A lightweight method to parallel k-means clustering

Kittisak Kerdprasop and Nittaya Kerdprasop

***Abstract*—**Traditional k-means clustering iteratively performs two major steps: data assignment and calculating the relocation of mean points. The data assignment step sends each data point to a cluster with closest mean, or centroid. Normally, the measure of closeness is the Euclidean distance. On clustering large datasets, the k-means method spends most of its execution time on computing distances between all data points and existing centroids. 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 the parallel method as well as its approximation scheme to the k-means clustering. The parallelism is implemented through the message passing model using a concurrent functional language, Erlang. The experimental results show the speedup in computation of parallel k-means. The clustering results of an approximated parallel method are impressive when taking into account its fast running time.

***Keywords*—**Parallel k-means, lightweight process, concurrent functional program, Erlang.

I. Introduction

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 [10], [11], [18], [19].

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 [9] since the last two decades. This paper also focuses on parallelizing k-means algorithm, but we base our study on the multi-core architecture. We implement our 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.

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 and the approximation method, are explained in Section 3. The implementation (a complete source code is available in the appendix) and experimental results are demonstrated in Section 4. The conclusion as well as future research direction appears as a last section.

II. Related Work

A serial k-means algorithm was proposed by J.B. MacQueen in 1967 [11] and since then it has gained mush interest from data analysts and computer scientists. The algorithm has been applied to variety of applications ranging from medical informatics [7], genome analysis [12], image processing and segmentation [16], [18], to aspect mining in software design [2]. 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 [5], [8]. To obtain acceptable computational speed on huge datasets, most researchers turn to parallelizing scheme.

Li and Fang [9] 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 [3] proposed a distributed k-means that runs on a multiprocessor environment. Kantabutra and Couch [6] 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 [14] proposed the method for initial cluster center selection and the design of parallel k-means algorithm.

Zhang and colleagues [19] 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 [13] parallelized the k-means algorithm on a distributed memory multi-processors using the message passing scheme. Farivar and colleagues [4] studied parallelism using the graphic coprocessors in order to reduce energy consumption of the main processor.

Zhao, Ma and He [20] proposed parallel k-means method based on map and reduce functions to parallelize the computation across machines. Tirumala Rao, Prasad and Venkateswarlu [15] 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, but our implementation does not rely on threads. The virtual machine that we use in our 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. Proposed Algorithms

A. Parallel k-Means

Serial k-means algorithm [11] 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 Algorithm1.

```
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 = <$C_1$, $C_2$, …, $C_K$>
    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 = <$C_1$, $C_2$, …, $C_K$>
    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 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 *difference(C, C')*. Communication between the master process and the created processes can be graphically shown in Fig. 1.

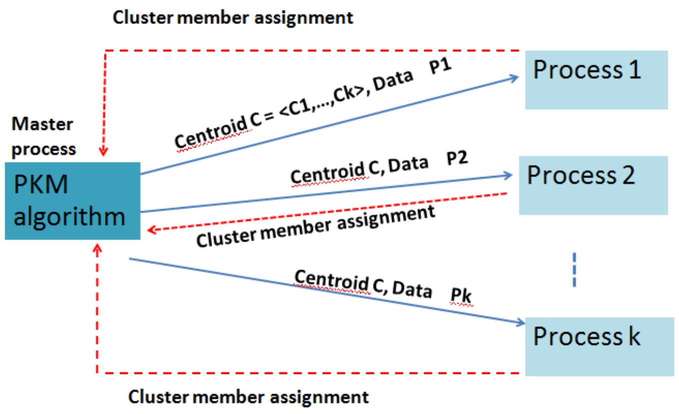

Cluster member assignment
Master process
PKM algorithm
Centroid C = <C1,...,Ck>, Data P1
Process 1
Centroid C, Data P2
Process 2
Cluster member assignment
Centroid C, Data Pk
Process k
Cluster member assignment

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 = <$C_1$, $C_2$, …, $C_K$>
  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' = 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.

IV. Implementation and Experimental Results

A. Implementation with Erlang

We implement the proposed 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 Figs. 2 and 3. To compile the program, we use the command

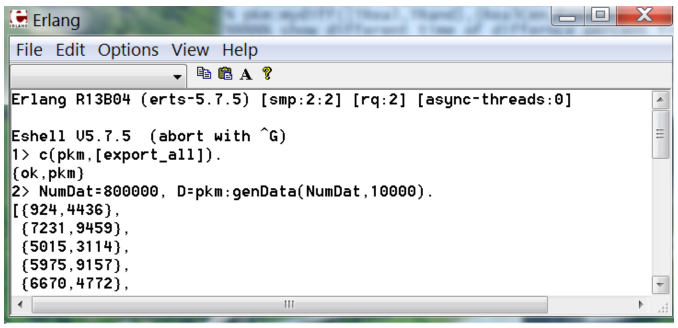
c(pka, [export_all]).

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 Fig. 2 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)].
```

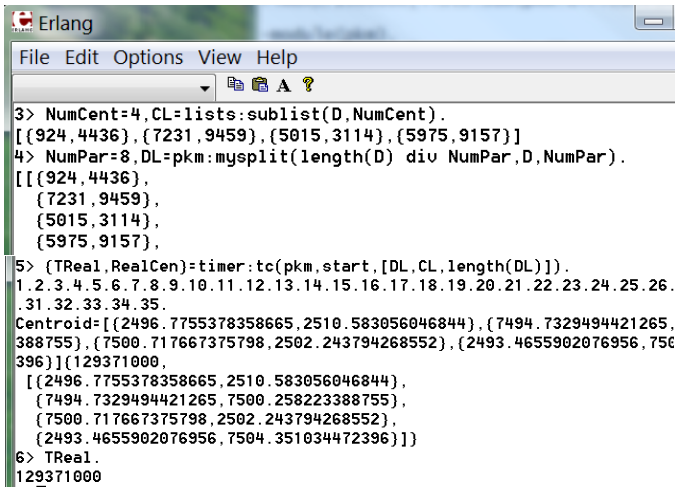
A function genData takes two arguments: number of data points and maximum value of a data point in each dimension (minimum value is 1 by default). Therefore, the second command in Fig. 2 generates 800,000 two-dimensional data points. A data value in each dimension is randomly ranged from 1 to 10,000. 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 V5.7.5  (abort with ^G)
1> c(pkm,[export_all]).
{ok,pkm}
2> NumDat=800000, D=pkm:genData(NumDat,10000).
[{924,4436},
 {7231,9459},
 {5015,3114},
 {5975,9157},
 {6670,4772},
```

Fig. 2 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=pkm:mysplit(length(D) div NumPar,D,NumPar).
[[{924,4436},
  {7231,9459},
  {5015,3114},
  {5975,9157},
5> {TReal,RealCen}=timer:tc(pkm,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,
388755},{7500.717667375798,2502.243794268552},{2493.4655902076956,750
396}]{129371000,
 [{2496.7755378358665,2510.583056046844},
  {7494.7329494421265,7500.258223388755},
  {7500.717667375798,2502.243794268552},
  {2493.4655902076956,7504.351034472396}]}
6> TReal.
129371000
```

Fig. 3 A series of line commands to clustering and recording running time

The screenshot in Fig. 3 illustrates commands to create 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 call a variable *TReal* to display the 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.

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

B. *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 Table1. 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 Fig. 4. Percentage of running time speedup in is also provided in Fig. 5. Speedup advantage is very high (more than 30%) at dataset of size between 50,000 to 200,000 points.

C. *Performance of Approximate Parallel k-Means*

The experiments on approximated parallel k-means 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).

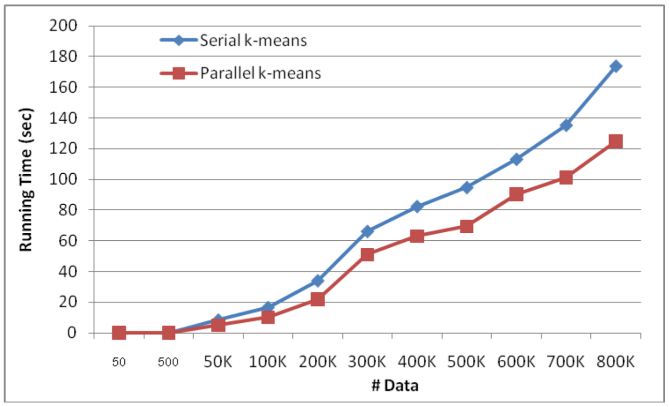


Fig. 4 Running time comparisons of serial versus parallel k-means

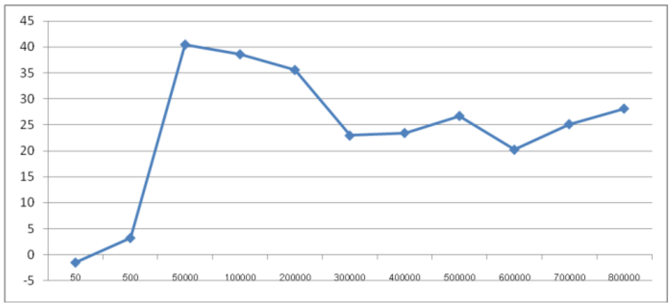


Fig. 5 Percentage of running time speedup at different data sizes

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 approximate parallel k-means.

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.

Percentage execution time speedup of the two sampling schemes is shown in Fig. 6. 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 Fig. 7. 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.

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

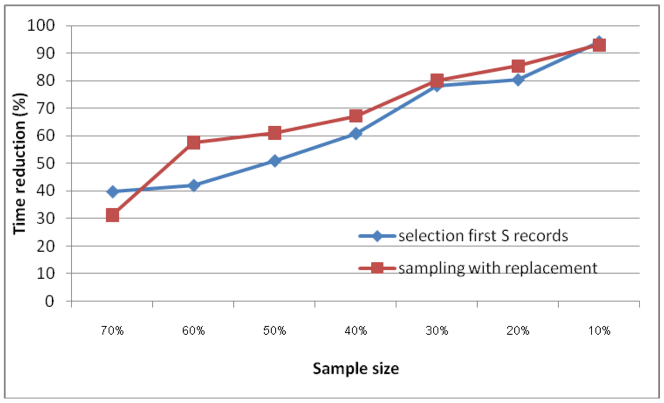


Fig. 6 Time reduction comparison of the two sampling schemes on approximate parallel k-means

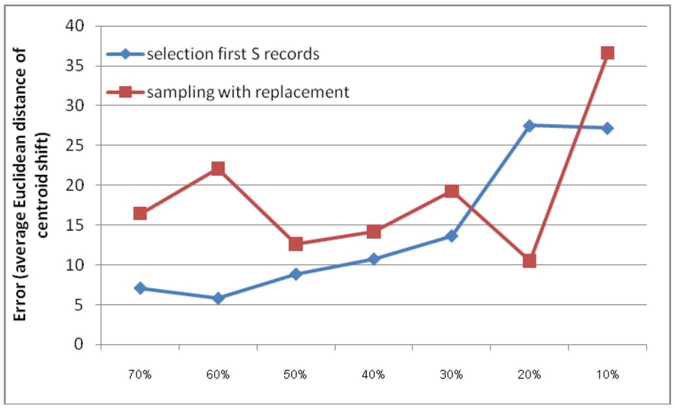


Fig. 7 Error comparison of the two schemes of approximate parallel k-means methods

V. Conclusion

Data clustering 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 clustering data.

The k-means algorithm is simple but it performs intensive calculation on computing distances between data points and cluster central points. For the dataset with n data points and 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, k-means clustering is a good candidate for parallelism.

In this paper we propose the design and implementation of two parallel algorithms: PKM and APKM. 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. We design this algorithm for streaming data applications.

The experimental results reveal that the parallel method considerably speedups the computation time, especially with 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

Source codes presented in this section are 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 firts  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:mysplit(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(_NRand) ->
          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) ).
```

ACKNOWLEDGMENT

This research has been funded by grants from the National Research Council of Thailand (NRCT) and the Thailand Research Fund (TRF, grant number RMU5080026). The DEKD Research Unit has been supported by Suranaree University of Technology.

REFERENCES

[1] J. Armstrong, *Programming Erlang: Software for a Concurrent World*, Raleigh, North Carolina, The Pragmatic Bookshelf, 2007.

[2] G. Czibula, G. Cojocar, and I. Czibula, Identifying crosscutting concerns using partitional clustering, *WSEAS Transactions on Computers*, Vol.8, Issue 2, February 2009, pp. 386-395.

[3] 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.

[4] 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.

[5] M. Joshi, Parallel k-means algorithm on distributed memory multiprocessors, *Technical Report*, University of Minnesota, 2003, pp. 1-12.

[6] S. Kantabutra and A. Couch, Parallel k-means clustering algorithm on NOWs, *NECTEC Technical Journal*, Vol.1, No.6, 2000, pp. 243-248.

[7] 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.

[8] 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.

[9] X. Li and Z. Fang, Parallel clustering algorithms, *Parallel Computing*, Vol.11, Issue 3, 1989, pp. 275-290.

[10] 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.

[11] 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.

[12] 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.

[13] A. Prasad, Parallelization of k-means clustering algorithm, *Project Report*, University of Colorado, 2007, pp. 1-6.

[14] J. Tian, L. Zhu, S. Zhang, and L. Liu, Improvement and parallelism of k-means clustering algorithm, *Tsignhua Science and Technology*, Vol. 10, No. 3, 2005, pp. 277-281.

[15] 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.

[16] 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.

[17] H. Xiao, Towards parallel and distributed computing in large-scale data mining: A survey, *Technical Report*, Technical University of Munich, 2010, pp. 1-30.

[18] 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.

[19] 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.

[20] W. Zhao, H. Ma, and Q. He, Parallel k-means clustering based on MapReduce, *Proceedings of the First International Conference on Cloud Computiong (CloudCom)*, 2009, pp. 674-679.

Kittisak Kerdprasop is an associate professor at the school of computer engineering, Suranaree University of Technology. His current address is School of Computer Engineering, Suranaree University of Technology, 111 University Avenue, Nakhon Ratchasima 30000, Thailand, Tel. +66-44-224349, e-mail address KittisakThailand@gmail.com.

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, Artificial Intelligence, Functional Programming, Computational Statistics.

Nittaya Kerdprasop is an associate professor at the school of computer engineering, Suranaree University of Technology, Thailand. She received her B.S. 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, USA, in 1999. She is a member of ACM and IEEE Computer Society. Her research of interest includes Knowledge Discovery in Databases, AI, Logic Programming, Deductive and Active Databases.