Designing a Synchronization-reducing Clustering Method on Manycores: Some Issues and Improvements

Weijian Zheng
Department of Computer Science
Indiana University-Purdue University
Indianapolis, Indiana
wz26@iupui.edu

Fengguang Song
Department of Computer Science
Indiana University-Purdue University
Indianapolis, Indiana
fgsong@iupui.edu

Lan Lin
Department of Computer Science
Ball State University
Muncie, Indiana
llin4@bsu.edu

ABSTRACT

The k-means clustering method is one of the most widely used techniques in big data analytics. In this paper, we explore the ideas of software blocking, asynchronous local optimizations, and heuristics of simulated annealing to improve the performance of k-means clustering. Like most of the machine learning methods, the performance of k-means clustering relies on two main factors: the computing speed (per iteration), and the convergence rate. A straightforward realization of the software-blocking synchronization-reducing clustering algorithm, however, sees sporadic slower convergence rate than the standard k-means algorithm. To tackle the issues, we design an annealing-enhanced algorithm, which introduces the heuristics of stop conditions and annealing steps to provide as good or better performance than the standard k-means algorithm. This new enhanced k-means clustering algorithm is able to offer the same clustering quality as the standard k-means. Experiments with real-world datasets show that the new parallel implementation is faster than the open source HPC library of Parallel K-Means Data Clustering (e.g., 19% faster on relatively large datasets with 32 CPU cores, and 11% faster on a large dataset with 1,024 CPU cores). Moreover, the extent to which the program performance improves is largely determined by the actual convergence rate of applying the algorithm to different datasets.

CCS CONCEPTS

• **Computing methodologies** → **Massively parallel algorithms**; **Machine learning**; • **Computer systems organization** → **Parallel architectures**;

KEYWORDS

High performance computing, machine learning, synchronization-reducing clustering algorithms, simulated annealing, manycores

ACM Reference Format:
Weijian Zheng, Fengguang Song, and Lan Lin. 2017. Designing a Synchronization-reducing Clustering Method on Manycores: Some Issues and Improvements. In *Proceedings of Machine Learning in High Performance Computing Environments Workshop at SC'17 (MLHPC'17).* ACM, New York, NY, USA, Article 2, 8 pages. https://doi.org/10.475/123_4

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
MLHPC'17, November 2017, Denver, Colorado, USA
© 2017 Copyright held by the owner/author(s).
ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.1145/3146347.3146357

1 INTRODUCTION

Machine learning has gained a lot of attentions and achieved momentous results in both industry and academia. A main driving force behind this success is the wider deployment of high performance computing (HPC) systems and the high demands of big data analytics. Given the exponential growth rate of data generation at extreme scales, it is critical to summarize big data succinctly and approximately to discover new knowledge.

One of the most important computational data summarization methods is *clustering* [11]. To date, clustering has been used in many scientific, engineering, and industrial applications. In a clustering method, each data item is viewed as a point in a multidimensional space. Data points that are "close" in this space will be assigned to the same cluster. Clustering methods are divided into two categories: *hierarchical* and *partitional* algorithms. Hierarchical clustering algorithms generate a dendrogram to represent the nested grouping of data points in a hierarchy. Differently, partitional clustering algorithms generate a single partition for all data points. Since generating a dendrogram is time-consuming for large-scale data volumes, partitional clustering is often used in practice. In this work, we target designing a new partitional clustering algorithm, named *a tiled synchronization-reducing k-means clustering algorithm*. It builds on the widely used *software blocking* technique [13] and the successes of applying synchronization-reducing algorithms to classic computation-intensive HPC applications [2, 3, 15, 22].

The standard k-means clustering algorithm [9] works as follows: 1) Randomly choose k points among all the data points as the initial centers; 2) assign each data point to its closest center; 3) update the k centers according to the changed datapoint membership; and 4) check if the convergence condition is met, if not, repeat 2) and 3). In this work, we use the standard *Sum of Squared Error* (*SSE*) metric to test clustering algorithms's exit condition and measure the quality of the clustering result.

There are several issues to design a scalable clustering algorithm for massively parallel systems. *First*, the global synchronizations between all points at every iteration will affect an algorithm's scalability when considering extreme-scale systems with a large number of CPUs and cloud systems with relatively slow communication networks. This inspires us to design a synchronization-reducing algorithm to decrease global synchronizations. *Second*, visiting all the data points at every iteration will result in a poor data locality and more data movements between slow memory and fast caches. Hence, we use *software blocking* to improve the program's data locality. These two modifications have led to our new tiled synchronization-reducing clustering algorithm. However, this algorithm does not work directly since its convergence rate sometimes

becomes slower than that of the standard k-means clustering. *Third*, new heuristics are hence needed to make the algorithm converge in the same or better rate than the standard k-means meanwhile computing the same SSE cost.

We first design a *sequential* version of the tiled synchronization-reducing algorithm. The sequential algorithm divides the input into a number of data blocks (also known as *tiles*). Each block will be processed by a local optimization algorithm repeatedly until the specific block finds a local optimal clustering result. After each block has independently found its own best clustering, their local centers will be reduced (or merged) to form the global centers. In addition, we extend the algorithm with a heuristic to attain a better convergence rate. The heuristic is motivated by the idea of *simulated annealing* but does not use simulated annealing's expensive neighborhood searches. This new heuristic enables a so-called "annealing-enhanced tiled synchronization-reducing algorithm", which can be faster than the k-means clustering algorithm. The annealing-enhanced algorithm is essentially an adaptive method. If the number of annealing steps is large, the algorithm becomes the k-means clustering algorithm; if the number of annealing steps equals one, it becomes the tiled synchronization-reducing algorithm.

Next, we develop a parallel implementation for the annealing-enhanced algorithm. The parallel implementation uses a static data distribution method to allocate an equal number of data blocks to each thread. The parallel algorithm is designed as follows: 1) Every thread computes clustering for each assigned block; 2) every thread derives k centers based on its assigned blocks' clustering results; 3) all threads on the same compute node will merge their centers (i.e., node-level); and 4) all compute nodes will merge their centers (i.e., system-level), which eventually lead to a set of global centers. These four steps will be repeated until the global SSE cost cannot be decreased any more. Our paper also discusses different strategies to merge local centers to global centers and their effectiveness.

We conducted experiment with four real-world datasets of MNIST, CIFAR-10, CIFAR-100, and PLACES-2 for both the sequential algorithm and the parallel algorithm on manycore systems. With the parallel implementation, we can outperform the MPI-based k-means library by up to 19% on 32 CPU cores, and 11% on 1,024 CPU cores.

The rest of the paper is organized as follows. Next section introduces the related work. Section 3 presents the tiled synchronization-reducing algorithm as a sequential algorithm first. Based on the tiled synchronization-reducing algorithm, an annealing-enhanced algorithm is constructed in Section 4. Section 5 and Section 6 describes how we realize a parallel implementation on distributed-memory multicore systems. Section 7 discusses the effects of the algorithm's parameters and shows the experimental results. Finally, Section 8 concludes this work.

2 RELATED WORK

Zhao et al. used the MapReduce programming model to develop a parallel k-means program [26], but did not achieve optimal performance since the MapReduce model uses expensive disk I/O to transfer data. Spark [23] provides a machine learning library called MLlib [18]. However, Spark's runtime overhead (e.g., task start delay, scheduler delay, inter-stage barrier, and so on) is still significant compared to the MPI-based libraries, as observed by researchers from the University of California at Berkeley [8].

On the other hand, there are several HPC-based k-means clustering libraries [5, 16]. They use collective MPI operations at every iteration to update global centers. Moreover, k-means clustering methods have also been implemented on Intel MIC (Many Integrated Core) Architecture [24] and Nvidia GPUs [6, 25].

The ideas of synchronization-reducing and lazy synchronizations have already been applied to a few machine learning algorithms. For instance, Xing et al. applied the asynchronous communication approach to a variety of machine learning problems such as topic models and low-rank matrix factorizations in the context of Parameter Server Frameworks [4, 10]. However, k-means clustering is not studied in their work.

Fatta et al. designed a fault tolerant epidemic clustering algorithm which does not require global communication [7]. However, their research is focused on handling network failures and does not provide the same SSE cost as the standard k-means clustering algorithm.

Simulated annealing algorithms have been developed [17, 19, 21] to further improve the clustering methods' quality. They generate and compare random solutions near the neighborhood to search for better solutions, but take much longer time than the k-means clustering method. Differently, we extend the k-means clustering method to achieve the same SSE cost and faster performance.

Mini-batch k-means [20] is a sampling-based k-means clustering algorithm. At each iteration, it picks only b sample points to *approximate* centers by using projected gradient descent. K-means++ [1] is a method designed to help find a better seed, which is used as a preprocessing step to improve the accuracy and convergence rate of the standard k-means clustering method.

3 DESIGN OF A SYNCHRONIZATION-REDUCING CLUSTERING ALGORITHM

The new algorithm we design uses a tiled data layout and consists of two functions: 1) the local optimization function to cluster each data block, and 2) the merge function to calculate global centers.

A tiled data layout: In a machine learning application, each data point is composed of n-dimensional attributes such that all data points may be viewed as a matrix. By using a tiled data layout, we divide all data points into rectangular tiles. Each tile stores a consecutive number of data points (i.e., a number of rows in the matrix). Also, each tile is regarded as an individual unit to which a *local optimization function* can be applied as many times as possible to optimize cache locality and communication cost.

3.1 The Local Optimization Function

The idea of local optimization is to optimistically consider the points in one data block being representative of the dataset such that the locally clustered result can approximate the global centers. The local optimization function guarantees that the SSE cost keeps decreasing. The function exits when it cannot reduce the SSE cost any longer. Hence, a better solution is conveniently obtained without any overhead of communication and synchronization. However,

Algorithm 1 The Local Optimization Function

```
1:  /* Local optimization on one block */
2:  local_optimization_block(points, g_centers,
3:   g_centers_size, membership, threshold)
4:  ▷ step 1: Use the global centers as a new seed
5:  g_centers′ ← g_centers;
6:  g_centers_size′ ← g_centers_size;
7:  for each center c do
8:      g_centers_sum′[c] ← g_centers[c] * g_centers_size[c];
9:  end for
10: blk_cost_new = MAXIMUM;
11: while true do
12:     blk_local_sum = 0;
13:     blk_local_size = 0;
14:     blk_cost_old = blk_cost_new;
15:     blk_cost_new = 0;
16:     ▷ step 2: Use the local points to improve centers
17:     for each point i in points do
18:         ▷ step 2.1: find the closest center for each point
19:         (dist, new_center) ← find_nearest_center
20:         (points[i], g_centers′);
21:         pre_center = membership[i];
22:         ▷ step 2.2: update affected global centers
23:         if (pre_center != new_center) then
24:             g_centers_size′[new_center] += 1;
25:             g_centers_size′[pre_center] -= 1;
26:             g_centers_sum′[new_center] += points[i];
27:             g_centers_sum′[pre_center] -= points[i];
28:         end if
29:         membership[i] = new_center;
30:         ▷ step 2.3: each block has a partial sum for each center
31:         blk_local_sum[new_center] += points[i];
32:         blk_local_size[new_center]+=1;
33:         blk_cost_new += dist;
34:     end for
35:     ▷ step 2.4: Recalculate new centers g_center′ */
36:     for each center c do
37:         g_centers′[c] = g_centers_sum′[c] / g_centers_size′[c];
38:     end for
39:     ▷ step 3: Check if the optimization should stop
40:     if blk_cost_new >= blk_cost_old * threshold then
41:         break;
42:     end if
43: end while
44: return {blk_local_sum, blk_local_size, blk_cost_new}
```

Algorithm 2 The Center-Merge Function

```
merge_block(blk_local_sum, blk_local_size, blk_local_cost)
/* Update global cost */
g_cost += blk_local_cost;
/* Update each global center and its size */
for each center c do
    g_centers_sum[c] += blk_local_sum[c];
    g_centers_size[c] += blk_local_size[c];
end for
return g_centers_sum, g_centers_size  g_cost
```

it is not trivial to obtain good global centers from a set of local centers. Sections 3.2 and 6 will discuss how to merge centers.

In Algorithm 1, we display the local optimization function in three steps:

Step 1: Reset seed to the most updated global centers. Whenever entering the function, we utilize the current global centers as a starting point to search for newer and better centers.

Step 2: Improve global centers based on local points. The second step will do the reassignment for each data point in a data block. In *step 2.1*, a point will find its closest global center. In *step 2.2*, If the center has changed, the function will update two affected global centers' sizes and sums of coordinates. *Step 2.3* monitors the number of local points for each global center. In *Step 2.4*, the global centers will be calculated based on the local points' reassignment.

Step 3: Check the stop condition. At the end of every iteration (i.e., line 39), a new SSE cost will be calculated. If $\frac{new\_cost}{old\_cost}$ is less than a small value, we determine that local optimization has reduced the cost successfully, and will continue the local optimization step. If the ratio is close to 1, we consider the improvement is too small to be worth further local optimizations. Here we use a *threshold* to control when to exit the local optimization function.

3.2 Merging Centers

After every data block finds its own version of the best centers, the next step is to derive a unique version of the best global centers. We call it *the center-merge step*. The goal of the merge function is to put together all the partial SSE costs, the k centers' partial coordinate sums, and the sizes from each data block to determine the overall global SSE cost and global centers. Algorithm 2 shows the specific function to merge one data block. After merging all the data blocks, we will obtain the new global centers.

3.3 Main Body of the Algorithm

The previous functions of *local_optimization_block* and *merge_block* of centers are called by the main function of the tiled synchronization-reducing algorithm, as shown in Algorithm 3. After choosing an initial set of k clustering centers, the algorithm enters an iterative process (lines 10–30), which executes the following steps:

(1) Apply local optimization to each block. It adjusts the global centers based on its own data points. When the local optimization is finished, the function will return a snapshot of the local optimization. The returned information includes the partial SSE cost, partial coordinate sum, and each cluster's size that are calculated solely based on the block's local points.
(2) Next, the block-local information gathered from the previous step is reduced to determine how many points are in each global center and the coordinate sum for all the points in each center globally.
(3) Finally, the global center information from the second step is used to calculate a set of new global centers. If the new global centers have a better SSE than the previous global centers, the iterative process will continue.

Algorithm 3 Tiled Synchronization-reducing Clustering Alg.

```
1:  /* m : number of points , n_b: number of blocks */
2:  /* k centers */
3:  tiled_sync_reducing_clustering(points, m, n_b, k, threshold)
4:  /* set the initial k centers */
5:  for each center c ← 0 to k-1 do
6:      g_centers[c] = points[c]
7:  end for
8:  Decide each cluster's initial size in g_size_new.
9:  g_cost_new = MAXIMUM
10: repeat
11:     /* back up the previous iteration's centers information */
12:     g_size_old = g_size_new;
13:     g_cost_old = g_cost_new;
14:     g_size_new = 0;
15:     g_cost_new = 0;
16:     g_sum_new = 0;
17:     for each block i ← 0 to n_b - 1 do
18:         ▷ step 1: Run local optimization on each block
19:         (local_sum, local_size, local_cost) ←
20:         local_optimization_block(i-th block, g_centers,
21:         g_size_old, membership, threshold);
22:         ▷ step 2: Merge each block's coordinate sums into global
23:         (g_sum_new, g_size_new, g_cost_new) ←
24:         merge_block(local_sum, local_size, local_cost);
25:     end for
26:     ▷ step 3: Calculate the global centers
27:     for each center c do
28:         g_centers[c] = g_sum_new[c] / g_size_new[c];
29:     end for
30: until g_cost_new >= g_cost_old
```

3.4 Observation on the New Algorithm

The main differences between the new synchronization-reducing clustering algorithm and the k-means clustering algorithm are twofold: 1) The new algorithm is more relaxed since each data block can execute a number of iterations to optimize the global centers autonomously without synchronizing with any other data blocks; 2) The new algorithm has better data locality as it can apply as many computations as possible to the same data points.

We apply the new algorithm to a variety of datasets, and find that the tiled synchronization-reducing algorithm sometimes converges faster but sometimes converges slower than the strictly synchronous k-means algorithm. As detailed in the next section, the reason is that each block works on its own local data independently. For instance, in an extreme case, if all the blocks have entirely different centers, it might require more iterations to converge to the globally optimal centers. To this end, we design an enhanced synchronization-reducing algorithm (based on new heuristics) that is able to converge as same or faster than the k-means algorithm (see details in the next section).

4 AN ANNEALING-ENHANCED ALGORITHM

We take advantage of the synchronous standard k-means algorithm's faster convergence rate and the synchronization-reducing algorithm's low communication overhead to design a new algorithm called *annealing-enhanced tiled synchronization-reducing* algorithm. The annealing-enhanced algorithm is inspired by an idea similar to

Algorithm 4 Annealing-Enhanced Tiled Synchronization-reducing Algorithm

```
1:  /* Annealing-enhanced algorithm */
2:  g_cost_new = MAXIMUM;
3:  for each center c ← 0 to k-1 do
4:      g_centers[c] = points[c]
5:  end for
6:  repeat
7:      g_old_size = g_new_size;
8:      g_new_size = 0;
9:      g_new_sum = 0;
10:     g_cost_old = g_cost_new;
11:     g_cost_new = 0;
12:     for each block i ← 0 to n_b-1 do
            if loop <= annealing_step
            /* annealing step */
13:         (blk_local_sum, blk_local_size, blk_local_cost) ←
            blocked_sync_kmeans(points, g_centers, member-
            ship)
14:         /* do local optimization */
15:         else
16:           (blk_local_sum, blk_local_size, blk_local_cost) ←
17:           local_optimization_block(points, g_centers,
18:           g_old_size, membership, threshold);
19:         end if
20:         /* merge this block */
21:         (g_sum_new, g_size_new, g_cost_new) ←
22:         merge_block(blk_local_sum, blk_local_size,
23:         blk_local_cost);
24:     end for
25:     /* update global center */
26:     for each center c do
27:         g_centers[c] = g_new_sum[c] / g_new_size[c];
28:     end for
29: until g_cost_new >= g_cost_old
```

simulated annealing. By intuition, the problem we want to solve is analogous to the game of getting a ping-pong ball into the lowest crevice in a bumpy surface. If we let the ball roll, it will stop at a local minimum. But if we shake the surface harder, it can pop out of the local minimum that is stuck somewhere in the middle. Therefore, we want to shake the surface violently at the beginning so that the ball will have the opportunities to try many different rolling directions. After the ball almost settles down, we will shake the surface gently or perhaps even let the ball roll by itself (i.e., no shaking) to reach the lowest crevice.

Driven by the intuition, we employ *frequent synchronizations* to simulate "hard shaking", and employ *local optimizations* to simulate "ball rolling on its own". We also let shaking occur in the first stage and rolling occur in the second stage in our algorithm design.

4.1 Design of the Algorithm

The new annealing-enhanced algorithm is shown in Algorithm 4. It calls two types of functions: 1) synchronous k-means (i.e., blocked_sync_kmeans), which simply applies the k-means algorithm to each data block for *only one iteration*; and 2) the local optimization function as described in Algorithm 1.

The beginning of Algorithm 4 initializes a set of global centers and then enters a loop in line 6. The algorithm is almost the same as the tiled synchronization-reducing algorithm (Algorithm 3) except for line 13. Line 13 shows that if the global iteration number is smaller than the number of annealing steps, the synchronous algorithm will be called to simulate the annealing steps, which is shown in the rectangular box. If the global iteration number is larger than the number of annealing steps (line 16-18), the algorithm executes the same operations as the first tiled synchronization-reducing algorithm which performs local optimizations.

The number of annealing steps plays an important role here. If the number of annealing steps is too large, the algorithm is essentially the same as the k-means clustering algorithm. On the other hand, if the number of annealing steps is equal to one, the algorithm turns into the previous tiled synchronization-reducing clustering algorithm. In the next subsection, we will introduce how to determine the number of annealing steps dynamically.

4.2 How to Determine the Number of Annealing Steps

We use a *progression-state* based heuristic to decide when to stop the synchronization-intensive annealing steps at runtime. The progress of the clustering algorithm can be measured by the following metrics:

(1) *Cost improvement*: We measure the percentage by which the SSE cost has been reduced from the previous annealing step to the current step. If the improvement becomes negligible, we regard that the global solution has settled down and the local optimization should start to refine the centers without synchronization.
(2) *Number of point reassignments*: We measure the number of data points that switch from one cluster to another. It is based on the observation that the entire process of searching becomes more and more stable as a general trend. There are two variants for measuring this metric: Using an absolute number or using a percentage number.

We perform different experiments with all the above metrics, and find that the most effective metric is to use the absolute number of *point reassignments* to detect when to stop the annealing steps.

4.3 The Design of a "Valley-Searching" Heuristic Scheme

We use a *historical histogram* to keep track of the total number of reassigned points in each annealing step. The histogram is able to show to which extent the state changes in each time step. The heuristic we design will search for the "valley" region to stop the annealing since we consider the valley represents a settled-down state. The heuristic function runs like a parking car, and goes through 3 steps as follows.

- *Throttle down:* When the number of points that have been reassigned is large, the annealing step is considered effective and will continue running. Otherwise, the algorithm will start to prepare for stopping the annealing steps. The next stage is to seek an exact place to stop.
- *Find an exact stop place:* A "good" stop place should satisfy the following condition: The valley (or a window) has a width of 10, and all points in the valley have a derivate less than 10. In other words, there exist ten consecutive annealing steps with similar numbers of reassigned points.
- *Halt:* Once the stop place (i.e., a valid valley) is found, the annealing steps will stop in that iteration.

According to our experiments, using the above heuristic method can obtain a similar or faster performance compared to the k-means clustering algorithm without any loss of quality (detailed results are presented in Section 7).

5 PARALLEL IMPLEMENTATION OF THE ALGORITHM

We have implemented the annealing-enhanced Algorithm 4 on distributed-memory multicore HPC systems. This section introduces the parallel implementation, which uses a hybrid MPI/Pthread computing model.

The parallel program is listed in Algorithm 5 and executes a number of multithreaded MPI processes. First, the root process reads data from a local file and sends it to the other processes. Given m data points and P processes, each process will have $\frac{m}{P}$ data points. Second, each process assigns its local points to every thread based on a given block size. If the block size is equal to B and there are T threads per process, each thread will have m/P/T/B data blocks.

After getting a subset of data, every process launches a number of T threads and executes the thread function *do_clustering_thread*. The thread entry function *do_clustering_thread* simply follows the sequential Algorithm 4 except that each thread just visits and computes its allocated blocks one by one in a loop. The other difference is that *do_clustering_thread* uses a multi-level method to merge each thread's local centers to obtain the global centers. In the first level, each thread's own blocks are merged to get the thread-local

Algorithm 5 Parallel annealing-enhanced tiled synchronization reducing algorithm

```
1:  parallel_aynch_clustering(points, m, B, k, threshold, P, T)
2:  /* B: block size */
3:  /* P: #processes; T: #threads */
4:  ▷ Read and distribute data points
5:  if pid == 0 and tid ==0 then
6:      n_b = m / B / P / T; /*#blocks per thread*/
7:      Read a file and send a subset of data to each process
8:      /* select first k data points as initial cluster centers */
9:      for each center c ← 0 to k-1 do
10:         g_centers[c] = points[c]
11:     end for
12: else
13:     Receive a subset of data from P0, store in points_proc
14: end if
15: ▷ Each process launches T threads to run a thread kernel
16: for each thread tid ← 0 to T do
17:     Allocate points_proc to each thread, store in points_thrd
18:     call do_clustering_thread(points_thrd, n_b, threshold, tid,
19:     g_centers)
20: end for
```

result. In the second level, each thread's result is merged with other threads that belong to the same process. Finally, the third level is conducted among different processes to obtain the global centers.

6 ISSUES AND DISCUSSIONS ON USING DIFFERENT METHODS TO MERGE LOCAL CENTERS

In our synchronization-reducing algorithms, we merge the local optimization results of every block to get the global centers by computing each cluster's coordinate sum followed by dividing the sum by the number of points in the cluster. It is similar to computing a *weighted average* on all the data blocks.

There are other possible options to do the same work. In this section, we study using different methods to get the global centers.

Here, we design and evaluate two new merging methods:

- Average of local centers: Since each block can compute its own version of global centers independently (via local optimization), we can use a much simpler method to compute the average of all the centers from all the blocks directly. This method assumes that all the blocks' centers have an equal weight.
- Double k-means: This idea is inspired by the extensively studied sampling-based methods. In this method, we consider the k centers computed from each block as k sample points. Hence, N blocks will contribute $N \times k$ samples. Next, the $N \times k$ samples will call the regular k-means algorithm to find the global centers. Note that the sampling method is only used to merge local centers to global centers, not to compute approximate solutions.

In Table 1, we compare our weighted-average method to the above new methods using the MNIST dataset (information of the dataset is provided in Section 7). Table 1 shows the performance of each method in terms of execution time, number of global iterations (not considering each block's local optimization iterations), and the clustering result's SSE cost.

From the table, we can see that the weighted-average method and the *double k-means* method provide good SSE costs. Also, between the weighted-average method method and the double k-means method, weighted-average has a better SSE cost and faster execution time.

Note that the *average-of-local-centers* is the fastest method, but its cost is much worse than the other methods. Nevertheless, if a user is only interested in approximate solutions, the *average-of-local-centers* method will help. Since this work targets high-quality clustering results, we choose to use the *weighted-average* method.

Table 1: Comparison of different merging methods.

	Weighted average	Average of local centers	Double k-means
Time (seconds)	5.11	0.48	5.78
Global iterations	28	3	31
SSE cost	25.332×10^9	27.164×10^9	25.337×10^9

Table 2: Specifications of the Cray HPC System.

Compute nodes	1,020 (max 2048 cores per pbs job)
Memory per node	64 GB
Processors per node	2
Cores per processor	16
Processor	AMD Opteron 6380 2.5GHz
Interconnect	Cray Gemini
MPI	Cray-MPICH 7.2.5

7 EXPERIMENTAL RESULTS

We examine the performance of the annealing-enhanced algorithm by conducting the following three types of experiments:

(1) Effect of selecting different parameters,
(2) Investigation of the sequential implementation in terms of execution time and SSE cost,
(3) Our parallel implementation compared with the open source MPI k-means library from the Northwestern University [16].

All the experiments are conducted on a Cray XE6/XK7 system, whose information is shown in Table 2.

The experiments used four real world datasets, which are summarized in Table 3. MNIST is a dataset used for hand-writing digit recognition [14]. It consists of 10,000 black and white training images. Each image represents one of the ten hand written digits. CIFAR-10 and CIFAR-100 datasets are used for feature learning and object recognition [12]. Each dataset includes 60,000 color images with labels. PLACES-2 is generated from *Places 2* [27]. It is a large collection of color pictures for different sceneries. We use the PLACES-2 dataset to do a large 1024-core experiment.

Table 3: The datasets used in our experiments.

	MNIST	CIFAR-10	CIFAR-100	PLACES-2
K	10	10	100	50
#Data points	10,000	60,000	60,000	1,024,000
Dimensions	784	3,072	3,072	1,024
Dataset size	17MB	626MB	629MB	3.25GB

7.1 Effect of Parameters

We present how a parameter may affect the performance of our algorithm. In particular, we study the parameters of the number of annealing steps and the stop condition in local optimizations.

Annealing steps: Annealing steps are the "hard-shaking" operations that invoke more frequent synchronizations. Table 4 shows the effect of the annealing steps on the CIFAR-100 dataset. From the table, we can see that the algorithm converges the fastest with 100 annealing steps. But when using 125 annealing steps, the algorithm becomes much slower. This result shows that the algorithm is sensitive to the number of annealing steps. That is, stopping too early or too late will lead to suboptimal performance. Hence, the annealing step must stop at an appropriate "valley" location. Such a special phenomenon has led us to take efforts to develop the "valley-searching" heuristic scheme as presented in Section 4.

Stop conditions: The local optimization function can execute a number of iterations by itself without any communications. Table 5

Table 4: Effect of the number of annealing steps.

Annealing steps	SSE cost	Global #iteration	Time (seconds)
50	398,835,242,961.11	140	613.72
75	398,845,351,171.04	151	502.88
100	398,861,208,887.09	124	238.8
125	398,838,864,406.40	208	540.33

Table 5: Effect of stop condition.

Threshold	SSE cost	Global #iteration	Time (seconds)
90%	25,331,986,841.05	32	12.59
95%	25,331,986,841.05	32	12.59
99%	25,334,655,274.55	26	10.22
100%	25,541,238,298.66	21	14.33

shows the effect of the stop condition that controls when to exit the local optimization on the MNIST dataset. When the stop condition threshold = p%, local optimization will not exit until the new cost is greater than p% of the old cost. From various experiments, we find that the threshold of 99% typically provides the best performance. Hence, we set local optimization stop condition to be 99% in our experiments.

7.2 Performance Evaluation of the Sequential Implementation

We compare the performance of our sequential annealing-enhanced implementation with the k-means algorithm using the datasets of MNIST, CIFAR-10, and CIFAR-100. This experiment will examine not only the total execution time but also the SSE cost. Table 6 shows execution time in seconds, and Table 7 shows the SSE cost and the number of global iterations measured for the same experiment.

MNIST: From Table 6, we can see that our algorithm is 74% faster than k-means. As revealed by Table 7, this speedup is due to the reduced number of global iterations (i.e., 54 iterations versus 106 iterations). The difference in their SSE costs is 0.026%.

CIFAR-10: From Table 6, our algorithm is slightly faster than k-means by 3%. The speedup is limited because both algorithms have a similar number of global iterations (i.e., 80 versus 88 as shown in Table 7). Their SSE cost difference is almost equal to zero.

CIFAR-100: For CIFAR-100, our algorithm is 18% faster than k-means, as shown in Table 6. From Table 7, we can find that our algorithm has 124 global iterations while k-means has 201 global iterations, and therefore runs faster. The performance speedup is not linearly proportional to the reduced number of global iterations since each global iteration also includes the local optimization function which has a number of iterations. Their corresponding SSE difference is 0.005%.

7.3 Performance Evaluation of the Parallel Implementation

To evaluate the scalability of our parallel program, we did experiments with the MNIST, CIFAR-10 and CIFAR-100 datasets using

Table 6: Performance comparison for sequential algorithms (Table 7 also shows their SSE cost and number of iterations).

	K-means (seconds)	Annealing-enhanced algorithm (seconds)
MNIST	15.6	8.9
CIFAR-10	303.5	295
CIFAR-100	6700.6	5650

4 to 32 CPU cores. In addition, for the largest dataset PLACES-2 (3.25GB), we used 1024 CPU cores to solve the problem quickly.

Figure 1 shows our experimental results with the MNIST dataset. Given one CPU core, our parallel algorithm is 74% faster than the MPI k-means library. When using two and four CPU cores, our parallel program is faster than the MPI k-means library by 80%. From one core to 4 cores, we are able to reduce the total execution time from 9 seconds to 2.2 seconds. As for the CIFAR-10 dataset, our parallel implementation has similar performance to the parallel k-means implementation (i.e., 3% faster).

Figure 2 shows the experimental results with the CIFAR-100 dataset. From one CPU core to 32 CPU cores, our algorithm is able to outperform the MPI k-means by 19%. Also, our parallel implementation shows a good scalability. By using 32 cores, we can decrease the execution time from 5651 seconds to 180 seconds. In addition, the SSE costs of the parallel implementation are the same as the costs of the sequential implementation that are shown in Table 7, thus they are not shown here again.

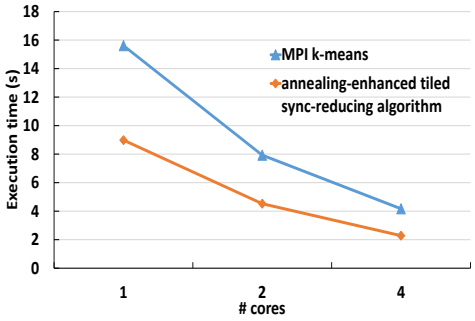

Figure 1: Scalability experiment on the MNIST dataset.

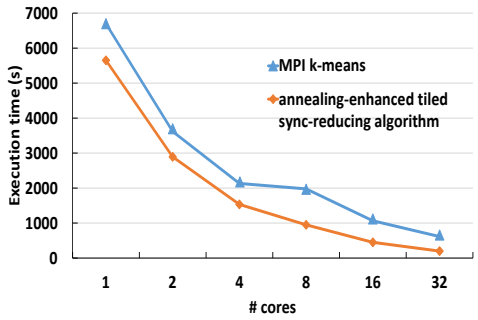

Figure 2: Scalability experiment on the CIFAR-100 dataset.

Table 7: SSE cost and the number of global iterations for k-means and our annealing-enhanced algorithm.

	K-means		Annealing-enhanced algorithm	
	SSE cost	Iterations	SSE cost	Iterations
MNIST	25,322,079,191.39	106	25,328,820,274.91	54
CIFAR-10	474,370,518,296.29	88	474,370,686,193.56	80
CIFAR-100	398,838,513,925.16	201	398,861,208,887.09	124

Our largest experiment was conducted with the 3.25GB PLACES-2 dataset using 1,024 CPU cores. On 1,024 CPU cores, the MPI k-means library takes 30.1 seconds while our parallel implementation takes 26.7 seconds to cluster PLACES2. The performance improvement is around 11%. Also, the two programs' SSE costs are almost identical (less than 0.005% difference, i.e., SSE=2,228,653,471,842.53 versus SSE=2,228,653,642,874.45). Note that experiments with different datasets have different speedups due to the fact that the actual convergence rate is dependent on the datapoint distribution in the input data.

8 CONCLUSION

This paper presents an annealing-enhanced tiled synchronization-reducing clustering algorithm that is designed for manycore HPC systems. This work makes three contributions. First, it combines the design of tiles and the asynchronous local optimization to design a new algorithm. In the algorithm, an iterative local optimization is computed on each data block to obtain a clustering result. Each block's local cluster result is later merged by different merging methods. By combining the two techniques of blocking and asynchronous clustering, we can reduce the number of synchronizations among threads and processes. Second, to avoid the possible slower convergence rate incurred by reduced synchronizations, we introduce new techniques of stop condition and annealing to achieve as good or better performance than the standard k-means clustering algorithm while keeping nearly the same SSE cost. Stop condition is introduced to control the time spent on each block's local optimization. Annealing is introduced as a heuristic to resolve the issue of slower convergence rate due to reduced synchronizations. Third, we design and implement a new parallel implementation to demonstrate the performance of the annealing-enhanced algorithm. The techniques developed in the work are generic, and can be extended and applied to support other parallel machine learning methods on both HPC and cloud systems.

REFERENCES

[1] David Arthur and Sergei Vassilvitskii. 2007. k-means++: The advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 1027–1035.

[2] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaeif, P. Luszczek, A. YarKhan, and J. Dongarra. 2011. Flexible Development of Dense Linear Algebra Algorithms on Massively Parallel Architectures with DPLASMA. In *Proceedings of the Workshops of the 25th IEEE International Symposium on Parallel and Distributed Processing (IPDPSW 2011)*. IEEE, 1432–1441.

[3] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Herault, and Jack J Dongarra. 2013. PaRSEC: Exploiting Heterogeneity to Enhance Scalability. *Computing in Science & Engineering* 15, 6 (2013), 36–45.

[4] Wei Dai, Abhimanu Kumar, Jinliang Wei, Qirong Ho, Garth Gibson, and Eric P Xing. 2014. High-performance distributed ML at scale through parameter server consistency models. *arXiv preprint arXiv:1410.8043* (2014).

[5] Inderjit S Dhillon and Dharmendra S Modha. 2002. A data-clustering algorithm on distributed memory multiprocessors. In *Large-Scale Parallel Data Mining*. Springer, 245–260.

[6] Reza Farivar, Daniel Rebolledo, Ellick Chan, and Roy H Campbell. 2008. A Parallel Implementation of K-Means Clustering on GPUs.. In *Pdpta*, Vol. 13. 212–312.

[7] Giuseppe Di Fatta, Francesco Blasa, Simone Cafiero, and Giancarlo Fortino. 2013. Fault tolerant decentralised K-Means clustering for asynchronous large-scale networks. *J. Parallel and Distrib. Comput.* 73, 3 (2013), 317 – 329. https://doi.org/10.1016/j.jpdc.2012.09.009 Models and Algorithms for High-Performance Distributed Data Mining.

[8] Alex Gittens, Aditya Devarakonda, Evan Racah, Michael Ringenburg, Lisa Gerhardt, Jey Kottalam, Jialin Liu, Kristyn Maschhoff, Shane Canon, Jatin Chhugani, et al. 2016. Matrix Factorization at Scale: a Comparison of Scientific Data Analytics in Spark and C+ MPI Using Three Case Studies. *arXiv preprint arXiv:1607.01335* (2016).

[9] John A Hartigan and JA Hartigan. 1975. *Clustering algorithms*. Vol. 209. Wiley New York.

[10] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing. 2013. More effective distributed ML via a stale synchronous parallel parameter server. In *Advances in neural information processing systems*. 1223–1231.

[11] Anil K Jain, M Narasimha Murty, and Patrick J Flynn. 1999. Data clustering: a review. *ACM computing surveys (CSUR)* 31, 3 (1999), 264–323.

[12] Alex Krizhevsky and Geoffrey Hinton. 2009. Learning multiple layers of features from tiny images. (2009).

[13] Monica D Lam, Edward E Rothberg, and Michael E Wolf. 1991. The cache performance and optimizations of blocked algorithms. In *ACM SIGARCH Computer Architecture News*, Vol. 19. ACM, 63–74.

[14] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[15] Charles E Leiserson. 2010. The Cilk++ Concurrency Platform. *The Journal of Supercomputing* 51, 3 (2010), 244–257.

[16] Weikeng Liao. 2017. Parallel K-Means Data Clustering for Large Data Sets. http://www.ece.northwestern.edu/~wkliao/Kmeans/index.html. (2017).

[17] Stephen Merendino and M Emre Celebi. 2013. A Simulated Annealing Clustering Algorithm Based On Center Perturbation Using Gaussian Mutation.. In *FLAIRS Conference*.

[18] MLlib. 2017. http://spark.apache.org/mllib/. (2017).

[19] Gabriela Trazzi Perim, Estefhan Dazzi Wandekokem, and Flávio Miguel Varejão. 2008. K-means initialization methods for improving clustering by simulated annealing. In *Ibero-American Conference on Artificial Intelligence*. Springer, 133–142.

[20] David Sculley. 2010. Web-scale k-means clustering. In *Proceedings of the 19th international conference on World wide web*. ACM, 1177–1178.

[21] Shokri Z Selim and K1 Alsultan. 1991. A simulated annealing algorithm for the clustering problem. *Pattern recognition* 24, 10 (1991), 1003–1008.

[22] Fengguang Song, Asim YarKhan, and Jack Dongarra. 2009. Dynamic Task Scheduling for Linear Algebra Algorithms on Distributed-Memory Multicore Systems. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, New York, NY, USA, 1–11.

[23] Spark. 2017. http://spark.apache.org/. (2017).

[24] Fuhui Wu, Qingbo Wu, Yusong Tan, Lifeng Wei, Lisong Shao, and Long Gao. 2013. A vectorized k-means algorithm for intel many integrated core architecture. In *International Workshop on Advanced Parallel Processing Technologies*. Springer, 277–294.

[25] Mario Zechner and Michael Granitzer. 2009. Accelerating k-means on the graphics processor via CUDA. In *Intensive Applications and Services, 2009. INTENSIVE'09. First International Conference on*. IEEE, 7–15.

[26] Weizhong Zhao, Huifang Ma, and Qing He. 2009. Parallel k-means clustering based on mapreduce. In *IEEE International Conference on Cloud Computing*. Springer, 674–679.

[27] Bolei Zhou, Aditya Khosla, Agata Lapedriza, Antonio Torralba, and Aude Oliva. 2016. Places: An image database for deep scene understanding. *arXiv preprint arXiv:1610.02055* (2016).