

Declarative Parallelized Techniques for K-Means Data Clustering

Kittisak Kerdprasop, Surasith Taokok, and Nittaya Kerdprasop

Abstract—The k-means clustering algorithm is an unsupervised learning method for non-hierarchical assigning data points into groups. K-means algorithm performs in an iterative manner the data assignment and central point calculation steps until data points do not move from one group to another. On clustering large datasets, the k-means method spends most of its execution time on computing distances between all data points and existing central points. It is obvious that distance computation of one data point is irrelevant to others. Therefore, data parallelism can be achieved in this case and it is the main focus of this paper. We propose parallel methods, including the approximation scheme, to the k-means clustering. Then demonstrate the implementation of parallelism through the message passing model using a concurrent functional language, Erlang, and also through the multi-threading technique using Prolog. Both Erlang and Prolog are declarative method that efficiently support rapid prototyping. The experimental results of both parallelized implementation techniques show the speedup in computation. The clustering results of an approximated parallel method are impressive in terms of its fast execution time.

Keywords— Parallel k-means, Concurrency, Multi-thread, Functional program, Erlang, Declarative method, Prolog.

I. INTRODUCTION

THE k-means algorithm has been proposed by J.B. MacQueen since 1967 and gained much interest as a data clustering method. Data clustering is an unsupervised learning problem widely studied in many research areas such as statistics, machine learning, data mining, pattern recognition. The objective of clustering process is to partition a mixture of large dataset into smaller groups with a general criterion that data in the same group should be more similar or closer to each other than those in different groups. The clustering problem can be solved with various methods, but the most widely used one is the k-means method [13], [14], [22], [23].

Manuscript received March 10, 2012; Revised version received June 12, 2012. This work was supported by grants from the National Research Council of Thailand (NRCT) and Suranaree University of Technology through the funding of Data Engineering Research Unit.

K. Kerdprasop is with the School of Computer Engineering and Data Engineering Research Unit, Suranaree University of Technology, Nakhon Ratchasima, Thailand (e-mail: KittisakThailand@gmail.com).

S. Taokok is a master student with the School of Computer Engineering and Data Engineering Research Unit, Suranaree University of Technology, Nakhon Ratchasima, Thailand (e-mail: taokok@gmail.com).

N. Kerdprasop is an associate professor and the director of Data Engineering Research Unit, School of Computer Engineering, Suranaree University of Technology, 111 University Avenue, Muang District, Nakhon Ratchasima 30000, Thailand (phone: +66-44-224-432; fax: +66-44-224-602; e-mail: nittaya@sut.ac.th).

The popularity of k-means algorithm is due to its simple procedure and fast convergence to a decent solution. Computational complexity of k-means is $O(nkt)$, where n is the number of data points or objects, k is the number of desired clusters, and t is the number of iterations the algorithm takes for converging to a stable state. To efficiently apply the method to applications with inherent huge number of data objects such as genome data analysis and geographical information systems, the computing process needs improvements.

Parallelization is one obvious solution to this problem and the idea has been proposed [6], [9], [11], [21] since the last two decades. This paper also focuses on parallelizing k-means algorithm, but we base our study on the multi-core architecture. Our focus is on the parallelize implementation techniques using message-passing and multi-threading schemes.

We implement our first extension of the k-means algorithm using Erlang language (www.erlang.org), which uses the concurrent functional paradigm and communicates among hundreds of active processes via a message passing method [1]. To create multiple processes in Erlang, we use a spawn function as in the following example.

```
-module(example1).
-export([start/0]).
start() ->
    Pid1 = spawn(fun run/0),
    io:format("New process ~p~n", [Pid1]),
    Pid2 = spawn(fun run/0),
    io:format("New process ~p~n", [Pid2]).
run() -> io:format("Hello ! ~n", []).
```

The start function in a module example1, which is the main process, creates two processes with identifiers Pid1 and Pid2, respectively. The newly created processes execute a run function that prints the word “Hello !” on the screen. The output of executing the start function is as follows:

```
New process <0.53.0>
Hello !
New process <0.54.0>
Hello !
```

The numbers <0.53.0> and <0.54.0> are identifiers of the newly created two processes. Each process then independently invokes the run function to print out a word “Hello!” on the screen.

The processes in Erlang virtual machine are lightweight and do not share memory with other processes. Therefore, it is an ideal language to implement a large scale parallelizing algorithm. To serve a very large data clustering application, we also propose an approximate method to the parallel k-means. Our experimental results confirm efficiency of the proposed algorithms.

We also propose the second extension of the k-means algorithm using Prolog language. Prolog is a general purpose logic programming language. Many Prolog compilers support parallelization through multi-threading such as SWI-Prolog, SICStus Prolog, CIAO Prolog, and Qu-Prolog. In this paper, we use SWI-Prolog that provides preemptive threads [20] to implement k-means clustering algorithm. SWI-Prolog is an open source and multi-threading support available for Linux, Windows and Macintosh platforms. We can profit multi-thread Prolog by splitting a large task into subtasks that can speedup computation time on multi-core processors.

The organization of the rest of this paper is as follows. Discussion of related work in developing a parallel k-means is presented in Section 2. Our proposed algorithms, a lightweight parallel k-means including the approximation method and a multi-thread parallel k-means, are explained in Section 3. The implementation with a declarative method using Erlang and Prolog languages is demonstrated in Section 4 (program source code is available in the appendix). Experimental results confirming good performance of the proposed algorithms are shown in Section 5. The conclusion as well as future research direction appears as a last section of this paper.

II. RELATED WORK

Serial k-means algorithm was proposed by J.B. MacQueen in 1967 [14] and since then it has gained much interest from data analysts and researchers. The algorithm has been applied to variety of applications ranging from medical informatics [9], genome analysis [15], image processing and segmentation [6], [19], [22] to aspect mining in software design [3]. Despite its simplicity and great success, the k-means method is known to degrade when the dataset grows larger in terms of number of objects and dimensions [7], [10]. To obtain acceptable computational speed on huge datasets, most researchers turn to parallelizing scheme.

Li and Fang [12] are among the pioneer groups on studying parallel clustering. They proposed a parallel algorithm on a single instruction multiple data (SIMD) architecture. Dhillon and Modha [4] proposed a distributed k-means that runs on a multiprocessor environment. Kantabutra and Couch [8] proposed a master-slave single program multiple data (SPMD) approach on a network of workstations to parallel the k-means algorithm. Their experimental results reveal that when on clustering four groups of two dimensional data the speedup advantage can be obtained when the number of data is larger than 600,000. Tian and colleagues [17] proposed the method for initial cluster center selection and the design of parallel k-means algorithm.

Zhang and colleagues [23] presented the parallel k-means with dynamic load balance that used the master/slave model. Their method can gain speedup advantage at the two-dimensional data size greater than 700,000. Prasad [16] parallelized the k-means algorithm on a distributed memory multi-processors using the message passing scheme. Farivar and colleagues [5] studied parallelism using the graphic coprocessors in order to reduce energy consumption of the main processor.

Zhao, Ma and He [24] proposed parallel k-means method based on map and reduce functions to parallelize the computation across machines. Tirumala Rao, Prasad and Venkateswarlu [18] studied memory mapping performance on multi-core processors of k-means algorithm. They conducted experiments on quad-core and dual-core shared memory architecture using OpenMP and POSIX threads. The speedup on parallel clustering is observable.

In this paper, we also study parallelism on the multi-core processors. We investigate the implementation schemes using both threads (in Prolog [2], [20]) and non-threads (in Erlang). The virtual machine that we use in our Erlang experiments employs the concept of message passing to communicate between parallel processes. Each communication carries as few messages as possible. This policy leads to a lightweight process that takes less time and space to create and manage.

III. PARALLELIZED K-MEANS ALGORITHMS

A. Parallel K-Means Based On Message-Passing

Serial k-means algorithm [14] starts with the initialization phase of randomly selecting temporary k central points, or centroids. Then, iteratively assign data to the nearest cluster and then re-calculate the new central points of k clusters. These two main steps are shown in Algorithm 1.

Algorithm 1. Serial k-means

Input: a set of data points and the number of clusters, K
 Output: K -centroids and members of each cluster

Steps

1. Select initial centroid $C = \langle C_1, C_2, \dots, C_K \rangle$
 2. Repeat
 - 2.1 Assign each data point to its nearest cluster center
 - 2.2 Re-compute the cluster centers using the current cluster memberships
 3. Until there is no further change in the assignment of the data points to new cluster centers
-

The serial algorithm takes much computational time on calculating distances between each of N data points and the current K centroids. Then iteratively assign each data point to the closest cluster. We thus improve the computational efficiency by assigning P processes to handle the clustering task on a smaller group of N/P data points. The centroid update is responsible by the master process. The pseudocode of our parallel k-means is shown in Algorithm 2.

Algorithm 2. Parallel k-means (PKM)

Input: a set of data points and the number of clusters, K
 Output: K -centroids and members of each cluster

Steps

1. Set initial global centroid $C = \langle C_1, C_2, \dots, C_K \rangle$
2. Partition data to P subgroups, each subgroup has equal size
3. For each P ,
4. Create a new process
5. Send C to the created process for calculating distances and assigning cluster members
6. Receive cluster members of K clusters from P processes
7. Recalculate new centroid C'
8. If $\text{difference}(C, C')$
9. Then set C to be C' and go back to step 2
10. Else stop and return C as well as cluster members

The PKM algorithm is the master process responsible for creating new parallel processes, sending centroids to the created processes, receiving the cluster assignment results, and recalculating the new centroids. The steps repeat as long as the old and the new centroids do not converge. The convergence criterion can be set through the function $\text{difference}(C, C')$. Communication between the master process and the created processes can be graphically shown in Figure 1.

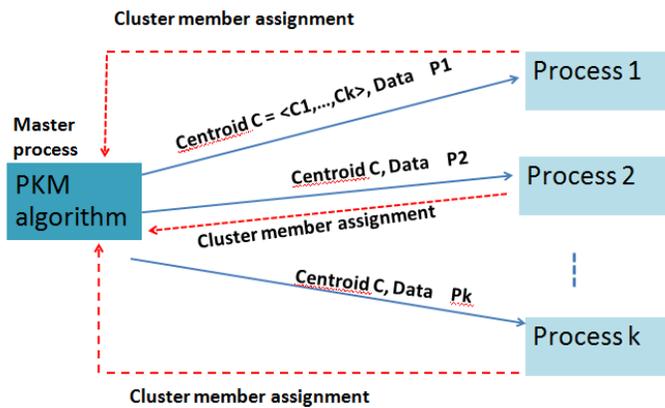


Fig. 1 A diagram illustrating the communication between master and created processes

B. Approximate Parallel K-Means

For the case of very large datasets or streaming data, we also design the approximation method (Algorithm 3) in order to obtain a timely and acceptable result.

Algorithm 3. Approximate parallel k-means (APKM)

Input: a set of data points, the number of clusters (K), and the sample size (S)

Output: approximate K -centroids and cluster members

Steps

1. Set initial centroid $C = \langle C_1, C_2, \dots, C_K \rangle$
2. Sampling data to be of size S
3. Partition S into P subgroups, each subgroup has equal size
4. For each P , create a new process and send C to all processes for calculating distances and assigning cluster members
5. Receive cluster members of K clusters from P processes
6. Recalculate new centroid $C' = \text{average } C$
7. If C' is diverge, then go back to step 2
8. else stop and return C' as well as cluster members

Our approximation scheme is based on the random sampling approach with the basis assumption that the incoming data are uniformly distributed. The data distribution takes other forms (such as Zipf, Gaussian), the proposed algorithm can be easily adapted by changing step 2 of the algorithm APKM to use different approach such as density-biased sampling.

C. Multi-thread Parallel K-Means

For a parallelize scheme using the concept of multi-thread, we design the concurrency process at a finer grain than in the message-passing scheme. The concurrency in this scheme is the distribution of data points in each K cluster to the parallel process. The new central point of each group is then reported to the master process. The graphical scheme can be shown as in Figure 2.

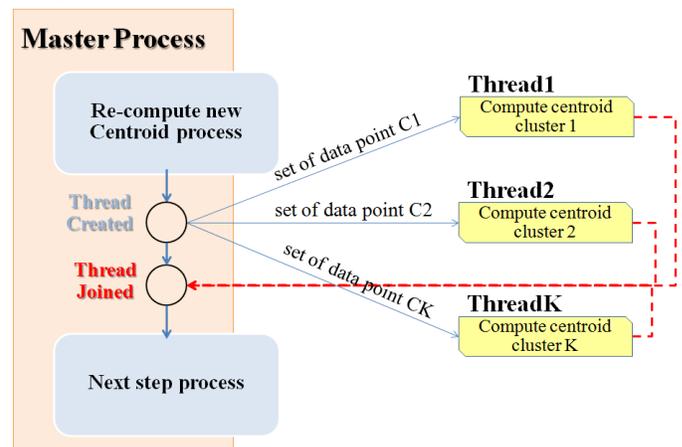


Fig. 2 Process communication in the multi-thread scheme

The multi-thread parallelization steps can be shown in Algorithm 4. The multi-thread k-means (MTK) algorithm asserts the multi-thread process at the centroid re-computation step. This re-computation is the main process responsible for creating threads, sending a set of data points in each cluster along with the thread, and recalculating the new centroids. The re-computation process repeats as long as the old and the new centroids do not converge and multi-threading process will be invoked every time the re-computation process has started.

Algorithm 4. Multi-thread k-means (MTK)

Input: number of clustering and a set of data points

Output: k-centroids and members of each cluster

Steps

1. Select initial centroid $C = \langle C_1, C_2, \dots, C_K \rangle$
 2. Assign each data point to nearest cluster center
 3. Create threads process $T = \langle T_1, T_2, \dots, T_K \rangle$ for centroid $C = \langle C_1, C_2, \dots, C_K \rangle$
 4. For each Thread ($T_{i=1}$ to T_K)
 - 4.1 Re-compute cluster centers $\langle T_i: \text{cal}(C_i) \rangle$
 - 4.2 Return a new centroid C_i to set C'
 5. Check stable of centroids
 - 5.1 if $C \neq C'$ then set $C = C'$ go to step 2
 - 5.2 if $C == C'$ then stop and return C and cluster members
-

IV. IMPLEMENTATION WITH DECLARATIVE METHOD

A. Implementation with Erlang

We implement the serial k-means, the proposed PKM and APKM algorithms with Erlang language. Each process of Erlang does not share memory and it works concurrently in an asynchronous manner. The implementation of PKM and APKM algorithms as an Erlang program is given in appendix.

Some screenshots of compiling and running the program (with Erlang release R13B04) are given in Figures 3 and 4. To compile the program, we use the command:

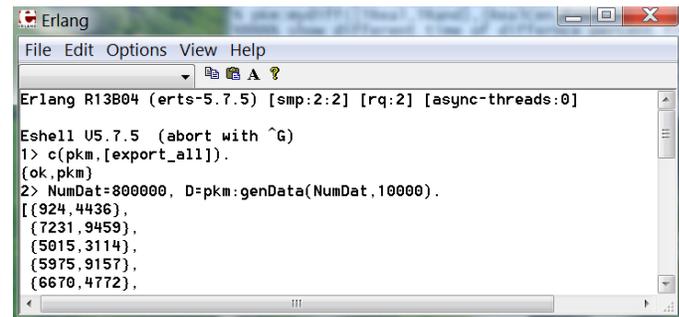
$$c(pka, [export_all]).$$

The first argument, *pka*, is the name of a module. The second argument, *[export_all]*, is a compiler directive meaning that every function in the *pka* module is visible and can be called from the Erlang shell. The second command in Figure 3 calls a function *genData* to generate a synthetic two dimensional dataset containing 800,000 data points with value randomly ranging from 1 to 10,000. Each data point is a tuple, which is a data structure enclosed by curly brackets, and all 800,000 points are contained in a single list.

Data points used in our experiments are randomly generated with the following function:

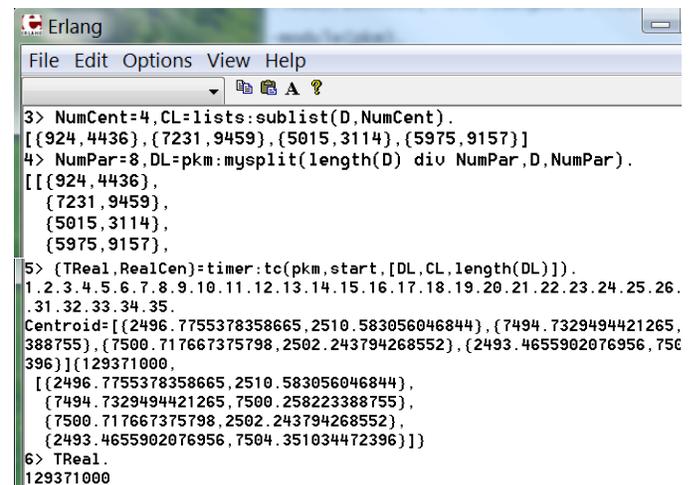
```
genData(0, _) -> [];
genData(Count, Max) ->
    [{uniform(Max), uniform(Max)} |
    genData(Count-1, Max)].
```

A function *genData* takes two arguments: number of data points and maximum value of a data point in each dimension (minimum value is 1 by default). Therefore, the second command in Figure 3 generates 800,000 two-dimensional data points. A data value in each dimension is randomly ranged from 1 to 10,000. For instance, the first generated data point is (924, 4436). All 800,000 data points are stored in a list structure that is represented by bracket symbol.



```
Erlang R13B04 (erts-5.7.5) [smp:2:2] [rq:2] [async-threads:0]
Eshell U5.7.5 (abort with ^G)
1> c(pka, [export_all]).
(ok, pka)
2> NumDat=800000, D=pka:genData(NumDat, 10000).
[[{924, 4436},
 {7231, 9459},
 {5015, 3114},
 {5975, 9157},
 {6670, 4772},
```

Fig. 3 A screenshot to illustrate compiling Erlang program and generating data points



```
3> NumCent=4, CL=lists:sublist(D, NumCent).
[[{924, 4436}, {7231, 9459}, {5015, 3114}, {5975, 9157}]
4> NumPar=8, DL=pka:mypsplit(length(D) div NumPar, D, NumPar).
[[{924, 4436},
 {7231, 9459},
 {5015, 3114},
 {5975, 9157},
 {2496, 7755378358665, 2510, 583056046844},
 {7494, 7329494421265, 7500, 258223388755},
 {7500, 717667375798, 2502, 243794268552},
 {2493, 4655902076956, 7504, 351034472396}]]
5> {TReal, RealCen}=timer:tc(pka, start, [DL, CL, length(DL)]).
1.2.3.4.5.6.7.8.9.10.11.12.13.14.15.16.17.18.19.20.21.22.23.24.25.26.
31.32.33.34.35.
Centroid:[{2496, 7755378358665, 2510, 583056046844},
 {7494, 7329494421265, 7500, 258223388755},
 {7500, 717667375798, 2502, 243794268552},
 {2493, 4655902076956, 7504, 351034472396}]
6> TReal.
129371000
```

Fig. 4 A series of line commands to clustering and recording running time

Figure 4 illustrates creation of four initial centroids (command 3), then partition 800,000 data points into eight subgroups to send to the eight processors (command 4). A parallel k-means starts at command 5. The outputs of parallel k-means shown on a screen are the number of iteration (which is 35 in this example) and the mean points of four clusters. The last command calls a variable *TReal* to display running time of the whole process, which is 129371000 microseconds or 129.371 seconds. This time includes sending and receiving messages between master and the eight concurrent processes.

B. Implementation with Prolog

For the multi-thread scheme of parallelization, we do the implementation of both serial k-means (called KM) and multi-thread k-means (called MTK) as a Prolog program and the source code is also given in Appendix. A screenshot of running the program (SWI-Prolog Multi-threaded, 32 bits, Version 5.10.5) is in Figure 5. To run the program, we use the command:

cluster(K).

The argument K is the number of clusters and before running the program the data file 'points.pl' must exist in working directory. The data format (three dimensions) is as follows:

item([[p1],[p2],...,[pk]]).

or

item([
 $[-4,8,-7],[-9,0,-5],[8,4,4],$
 $[9,5,6],[-4,-5,-7],[-2,-1,3],$
 $[10,11,0],[0,-15,7],[2,-1,3]]).$

A predicate *item([[p1],[p2],...,[pk]])* is a set of data points for clustering with the KM and MTK implementations. The screenshot in Figure 5 shows a command to run the MTK program for two clusters. Time usage is also shown at the bottom line before the true predicate.

```

SWI-Prolog -- c:/Users/ACER/Desktop/2 be a Master/k-means/ex/new_multi-thread_kmeans.pl
File Edit Settings Run Debug Help
Welcome to SWI-Prolog (Multi-threaded, 32 bits, Version 5.10.5)
Copyright (c) 1990-2011 University of Amsterdam, VU Amsterdam
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic), or ?- apropos(Word).

1 ?- cluster(2).
% points.pl compiled 0.00 sec, 12,884 bytes
[[24,0,40],[24,0,48]]
[[35.2,15.2,40],[67.648,51.488,69.632]]
[[42.08,18.56,50.88],[72.19047619047619,57.6,72.68571428571428]]
[[47.68421052631579,22.31578947368421,54.526315789473685],[74.1304347826087,61.56521739130435,74.26086]
[[51.91836734693877,24.897959183673468,56.734693877551024],[75.1604938271605,65.33333333333333,75.6045]
[[75.75172413793103,68.35862068965517,76.9103448275862],[54.608695652173914,27.06086956521739,57.87826]
[[56.28,787401574803148,58.45669291338583],[76.33082706766918,70.43609022556392,78.07518796992481]]
[[76.7741935483871,72.79.09677419354838],[56.94117647058823,30.11764705882353,58.8235294117647]]
[[56.94117647058823,30.11764705882353,58.8235294117647],[76.7741935483871,72.79.09677419354838]]
time:0.06100010899535846
true.
2 ?-

```

Fig. 5 Running the MTK program with 2 clusters

V. EXPERIMENTATION AND RESULTS

A. Performance of Parallel K-Means

We evaluate performances of the proposed PKM and APKM algorithms on synthetic two dimensional dataset. The computational speed of parallel k-means as compared to serial k-means is given in Table 1. Experiments are performed on personal computer with processor speed GHz and GB of memory.

It is noticeable from Table 1 that when dataset is small ($N=50$), running time of parallel k-means is a little bit longer than the serial k-means. This is due to the overhead of spawning concurrent processes. At data size of 900,000 points, running time is unobservable because the machine is out of memory. Running time comparison of parallel against serial k-means is graphically shown in Figure 6. Percentage of running time speedup in is also provided in Figure 7. Speedup advantage is very high (more than 30%) at dataset of size between 50,000 to 200,000 points.

Table 1. Execution time of serial versus parallel k-means clustering

# Data points (N)	Time (Ts, sec) Serial k-means	Time (Tp, sec) Parallel k-means (dual cores)	Time Difference (Ts - Tp) (sec)	Speedup (%)
50	0.000	0.0149	- 0.0149	- 1.49
500	0.031	0.030	0.001	3.23
50,000	8.45	5.03	3.42	40.47
100,000	16.59	10.18	6.40	38.60
200,000	34.03	21.92	12.10	35.58
300,000	66.09	50.92	15.17	22.95
400,000	82.34	63.03	19.31	23.45
500,000	94.67	69.35	25.31	26.73
600,000	113.06	90.18	22.87	20.23
700,000	135.20	101.18	34.01	25.15
800,000	173.67	124.79	48.87	28.14
900,000	N/A	N/A	N/A	N/A

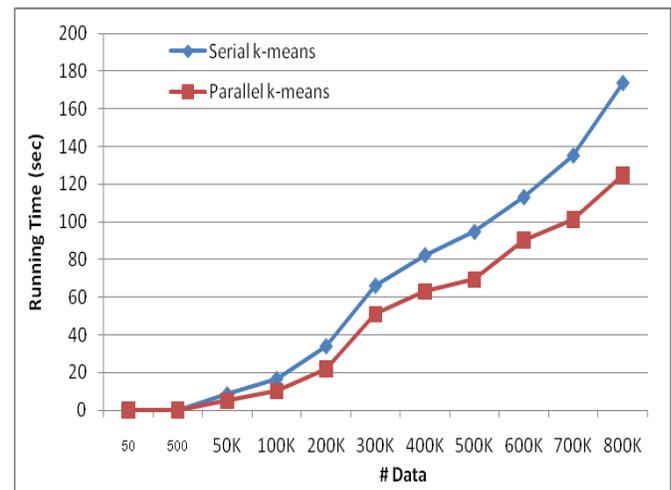


Fig. 6 Running time comparisons of serial versus parallel k-means (PKM)

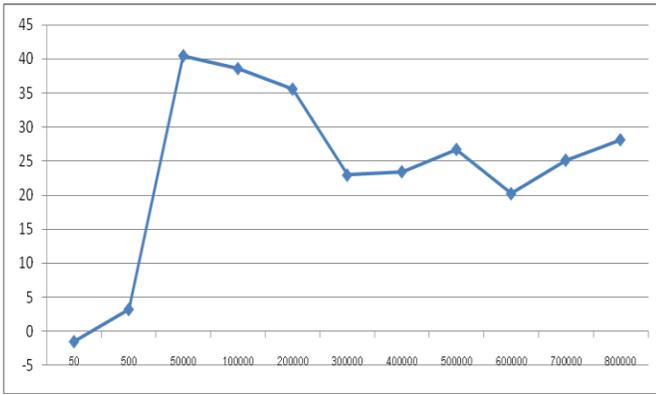


Fig. 7 Percentage of running time speedup at different data sizes

B. Performance of Approximate Parallel K-Means (APKM)

The experiments on APKM algorithm have been conducted to observe running time of a complete dataset (sample size = 100%) versus a reduced sample at different sizes (S). Dataset used in this series of experiments is 500,000 two-dimensional data points, four clusters, and run concurrently with eight processes (running time is 64.67 seconds). At each sample size, the Euclidean distance of centroid shift is also computed by averaging the distances of four centroids that reported as the output of APKM.

We test two schemes of sampling technique. The simple one is a fixed number of samples appeared as the first S records in the data stream. Another tested sampling technique is a randomly picked sample across the dataset (uniform random sampling with replacement). Centroid distance shift of both techniques are reported as “mean shift” in Table 2. The centroid shift may be considered as an error of the approximation method; the lower the distance shift, the better the sampling scheme. It turns out that a simple scheme of picking samples from the first part of dataset performs better than a uniform random sampling across the entire dataset.

Table 2. Performances of approximate parallel k-means

Sample Size (N = 500K)	Sample = the first S records in data stream			Uniform random sampling with replacement		
	Time (sec)	Time reduction (%)	Mean shift	Time (sec)	Time reduction (%)	Mean shift
70%	38.96	39.75	7.07	44.50	31.19	16.45
60%	37.48	42.03	5.78	27.45	57.54	22.11
50%	31.71	50.95	8.83	25.18	61.05	12.62
40%	25.31	60.86	10.73	21.23	67.16	14.15
30%	14.06	78.26	13.64	12.87	80.08	19.26
20%	12.70	80.35	27.49	9.40	85.33	10.49
10%	3.82	94.08	27.19	4.57	92.92	36.61

Percentage execution time speedup of the two sampling schemes is shown in Figure 8. The error (or centroid shift computed from the average Euclidean distances of all mean points) of both schemes to approximate parallel k-means is shown in Figure 9. It can be noticed from the experimental results that the random sampling with replacement scheme gains a little higher percentage of running time reduction than the simple scheme of selection the first S records from dataset.

When compare the clustering error measured as the average centroid shift, the random sampling with replacement scheme shows worse performance than the simple scheme of selection the first S records from the dataset. There is only one exception at the sampling size 20% that random sampling with replacement produces a better result than the simple scheme of selection the first S records. This could happen from the nature of uniform sampling that sometimes a set of good representatives has been selected.

This series of experiments, however, have been conducted on a uniformly distributed data. For other forms of data distribution, the experimental results can be different from the ones reported in this paper.

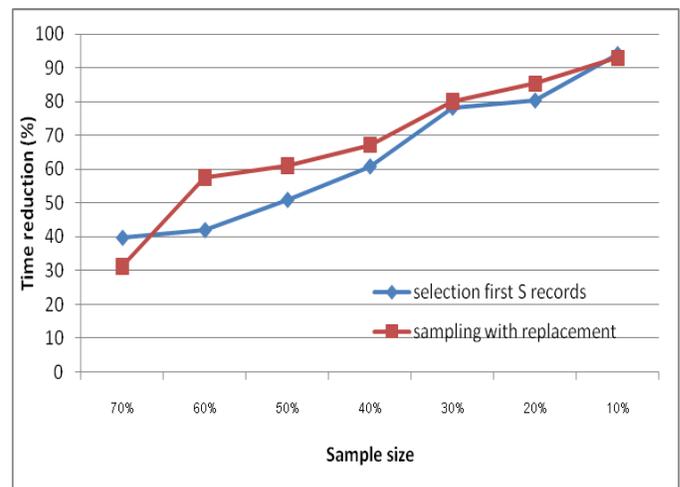


Fig. 8 Time reduction comparison of the two sampling schemes on approximate parallel k-means

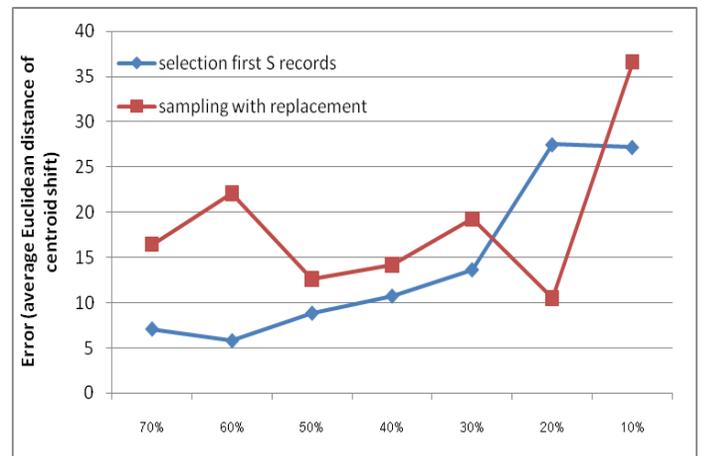


Fig. 9 Error comparison of the two schemes of approximate parallel k-means methods

C. Performance of Multi-thread Parallel K-Means

For the multi-thread scheme, we evaluate performance of the proposed KM and MTK algorithms on synthetic three dimensional dataset. The computational speed of k-means as compared to multi-thread k-means is given in Table 3. Experimentation has been performed on Laptop computer with the processor Intel(R) Core(TM) i5-2410 2.3GHz, 4Gb of memory, and Windows 7 32-bit operating system. The number of synthetic data points is 10,000 points. Running time comparison and percentage of speedup are also shown in Figures 10 and 11, respectively.

Table 3. Execution time of KM versus MTK with 10,000 data points (Number of clusters is equal to the number of threads)

Number of Clusters	Time (seconds)		Speedup (%)
	KM	MTK	
2	2.1	1.4	33.33
3	5.3	3.7	30.19
4	9.0	6.1	32.22
5	12.7	8.3	34.65
6	25.4	17.6	30.71
7	29.9	22.2	25.75
8	32.2	20.9	35.09
9	40.8	26.5	35.05
10	57.1	36.9	35.38

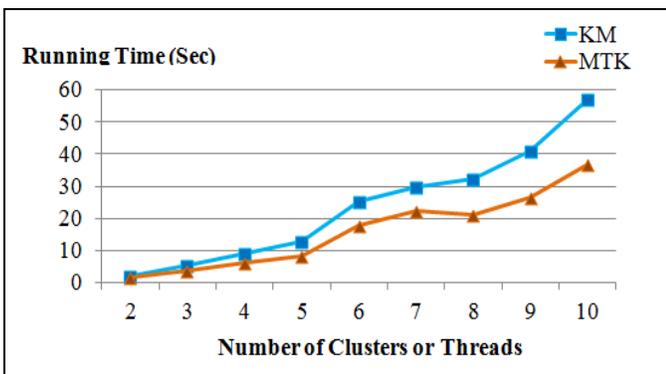


Fig. 10 Running time comparisons of KM versus MTK with 10,000 data points

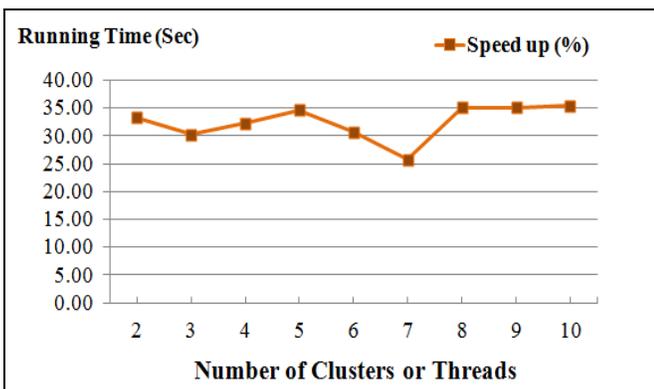


Fig. 11 Percentage of running time speedup on different number of clusters with 10,000 data points

We also test the MTK performance on varying number of data points. We prepare series of data sets ranging from 500, 1000, 2000, 3000, 4000, 5000, 8000, 10000, to 12000 points of 3-dimensional data. The experimentation on these data uses different number of clusters: 2, 4, 6, 8, and 10 clusters. The running results are graphically shown in Figure 12.

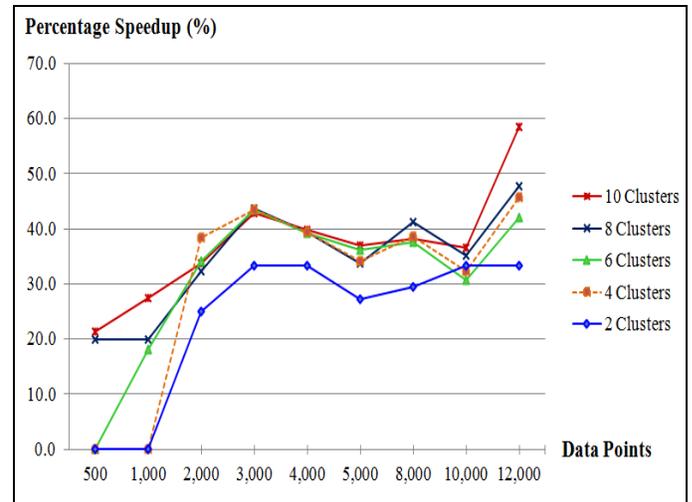


Fig. 12 Percentage of running time speedup of MTK over the serial k-means on different numbers of clusters with varying sizes of three-dimensional data points

VI. CONCLUSION

Clustering similar data points into the same subgroups is now a common task applied in many application areas such as grouping similar functional genomes, segmenting images that demonstrate the same pattern, partitioning web pages showing the same structure, and so on. K-means clustering is the most well-known algorithm commonly used for such tasks. Even though the k-means algorithm is simple, it performs intensive calculation on computing distances between data points and cluster central points. For the dataset with n data points to be grouped into k clusters, each iteration of k-means requires as much as $(n \times k)$ computations. Fortunately, the distance computation of one data point does not interfere the computation of other points. Therefore, parallel computation can be applied to the k-means clustering.

In this paper we propose the design and implementation of three parallel algorithms: PKM, APKM, and MTK. The PKM algorithm parallel the k-means method by partitioning data into equal size and send them to processes that run distance computation concurrently. The parallel programming model used in our implementation is based on the message passing scheme. The APKM algorithm is an approximation method of parallel k-means. Unlike PKM and APKM, the MTK algorithm has been design for the multi-thread parallelization.

The implementation of both serial k-means and the proposed parallel k-means has been done with the declarative method using Erlang and Prolog languages. The experimental results reveal that the proposed three parallel methods can

considerably speedup computation time, especially when the programs have been tested with multi-core processors. The approximation scheme also produces acceptable results in a short period of running time. Our future work will focus on the real applications. We are currently testing our algorithms with the genome dataset and the preliminary outcome is quite promising.

APPENDIX

A. Erlang Programs

Source code presented in this section is in Erlang format. Erlang is a functional language. Each function takes a format:

functionName(Arguments) -> functionBody.

A line preceded with ‘%’ is a comment. We provide two versions of clustering programs: serial k-means and approximate parallel k-means. Each program starts with comments explaining how to compile and run the program.

Serial k—means

```
%-----k-means clustering -----
% data file "points.dat" must exist in working directory
% example of data file:
% [2,7]. [3,6]. [1,6]. [3,7]. [2,6].
% [21,25]. [16,29]. [29,25]. [18,23]. [16,33].
% Then test a program with these commands:
% c(cluster).      %% compile a program
% cluster:go().   %% then run
```

```
-module(cluster).
-export([go/0, clustering/3]).
```

```
go() ->
  {_, DataList} = file:consult("points.dat"),
  file:close("points.dat"),
  kMeans(DataList).
```

```
% -----
% start k-means clustering
% -----
kMeans(PL) ->
  {_,N} = io:read('enter number of clusters:> '),
  % for this example input "2"
  % then select initial centroids
  CL = take(N, PL),
  io:format("~n AllPoints = ~w ~n",[PL]),
  io:format("~n Initial Centroid = ~w~n",[CL]),
  % report data and initial centroids
  % start clustering process with
  % cluster number 1
  % then move on to cluster number 2
  % and so on

  {TT,{Centroid,DataGroup}} = timer:tc(cluster,
  clustering,1,CL,PL),
  T = TT/1000000,
  % record running time and report time
  % in a unit of seconds
  io:format("~n~n__Time for k-means is ~w
  second",[T]),
  io:format("~n~n__Calculated Centroid=~w~n~n",
  [Centroid]),
  printCluster(1, N, DataGroup).
```

```
% .....
% supporting clauses for kMeans
%
% These clauses take first distinct-n element of list
take(0,_) -> [];
take(N,[H|T]) -> [H|take(N-1,T)].
```

```
% to print cluster nicely
printCluster(_,_,[]) -> end_of_clustering;
printCluster(_,0,_) -> end_of_clustering;
printCluster(I,N, [H|T]) ->
  {Centroid, ClusterMember} = H,
  io:format("~n__Cluster:~w Mean point =
  ~w~n",[I,Centroid]),
  io:format(" Cluster member is
  ~w~n",[ClusterMember]),
  printCluster(I+1,N-1,T).
```

```
% -----
% repetitive data clustering
% -----
clustering(N,CL,PL)->
  L1 = lists:map( fun(A) -> nearCentroid(A,CL)
  end,
  PL),
  L2 = transform(CL,L1),
  NewCentroid = lists:map(fun({_,GL}) ->
  findMeans(GL)
  end,
  L2),
  if NewCentroid==CL ->
  io:format("~nNo cluster changes~n"),
  io:format("From Loop1->stop at
  Loop~w~n",[N]),
  {NewCentroid,L2};
  % return new centroids and
  % cluster members as a list L2
  N>=90 -> % max iterations=90
  io:format("Force to stop at Loop
  ~w~n",[N]),
  io:format("Centroid = ~w",
  [NewCentroid]),
  {NewCentroid,L2};
  % return new centroids and
  % cluster members as a list L2

  true -> % default case
  io:format("~nLoop=~w~n",[N]),
  io:format("~nNewCentroid=~w
  ~n",[NewCentroid]),
  clustering(N + 1, NewCentroid, PL)
  end.
  % end if and end clustering function
```

```
% transform a format "Point-CentroidList"
% to "Centroid-PointList"
% example,
% transform([[1]],[{2,[1]},{3,[1]}]).
% --> [{1],[2],[3]} ]
```

```
transform([], _) -> [];
transform([C|TC], PC) ->
  [{C, t1(C, PC)} | transform(TC, PC)].
```

```
t1(_, []) -> [];
t1(C1, [H|T]) ->
  {P,C} = H,
  if C1==C -> [P| t1(C1, T) ];
  C1/=C -> t1(C1, T)
  end.
```

```

% -----
% Given a data point and a centroidList,
% the clause nearCentroid computes a nearest
% centroid and then returns
% a tuple of Point-Centroid
% example:
% nearCentroid( [1], [[2],[3],[45],[1]] ).
% -----> [ [1], [1] ]

nearCentroid(Point, CentroidL)->
  LenList = lists:zip(
    lists:map(fun(A) ->
      distance(Point,A)
    end,
      CentroidL),
    CentroidL),
  [ {_, Centroid} | _ ] = lists:keysort(1,LenList),
  {Point, Centroid}.
% return this tuple to caller

% -----
% compute Euclidean distance
% -----
distance([], []) -> 0;
distance([X1 | T1], [X2 | T2]) ->
  math:sqrt((X2-X1)*(X2-X1) + distance(T1,T2) ).

% -----
% calculate mean point (or centroid)
% -----
% example,
% findMeans([[1,2], [3,4]]). --> [2.0,3.0]

findMeans(PointL) ->
  [H | _] = PointL,
  Len = length(H),
  AllDim = lists:reverse( allDim(Len,PointL) ),
  lists:map(fun(A)-> mymean(A) end, AllDim ).

allDim(0, _) -> [];
allDim(D, L) -> [ eachDimList(D,L) | allDim(D-1,L) ].

eachDimList(_, []) -> [];
eachDimList(N, [H | T]) ->
  [ lists:nth(N, H) | eachDimList(N, T) ].

mymean(L) -> lists:sum(L) / length(L).

% ----- End of Serial k-means program -----
% -----
% Running example:
% -----
% 1> c(cluster).
% {ok,cluster}
% 2> cluster:go().
% enter number of clusters:> 2.
% AllPoints = [[2,7],[3,6],[1,6],[3,7],[2,6],[1,5],[3,5],
% [2,5],[2,6],[1,6],[21,25],[16,29],
% [29,25], [18,23],[16,33],[25,32],
% [20,24],[27,21],[16,21],[19,34]]
% Initial Centroid = [[2,7],[3,6]]
% Loop = 1
% NewCentroid = [ [1.75,6.0],
% [17.75,23.166666666666668]]
% Loop = 2
% NewCentroid = [ [2.0,5.9], [20.7,26.7] ]
% No cluster changes
% From Loop1->stop at Loop3

% -----
% Time for k-means is 0.031 second
% Calculated Centroid=[[2.0,5.9],[20.7,26.7]]
% Cluster:1 Mean point = [2.0,5.9]
% Cluster member is [ [2,7],[3,6],[1,6],[3,7],
% [2,6],[1,5],[3,5],
% [2,5],[2,6],[1,6]]
% Cluster:2 Mean point = [20.7,26.7]
% Cluster member is [ [21,25],[16,29],[29,25],
% [18,23],[16,33],[25,32],
% [20,24],[27,21],[16,21],
% [19,34]]
% end_of_clustering

Approximate parallel k—means
% A parallel k-means program
% Compile program with a command
% c(pkm,[export_all]).
% To unbinding variables from the previous run
% use a command
% f(Var) % means clear Var
% Start experimentation by calling a function
% genData to generate 8000 synthetic data points
% f(), NumDat = 8000,
% D = pkm:genData(NumDat,10000).
% Then identify number of clusters
% f(NumCent), f(CL),
% NumCent = 4,
% CL = lists:sublist(D, NumCent).
% Start parallelization by identifying
% number of data partitions
% f(NumPar), f(DL), NumPar=8,
% DL = pkm:mypsplit(length(D) div NumPar,
% D, NumPar).
% Record running time with the command
% {TReal,RealCen} = timer:tc(pkm,
% start,[DL,CL,length(DL)]).
% Then record the running time of approximate parallel
% k-means (in this example apply 50%
% data sampling scheme)
% f(RDL),
% RDL=pkm:mrand(DL,50),
% {TRand,RandCen} = timer:tc(pkm,
% start,[RDL,CL,length(RDL)]).
% Calculate time difference between the parallel
% k-means and the approximate (50% data points)
% parallel k-means with a command
% pkm:mydiff({TReal,TRand},{RealCen,RandCen}).
% To show different time of different percentages
% from the same Centroid use the following commands
% f(RDL), f(Rand),
% RDL = pkm:mrand(DL,40),
% Rand = pkm:start(RDL,CL,length(RDL)),
% f(RealDL), f(Real),
% RealDL = pkm:mrand(DL,100),
% Real = pkm:start(RealDL,CL,length(RealDL)).
% f(RDL), f(Rand), f(TimeR),
% f(TimeReal), f(Per),
% Per = 40, RDL = pkm:mrand(DL,Per),
% {TimeR,Rand} = timer:tc(pkm,
% start,[RDL,CL,length(RDL)]),
% f(RealDL), f(Real),

```

```

%      RealDL = pkm:mrand(DL,100),
%      {TimeReal,Real} = timer:tc(pkm,
%          start,[RealDL,CL,length(RealDL)]).
%
% To compute percentage of time difference,
% use a command
%   io:format("___For ~w Percent, diff.time =
%       ~w sec,length=~w",
%           [Per, (TimeReal-TimeR) / 1000000,
%             lists:sum(pkm:diffCent(Real,Rand))]).
%
% All of the commands in clustering experimentation
% are also included in the test function
%
-module(pkm).
-import(lists, [seq/2,sum/1,flatten/1,split/2,nth/2]).
-import(io, [format/1,format/2]).
-import(random, [uniform/1]).

%---- for clustering experimentation -----
test(_N Rand) ->
    NumDat = 8000,
    D = pkm:genData(NumDat,10000),
    NumCent = 4,
    CL = lists:sublist(D, NumCent),
    NumPar=8,
    DL=pkm:mysplit(length(D) div NumPar,
        D, NumPar),
    {TReal, RealCen} = timer:tc(pkm,
        start, [DL,CL, length(DL)]),
    RDL = pkm:mrand(DL,50),
    {TRand,RandCen} = timer:tc(pkm,
        start, [RDL,CL, length(RDL)]),
    pkm:mydiff({TReal, TRand},
        {RealCen, RandCen}).

% ---- spawn a new process
% and start the newly created process
% with a function c(Pid)

myspawn(0) -> [] ;
myspawn(N) ->
    [spawn(?MODULE, c, [self()]) | myspawn(N-1) ].

% random sampling without replacement
%
myrand(_, 0) -> [];
myrand(L, Count) ->
    E = nth((uniform(length(L))), L),
    L1 = L -- [E],
    [E | myrand(L1, Count-1)].

% for 100 percent sampling
mrand(L, 100) -> L;
% random in each partition
mrand([], _) -> [];
mrand([HL | TL], X) ->
    [myrand(HL, trunc(length(HL)/(100/X)) ) |
        mrand(TL,X)].

mysend(LoopN, [CidH | CT], Cent, [DataH | DT]) ->
    CidH ! {LoopN, Cent, DataH},
    mysend(LoopN, CT, Cent, DT);

mysend( _, [], _ ,_) -> true.

% Compute difference between centroids
%
diffCent( [H1 | T1], [H2 | T2]) ->
    [ abs(H1-H2) | diffCent(T1,T2) ];

diffCent( [], _ )->[].
mystop( [CH | CT] ) ->
    CH ! stop,
    mystop(CT);
mystop([]) -> true.

myrec( _, 0) -> [];
myrec(LoopN, Count) ->
    receive
        {LoopN, L} -> [L | myrec(LoopN,Count-1) ];
        Another -> self() ! Another % send to myself
    end.

% generate 2 dimensional data points
% example: [{2,76},...]
%
genData(0, _) -> [];
genData(Count, Max) ->
    [ {uniform(Max), uniform(Max)} |
        genData(Count-1, Max)].

mysplit( _, _, 0) -> [];
mysplit(Len, L, Count) ->
    {H, T} = split(Len, L),
    [ H | mysplit(Len, T, Count-1) ].

start( DataL, Cent, NumPar) ->
    CidL = myspawn(NumPar),
    LastC = myloop(CidL,Cent,DataL,NumPar,1),
    format("~nCentroid=~w",[LastC]),
    LastC.

myloop(CidL, Cent, DataL, NumPar, Count) ->
    mysend(Count, CidL, Cent, DataL),
    L = flatten(myrec(Count, NumPar)),
    C_ = calNewCent(Cent, L),
    format("~w.", [Count]),
    if Count > 100 -> mystop(CidL),
        C_ ;
        Cent/= C_ -> myloop(CidL, C_, DataL,
            NumPar, Count+1);
    true -> mystop(CidL),
        C_
    end.

c(Sid) ->
    receive
        stop -> true;
        {LoopN, Cent, Data} -> L = locate(Data,Cent),
            Sid ! {LoopN,L},
            c(Sid)
    end.

calNewCent(Cent, RetL) ->
    LL = group(Cent, RetL),
    avgL(LL).

%---- supplementary functions-----
%
mydiff( {TReal,TRand}, {RealCen,RandCen} ) ->
    { (TReal-TRand)/ 1000000,
        mdiff(RealCen, RandCen) / length(RandCen) }.

mdiff( [ {X,Y}|T1], [ {X1,Y1}|T2] ) ->
    distance({X,Y}, {X1,Y1}) + mdiff(T1,T2);
mdiff([], _ ) -> 0.

group([H | T], RetL) ->
    [ [X | | {X,M} <- RetL , M==H ] | group(T, RetL)];

```

```

group([],_) -> [].

avgL( [HL|TL] ) ->
    N = length(HL),
    [ {sumX(HL) / N, sumY(HL) / N} | avgL(TL)];
avgL([]) -> [].

sumX( [ {X, _} | T ] ) -> X + sumX(T);
sumX([]) -> 0.

sumY( [ {_,Y} | T ] ) -> Y + sumY(T);
sumY([]) -> 0.

locate( [H|T], C ) ->
    NearC = near(H,C),
    [ {H, NearC} | locate(T, C) ];
locate( [], _ ) -> [].

near(H, C ) ->
    mynear(H, C, {0,1000000000} ).

mynear(D, [H|T], {MinC, Min}) ->
    Min_ = distance(D, H),
    if Min>Min_ -> mynear(D, T, {H, Min_} );
    true -> mynear(D, T, {MinC, Min} )
end ;
mynear( _ , [], {MinC, _ } ) -> MinC.

distance( {X, Y}, {X1, Y1} ) ->
    math:sqrt( (X-X1)*(X-X1) + (Y-Y1)*(Y-Y1) ).

% ===== End of Erlang Part =====

B. Prolog Programs

The following source code is in SWI-Prolog format. We
provide two versions of clustering programs: k-means and multi-
thread k-means. Each program starts with comments explaining how
to run the program.

K-means Clustering

% files "points.pl" must exist in working directory
% example of data file:
% item([ [-4,8,-7], [-9,0,-5], [8,4,4], [9,5,6], [-4,-5,-7],
%        [-2,-1,3], [10,11,0],[0,-15,7],[2,-1,3]]).

% Then test a program with this command
% cluster(2). %% use 2 or more

%% -- Reserve memories
:-set_prolog_stack(global,limit(3*10**9)),
set_prolog_stack(local,limit(4*10**9)).

%% -- Main program
cluster(K):-
    ensure_loaded('points.pl'),
    pc_time(H1-M1-S1),
    item(Item),
    initial(Item,K,Mean),
    writeln(Mean),
    kmean(Item,Mean),
    pc_time(H2-M2-S2),
    TS1 is H1*60*60+M1*60+S1,
    TS2 is H2*60*60+M2*60+S2,
    DTS is TS2 - TS1,
    writeln(time-DTS).

%% -- Return the execution times

%% -- example time-15.9
pc_time(CT):-
    get_time(T),
    stamp_date_time(T,
        date( _ , _ , H, M, S, 0, 'UTC' , -),
        'UTC'),
    CT = H-M-S.

%% -- Initial Centroid pick from a set of data lis
initial( _ ,[]):-!.
initial([Hitem|Titem],K,[Hitem|Tmean]):-
    Nk is K - 1,
    initial(Titem,Nk,Tmean).

%% -- K-means work
kmean(Item,Mean):-
    calculate_dist(Item,Mean,CaledItem),
    split_item(Mean,CaledItem,SplitItem),
    calculate_mean(SplitItem,NewMean),
    writeln(NewMean),
    ( Mean = NewMean ->
        true,;
        kmean(Item,NewMean) ),!.

%% -- Calculate distance and assign
%% -- each point to nearest cluster
calculate_dist([],_,[]):-!.
calculate_dist([Hitem|Titem],Mean,[Hitem-
SelMean|TSelMean]):-
    calculating(Hitem,Mean,Dist),
    select_cluster(Dist,Mean,SelMean),
    calculate_dist(Titem,Mean,TSelMean).

%% -- Euclidian distance with 3 Dimensional data
calculating( _ ,[],[]):-!.
calculating([Hi1,Hi2,Hi3],
    [[Hm1,Hm2,Hm3]|Tmean],
    [Dist|Tdist]):-
    Caler is (Hi1-Hm1)^2 +
        (Hi2-Hm2)^2 +
        (Hi3-Hm3)^2,
    sqrt(Caler,Dist),
    calculating([Hi1,Hi2,Hi3],Tmean,Tdist).

%% -- Each point choose nearest cluster
select_cluster( _ ,[Mean],Mean):-!.
select_cluster([Hd1,Hd2|Tdist],
    [Hm1,Hm2|Tmean],
    SelMean):-
    (Hd1 < Hd2 -> select_cluster([Hd1|Tdist],
        [Hm1|Tmean],
        SelMean) ;
        select_cluster([Hd2|Tdist],
            [Hm2|Tmean],
            SelMean) ).

%% -- splited data to classify cluster
split_item([],_,[]):-!.
split_item([HM|Mean],
    CaledItem,[Splited|SplitItem]):-
    spliting(HM,CaledItem,Splited),
    split_item(Mean,CaledItem,SplitItem).

spliting( _ ,[],[]):-!.
spliting(Mean,[Hitem-SelMean|Titem],Splited):-
    spliting(Mean,Titem,TSplited),
    ( Mean = SelMean ->
        Splited = [Hitem|TSplited] ;
        Splited = TSplited).

%% -- Re-compute new Centroid value
calculate_mean([],[]):-!.
calculate_mean([HS|SplitItem],[HR|NewMean]):-

```

```

cal_mean(HS,HR),
calculate_mean(SplitItem,NewMean).
cal_mean(L,R):-
mean_me(0,[0,0,0],L,R).
mean_me(N,[Sx,Sy,Sz],[X,Y,Z] | T,R):-
NN is N + 1,
NSx is Sx + X,
NSy is Sy + Y,
NSz is Sz + Z,
mean_me(NN,[NSx,NSy,NSz],T,R).
mean_me(N,[Sx,Sy,Sz],[RSx,RSy,RSz]):-
RSx is Sx / N,
RSy is Sy / N,
RSz is Sz / N.
%----- End of K-means -----%

```

Multi-thread K-means Clustering

```

%-----K-means Clustering-----%
% data files "points.pl" must exist in working directory
% example of data file:
% item([ [-4,8,-7], [-9,0,-5], [8,4,4], [9,5,6], [-4,-5,-7],
% [-2,-1,3], [10,11,0],[0,-15,7],[2,-1,3]]).
% Then test a program with this command
% cluster(2). %% use 2 or more/ is a number of clusters
%% Reserve memories
:-set_prolog_stack(global,limit(2*10**9)),
set_prolog_stack(local,limit(2*10**9)).
%% -- Main program
cluster(K):-
ensure_loaded('points.pl'),
pc_time(H1-M1-S1),
item(Item),
initial(Item,K,Mean),
writeln(Mean),
kmean(Item,Mean),
pc_time(H2-M2-S2),
TS1 is H1*60*60+M1*60+S1,
TS2 is H2*60*60+M2*60+S2,
DTS is TS2 - TS1,
writeln(time-DTS).

%% -- Return the execution times
%% -- example time-15.9
pc_time(CT):-
get_time(T),
stamp_date_time(T,
date(_,_,_,H,M,S,0,'UTC',-),'UTC'),
CT = H-M-S.
%% -- Initial Centroid pick from a set of data list
initial(0,[]):-!.
initial([Hitem | Titem],K,[Hitem | Tmean]):-
Nk is K - 1,
initial(Titem,Nk,Tmean).
%% -- Multi-thread K-means work
kmean(Item,Mean):-
calculate_dist(Item,Mean,CaledItem),
split_item(Mean,CaledItem,SplitItem),
calculate_mean(SplitItem,TL),
wait_for_threads(TL,NewMean),
writeln(NewMean),
(intersection(Mean,NewMean,Mean) ->
true,! ;
kmean(Item,NewMean) ),!.
%% -- Calculate distance and assign each point
%% -- to nearest cluster
calculate_dist([],[]):-!.
calculate_dist([Hitem | Titem],Mean,[Hitem-
SelMean | TselMean]):-
calculating(Hitem,Mean,Dist),

```

```

select_cluster(Dist,Mean,SelMean),
calculate_dist(Titem,Mean,TselMean).
%% -- Euclidian distance with 3 Dimensional data
calculating(.,[],[]):-!.
calculating([Hi1,Hi2,Hi3],
[[Hm1,Hm2,Hm3] | Tmean],
[Dist | Tdist]):-
Caler is (Hi1-Hm1)^2 +
(Hi2-Hm2)^2 +
(Hi3-Hm3)^2,
sqrt(Caler,Dist),
calculating([Hi1,Hi2,Hi3],Tmean,Tdist).
%% -- Each point choose nearest cluster
select_cluster(.,[],[Mean],Mean):-!.
select_cluster([Hd1,Hd2 | Tdist],
[Hm1,Hm2 | Tmean],
SelMean):-
(Hd1 < Hd2 -> select_cluster([Hd1 | Tdist],
[Hm1 | Tmean],
SelMean) ;
select_cluster([Hd2 | Tdist],
[Hm2 | Tmean],
SelMean) ).
%% -- splited data to classify cluster
split_item([],[]):-!.
split_item([HM | Mean],
CaledItem,[Splited | SplitItem]):-
splitting(HM,CaledItem,Splited),
split_item(Mean,CaledItem,SplitItem).
splitting(.,[],[]):-!.
splitting(Mean,[Hitem-SelMean | Titem],Splited):-
splitting(Mean,Titem,TSplited),
(Mean = SelMean ->
Splited = [Hitem | TSplited] ;
Splited = TSplited).
%% -- Re-compute new Centroid value
%% -- In this section create on thread
%% -- per one cluster re-computer new centroid
calculate_mean([],[]):-!.
calculate_mean([HS | SplitItem],[T0 | TL1]):-
calculate_mean(SplitItem,TL1),
thread_create(cal_mean(HS), T0, []).
cal_mean(L):-
mean_me(0,[0,0,0],L,R),
assert(mean(R)).
mean_me(N,[Sx,Sy,Sz],[X,Y,Z] | T,R):-
NN is N + 1,
NSx is Sx + X,
NSy is Sy + Y,
NSz is Sz + Z,
mean_me(NN,[NSx,NSy,NSz],T,R).
mean_me(N,[Sx,Sy,Sz],[RSx,RSy,RSz]):-
RSx is Sx / N,
RSy is Sy / N,
RSz is Sz / N.
%% -- Wait for all thread completed work.
wait_for_threads([],[]):-!.
wait_for_threads([T | TL],NewMean) :-
(thread_join(T, true) ->
mean(NM),
retract(mean(NM)),
wait_for_threads(TL,NewMean),
NewMean = [NM | TM] ;
wait_for_threads([T | TL],NewMean) ).
% ===== End of Prolog part =====

```

REFERENCES

- [1] J. Armstrong, *Programming Erlang: Software for a Concurrent World*, Raleigh, North Carolina, The Pragmatic Bookshelf, 2007.
- [2] M. Carro and M. Hermenegildo, "Concurrency in Prolog using threads and a shared database," *Proceedings of International Conference on Logic Programming*, 1999, pp.320-334.
- [3] G. Czibula, G. Cojocar, and I. Czibula, "Identifying crosscutting concerns using partitioned clustering," *WSEAS Transactions on Computers*, Vol.8, Issue 2, February 2009, pp. 386-395.
- [4] I. Dhillon and D. Modha, "A data-clustering algorithm on distributed memory multiprocessors," *Proceedings of ACM SIGKDD Workshop on Large-Scale Parallel KDD Systems*, 1999, pp. 47-56.
- [5] R. Farivar, D. Rebolledo, E. Chan, and R. Campbell, "A parallel implementation of k-means clustering on GPUs," *Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, 2008, pp. 340-345.
- [6] B. Hohlt, "Pthread parallel k-means," *UC Berkeley*, 2001, pp. 1-9.
- [7] M. Joshi, "Parallel k-means algorithm on distributed memory multiprocessors," *Technical Report*, University of Minnesota, 2003, pp. 1-12.
- [8] S. Kantabutra and A. Couch, "Parallel k-means clustering algorithm on NOWs," *NECTEC Technical Journal*, Vol.1, No.6, 2000, pp. 243-248.
- [9] N. Kerdprasop and K. Kerdprasop, "Knowledge induction from medical databases with higher-order programming," *WSEAS Transactions on Information Science and Applications*, Vol.6, Issue 10, October 2009, pp. 1719-1728.
- [10] K. Kerdprasop, N. Kerdprasop, and P. Sattayatham, "Weighted k-means for density-biased clustering," *Lecture Notes in Computer Science*, Vol.3589, Data Warehousing and Knowledge Discovery (DaWaK), August 2005, pp. 488-497.
- [11] N. Kerdprasop and K. Kerdprasop, "A lightweight method to parallel k-means clustering," *International Journal of Mathematics and Computers in Simulations*, Vol. 4, issue 4, 2010, pp. 144-153.
- [12] X. Li and Z. Fang, "Parallel clustering algorithms," *Parallel Computing*, Vol.11, Issue 3, 1989, pp. 275-290.
- [13] C. Li and T. Wu, "A clustering algorithm for distributed time-series data," *WSEAS Transactions on Systems*, Vol. 6, Issue 4, April 2007, pp. 693-699.
- [14] J. MacQueen, "Some methods for classification and analysis of multivariate observations," *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability*, 1967, pp. 281-297.
- [15] F. Othman, R. Abdullah, N. Abdul Rashid, and R. Abdul Salam, "Parallel k-means clustering algorithm on DNA dataset," *Proceedings of the 5th International Conference on Parallel and Distributed Computing: Applications and Technologies (PDCAT)*, 2004, pp. 248-251.
- [16] A. Prasad, "Parallelization of k-means clustering algorithm," *Project Report*, University of Colorado, 2007, pp. 1-6.
- [17] J. Tian, L. Zhu, S. Zhang, and L. Liu, "Improvement and parallelism of k-means clustering algorithm," *Tsinghua Science and Technology*, Vol. 10, No. 3, 2005, pp. 277-281.
- [18] S. Tirumala Rao, E. Prasad, and N. Venkateswarlu, "A critical performance study of memory mapping on multi-core processors: An experiment with k-means algorithm with large data mining data sets," *International Journal of Computer Applications*, Vol.1, No.9, pp. 90-98.
- [19] H. Wang, J. Zhao, H. Li, and J. Wang, "Parallel clustering algorithms for image processing on multi-core CPUs," *Proceedings of International Conference on Computer Science and Software Engineering (CSSE)*, 2008, pp. 450-53.
- [20] J. Wielemaker, "Native preemptive threads in SWI-Prolog," *ICLP. Volume 2916 of Lecture Notes in Computer Science*, 2003, pp. 331-345.
- [21] H. Xiao, "Towards parallel and distributed computing in large-scale data mining: A survey," *Technical Report*, Technical University of Munich, 2010, pp. 1-30.
- [22] Z. Ye, H. Mohamadian, S. Pang, and S. Iyengar, "Contrast enhancement and clustering segmentation of gray level images with quantitative information evaluation," *WSEAS Transactions on Information Science and Applications*, Vol.5, Issue 2, February 2008, pp. 181-188.
- [23] Y. Zhang, Z. Xiong, J. Mao, and L. Ou, "The study of parallel k-means algorithm," *Proceedings of the 6th World Congress on Intelligent Control and Automation*, 2006, pp. 5868-5871.
- [24] W. Zhao, H. Ma, and Q. He, "Parallel k-means clustering based on MapReduce," *Proceedings of the First International Conference on Cloud Computing (CloudCom)*, 2009, pp. 674-679.

Kittisak Kerdprasop is an associate professor at the School of Computer Engineering and one of the principal researchers of Data Engineering Research Unit, Suranaree University of Technology, Thailand. He received his bachelor degree in Mathematics from Srinakarinwirot University, Thailand, in 1986, master degree in computer science from the Prince of Songkla University, Thailand, in 1991 and doctoral degree in computer science from Nova Southeastern University, USA, in 1999. His current research includes Data mining, Machine Learning, Artificial Intelligence, Logic and Functional Programming, Probabilistic Databases and Knowledge Bases.

Surasith Taokok is currently a master student with the School of Computer Engineering, Suranaree University of Technology, Thailand. He received his bachelor degree in computer engineering from Suranaree University of Technology, Thailand, in 2007. His research topic is related to parallelization techniques, data clustering, data mining, and declarative programming.

Nittaya Kerdprasop is an associate professor and the director of Data Engineering Research Unit, School of Computer Engineering, Suranaree University of Technology, Thailand. She received her B.S. in radiation techniques from Mahidol University, Thailand, in 1985, M.S. in computer science from the Prince of Songkla University, Thailand, in 1991 and Ph.D. in computer science from Nova Southeastern University, U.S.A., in 1999. She is a member of IAENG, ACM, and IEEE Computer Society. Her research of interest includes Knowledge Discovery in Databases, Data Mining, Artificial Intelligence, Logic and Constraint Programming, Deductive and Active Databases.