Parallelized Rough K-means Clustering with Erlang Programming

Weerasak Chongnguluam, Kanjana Intharachatorn, Prapatsorn Sinahawattana, and
Nittaya Kerdprasop, *Member, IAENG*

***Abstract*—Currently multi-core processors have been available on most personal computers. To get the maximum benefit of computational power from the multi-core architecture, we need a new design on existing algorithms and software. We propose the parallelization of the rough k-means clustering algorithm. In the rough k-means clustering algorithm, each cluster has been formed regarding the two approximations, a lower and an upper approximation. To make the rough k-means clustering be better parallelized, we employ Erlang as a language for concurrent programming. Sending and receiving messages between a master and the concurrently created process of the Erlang language are done in an asynchronous manner. Therefore, the implementation can be highly parallel and fault tolerant. The experimental results demonstrate considerable speedup rate of the proposed parallel rough k-means clustering method, compared to the serial rough k-means approach.**

***Index Terms*—Rough clustering, parallel rough k-means, Erlang, concurrent functional language**

I. Introduction

CLUSTERING algorithm is an unsupervised learning. In clustering methodologies, data are partitioned 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 method most widely used is the k-means method. But rough clustering produces different solutions because of the possibility of multiple cluster membership of objects.

Rough clustering is an extension of the theory of rough or approximation sets, introduced by Pawlak [4]. In the rough k-means clustering algorithm proposed by Lingras [3], each cluster has two approximations, a lower and an upper approximation. The lower approximation is a subset of the upper approximation. The members of the lower approximation belong to any other cluster. The data objects in an upper approximation may belong to the cluster. Since their membership is uncertain they must be member of an upper approximation of at least another cluster. New centroid of the rough k-means algorithm depends on three parameters *Wl*, *Wu* and ε threshold.

Parallelization is one obvious solution to this problem and several ideas have been proposed since the last two decades. This paper focuses on parallelizing rough k-means algorithm using Erlang language, which uses the concurrent functional paradigm and communicates among hundreds of active processes via a message passing method. 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.

The organization of the rest of this paper is as follows. Discussion of related work in rough clustering and parallel cluster in other algorithm is presented in Section 2. Our proposed algorithm, a lightweight parallel rough clustering is explained in Section 3. The implementation of the proposed algorithm as Erlang functions and experimental results are demonstrated in Section 4. The source code of our implementation is also provided in the appendix section. The conclusion as well as future research direction appears as the last section of this paper.

II. Related Work

Rough set theory introduced by Zdzislaw Pawlak [4] is a new mathematical tool to deal with vagueness and uncertainty. Rough set theory applied to the clustering method can take advantage of the multi-core processors in that programs implementing its methods may easily run on parallel computers. Some refinements of rough k-means clustering were proposed by Georg Peters in 2005 [5]. He analyze Lingras algorithm [3] with respect to objective function, numerical stability, the stability of the clusters and others. He suggests two solutions to tackle the problem of initial parameter selection in rough k-means clustering. Pawan Lingras [3] presented combination of the efficiency of rough K-means algorithm with the ability of genetic algorithms to find a near optimal solution based on a cluster quality measure.

The study of parallel k-means algorithm [8] adopted data parallel strategy and master/slave model in a parallel manner. Zhang and Colleagues [8] introduced dynamic load balance for enhancing the efficiency of parallel K-means. Their experiments demonstrate that presented parallel K-

Manuscript received December 18, 2010; revised January 20, 2011. This work was supported by the National Research Council of Thailand (NRCT) and Suranaree University of Technology.

W. Chongnguluam is a master student with the school of Computer Engineering, Suranaree University of Technology, Thailand (corresponding author, phone: +66-(0)86-677-4904; fax: +66-(0)44-224602; e-mail: singpor@gmail.com).

K. Intharachatorn is a master student with the school of Computer Engineering, Suranaree University of Technology, Thailand (e-mail: b4900313kanjana@ gmail.com).

P. Sinahawattana is a master student with the school of Computer Engineering, Suranaree University of Technology, Thailand (e-mail: prapatsorn.sin@gmail.com).

N. Kerdprasop is an associate professor with the school of Computer Engineering and the principal researcher of the Data Engineering and Knowledge Discovery (DEKD) research unit, Suranaree University of Technology, Thailand (e-mail: nittaya@sut.ac.th).

ISBN: 978-988-19251-2-1
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

means has higher efficiency and general usage. Prasad [6] modified the method to initial centroid selection by using electrostatic data partitioning and used node merging algorithm. Tian and colleagues [7] presented a refined initial cluster centers method and a parallel k-means scheme.

Li and Fang [2] presented two parallel clustering algorithms on a single instruction multiple data (SIMD) architecture. Kittisak Kerdprasop and Nittaya Kerdprasop [1] proposed the parallelization of the well-known k-means clustering algorithm by employing a single program multiple data (SPMD) approach based on a message passing model.

In this paper, we propose implementation of the rough k-means clustering algorithm in parallel by using Erlang Programming. We compare the performances of the parallel rough k-means clustering with the serial rough k-means clustering. We expect that the performance of parallel rough k-mean clustering algorithm is better than serial rough k-mean clustering algorithm.

III. Parallel Rough K-means Clustering Algorithm

In sequential rough k-means clustering, the process takes much time in determining the distance between each data points and in the step of calculating new mean points for each cluster. Both the distance and mean calculation processes must be repeated several times, so we design our algorithm to improve the performance by dividing the work of both of these processes into P processes, and then calculate the values from each P processes before running in the next step. Division processes of calculation into P process are the basis of the parallel programming and we will call our algorithm the Parallel Rough K-Means (PRKM) algorithm. The pseudo code of the PRKM algorithm is shown in Figure 1 and the program source codes are provided in appendix.

INPUT: data set,
number of clusters, K, and
Wl, *Wb*, and ε threshold constants
Output: K-centroids,
lower approximation and upper approximation of each cluster

Steps:
1. Set initial centroid C = <C1, C2, ..., Ck>
2. Partition data into P subgroups
3. For each P,
4. Create a new process for calculate distance
5. Send result back to parent process
6. Receive distances and initial members of lower initialize of K clusters from P
7. Calculate upper approximation of each member in K clusters
8. Recalculate new centroid C'
9. If difference(C, C')
10. Then set C to be C' and go back to step 3
11. Else stop and return C as well as cluster members

Fig.1. Parallel Rough K-Means (PRKM) algorithm

The PRKM algorithm is the main process responsible for creating new parallel processes, sending centroids to the created processes and receiving the cluster distances. The steps repeat as long as the old and new centroids do not converge or do not reach the maximum iteration limit.

IV. Implementation and Experimental Results

We implement our algorithm with the Erlang language (http://www.erlang.org, release R14B). Each process of Erlang is a lightweight process. Thai is each process does not use shared memory and running in the concurrently manner. Main part of the PRKM implementation as an Erlang program is presented in Figure 2. Main function of the given program is the function *go()*. That dataset to be used in this clustering program is stored in the *sample4.txt* file. The two definitions of the *rough_cluster* functions are responsible for the clustering method.

```
13 rough_cluster(DataSet,T,Wl,Wu,N) ->
14   Means = [E ||
15       E <- lists:zip(lists:seq(1,N),
16           get_centroid(DataSet,N))],
17   DataSetList = mysplit(DataSet,8,
18       (length(DataSet) div 8),[]),
19   {Time,Result} = timer:tc(?MODULE,
20       rough_cluster, [DataSetList,N,T,
21           Means,Wl,Wu,100,0]).
22
23 rough_cluster(_,_,_,Means,_,_,0,_) ->
24   Means;
25 rough_cluster(DataSet,N,T,Means,Wl,Wu,
26   I,Count) -> Closest =
27     determine_closest2(DataSet,Means),
28   Approx =
29     determine_approximation(Closest,
30         Means,T,
31         [{K,{[],[]}} || K <- lists:seq(1,N)]),
32   NewMeans =
33     calculate_means_rough(Approx,N,
34         Wl,Wu,[]),
35   AllTrue =
36       lists:all(fun(X)->lists:member(X,
37         Means) end,NewMeans),
38   if AllTrue == true -> NewMeans;
39       true -> rough_cluster(DataSet,N,T,
40         NewMeans,Wl,Wu,I-1,Count+1)
41   end.
42 go() -> {_,N} =
43   io:read("enter number of clusters:> "),
44   {ok,DataSet} =
45       file:consult("sample4.txt"),
46   rough_cluster(DataSet,0.2,0.7,0.3,N).
```

Fig.2. Main module of PRKM program coding in Erlang language

ISBN: 978-988-19251-2-1
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

```
Eshell V5.8.1  (abort with ^G)
1> roughcluster_parallel:go().
enter number of clusters:> 4.
Loop=0
Loop=1
Loop=2
Loop=3
Loop=4
Loop=5
{9712938,
 [{1,[7552.514983497094,2871.9109998336967]},
  {2,[2625.996081532821,7873.6143174056]},
  {3,[7584.151573466377,7787.356411575578]},
  {4,[2558.9672124867084,2832.520280533398]}]}
```

Fig.3. Compiling and running the PRKM program

The screenshot of compiling and running the program is shown in Figure 3. The program is within the module *roughcluster_parallel*. Therefore, invoking the *go()* function has to be done via the command *roughcluster_parallel:go()*.

In the experiments we have identified a number of processes to be 8 processes and set the values of *Wl* to be 0.7, *Wb* to be 0.3, ε threshold to be 0.2. The third line in figure 3 shows a command to divide the dataset into 4 clusters. When the program has finished, it will show the time used for calculations (9,712,938 microseconds or 9.712938 seconds as shown in the tenth line of figure 3), together with the new centroid of each cluster at the end of the clustering process. In the experiment we generate synthetic two-dimensional data. The final output of our clustering program is therefore the two-dimensional centroid, or central point, of each cluster.

Each data point is stored as a list, then it was divided into 8 parts and then it will be calculated for the rough k-means clustering. The data used in our experiments will be randomly generated by determining the value of the two-dimensional data point in the range of 1 to 10000. The number of experimental data were 50, 500, 1000, 10000, 50000, 100000, and 500000. The execution time of sequential rough k-means versus the parallel scheme, PRKM, is shown in Table 1. We perform a series of experiments with the laptop computer Intel Core 2 Duo 2.26GHz/core with 3GB memory space.

Table1. The execution time of sequential rough k-means versus parallel rough k-means (PRKM)

Data	Time(ms) Serial K-mean	Time(ms) Parallel K-mean	Time(ms) Difference
50	6.013	22.128	-16.115
500	126.354	109.799	16.555
1,000	174.218	291.600	-117.382
10,000	11,181.424	10,907.456	273.968
50,000	59,867.389	53,808.399	6,058.990
100,000	121,865.557	111,117.071	10,748.486
500,000	637,863.006	588,623.303	49,239.703

From the experimental results shown in Table 1, it can be noticed that when the number of data were small (50 and 1000 data points), the parallel program run slower than the sequential one. This may be the results of the overhead in the parallel initialization step to spawn concurrent processes. The effect of running time saving is observable when the number of data points is large (more than 100,000 two-dimensional data points). Running time comparison of parallel versus sequential rough k-means is graphically shown in Figure 4. Percentage of running time speedup is also provided in Figure 5.

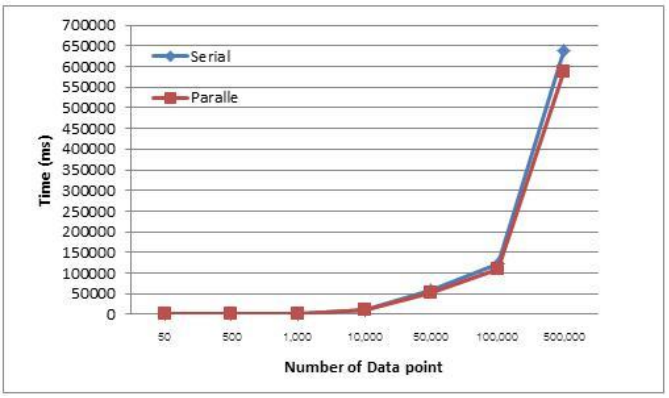

Fig.4. Running time comparison of parallel versus sequential rough k-means

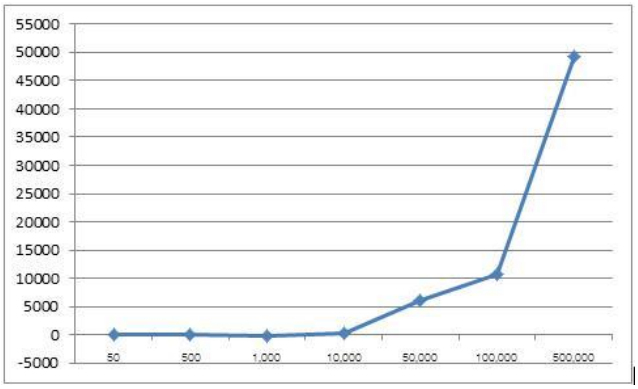

Fig.5 Percentage of running time speedup at different data sizes

V. Conclusion

Rough k-means clustering is a clustering method that is discovered recently base on rough set theory to help in approximate clustering with data with uncertain membership states. The clustering results are the lower and upper approximation. Rough clustering is gaining popularity for the potential use in Web mining and Web usage mining.

In this paper, we propose the algorithm and the implementation of rough k-means clustering algorithm in parallel style by using Erlang Programming language. Parallelization is achieved by dividing data into groups equally and then dividing processes to find the distances and the mean of each cluster.

The results show that by working parallel, the speed of calculation increases. The speed is increased significantly as compared to the sequential program when the number of data increases. Our future work will focus on the real applications in web mining.

ISBN: 978-988-19251-2-1
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

APPENDIX

```
-module(roughcluster_parallel).
-compile(export_all).

mysplit(DataSet,_Num,NumPart,Result)
   when   length(DataSet) =< NumPart ->
      lists:reverse([DataSet|Result]);
mysplit(DataSet,0,_NumPart,Result) ->
   lists:reverse([DataSet|Result]);
mysplit(DataSet,Num,NumPart,Result) ->
   {Part,Rest} = lists:split(NumPart,DataSet),
   mysplit(Rest,Num-1,NumPart,[Part|Result]).

rough_cluster(DataSet,T,Wl,Wu,N) ->
   Means = [E || E <- lists:zip(lists:seq(1,N),
              get_centroid(DataSet,N))],
      DataSetList =
         mysplit(DataSet,8,(length(DataSet) div 8),[]),
      {Time,Result} =
         timer:tc(?MODULE,rough_cluster,
                  [DataSetList,N,T,Means,Wl,Wu,100,0]).

rough_cluster(_,_,_,Means,_,_,0,_) -> Means;
rough_cluster(DataSet,N,T,Means,Wl,Wu,I,Count) ->
   Closest = determine_closest2(DataSet,Means),
   Approx = determine_approximation(Closest,Means,T,
              [{K,{[],[]}} || K <- lists:seq(1,N)]),
   NewMeans  =
      calculate_means_rough(Approx,N,Wl,Wu,[]),
   AllTrue =
      lists:all(fun(X)->lists:member(X,Means)end,
         NewMeans),
   if AllTrue == true ->   NewMeans;
      true -> rough_cluster(DataSet,N,T,NewMeans,
               Wl,Wu,I-1,Count+1)
      end.
go() ->
         {_,N} = io:read("enter number of clusters:> "),
         {ok,DataSet} = file:consult("sample4.txt"),
          rough_cluster(DataSet,0.2,0.7,0.3,N).
get_centroid(DataSet,Cluster)     ->
   get_centroid(DataSet,Cluster,[]).

get_centroid(_DataSet,0,R) -> lists:reverse(R);
get_centroid([],_,R) -> lists:reverse(R);
get_centroid([H|T],N,R) ->
   case lists:member(H,R) of
      true -> get_centroid(T,N,R);
      _     -> get_centroid(T,N-1,[H|R])
   end.

random_assign_approximation([],_,Result) -> Result;
random_assign_approximation([H|T],K,Result) ->
   N = random:uniform(K),
   {LowerApp,UpperApp} = proplists:get_value(N,Result),
   NewResult = [{N,{[H|LowerApp],UpperApp}} |
                proplists:delete(N,Result)],
random_assign_approximation(T,K,NewResult).

calculate_means([]) -> 0;
calculate_means(H) ->
   [HR|_T] = H,
   Dim = length(HR),
   GroupByDim = [[lists:nth(N,L) || L <- H] ||
                  N <- lists:seq(1,Dim)],
   [lists:sum(G)/length(G) || G <- GroupByDim].

calculate_means_rough(_,0,_,_,Result) -> Result;
calculate_means_rough(Cluster,K,Wl,Wb,Result) ->
   {LowerApp, UpperApp} =
      proplists:get_value(K,Cluster),
   Mk = case UpperApp of
      []      -> calculate_means(LowerApp);
      _Else ->
         LMeans =
            [Wl*M || M <- calculate_means(LowerApp)],
         UMeans =
            [Wb*M || M <- calculate_means(UpperApp)],
         [L+U || {L,U} <- lists:zip(LMeans,UMeans)]
      end,
   NewMean = [{K,Mk}|Result],
   calculate_means_rough(Cluster,K-1,Wl,Wb,NewMean).

euclidean_distance(X1,X2) -> math:pow((X1-X2),2).

distance_cluster(Xn,Mk) ->
   {K,M} = Mk,
   {K,math:sqrt(lists:sum(lists:map(fun({X1,X2}) ->
      euclidean_distance(X1,X2) end, lists:zip(Xn,M))))}.

min_distance(Xn,Means) ->
   Dist = [distance_cluster(Xn,Mk) || Mk <- Means],
      [FirstDist |_Rest] = Dist,
      {H,_} = lists:foldl(fun({K1,M1},{K2,M2}) ->
                  if M1 < M2 -> {K1,M1};
                     true      -> {K2,M2}
                  end
                end,FirstDist,Dist),H.


determine_closest([],_Means,Result) -> Result;
determine_closest([H|T],Means,Result) ->
   Min = min_distance(H,Means),
   {Lower,Upper} =
      proplists:get_value(Min,Result,{[],[]}),
   NewResult = [{Min,{[H|Lower],Upper}}
                  | proplists:delete(Min,Result)],
   determine_closest(T,Means,NewResult).

determine_closest2(DataSetList,Means) ->
   lists:foreach(fun(H) ->
      spawn(?MODULE,determine_closest_process,
            [self(),H,Means])
      end, DataSetList),
   determine_closest_response(length(DataSetList),
      length(Means),[]).
```

ISBN: 978-988-19251-2-1
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

```
determine_closest_process(Parent,DataList,Means) ->
   Parent ! determine_closest3(DataList,Means,[]).

determine_closest3([],_Means,Result) -> Result;
determine_closest3([H|T],Means,Result) ->
   Min = min_distance(H,Means),
   determine_closest3(T,Means,[{Min,H}|Result]).

determine_closest_response(0,LMean,Result) ->
   FlattenResult = lists:flatten(Result),
      lists:map(fun(M) ->
         Lowers = proplists:get_all_values(M,FlattenResult),
                  {M,{Lowers,[]}}
      end,
      lists:seq(1,LMean));

determine_closest_response(N,LMean,Result) ->
   receive
      R ->
         determine_closest_response(N-1,LMean,[R|Result])
   end.


determine_set_T(H,Mh,Means,Epsilon) ->
   lists:filter(fun(Mk) ->
      {_,Dist1} = distance_cluster(H,Mk),
      {_,Dist2} = distance_cluster(H,Mh),
      ((Dist1 - Dist2) =< Epsilon)
      end,
   Means).

determine_approximation([],_,_,Result) -> Result;
determine_approximation([H|T],Means,Epsilon,Result) ->
   {K,Data} = H,
   {Lower,_} = Data,
   Mh = proplists:get_value(K,Means),
   OtherMeans = proplists:delete(K,Means),
   NewResult = determine_set_T(Lower,{K,Mh},
                  OtherMeans,Epsilon,Result),
   determine_approximation(T,Means,Epsilon,NewResult).

determine_set_T([],_,_,_,Result) -> Result;
determine_set_T([H|T],Mh,Means,Epsilon,Result) ->
   {K,_} = Mh,
   SetT = determine_set_T(H,Mh,Means,Epsilon),

      NewResult = if
         SetT == [] ->
            {Lower,Upper}=
               proplists:get_value(K,Result,{[],[]}),
               [{K,{[H|Lower],Upper}} |
               proplists:delete(K,Result)];
         true ->
            upper_assign_approximation(H,[Mh|SetT],Result)
      end,
   determine_set_T(T,Mh,Means,Epsilon,NewResult).

upper_assign_approximation(_Data,[],Result) -> Result;
upper_assign_approximation(Data,[Mt|T],Result) ->
   {K,_} = Mt,
   {Lower,Upper}= proplists:get_value(K,Result,{[],[]}),
      NewResult = [{K,{Lower,[Data|Upper]}}
                  | proplists:delete(K,Result)],
      upper_assign_approximation(Data,T,NewResult).
```

REFERENCES

[1] K. Kerdprasop and N. Kerdprasop, "Parallelization of K-means Clustering on Multi-core Processors", in *Proceedings of 10th WSEAS International Conference on Applied Computer Science*, Japan, October 2010, pp.472-477.

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

[3] P. Lingras, "Evolutionary Rough K-Means Clustering", Department of Mathematics and Computing Science, Saint Mary's University, Halifax, 2009, pp. 68-75.

[4] Z. Pawlak, "Rough sets", *International Journal of Information and Computer Science*, Vol. 11, 1982, pp. 145–172.

[5] G. Peters, "Some refinements of rough k-means clustering", Department of Computer Science/Mathematics, Munich University of Applied Sciences, 2006.

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

[7] 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.

[8] Y. Zhang, Z. Xiong, J. Mao, and L. Ou, "The study of parallel k-means algorithm", in *Proceedings of the 6th World Congress on Intelligent Control and Automation*, 2006, pp. 5868-5871.

ISBN: 978-988-19251-2-1
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)