling preserved this property: The vectors were considered sequentially, and each vector had a 1/4 probability of being included in the sample. This order of vectors increased the chance of observing load imbalance in the parallel system.

[2]The original images and the GIST descriptor version can be downloaded from http://people.csail.mit.edu/torralba/tinyimages

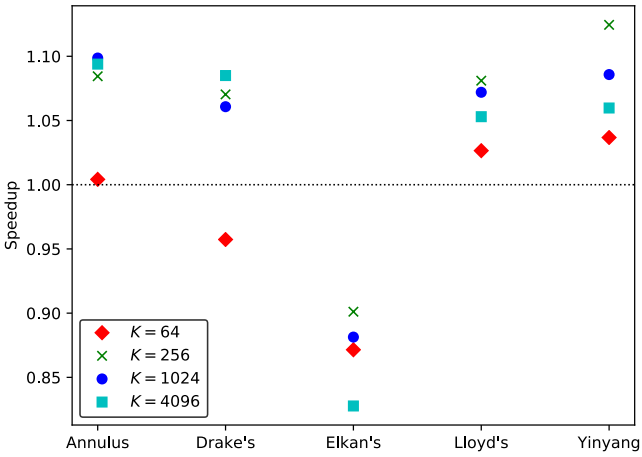


**FIGURE 11. Speedup of the versions using guided OpenMP loop scheduling over the versions using static scheduling for five algorithms and different values of $K$. The execution times for speedup computation are the median from five experiments running on 64 computing nodes (1536 cores total).**

### D. COMPARISON OF GUIDED AND STATIC LOOP SCHEDULING

We begin the presentation of our results with a comparison of two loop scheduling policies for the hybrid MPI/OpenMP parallelization. Fig. 11 shows the speedups of the versions using guided OpenMP loop scheduling over the versions using static scheduling. For each of the five algorithms and for each value of K, this speedup was computed as $t_{\text{static}}/t_{\text{guided}}$, where $t_{\text{static}}$ and $t_{\text{guided}}$ denote the median execution times of the static and guided versions running on 64 computing nodes, respectively. The horizontal dotted line indicates the two scheduling policies are of equal speed. A speedup greater than one indicates that the version using guided scheduling is faster. A speedup lower than one indicates that the version with static scheduling is faster.

The results from Fig. 11 show that most of the algorithms' gains due to guided scheduling of loop iterations, which results in a better load balance, outweigh the losses caused by the increased number of NUMA remote accesses to feature vectors and bounds. One notable exception to this rule is Elkan's algorithm, which requires the highest memory bandwidth due to the use of $K + 1$ bounds per feature vector. Interestingly, the static version of Lloyd's method, which should have the perfect load balance, performed worse than its guided counterpart. Perhaps in this case, the imbalance was caused by system noise or interference from other applications running on the computer cluster.

In further experiments reported in this section, we used the fastest scheduling for each of the algorithms, i.e., static scheduling for Elkan's method and guided scheduling for the others.

### E. EXECUTION TIMES AND STRONG SCALABILITY

In this subsection, we investigate the efficiency of the parallelization of five $K$-means algorithms. For this purpose, we ran the algorithms, increasing exponentially the number

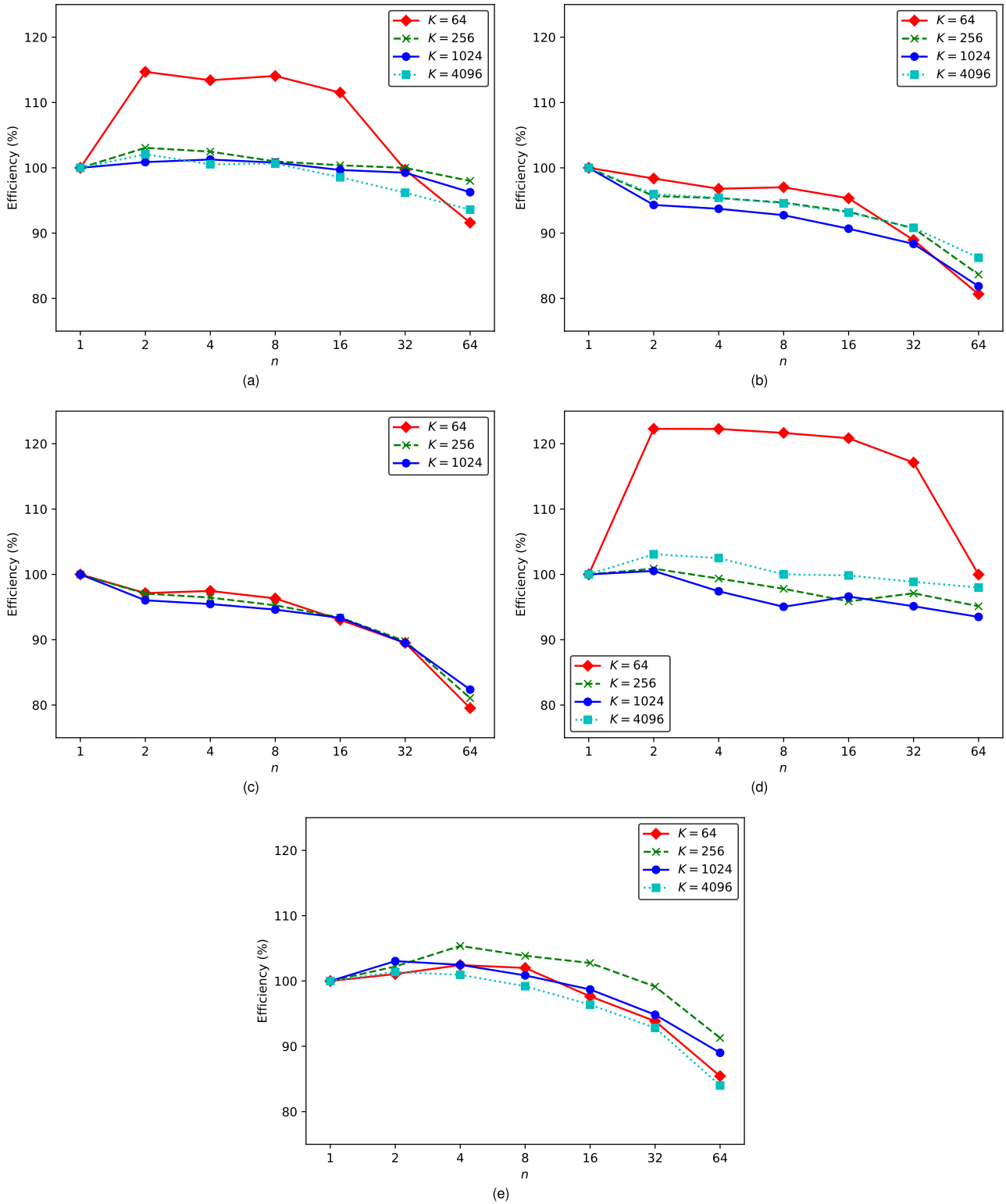


**FIGURE 12.** Parallel efficiency of (a) the Annulus algorithm, (b) Drake's algorithm, (c) Elkan's algorithm, (d) Lloyd's algorithm, (e) the Yinyang algorithm.

of allocated nodes $n$ from 1 to 64. Table 1 shows the obtained execution times.

Based on these results, we calculated strong scaling efficiency individually for each of the algorithms as follows. We denoted by $t_A(n, K)$ the execution time (median of five runs) on $n$ nodes of algorithm $A$ using $K$ clusters. The efficiency of algorithm $A$ (in percentages) was defined as the ratio of the parallel speedup (equal $t_A(1, K)/t_A(n, K)$) to the ideal

**TABLE 1.** Execution times (in seconds) for different variants of the $K$-means algorithm depending on the number of clusters $K$ and the number of nodes $n$. Each result is a median of five runs using the same initial solution. Due to the very high memory complexity of Elkan's algorithm, obtaining the results for $K = 4096$ and $n \in \{1, 2\}$ was not possible.

| $K$ | $n$ | Annulus | Drake's | Elkan's | Lloyd's | Yinyang |
|---|---|---|---|---|---|---|
| 64 | 1 | 526.70 | 387.81 | 377.11 | 1898.18 | 366.29 |
| | 2 | 229.60 | 197.12 | 194.11 | 776.07 | 181.22 |
| | 4 | 116.11 | 100.16 | 96.72 | 388.11 | 89.39 |
| | 8 | 57.72 | 49.96 | 48.94 | 195.03 | 44.89 |
| | 16 | 29.52 | 25.42 | 25.33 | 98.17 | 23.44 |
| | 32 | 16.51 | 13.63 | 13.17 | 50.65 | 12.20 |
| | 64 | 8.98 | 7.51 | 7.41 | 29.67 | 6.70 |
| 256 | 1 | 1876.01 | 594.41 | 531.62 | 5078.26 | 500.55 |
| | 2 | 910.11 | 310.63 | 273.84 | 2516.92 | 244.83 |
| | 4 | 457.66 | 155.82 | 137.80 | 1277.65 | 118.76 |
| | 8 | 232.21 | 78.48 | 69.75 | 649.07 | 60.24 |
| | 16 | 116.80 | 39.83 | 35.58 | 331.05 | 30.44 |
| | 32 | 58.63 | 20.47 | 18.50 | 163.41 | 15.78 |
| | 64 | 29.91 | 11.10 | 10.25 | 83.41 | 8.57 |
| 1024 | 1 | 12784.80 | 2712.17 | 2974.78 | 31614.60 | 1535.54 |
| | 2 | 6336.63 | 1437.83 | 1548.74 | 15722.90 | 745.07 |
| | 4 | 3156.13 | 723.39 | 779.03 | 8113.15 | 374.60 |
| | 8 | 1585.42 | 365.51 | 393.01 | 4157.83 | 190.31 |
| | 16 | 801.69 | 186.92 | 199.17 | 2045.11 | 97.23 |
| | 32 | 402.52 | 95.93 | 103.86 | 1038.58 | 50.60 |
| | 64 | 207.50 | 51.77 | 56.43 | 528.34 | 26.96 |
| 4096 | 1 | 57182.20 | 13147.40 | — | 116453.00 | 4076.61 |
| | 2 | 28009.90 | 6849.30 | — | 56482.40 | 2010.51 |
| | 4 | 14217.20 | 3444.58 | 3471.97 | 28402.00 | 1009.60 |
| | 8 | 7101.23 | 1737.91 | 1369.06 | 14555.00 | 513.65 |
| | 16 | 3626.10 | 882.14 | 699.76 | 7289.76 | 264.39 |
| | 32 | 1857.69 | 452.41 | 369.37 | 3680.88 | 137.25 |
| | 64 | 954.63 | 238.21 | 205.94 | 1856.86 | 75.82 |

linear speedup equal to $n$:

$$E_A(n, K) = \frac{t_A(1, K)}{n * t_A(n, K)} * 100\%. \tag{12}$$

An efficiency equal to 100% indicates an ideal linear speedup, whereas an efficiency greater than 100% indicates speedup on $n$ nodes greater than $n$ (super-linear speedup). In the context of data clustering, this may happen mostly because of the increased total size of the CPU cache available to the algorithms [59].

The efficiency plots are shown in Fig. 12. The plots indicate that for the moderate number of nodes $n$, some of the algorithms were able to achieve a super-linear speedup. This phenomenon manifests stronger for slower algorithms (Lloyd's and Annulus) than for the much faster Yinyang algorithm.

We recapitulate the efficiency comparison with Fig. 13, which compares the parallel efficiency of five algorithms for the highest number of nodes ($n = 64$). The comparison indicates that all the algorithms scaled well, with an efficiency higher than 80%, except in one case.

### F. ALGORITHMIC SPEEDUP OVER LLOYD'S APPROACH

In this subsection, we investigate the speedup of the four accelerated $K$-means algorithms over the standard Lloyd's method. The algorithmic speedup of algorithm $A$ over Lloyd's method is given as [16]:

$$S_A(n, K) = \frac{t_{\text{Lloyd}}(n, K)}{t_A(n, K)}. \tag{13}$$

Fig. 14 presents a comparison of this speedup for two extreme values of the number of nodes $n$. It is evident that the Yinyang algorithm is the fastest. Moreover, its advantage over Lloyd's method (and the other methods) increases with an increase in the number of clusters $K$. The comparison of Figs. 14a and 14b indicates that all four accelerated algorithms are able to maintain speedup over the Lloyd's method when the number of computing nodes $n$ is increased from 1 to 64.

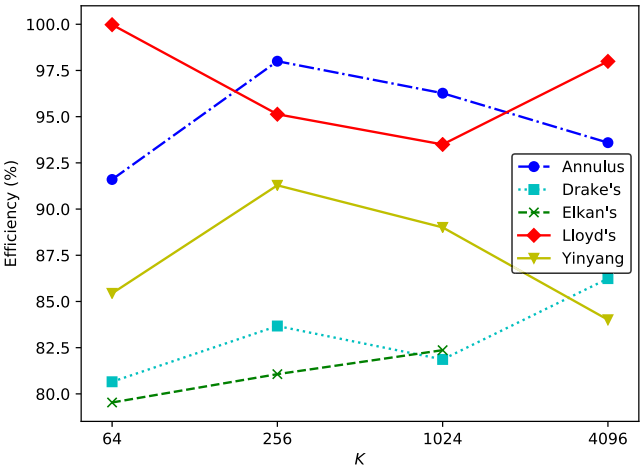


**FIGURE 13.** Comparison of the parallel efficiency of five $K$-means algorithms for the number of computing nodes $n = 64$ and different values of $K$.

### G. COMPARISON WITH THE PURE MPI VERSION

We continue the presentation of the experimental results with a comparison of the hybrid OpenMP/MPI parallelization with its pure MPI counterpart. We obtained the pure MPI version by disabling OpenMP compilation and using one MPI process per one core of a computing node, which yielded 24 MPI processes in a node and 1536 processes in 64 nodes. We ensured the CPU affinity of each MPI process by using `-bind-to thread` OpenMPI [54] mpirun script parameter. Fig. 15 shows the speedups of the hybrid OpenMP/MPI parallelization over the pure MPI parallelization. The plots indicate that for large $K$, the gains of hybrid parallelizations are very high, with the pure MPI version running several times slower. Such dramatic differences in execution times might indicate a misconfiguration of OpenMPI on the Tryton supercomputer. To verify such a possibility, and to check if the differences in the execution times were indeed caused by more efficient implementation of reduction step in the hybrid version, we repeated some of the experiments by performing single runs of Lloyd's and the Yinyang methods on a different system (64 nodes of Cray XC40 using Cray MPI and an Intel C++ compiler; each node was equipped with two 12-core Intel Xeons E5-2690 v3 2.6 GHz). The comparison

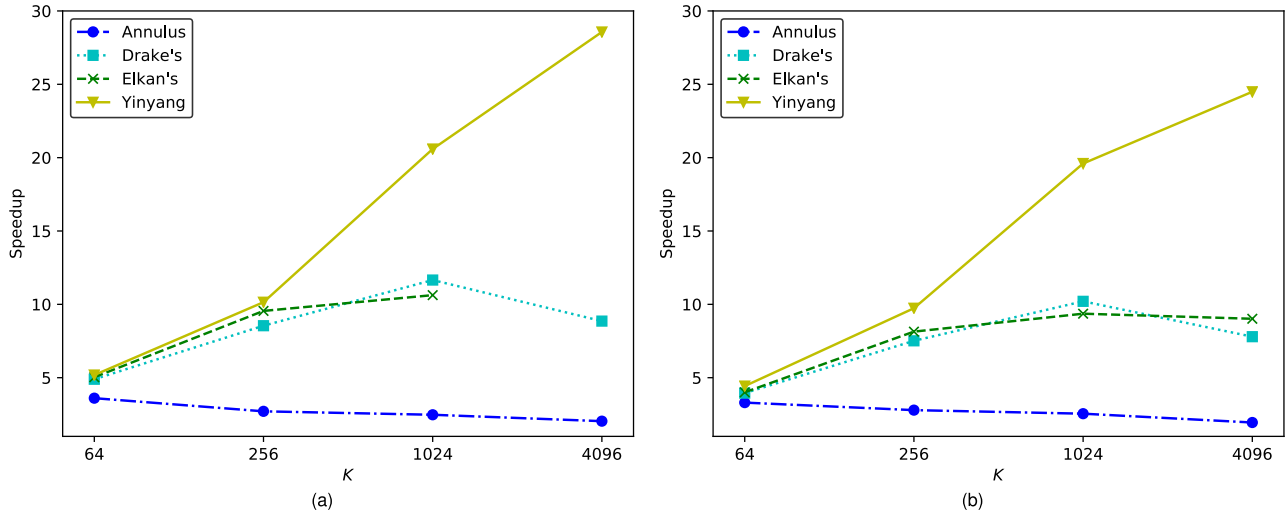


**FIGURE 14.** Algorithmic speedup of the four accelerated $K$-means algorithms over Lloyd's method for different values of $K$. (a) $n = 1$ (b) $n = 64$.

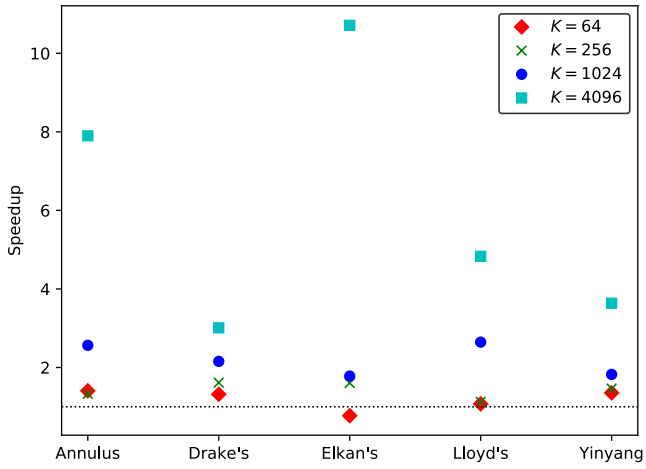


**FIGURE 15.** Speedup of the versions parallelized using the hybrid OpenMP/MPI method over the versions using pure MPI parallelization for the five algorithms and different values of $K$. The execution times for the speedup computation were the median from five experiments running on 64 computing nodes.

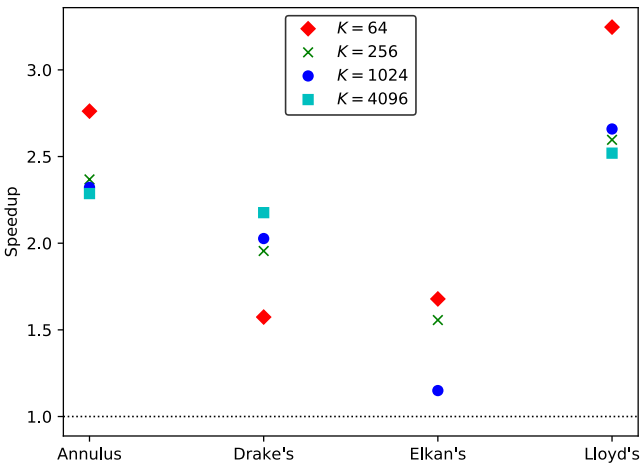


**FIGURE 16.** Algorithmic speedup of our implementation over *fast-kmeans*. Execution times for speedup computations were the medians from five experiments running on one computing node. The *fast-kmeans* package is parallelized using POSIX threads and thus cannot utilize more than one node. The Yinyang algorithm is not implemented in *fast-kmeans*. Obtaining results for Elkan's algorithm and $K = 4096$ was not possible due to excessive memory requirements.

of the hybrid OpenMP/MPI and pure MPI versions on the Cray system generated very similar results to those reported in Fig. 15. For the Yinyang algorithm, we measured speedups of the hybrid version equal to 1.50 and 3.49 for $K = 1024$ and $K = 4096$, respectively. For Lloyd's method, the speedups were equal to 2.51 and 4.87 for $K = 1024$ and $K = 4096$, respectively. Because of limited CPU time allocation, we could not extend the scope of the experiments on Cray XC40.

### H. COMPARISON WITH EXISTING SOFTWARE PACKAGES

In this subsection, we compare the performance of our implementation with two existing parallel implementations whose sources in C/C++ languages are publicly available. The first software package,[3] which we denote as *fast-kmeans*, was developed by Greg Hamerly and Jonathan Drake at Baylor University. Among the $K$-means algorithms considered in this paper, it contains implementations of Lloyd's, Drake's, Elkan's, and the Annulus methods. The algorithms were parallelized using the POSIX threads standard, allowing them to be run on a single node of a computer cluster. In [16], experiments with the package running on a system with 12 cores were reported.

The second software package,[4] which we denote as *parallel-kmeans* was developed by Wei-keng Liao at Northwestern University. It contains three versions of

[3]The source code is available at https://github.com/ghamerly/fast-kmeans

[4]The source code is available at http://www.ece.northwestern.edu/~wkliao/Kmeans/index.html

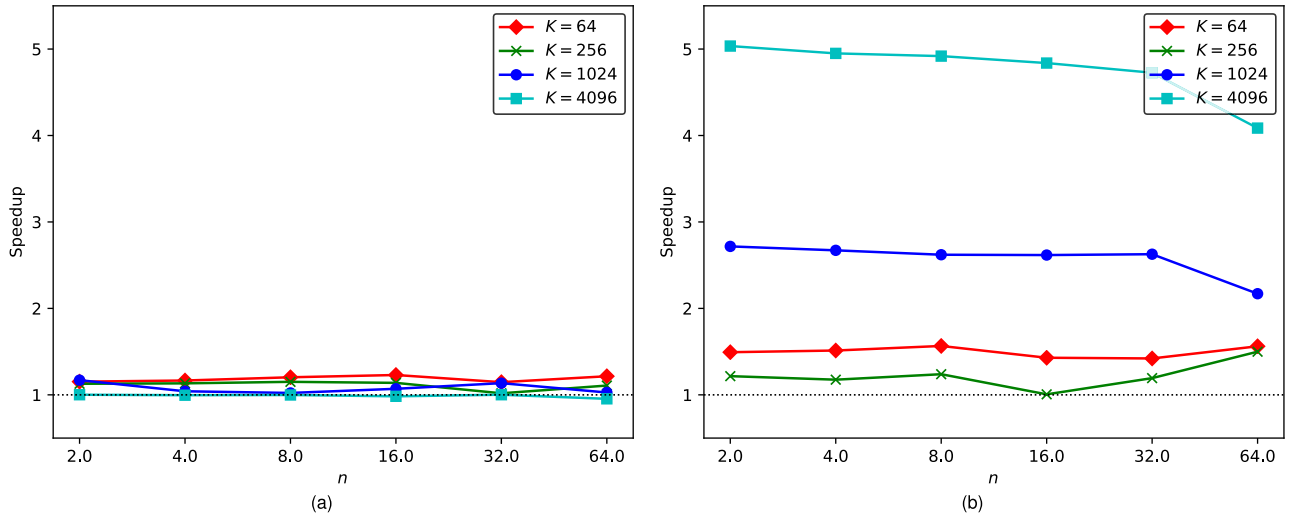


**FIGURE 17.** Algorithmic speedup of our flat MPI (a) and hybrid MPI/OpenMP (b) versions of Lloyd's algorithm over *parallel-kmeans* package depending on the number of nodes $n$. Execution times for speedup computations were the medians from five experiments.

Lloyd's algorithm: a sequential version, an OpenMP version limited to a single node, and a flat MPI version. For the comparison, we chose the MPI version, because it can be run on an arbitrary number of nodes.

To ensure a fair comparison we compiled *fast-kmeans* and *parallel-kmeans* using the same compiler, the same MPI library, and the same set of optimization options (described in section V-A) as our own code. We also tried to limit our modifications of source codes of *fast-kmeans* and *parallel-kmeans* as much as possible. However, to ensure a fair comparison, we had to use the same stopping criterion and representation of floating-point numbers for all three implementations. For these reasons, we changed the stopping criteria of *fast-kmeans* and *parallel-kmeans* to the criterion described in subsection V-B, which was used by our implementation. We also harmonized the representation of the dataset (using single precision floating numbers) and the centroids in $K$-means algorithms (using double precision floating-point numbers). After applying these changes to *fast-kmeans* and *parallel-kmeans*, we observed in our experiments that all three implementations needed the same number of iterations and found the same solution, given identical initial conditions.

Fig. 16 shows the algorithmic speedups of our implementation of $K$-means algorithms over the corresponding algorithms from the *fast-kmeans* package. The results demonstrate that our implementations run faster. There are several factors which might contribute to the higher speed of our implementations. One may be the use of the minimum spanning tree reduction, which has a lower complexity than the linear time reduction used in *fast-kmeans*. Another may be a careful tuning of our source code, including the use of OpenMP `#pragma omp simd` directive [7], which greatly aids the compiler in generating vectorized AVX2 instructions. Fig. 17 shows the algorithmic speedups of our flat MPI and hybrid MPI/OpenMP versions of Lloyd's algorithm over the *parallel-kmeans* package. Two plots using the same scale indicate that the advantage of our flat MPI implementation, which uses identical MPI communication patterns, over *parallel-kmeans*, is moderate. However, the advantage of the hybrid MPI/OpenMP version is much greater and manifests stronger for the large number of clusters $K$. We surmise that our more efficient implementation of the all-reduce step contributes greatly to this advantage.

## VI. CONCLUSIONS AND FUTURE WORK

In the study, we experimentally investigated the performance of five parallel variants of the $K$-means clustering algorithm. In the experiments, we used a sample of one of the largest publicly available datasets. The results strongly suggest that in contrast to pure MPI parallelization, our hybrid MPI/OpenMP parallelization allows all the algorithms to achieve very high efficiency. Moreover, the four accelerated algorithms, running on several dozens of multi-core nodes, maintained their advantage over Lloyd's method, as demonstrated in the previous single-node experiments [16], [17]. A comparison with two existing software packages demonstrated higher efficiency of our implementation.

In the experiments, the Yinyang algorithm was the fastest $K$-means method. However, it should be noted that in our comparison, we used one high-dimensional dataset, which is typical for the application of clustering to big-data analysis. Experiments with low-dimensional data could result in a different outcome.

Our work can be extended in several directions. Spark [10] is the de facto standard for distributed in-memory big-data processing. MLlib, its machine learning library [47], has implemented Lloyd's algorithm for $K$-means clustering. A recently proposed variant of Spark (RDMA-Spark) [60] can efficiently utilize high-performance interconnects, such as Infiniband. Currently, we are investigating the possibility of implementing the triangle inequality-accelerated algorithms in RDMA-Spark and comparing their performance

with our MPI/OpenMP implementation on the same hardware.

Another extension of our work could focus on implementation of triangle inequality-based methods on GPUs (or even on clusters of GPUs, since the dataset used in this study and the auxiliary data structures are unlikely to fit in the on-board memory of a single GPU accelerator). Previous studies [42], [43] demonstrated the superiority of Lloyd's algorithm implementation on GPUs over single-node CPU versions. It remains to be seen if the accelerated methods discussed in this paper, implemented on cluster of GPUs, can achieve a similar superiority over our multi-node multi-threaded implementation.

## ACKNOWLEDGMENTS

The calculations were carried out at the Academic Computer Centre in Gdansk, Poland.

## REFERENCES

[1] A. K. Jain, "Data clustering: 50 years beyond K-means," *Pattern Recognit. Lett.*, vol. 31, no. 8, pp. 651–666, 2010.
[2] R. Xu and D. Wunsch, II, "Survey of clustering algorithms," *IEEE Trans. Neural Netw.*, vol. 16, no. 3, pp. 645–678, May 2005.
[3] D. Aloise, A. Deshpande, P. Hansen, and P. Popat, "NP-hardness of Euclidean sum-of-squares clustering," *Mach. Learn.*, vol. 75, no. 2, pp. 245–248, 2009.
[4] S. Lloyd, "Least squares quantization in PCM," *IEEE Trans. Inf. Theory*, vol. IT-28, no. 2, pp. 129–137, Mar. 1982.
[5] J. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proc. 5th Berkeley Symp. Math. Statist. Probab.*, vol. 1. 1967, pp. 281–297.
[6] C. L. P. Chen and C.-Y. Zhang, "Data-intensive applications, challenges, techniques and technologies: A survey on big data," *Inf. Sci.*, vol. 275, pp. 314–347, Aug. 2014.
[7] OpenMP Architecture Review Board. (2015). *OpenMP Application Program Interface Version 4.5*. [Online]. Available: http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf
[8] Message Passing Interface Forum. (2015). *MPI: A Message-Passing Interface Standard Version 3.1*. [Online]. Available: http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf
[9] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.
[10] M. Zaharia *et al.*, "Apache Spark: A unified engine for big data processing," *Commun. ACM*, vol. 59, no. 11, pp. 56–65, 2016.
[11] H. Jin, D. Jespersen, P. Mehrotra, R. Biswas, L. Huang, and B. Chapman, "High performance computing using MPI and OpenMP on multi-core parallel systems," *Parallel Comput.*, vol. 37, no. 9, pp. 562–575, 2011.
[12] R. Rabenseifner, G. Hager, and G. Jost, "Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes," in *Proc. 17th Euromicro Int. Conf. Parallel, Distrib. Netw.-Based Process.*, Feb. 2009, pp. 427–436.
[13] J. Drake and G. Hamerly, "Accelerated k-means with adaptive distance bounds," in *Proc. 5th NIPS Workshop Optim. Mach. Learn.*, 2012, pp. 42–53.
[14] J. Drake, "Faster *k*-means clustering," M.S. thesis, Dept. Comput. Sci., Baylor Univ., Waco, TX, USA, 2013.
[15] C. Elkan, "Using the triangle inequality to accelerate k-means," in *Proc. 20th Int. Conf. Mach. Learn. (ICML)*, vol. 3. Menlo Park, CA, USA: AAAI, 2003, pp. 147–153.
[16] G. Hamerly and J. Drake, "Accelerating Lloyd's algorithm for k-means clustering," in *Partitional Clustering Algorithms*. Cham, Switzerland: Springer, 2015, pp. 41–78.
[17] Y. Ding, Y. Zhao, X. Shen, M. Musuvathi, and T. Mytkowicz, "Yinyang k-means: A drop-in replacement of the classic k-means with consistent speedup," in *Proc. 32nd Int. Conf. Mach. Learn. (ICML)*, 2015, pp. 579–587.
[18] X. Wu *et al.*, "Top 10 algorithms in data mining," *Knowl. Inf. Syst.*, vol. 14, no. 1, pp. 1–37, 2008.
[19] E. W. Forgy, "Cluster analysis of multivariate data: Efficiency versus interpretability of classifications," *Biometrics*, vol. 21, no. 3, pp. 768–769, 1965.
[20] D. Arthur and S. Vassilvitskii, "K-means++: The advantages of careful seeding," in *Proc. 18th Annu. ACM-SIAM Symp. Discrete Algorithms (SODA)*, 2007, pp. 1027–1035.
[21] B. Bahmani, B. Moseley, A. Vattani, R. Kumar, and S. Vassilvitskii, "Scalable k-means++," *Proc. VLDB Endowment*, vol. 5, no. 7, pp. 622–633, 2012.
[22] M. E. Celebi, H. A. Kingravi, and P. A. Vela, "A comparative study of efficient initialization methods for the k-means clustering algorithm," *Expert Syst. Appl.*, vol. 40, no. 1, pp. 200–210, 2013.
[23] J. Philbin, O. Chum, M. Isard, J. Sivic, and A. Zisserman, "Object retrieval with large vocabularies and fast spatial matching," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2007, pp. 1–8.
[24] J. Wang *et al.*, "Trinary-projection trees for approximate nearest neighbor search," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 36, no. 2, pp. 388–403, Feb. 2014.
[25] J. Wang, J. Wang, Q. Ke, G. Zeng, and S. Li, "Fast approximate k-means via cluster closures," in *Multimedia Data Mining Analytics*. Cham, Switzerland: Springer, 2015, pp. 373–395.
[26] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu, "An efficient K-means clustering algorithm: Analysis and implementation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 24, no. 7, pp. 881–892, Jul. 2002.
[27] D. Pelleg and A. Moore, "Accelerating exact k-means algorithms with geometric reasoning," in *Proc. 5th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 1999, pp. 277–281.
[28] T. Kaukoranta, P. Franti, and O. Nevalainen, "A fast exact GLA based on code vector activity detection," *IEEE Trans. Image Process.*, vol. 9, no. 8, pp. 1337–1342, Aug. 2000.
[29] G. Hamerly, "Making k-means even faster," in *Proc. SIAM Int. Conf. Data Mining*, 2010, pp. 130–140.
[30] T. Bottesch, T. Bühler, and M. Käechele, "Speeding up k-means by approximating Euclidean distances via block vectors," in *Proc. 33rd Int. Conf. Mach. Learn.*, 2016, pp. 2578–2586.
[31] I. S. Dhillon and D. S. Modha, "A data-clustering algorithm on distributed memory multiprocessors," in *Large-Scale Parallel Data Mining*. Berlin, Germany: Springer, 2002, pp. 245–260.
[32] L. M. Rodrigues, L. E. Zárate, C. N. Nobre, and H. C. Freitas, "Parallel and distributed kmeans to identify the translation initiation site of proteins," in *Proc. IEEE Int. Conf. Syst., Man, Cybern. (SMC)*, Oct. 2012, pp. 1639–1645.
[33] M. A. Bender, J. Berry, S. D. Hammond, B. Moore, B. Moseley, and C. A. Phillips, "k-means clustering on two-level memory systems," in *Proc. Int. Symp. Memory Syst.* New York, NY, USA: ACM, 2015, pp. 197–205.
[34] C. Böhm, M. Perdacher, and C. Plant, "Multi-core K-means," in *Proc. SIAM Int. Conf. Data Mining*, 2017, pp. 273–281.
[35] L. Li *et al.*, "Large-scale hierarchical k-means for heterogeneous many-core supercomputers," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage, Anal.*, 2018, pp. 160–170.
[36] M. Li, C. Yang, Q. Sun, W.-J. Ma, W.-J. Ma, W.-L. Cao, and Y.-L. Ao, "Enabling highly efficient k-means computations on the SW26010 many-core processor of Sunway TaihuLight," *J. Comput. Sci. Technol.*, vol. 34, no. 1, pp. 77–93, 2019.
[37] E. Strohmaier, J. Dongarra, H. Simon, and M. Meuer. *Top500 Supercomputer Sites*. 52nd ed. Accessed: Feb. 20, 2019. [Online]. Available: https://www.top500.org/list/2018/11/
[38] D. Pettinger and G. Di Fatta, "Scalability of efficient parallel K-means," in *Proc. 5th IEEE Int. Conf. E-Sci. Workshops*, Dec. 2009, pp. 96–101.
[39] G. Di Fatta and D. Pettinger, "Dynamic load balancing in parallel KD-tree K-means," in *Proc. 10th IEEE Int. Conf. Comput. Inf. Technol.*, Jun./Jul. 2010, pp. 2478–2485.
[40] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. Hellerstein, "Graphlab: A new framework for parallel machine learning," in *Proc. 26th Conf. Annu. Conf. Uncertainty Artif. Intell. (UAI)*. Corvallis, OR, USA: AUAI Press, 2010, pp. 340–349.
[41] J. Newling and F. Fleuret, "Fast k-means with accurate bounds," in *Proc. 33rd Int. Conf. Mach. Learn.*, 2016, pp. 936–944.
[42] K. J. Kohlhoff, V. S. Pande, and R. B. Altman, "K-means for parallel architectures using all-prefix-sum sorting and updating steps," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 8, pp. 1602–1612, Aug. 2013.

[43] Y. Li, K. Zhao, X. Chu, and J. Liu, "Speeding up k-means algorithm by GPUs," *J. Comput. Syst. Sci.*, vol. 79, no. 2, pp. 216–229, 2013.
[44] Z. Lin, C. Lo, and P. Chow, "K-means implementation on FPGA for high-dimensional data using triangle inequality," in *Proc. 22nd Int. Conf. Field Program. Logic Appl. (FPL)*, Aug. 2012, pp. 437–442.
[45] F. Winterstein, S. Bayliss, and G. A. Constantinides, "FPGA-based k-means clustering using tree-based data structures," in *Proc. 23rd Int. Conf. Field Program. Logic Appl. (FPL)*, Sep. 2013, pp. 1–6.
[46] R. R. Curtin *et al.*, "MLPACK: A scalable C++ machine learning library," *J. Mach. Learn. Res.*, vol. 14, pp. 801–805, Mar. 2013.
[47] X. Meng *et al.*, "MLlib: Machine learning in Apache Spark," *J. Mach. Learn. Res.*, vol. 17, no. 1, pp. 1235–1241, 2016.
[48] J. L. Reyes-Ortiz, L. Oneto, and D. Anguita, "Big data analytics in the cloud: Spark on Hadoop vs MPI/OpenMP on Beowulf," *Procedia Comput. Sci.*, vol. 53, pp. 121–130, 2015.
[49] S. Z. Selim and M. A. Ismail, "K-means-type algorithms: A generalized convergence theorem and characterization of local optimality," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. PAMI-6, no. 1, pp. 81–87, Jan. 1984.
[50] S. Phillips, "Acceleration of *k*-means and related clustering algorithms," in *Proc. Algorithm Eng. Exp., 4th Int. Workshop (ALENEX)* (Lecture Notes in Computer Science), vol. 2409, 2002, pp. 166–177.
[51] W. Kwedlo, "Two modifications of Yinyang K-means algorithm," in *Proc. Int. Conf. Artif. Intell. Soft Comput.* (Lecture Notes in Computer Science), vol. 10246. Cham, Switzerland: Springer, 2017, pp. 94–103.
[52] E. Chan, M. Heimlich, A. Purkayastha, and R. van de Geijn, "Collective communication: Theory, practice, and experience," *Concurrency Comput., Pract. Exper.*, vol. 19, no. 13, pp. 1749–1783, 2007.
[53] Intel Corporation. (2016). *Intel C++ Compiler 17.0 Developer Guide and Reference*. [Online]. Available: https://software.intel.com/en-us/intel-cplusplus-compiler-17.0-user-and-reference-guide
[54] E. Gabriel *et al.*, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proc. 11th Eur. Parallel Virtual Mach./Message Passing Interface Users'Group Meeting* (Lecture Notes in Computer Science), vol. 3241. Berlin, Germany: Springer, 2004, pp. 97–104.
[55] T. Hoefler and R. Belli, "Scientific benchmarking of parallel computing systems: Twelve ways to tell the masses when reporting performance results," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.* New York, NY, USA: ACM, 2015, pp. 1–12.
[56] A. Bhatele, K. Mohror, S. H. Langer, and K. E. Isaacs, "There goes the neighborhood: Performance degradation due to nearby jobs," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.* New York, NY, USA: ACM, 2013, p. 41.
[57] A. Torralba, R. Fergus, and W. T. Freeman, "80 million tiny images: A large data set for nonparametric object and scene recognition," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 30, no. 11, pp. 1958–1970, Nov. 2008.
[58] A. Oliva and A. Torralba, "Modeling the shape of the scene: A holistic representation of the spatial envelope," *Int. J. Comput. Vis.*, vol. 42, no. 3, pp. 145–175, 2001.
[59] D. P. Helmbold and C. E. McDowell, "Modelling speedup (n) greater than n," *IEEE Trans. Parallel Distrib. Syst.*, vol. 1, no. 2, pp. 250–256, Apr. 1990.
[60] X. X. Lu, M. W. Ur Rahman, N. Islam, D. Shankar, and D. K. Panda, "Accelerating Spark with RDMA for big data processing: Early experiences," in *Proc. IEEE 22nd Annu. Symp. High-Perform. Interconnects (HOTI)*, Aug. 2014, pp. 9–16.

**WOJCIECH KWEDLO** (M'11) was born in Bialystok, Poland, in 1972. He received the M.S. and Ph.D. degrees in computer science engineering from the Bialystok University of Technology, Poland, in 1996 and 2004, respectively.

From 1996 to 2004, he was a Teaching and Research Assistant with the Faculty of Computer Science, Bialystok University of Technology, where he has been an Assistant Professor with the Faculty of Computer Science, since 2004. His research interests include model-based and *K*-means clustering algorithms, hybridization of these algorithms with evolutionary methods, and parallel algorithms.

Dr. Kwedlo was a recipient of the DAAD Scholarship for Young Researchers for five months-long research stay at the University of Karlsruhe, Karlsruhe, Germany, in 1999.

**PAWEL J. CZOCHANSKI** was born in Bialystok, Poland, in 1991. He received the M.S. degree in computer science engineering from the Bialystok University of Technology, Poland, in 2015, where he was a Teaching and Research Assistant with the Faculty of Computer Science, from 2015 to 2017.

His research interests include machine learning algorithms and parallel processing.