Parallelization of K-Means Clustering on Multi-Core Processors

Kittisak Kerdprasop and Nittaya Kerdprasop
Data Engineering and Knowledge Discovery (DEKD) Research Unit
School of Computer Engineering, Suranaree University of Technology
111 University Avenue, Muang District, Nakhon Ratchasima 30000
THAILAND
KittisakThailand@gmail.com, nittaya@sut.ac.th

Abstract: - Multi-core processors have recently 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. In this paper we propose the parallelization of the well-known k-means clustering algorithm. We employ a single program multiple data (SPMD) approach based on a message passing model. Sending and receiving messages between a master and the concurrently created process 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 k-means clustering method, compared to the serial k-means approach.

Key-Words: - Parallel k-means, Multi-core processing, Concurrent programming, Erlang, Functional language, Clustering

1 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].

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]. 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 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 as Erlang functions and experimental results are demonstrated in Section 4. The conclusion as well as future research direction appears as a last section of the paper.

2 Related Work

A serial k-means algorithm was proposed by J.B. MacQueen in 1967 [10] and since then it has gained much interest from data analysts and computer scientists. The algorithm has been applied to variety of applications ranging from medical informatics [7], genome analysis [11], image processing and segmentation [15, 17], 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 [13] proposed the method for initial cluster center selection and the design of parallel k-means algorithm.

Zhang and colleagues [18] 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 [12] 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 [19] proposed parallel k-means method based on map and reduce functions to parallelize the computation across machines. Tirumala Rao, Prasad and Venkateswarlu [14] 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.

3 Proposed Algorithm

3.1 Parallel K-Means

The serial k-means algorithm [10] 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 1.

```
Algorithm 1. 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 \( C = \langle C_1, C_2, ..., C_K \rangle \)
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 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 \( \text{difference}(C, C') \).

3.2 Approximate Parallel K-Means

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

```
Algorithm 2. 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 their members

Steps
1. Set initial centroid \( C = \langle C_1, C_2, ..., C_K \rangle \)
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.

4 Implementation and Experiments

We implement the proposed algorithms with Erlang language (release R13B04). 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 Figure 1. A screenshot of compiling and running the program are given in Figure 2.

```erlang
start(DataList, Centroid, NumPar) ->
    CidL = myspawn(NumPar),
    LastC = myloop(CidL, Centroid, DataList, NumPar, 1),
    io:format("-nCentroid=-w", [LastC]),
    LastC.
```

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

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

near(H, C) ->
    distance(H, C, [0, 10000000000]).

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, Min]) ->
    MinC.

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

myloop(CidL, Cent, DataL, NumPar, Count) ->
    mysend(Count, CidL, Cent, DataL),
    L = flatten(mynear(Count, NumPar)),
    C = calNewCent(Cent, L),
    io:format("-w", [Count]),
    if Count > 100 ->
        mystop(CidL, C_);
        Cent/=C_ ->
            myloop(CidL, C_, DataL, NumPar, Count+1);
        true_ ->
            mystop(CidL, C_)
    end.
```

Fig. 1. Some main functions of PKM and APKM

Fig. 2. Compiling and running the pkm program

Figure 2 shows screenshots of line commands to compile and run the program. Command 1, `c(pka, [export_all])`, compiles the program with the option to export all functions of the program. Command 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. An Erlang function to generate data is as follows:

```erlang
MyData(Count, Max) ->
    [uniform(Max), uniform(Max)]
    genData(Count, Max).
```

Each data point is a tuple and all 800,000 points are contained in a single list. Command 3 then partitions data points into eight subgroups to send to the eight processors (command 4), parallel k-means starts at command 5 by calling the function `start`. It is called by included in the function `timer` that is used for recording running time. The outputs of parallel k-means 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 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 concurrent processes.

We evaluate performances of the proposed PKM and APKM algorithms on synthetic two dimensional dataset, four clusters, run concurrently on eight processes. The computational speed of parallel k-means as compared to serial k-means is given in Table 1. Experiments are performed on a desktop computer AMD Athlon 64 X2 Dual-Cores Processor 2.2 GHz and 1 GB of memory.
Table 1. The execution time of serial k-means versus parallel k-means

<table>
<thead>
<tr>
<th># Data points (N)</th>
<th>Time (Ts, sec) Serial k-means</th>
<th>Time (Tp, sec) Parallel k-means (dual cores)</th>
<th>Time Difference (Ts – Tp) (sec)</th>
<th>Speedup (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>0.0000</td>
<td>0.0149</td>
<td>- 0.0149</td>
<td>- 1.49</td>
</tr>
<tr>
<td>500</td>
<td>0.031</td>
<td>0.030</td>
<td>0.001</td>
<td>3.23</td>
</tr>
<tr>
<td>50,000</td>
<td>8.45</td>
<td>5.03</td>
<td>3.42</td>
<td>40.47</td>
</tr>
<tr>
<td>100,000</td>
<td>16.59</td>
<td>10.18</td>
<td>6.40</td>
<td>38.60</td>
</tr>
<tr>
<td>200,000</td>
<td>34.03</td>
<td>21.92</td>
<td>12.10</td>
<td>35.58</td>
</tr>
<tr>
<td>300,000</td>
<td>66.09</td>
<td>50.92</td>
<td>15.17</td>
<td>22.95</td>
</tr>
<tr>
<td>400,000</td>
<td>82.34</td>
<td>63.03</td>
<td>19.31</td>
<td>23.45</td>
</tr>
<tr>
<td>500,000</td>
<td>94.67</td>
<td>69.35</td>
<td>25.31</td>
<td>26.73</td>
</tr>
<tr>
<td>600,000</td>
<td>113.06</td>
<td>90.18</td>
<td>22.87</td>
<td>20.23</td>
</tr>
<tr>
<td>700,000</td>
<td>135.20</td>
<td>101.18</td>
<td>34.01</td>
<td>25.15</td>
</tr>
<tr>
<td>800,000</td>
<td>173.67</td>
<td>124.79</td>
<td>48.87</td>
<td>28.14</td>
</tr>
<tr>
<td>900,000</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

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

Fig. 3. Running time comparisons of serial versus parallel k-means

Fig. 4. Percentage of running time speedup at different data sizes

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). 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 samples 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

<table>
<thead>
<tr>
<th>Sample Size (N=500K)</th>
<th>Sample = the first S records in data stream</th>
<th>Uniform random sampling with replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time (sec)</td>
<td>Time Reduction (%)</td>
</tr>
<tr>
<td>70%</td>
<td>38.96</td>
<td>39.75</td>
</tr>
<tr>
<td>60%</td>
<td>37.48</td>
<td>42.03</td>
</tr>
<tr>
<td>50%</td>
<td>31.71</td>
<td>50.95</td>
</tr>
<tr>
<td>40%</td>
<td>25.31</td>
<td>60.86</td>
</tr>
<tr>
<td>30%</td>
<td>14.06</td>
<td>78.26</td>
</tr>
<tr>
<td>20%</td>
<td>12.70</td>
<td>80.35</td>
</tr>
<tr>
<td>10%</td>
<td>3.82</td>
<td>94.08</td>
</tr>
</tbody>
</table>
Percentage execution time speedup of the two sampling schemes is shown in Figure 5. 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 6. 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.

5 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 structure, 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×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 when 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.

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

References:


