Fast Parallel Initialization for k-means

PLEASE DO NOT DISTRIBUTE

Dipankar Das
Intel Labs Bangalore
dipankar.das@intel.com

Jatin Garg
IIT Delhi
cs1110223@cse.iitd.ac.in

Sachin Goel
IIT Delhi
cs1110249@cse.iitd.ac.in

Ragesh Jaiswal[*]
IIT Delhi
rjaiswal@cse.iitd.ac.in

Sandeep Sen
IIT Delhi
ssen@cse.iitd.ac.in

Abstract

We provide a fast data-parallel initialization routine for k-means algorithm. The algorithm is based on sampling techniques that are easy to implement and parallelize. These parallel algorithms are implemented on a dual-socket Intel®Xeon®E5-2680v2. Empirical evaluations on large real and synthetic datasets demonstrates the practical utility of our techniques.

1 Introduction

k-means Clustering is one of the most basic data processing tasks. So, any improvement in quality or efficiency in the popular clustering techniques has a significant impact. We consider the k-means problem and the k-means algorithm (also known as the Lloyds algorithm) that are the one of the most popular clustering problems and solutions respectively, known in the Machine Learning literature. The k-means problem is defined in the following manner: given a set of points $X \in \mathbb{R}^d$ and an integer k, find a set of centers $C \subset \mathbb{R}^d$ such that the cost

$$\Phi_C(X) \overset{def.}{=} \sum_{x \in X} \min_{c \in C} ||x - c||^2$$

is minimized. These k centers induce a natural partition of the data points (points that are closest to the same center are in the same partition). This is also known as the *Voronoi* partition. The computational hardness of this problem is well understood and it is known that the problem is NP-hard [5]. Various approximation algorithms for this problem is known [8, 4]. However, the algorithm that is used to solve the k-means problem is practice is the k-means algorithm.[1] This is a local search algorithm that starts with an arbitrary set of k centers and then locally shifts these centers so that the cost of decreases and continue until no local improvement is possible. Even though this algorithm gives good results in practice, the algorithm does not provide any guarantees in terms of performance or quality with respect to the the cost function. This basically means that the k-means algorithm may take a huge amount of time to converge or the the solution produced has arbitrarily

[*]Thanks to funding agencies.

[1]Unfortunately the algorithm has the same name as the problem. So, the reader should be careful while reading.

higher cost compared to the best solution. One way to ensure solution quality is to initialise the k-means algorithm with k centers that satisfies some minimal quality guarantee. It is known that the k-means algorithm only improves the cost while execution. So, at the end, the quality guarantee follows from the initial centers. In some sense, we get the best of both worlds – k-means algorithm ensures good practical performance and initial centers ensures some minimal quality guarantee.

Seeding So now the question shifts to how to pick the initial centers for the k-means algorithm? This is sometimes known as the "seeding" algorithm. The requirements of such a seeding algorithm is that it should be (i) fast, (ii) simple, and (iii) give some quality guarantee. [2] There has been a significant amount of work in obtaining algorithms that give quality guarantees. That is, obtaining approximation algorithms. So, as far as requirement (iii) is concerned, there are many candidate algorithms. However, most of these algorithms do not satisfy (i) and (ii). One of the most popular algorithms that seem to satisfy all the properties and is popularly used as a seeding algorithm for the k-means algorithm is the k−`means++seed` algorithm [2] that uses a simple sampling based approach. Here, the k points are sampled in the following manner: Pick the first center uniformly at random and for $i > 1$, pick a point p to be the i^{th} center with probability proportional to the squared Euclidean distance of p from the nearest center from among the set of previously chosen $(i-1)$ centers C_{i-1} (i.e., with probability $\frac{\Phi_{C_{i-1}}(\{p\})}{\Phi_{C_{i-1}}(X)}$). This sampling procedure is also called the D^2-sampling [7]. Arthur and Vassilvitskii [2] showed that this sampling procedure gives an approximation factor of $O(\log k)$ in expectation. This sampling based idea was further developed by Jaiswal *et al.* [7] and they gave a sampling based algorithm that gives a solution that is arbitrarily close to the optimal solution in terms of the cost. That is, they give a $(1+\varepsilon)$-approximation algorithm for arbitrary small ε. Moreover, their algorithm has another advantage that the algorithm has an inherent massively parallel version. However, the running time of their algorithm is $O(nd \cdot 2^{\tilde{O}(k/\varepsilon)})$ which is large for practical relevance even with parallelisation. In this work, we explore a simple modified version of their algorithm with much smaller running time. Note that the strong approximation guarantee of the original algorithm is compromised in this process. However, through empirical analysis on large datasets we build some confidence that the quality of the solution produced is good. We do extensive comparative analysis with the other seeding techniques. The modified algorithm remains inherently parallel and we do a parallel implementation of our algorithm that we discuss next. Note that the k-means algorithm runs in a number of rounds and these rounds are sequential. So, if the seed k centers result in smaller number of rounds and furthermore the seeding algorithm is parallel, then that means that we are effectively using the parallel architecture for k-means clustering. We also do a comparative analysis of the decrease in the number of rounds of the k-means algorithm (i.e., number of Lloyd iterations).

Parallel algorithm The parallel algorithms discussed in this work are designed for a shared memory parallel programming model, where all parallel computation threads/processes can view data written into the shared memory by any of the threads/processes. Such shared-memory multi/many-core processors are widely available these days. Designing parallel algorithms for important tasks such as clustering that exploit this readily available parallelism has gained importance.

2 The Algorithm

We start with a discussion of the algorithm of Jaiswal *et al.* [7] since our algorithm will be a modification of their algorithm. Given a dataset X, let the optimal clusters be denoted by $X_1, ..., X_k$. Suppose we have i centers $C_i = \{c_1, ..., c_i\}$ such that these centers are good centers for some i optimal clusters. That is, there exists distinct indices $j_1, ..., j_i \in \{1, ..., k\}$ such that $\forall l \leq i, \Phi_{c_l}(X_{j_l}) \leq (1 + O(\varepsilon)) \cdot \Delta(X_{j_l})$, where $\Delta(X_{j_l})$ denotes the optimal 1-means cost of the cluster X_{j_l}. Suppose at this time, we sample a set S of N points independently with the sampling technique of k−`means++seed`. A formal description of k−`means++seed` is given in Figure 3.1 Since the sampling technique gives preference to points that are further away from the current centers, we are likely to sample points from clusters whose indices do not appear in the set $\{j_1, ..., j_i\}$.

[2]Simplicity is a subjective measure and hence data scientists might not be comfortable with such a requirement. However, it is an important measure since simpler techniques have advantages when it comes to implementation and debugging.

Let us call such clusters "uncovered" clusters. So, there is a good chance that a significant number of sampled points will be from uncovered clusters. The algorithm of Jaiswal *et al.* [7] just considers all possible subset of points of size $M < N$ and considers the centroid of these points as the $(i+1)^{\text{th}}$ center. The main idea being that at least one of these subsets will behave as a uniform sample of points from some uncovered cluster and known results [6] suggests that the centroid of such a sample will be a good center for this uncovered cluster. However, this results in the exponential running time since each time sampling is done, there are more than 1 subsets to consider. [3]

Our algorithm follows the same idea as that of Jaiswal *et al.* [7] given in the above paragraph. However, while choosing the $(i+1)^{\text{th}}$ center, we do not try more than one possibility. Instead of considering all possible subsets of S, we cluster S into k clusters using the k-`means++seed` algorithm and then pick the centroid of the largest cluster as the $(i+1)^{th}$ center. [4] Figure 2.1 gives a formal description of our algorithm. Figure 1 gives an illustration of one iteration of our algorithm. This algorithm has inherent data level parallelism. The parallel version of our algorithm is given in Figure 2.2. The rationale behind this choice for the $(i+1)^{th}$ center is the following: Let X_l denote the uncovered cluster that is farthest from the current set C_i of centers. That is, $\Phi_{C_i}(X_l)$ is the largest amongst the uncovered clusters. Note that S is likely to have more points from X_l than any other optimal cluster. Ideally, we would like to isolate the points from X_l present in S and then take the centroid of these points. In Jaiswal *et al.* [7] this was done by taking all possible subsets to ensure that one of these subsets contain all points from X_l. What we essentially do here is to cluster the points in S, the hope being that all points in X_l that are present in S will form a cluster. Since our algorithm picks the largest cluster it would be able to isolate the points of X_l present in S.

Note that our suggested algorithm is a heuristic and we do not give any provable guarantees for our algorithm. So, the only way to validate the effectiveness of our algorithm is to do a detailed experimental analysis on real and synthetic datasets. This is what we discuss in the remaining paper.

```
$D^2$-Seeding($X, k, N$)
  (1) $C_0 \leftarrow \{\}$
  (2) For $i$ = 1 to $k$
      (a) Sample a multiset $S$ of $N$ points from $X$ using $D^2$ sampling with respect to $C_{i-1}$. [a]
      (b) Let $T_l$ denote the largest cluster obtained by running $k$-means++seed
          algorithm on inputs $S$ and $k$.
      (c) $c_i \leftarrow \Gamma(T_l)$ and $C_i \leftarrow C_{i-1} \cup \{c_i\}$. [b]
  (3) Output $C_k$
```

[a]When $i = 1$, D^2-sampling is the same as uniform sampling.

[b]$\Gamma(T)$ denote the centroid of the points in T i.e., $\Gamma(T) = \frac{\sum_{t \in T} t}{|T|}$.

Algorithm 2.1: Our seeding algorithm. N is an input parameter that may be adjusted for performance/quality.

3 Experimental Analysis

3.1 Datasets and machine

We use the following datasets for evaluating our algorithm. Note that the first three datasets are real datasets and the last three are synthetically generated datasets. The sequential algorithms in this work were implemented and evaluated on a single workstation with a dual-socket Intel®Xeon®E5-2680v2, with 20 cores and 40 threads, and operating at 2.8GHz running Ubuntu 14.04. We build our code using the GCC 4.8.2 and use OpenMP based thread parallelism.

[3]Even if we consider only two possibilities in each step, the running time will be 2^k since we have the sample k times.

[4]Ties are broken arbitrarily.

```
Par-$D^2$-Seeding$(X, k, N)$
  (1) $C_0 \leftarrow \{\}$
  (2) For all $1 \leq j \leq M$, let $P_j = \left\{X\left[\frac{(j-1)\cdot|X|}{M}\right], ..., X\left[\frac{j\cdot|X|}{M}\right]\right\}$. [a]
  (3) For $i \leftarrow 1$ to $k$:
      (a) $S \leftarrow \{\}$        // $S$ here denotes a multiset.
      (b) Parallel-for $j \leftarrow 1$ to $M$:
          (i) For all $1 \leq l \leq |X|/M$, $D_j[l] \leftarrow \Phi_{C_{i-1}}(\{P_j[l]\})$.
          (ii) $D[j] \leftarrow \Phi_{C_{i-1}}(P_j)$.
      (c) Let $\mathcal{D}$ denote the distribution over $\{1, ..., M\}$ defined by $D[1], ..., D[M]$.
      (d) For all $j$, let $\mathcal{D}_j$ denote a distribution over $\left\{1, ..., \frac{|X|}{M}\right\}$ defined by $D_j[1], ..., D_j\left[\frac{|X|}{M}\right]$.
      (e) Parallel-for $j \leftarrow 1$ to $N$:
          (i) Sample $r$ from $\mathcal{D}$ and then $t$ from $\mathcal{D}_r$.
          (ii) $S \leftarrow S \cup P_r[t]$.
      (f) Let $T_l$ denote the largest cluster obtained by running $k$-means++seed
          algorithm on inputs $S$ and $k$.
      (g) $c_i \leftarrow \Gamma(T_l)$ and $C_i \leftarrow C_{i-1} \cup \{c_i\}$.
  (4) Output $C_k$
```

[a] $X[i]$ denotes the i^{th} element of the dataset X

Algorithm 2.2: Parallel version of our seeding algorithm. N is an input parameter and M is a global environment variable indicating the number of hardware-level threads available.

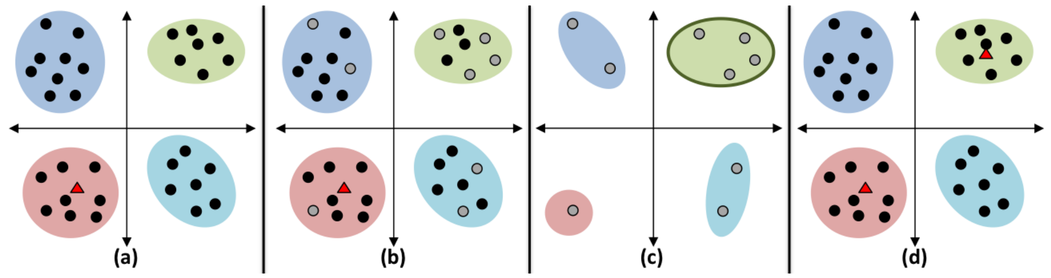

Figure 1: (a) First center (red triangle) has been chosen in iteration 1 (b) Points (in gray) are sampled with D^2-sampling (c) The sampled points are clustered (d) Centroid of the largest cluster is chosen as the second center.

3.2 Parameter N

Before we start the comparative analysis, we would like to discuss the parameter N of our D^2-seeding algorithm. We will pick $N > k$.[5] How large should N be as a function of k? Note that the larger the number of points from an uncovered cluster that we are able to isolate, the better is the centroid of these isolated points with respect to this uncovered cluster. The number of points from each uncovered cluster will be clearly larger for larger values of N. However, the running time of the algorithm grows as N increases. So, there is a quality/performance tradeoff and the user should pick a value of N that gives the best results. We give cost analysis for values of N that are multiples of k in Table 2. Since our algorithm is randomized, we give the mean and standard deviation of the cost over 20 runs over the datasets.

We note that there is a significant improvement as N is increased from k to $10k$. However, the improvement is not that significant when comparing $N = 10k$ and $N = 100k$. We suggest using $N = 10k$ as a standard input for our algorithm. However, the user may adjust this parameter

[5]This would avoid problems in step 2(b). However, note that if $N \leq k$, then our algorithm is basically the k-means++seed algorithm. This is because in each iteration, we pick a point sampled with D^2-sampling as the next center. So, k-means++seed algorithm is one of the extreme cases of our algorithm.

Dataset	**Details**	n	d	k
MNIST	Handwriting dataset [11]	70000	784	10
CIFAR	CIFAR-10 Image dataset [10]	60000	3072	10
3D	3D Road Network dataset [9]	434874	3	5
Birch1	2D Gaussians in a grid [1]	100000	2	100
Birch2	2D Gaussians on a sine curve [1]	100000	2	100
Birch3	2D Gaussians at random locations [1]	100000	2	100

Table 1: Datasets used for evaluation in this work.

depending on the database for best performance. The supplementary material gives a finer grained analysis of cost w.r.t. N.

N		**MNIST** ($\times 10^{10}$)	**CIFAR** ($\times 10^{10}$)	**3D** ($\times 10^{6}$)	**Birch1** ($\times 10^{12}$)	**Birch2** ($\times 10^{10}$)	**Birch3** ($\times 10^{12}$)
k	mean	32.39	80.90	15.98	187.67	158.02	66.45
	sd	1.31	6.11	5.05	7.88	16.68	3.47
$10k$	mean	21.19	53.92	11.89	120.39	49.76	45.35
	sd	4.45	1.71	2.71	3.48	1.16	1.10
$100k$	mean	21.18	53.37	11.51	121.50	50.30	45.59
	sd	4.12	1.37	1.76	3.41	1.80	1.09

Table 2: This table gives the cost of the solution produced by our seeding algorithm as a function of the input parameter N.

3.3 Cost, number of lloyds iterations, and running time

Recall that for a given set C of centers, the cost is given by $\Phi_C(X) = \sum_{x \in X} \min_{c \in C} ||x - c||^2$. Given cost of the solution as an evaluation metric, we compare our algorithm with k-`means++seed` which is widely believed to give the best results and is used in practice to obtain the seed for the k-means algorithm. For comparison purposes, we also give results for the `Random` procedure that picks k centers uniformly at random and is the simplest seeding algorithm. Since all the three procedures are randomized, we give the mean and standard deviation measured over 20 runs. The cost comparison is given in Table 3. We note that the our algorithm (with $N = 10k$) gives much better results compared to the other algorithms.

Algorithm		**MNIST** ($\times 10^{10}$)	**CIFAR** ($\times 10^{10}$)	**3D** ($\times 10^{6}$)	**Birch1** ($\times 10^{12}$)	**Birch2** ($\times 10^{10}$)	**Birch3** ($\times 10^{12}$)
`Random`	mean	31.99	78.87	55.32	289.97	713.26	152.59
	sd	1.24	4.81	44.26	30.91	203.07	26.17
k-`means++seed`	mean	32.03	79.46	15.66	190.82	167.57	67.36
	sd	1.09	6.17	4.33	7.68	22.18	4.65
D^2-`seeding` ($N = k$)	mean	32.39	80.90	15.98	187.67	158.02	66.45
	sd	1.31	6.11	5.05	7.88	16.68	3.47
D^2-`seeding` ($N = 10k$)	mean	21.19	53.92	11.89	120.39	49.76	45.35
	sd	0.44	1.71	2.71	3.48	1.16	1.10
D^2-`seeding` ($N = 100k$)	mean	21.18	53.37	11.51	121.50	50.30	45.59
	sd	0.41	1.37	1.76	3.41	1.80	1.09

Table 3: This table gives the cost of the solution produced by the seeding algorithms.

We also compare cost of the solution obtained on running Lloyds iterations until convergence, post the seeding step. The results of this comparison are given in Table 4. We note that for the synthetic datasets, we get a significant improvement. However, for the real datasets the final costs obtained are comparable. Another interesting observation to be made when studying the experimental results in Tables 3 and 4 is that cost of the solution produced by our seeding algorithm (with $N = 10k$) in Table 3 is consistently close to the best solution produced by the Lloyds algorithm in Table 4.

Algorithm		MNIST ($\times 10^{10}$)	CIFAR ($\times 10^{10}$)	3D ($\times 10^{6}$)	Birch1 ($\times 10^{12}$)	Birch2 ($\times 10^{10}$)	Birch3 ($\times 10^{12}$)
`Random`	mean	17.93	47.43	8.82	111.75	159.36	49.33
	sd	0.0562	0.0599	0.0001	5.8873	21.6450	5.4267
k`-means++seed`	mean	17.90	47.46	8.82	106.24	84.49	40.41
	sd	0.0556	0.1048	0.0004	3.4275	8.8590	1.3834
D^2`-seeding` ($N = k$)	mean	17.93	47.45	8.82	106.31	84.56	40.45
	sd	0.0546	0.0711	0.0005	3.3346	10.6210	1.0226
D^2`-seeding` ($N = 10k$)	mean	17.91	47.46	8.82	98.11	45.67	38.95
	sd	0.0706	0.0667	0.0005	2.1194	3.8557	0.5371
D^2`-seeding` ($N = 100k$)	mean	17.90	47.44	8.82	97.58	45.88	39.02
	sd	0.0445	0.0606	0.0005	2.3682	0.94	0.54

Table 4: This table gives the cost of the solution produced by the Lloyds algorithm with the initial centers obtained using the above seeding algorithms.

Since our goal is to use our algorithm as a seeding algorithm for the k-means algorithm which is inherently sequential, the number of iterations of Lloyds algorithm that need to be run for convergence is an important metric. We say that the k-means algorithm has converged if the decrease in the cost is less than 0.0001 times the current cost. We do a comparative analysis of the number of Lloyds iterations. The results of this comparison is given in Table 5. We note that our algorithm (with $N = 10k$) gives improvements for all datasets. These improvements are significant for the synthetic Birch datasets whereas the improvements are small for the real datasets.

Algorithm		MNIST	CIFAR	3D	Birch1	Birch2	Birch3
`Random`	mean	19.48	22.32	32.38	41.54	19.56	51.78
	sd	7.94	6.51	6.25	15.82	9.06	16.74
k`-means++seed`	mean	22.14	24.18	23.18	30.28	16.30	41.78
	sd	6.60	7.40	10.04	12.05	6.55	12.67
D^2`-seeding` ($N = k$)	mean	20.28	23.18	21.2	30.46	13.46	40.32
	sd	6.52	7.53	8.98	10.05	5.21	10.76
D^2`-seeding` ($N = 10k$)	mean	20.88	23.36	20.12	13.92	2.42	32.58
	sd	7.47	6.90	7.65	7.66	0.49	10.35
D^2`-seeding` ($N = 100k$)	mean	23.26	21.88	16.8	12.20	3.00	33.90
	sd	8.06	7.55	6.68	5.41	0.79	7.75

Table 5: This table gives the number of lloyds steps until convergence when the algorithm is executed with the seed produced by the above algorithms.

We also do a running time comparison of the three algorithms. The results are given in Table 6. Since our algorithm does more sampling work than k`-means++seed`, the running time of our algorithm is of course larger. However, one should note that the running time is not more than twice as that of the k`-means++seed` algorithm.

3.4 Parallel implementation and M

Consider `Par-`D^2`-seeding`, the parallel version of our algorithm. M denotes the number of parallel threads executed. The parallel steps in the algorithm are steps 3(b) and 3(e). We do an analysis of the running time as a function of the number of threads used in this parallel step. The maximum number of threads in our analysis is 40 since that is the total number of hardware level threads available in the architecture that we use. This is just to understand the behaviour of our algorithm in terms of the amount of parallelism available.

Parallel architecture The `Par-`D^2`-seeding` algorithm and other parallel algorithms discussed in this work are designed for a shared memory parallel programming model, where all parallel computation threads/processes can view data written into the shared memory by any of the

Algorithm		MNIST	CIFAR	3D	Birch1	Birch2	Birch3
Random	mean	0.0015	0.0009	0.0071	0.0032	0.0031	0.0030
	sd	0.0004	0.0001	0.0009	0.0005	0.0007	0.0005
k-means++seed	mean	29.8301	96.0875	1.8191	136.9317	133.9361	134.245
	sd	0.2941	0.6122	0.1252	0.4061	0.3542	0.4413
D^2-seeding ($N = k$)	mean	29.8118	96.6231	1.8755	152.0806	149.1547	149.36
	sd	0.3951	0.6524	0.1060	0.4001	0.4189	0.3795
D^2-seeding ($N = 10k$)	mean	30.2519	98.0837	1.8372	274.9108	271.5379	272.50
	sd	0.3230	0.4534	0.1325	0.5928	0.4878	0.6575
D^2-seeding ($N = 100k$)	mean	34.6420	115.7657	1.9183	1509.3000	1545.4000	1511.4000
	sd	0.4269	0.6319	0.1915	20.5850	28.8680	20.8380

Table 6: Running time (in seconds) of D^2-seeding.

# Machines		MNIST	CIFAR	3D	Birch1	Birch2	Birch3
$M = 1$	mean	30.25	98.08	1.83	274.91	271.53	272.50
	sd	0.32	0.45	0.13	0.59	0.48	0.65
$M = 5$	mean	7.02	22.31	0.99	152.64	151.73	155.62
	sd	1.19	2.28	0.20	5.98	5.72	6.04
$M = 10$	mean	3.53	11.05	0.56	87.51	87.22	86.47
	sd	0.55	0.49	0.06	3.81	4.93	3.66
$M = 20$	mean	3.67	8.53	0.35	46.77	47.53	48.04
	sd	1.27	2.45	0.06	2.07	2.37	1.87
$M = 40$	mean	2.26	6.72	0.27	37.16	36.51	38.50
	sd	0.60	0.77	0.06	1.26	1.35	1.52

Table 7: Running time (in seconds) of Par-D^2-seeding as a function of M. N is set to be $10k$.

threads/processes. Examples of such systems are shared-memory multi-/many-core processors like Intel®Xeon®CPUs and Xeon Phi™coprocessors, and distributed shared memory systems like those using global arrays and partitioned global address space (PGAS) (http://www.pgas.org). In this work we implement the algorithms and conduct experiments on an Intel®Xeon®E5-2680v2 processor which supports upto 40 concurrent threads programmed by C++/OpenMP. The machine has 2 sockets each with 10 cores. Each core can support 2 hyperthreads. So, the machine has effectively 20 cores and 40 threads. Note that this is the reason for a disproportionate running time decrease when M increases from 20 to 40. Moreover there is a serial segment in our algorithm since we choose not to parallelize step 3(f) (k-means++seed step), and this becomes significant when k (and hence N) is large as in case of Birch data-sets. Hence we do not witness linear scaling with increase in number of threads. Finally, we believe that the algorithms proposed in this work are generic and applicable to any other system having shared memory parallelism, and can be easily extended to a message passing paradigm as well.

3.5 Analysis of Par-D^2-Seeding

As far as parallel seeding algorithms for k-means is concerned, the most notable and recent algorithm is the k-means|| algorithm of Bahmani *et al.* [3]. We compare the running time and cost of our parallel algorithm with parallel version of k-means++seed, and k-means||. The pseudocodes for k-means++seed and its parallel version are are given in Algorithm 3.1. Algorithm 3.2 gives the formal description of the k-means|| (Algorithm 2 in [3]) customised for our parallel setting.

The experimental results doing a comparison of cost and running time is given in Table 8. We note that our algorithm does significantly better with respect to the cost. As far as running time is concerned, on the real datasets our algorithm runs faster and this is reversed for the Birch datasets. This is due to large values of k in the Birch datasets. Note that our algorithm runs in k iterations that are inherently sequential. This property is also present in the k-means++seed algorithm. k-means|| was designed precisely to remove this sequential iteration of length k from k-means++seed. So,

```
k-means++seed(X, k) :
  1. $C_0 \leftarrow \{\}$
  2. For $i \leftarrow 1$ to $k$
     (a) Sample a point $p$ from $X$ using
         $D^2$-sampling w.r.t. $C_{i-1}$
     (b) $c_i \leftarrow p$; $C_i \leftarrow C_{i-1} \cup \{c_i\}$
  3. Return $C_k$
```

```
Par-k-means++(X, k) :
  1. $C_0 \leftarrow \{\}$
  2. Let $P_j = \left\{X\left[\frac{(j-1)|X|}{M}\right], ..., X\left[\frac{j|X|}{M}\right]\right\}$ for all $j$
  3. For $i \leftarrow 1$ to $k$
     (a) Parallel-for $j \leftarrow 1$ to $M$
         (i) Sample a point $p_j$ from $P_j$ using $D^2$-sampling
             w.r.t. $C_{i-1}$
     (b) Let $D$ denote the distribution over $\{1, ..., M\}$ s.t.
         $\mathbf{Pr}[j \leftarrow D] = \frac{\Phi_{C_{i-1}}(P_j)}{\Phi_{C_{i-1}}(X)}$
     (c) Sample $j$ from $D$
     (d) $c_i \leftarrow p_j$; $C_i \leftarrow C_{i-1} \cup \{c_i\}$
  4. Return $C_k$
```

Algorithm 3.1: The k-means++seed and Par-k-means++ algorithms. In the parallel implementation, M indicates the number of hardware level threads available.

```
k-means||(X, k, ℓ, r)
  (1) $C \leftarrow$ sample a point uniformly at random from $X$.
  (2) For $i \leftarrow 1$ to $r$:
      (a) $S \leftarrow \{\}$.    S is a multiset.
      (b) Use data-level parallelism as in step 3(b) of Par-$D^2$-seeding to compute $\Phi_C(\{p\})\ \forall p$.
      (c) Parallel-for $j \leftarrow 1$ to $|X|$:
          (i) Add point $X[j]$ to $S$ with with probability $\frac{\ell \cdot \Phi_C(\{X[j]\})}{\Phi_C(X)}$.
      (d) $C \leftarrow C \cup S$.
  (5) For each $c \in C$ compute the weight of $c$ as the number of points in $X$ that has $c$ as the
      closest center in $C$.
  (6) Cluster the weighted set $C$ using a weighted version of k-means++seed and output the
      resulting $k$ centers.
```

Algorithm 3.2: The k-means|| algorithm customized for our parallel setting. Recommended values [3] of ℓ and r are $\ell = 2k$ and $r = 5$. M indicates the number of parallel threads available.

it is not surprising that whenever the value of k is large, the k-means|| algorithm will run faster. However, our algorithm is more targeted towards getting a better quality solution which can also be seen from our experimental results.

4 Conclusion

We suggest an initialization heuristic for k-means obtained by a suitable modification of the sampling based algorithm of Jaiswal *et al.* [7]. Sampling based algorithms, apart from being simple, have the advantage the they are inherently parallel. We give a parallel implementation of our algorithm. We do a comparative analysis on large real and synthetic datasets comparing our results with the state-of-art algorithms. Our experiments show notable cost and running time benefits. We would like to recommend our algorithm as an alternative for the k-means++seed algorithm as an initialization algorithm on any standard multicore platform since our algorithm exploits the available parallelism easily and efficiently to give very good quality solutions.

References

[1] Birch datasets. `http://cs.joensuu.fi/sipu/datasets/`.

[2] David Arthur and Sergei Vassilvitskii. k-means++: the advantages of careful seeding. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 1027–1035, 2007.

[3] Bahman Bahmani, Benjamin Moseley, Andrea Vattani, Ravi Kumar, and Sergei Vassilvitskii. Scalable k-means++. *Proc. VLDB Endow.*, 5(7):622–633, March 2012.

Algorithm		**MNIST**		**CIFAR**		**3D**	
		Cost ($\times 10^{10}$)	Time (in sec.)	Cost ($\times 10^{10}$)	Time (in sec.)	Cost ($\times 10^{6}$)	Time (in sec.)
Par-k-means++	mean	32.42	1.91	80.45	5.68	17.82	0.25
	sd	1.16	0.51	6.43	0.47	6.36	0.03
k-means|| ($\ell = 2k, r = 5$)	mean	31.21	10.68	68.55	34.29	15.72	2.92
	sd	1.02	1.28	2.89	3.51	5.06	0.48
Par-D^2-Seeding ($N = 10k$)	mean	21.23	2.26	53.80	6.72	11.98	0.27
	sd	0.49	0.60	1.33	0.77	2.02	0.06

Algorithm		**Birch1**		**Birch2**		**Birch3**	
		Cost ($\times 10^{12}$)	Time (in sec.)	Cost ($\times 10^{10}$)	Time (in sec.)	Cost ($\times 10^{12}$)	Time (in sec.)
Par-k-means++	mean	189.54	27.39	165.16	26.83	67.78	25.55
	sd	8.76	2.33	21.25	2.27	4.72	3.24
k-means|| ($\ell = 2k, r = 5$)	mean	190.09	9.93	156.26	10.15	67.73	10.21
	sd	10.86	0.65	20.28	0.72	3.43	0.76
Par-D^2-Seeding ($N = 10k$)	mean	120.63	37.16	50.40	36.51	45.66	38.50
	sd	8.76	2.33	1.84	1.35	1.09	1.52

Table 8: Running time and cost for parallel algorithms. The number of machine threads used is $M = 40$.

[4] Moses Charikar, Sudipto Guha, va Tardos, and David B. Shmoys. A constant-factor approximation algorithm for the k-median problem. *Journal of Computer and System Sciences*, 65(1):129 – 149, 2002.

[5] Sanjoy Dasgupta. The hardness of k-means clustering. Technical Report CS2008-0916, Department of Computer Science and Engineering, University of California San Diego, 2008.

[6] Mary Inaba, Naoki Katoh, and Hiroshi Imai. Applications of weighted voronoi diagrams and randomization to variance-based k-clustering: (extended abstract). In *Proceedings of the Tenth Annual Symposium on Computational Geometry*, SCG '94, pages 332–339, New York, NY, USA, 1994. ACM.

[7] Ragesh Jaiswal, Amit Kumar, and Sandeep Sen. A simple D^2-sampling based PTAS for k-means and other clustering problems. *Algorithmica*, 70(1):22–46, September 2014.

[8] Tapas Kanungo, David M. Mount, Nathan S. Netanyahu, Christine D. Piatko, Ruth Silverman, and Angela Y. Wu. A local search approximation algorithm for k-means clustering. In *Proc. 18th Annual Symposium on Computational Geometry*, pages 10–18, 2002.

[9] M. Kaul, Bin Yang, and C.S. Jensen. Building accurate 3D spatial networks to enable next generation intelligent transportation systems. In *Mobile Data Management (MDM), 2013 IEEE 14th International Conference on*, volume 1, pages 137–146, June 2013.

[10] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009.

[11] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. The MNIST database of handwritten digits. `http://yann.lecun.com/exdb/mnist/`.