A Grid-based Approach to Continuous Clustering of Moving Objects

Tongyu Zhu, Yuan Zhang, Weifeng Lv, Fei Wang
State Key Laboratory of Software Development Environment
Beihang University, Beijing, China
{zhutongyu, yuanzhang, lwf, wangfei }@nlsde.buaa.edu.cn

Abstract—**With the rapid advances in wireless devices and positioning technologies, tracking and clustering of moving objects has drawn increasing attention. Previous methods of clustering moving objects merge clusters by searching all the existing clusters, which have an obvious decline in efficiency as the number of clusters increases. This paper proposes a grid-based approach to continuous clustering of moving objects. We first employ dynamic grid to narrow the searching area when merging clusters, and then develop an efficient split algorithm to handle the split of clusters, which avoids multiple splits of one cluster during a period of time. At last, a comprehensive experimental evaluation has been conducted to validate our approach, and the results indicate the efficiency and effectiveness of our algorithm, especially for large data set.**

Keywords- Moving object; clustering; grid; data mining

I. INTRODUCTION

Due to the growing popularity of wireless devices (e.g., PDAs, mobile phones, navigation devices) and the rapid advances in wireless communication and positioning technologies (e.g., GPS), tracking the behaviors and movements of individuals becomes increasingly available, which boosts various kinds of services exploiting knowledge of object movement.

Clustering analysis aims to group similar data into the same group and different data into distinct groups, which provides a summary of data distribution patterns and discovers data correlations in dataset. Early clustering techniques mainly focused on analyzing static datasets [1], [2], [3]. Recently, clustering moving objects has attracted increasing attention [4], [5], [6], [7] which has various kinds of applications, including mobile computing, targeted sales, traffic jam prediction, weather forecast and animal migration analysis.

As the positions of moving objects continuously change, treating moving objects as static ones and periodically clustering them with the methods for static datasets is a brute-force approach which does not consider the information of the movement. Some incremental clustering schemes have been proposed [4], [6], [7], and they mainly focus on dynamically maintaining a small set of *moving micro-clusters* (MMCs) [4]. The concept of MMC is a group of objects that are not only close to each other at current time, but also likely to move together for a while.

The *split* and *merge* operations are central parts of the schemes for incremental clustering of moving objects. As has been observed in the literature, there are two kinds of methods to deal with the split of a MMC. One is to delete the *extreme* object [4] or the farthest object from the center of MMC [7]. The other is to divide the MMC into two MMCs [6]. Neither of them considers the situation that a MMC may continuously split during a period of time, and when this situation occurs, it will take a long time to handle the multiple splits of a MMC. Moreover, when checking whether a MMC can be merged with other MMCs, previous schemes search all the existing MMCs, which have an obvious decline in efficiency as the number of MMCs increasing.

In this paper, we present a grid-based approach to continuous clustering of moving objects. First, we develop an efficient split algorithm to handle the split of a MMC, which avoids multiple splits of one MMC during a period of time. Then we employ hierarchical grids similar to that used in [8] in order to narrow the searching area during *merge* operation. The spatial area is divided into square grids, and each grid will be dynamically divided or combined according to the number of MMCs belonging to it.

Our contributions can be summarized as follows: we develop an efficient split scheme which can avoid multiple splits of one MMC during a period of time. We employ hierarchical grids to narrow the searching area during *merge* operation. We present the algorithms of maintaining the dynamic grids and applying the dynamic grids to MMCs.

II. RELATED WORK

As one of the most important analysis techniques, clustering has been an active area of research in the field of data mining. A lot of clustering techniques have been proposed for static data sets [1], [2], [3], [8], [9], [10], [11]. They can be classified into the partitioning, hierarchical, density-based, grid-based and model-based method. The k-means algorithm [1] is representative of partitioning method, which aims at dividing the objects into *k* clusters in order to minimize the metric relative to the centroids of the clusters. The Birch algorithm [2] is a comprehensive hierarchical method, which originally proposed the concept of *micro-clustering* and the notion of *clustering feature* (CF) and *CF tree*. The STING algorithm [8] is a grid-based method, which divides the spatial area into rectancle cells and employs a hierarchical stucture. It has high efficency especially for large data set. To deal with the moving objects, our approach extends the grid in STING to a dynamic one, which will be dynamically divided or combined according to the number of MMCs belonging to it.

Recently, clustering of moving objects has drawn increasing attention. Har-Peled [12] aims to show that moving objects can be clustered once and the resulting clusters are competitive at any future time during the motion. Li et al. [4] first addressed clustering of moving objects by applying *micro-clustering* and dynamically maintaining bounding boxes of clusters. However, the bounding boxes of the cluster are likely to be exceeded frequently, which makes the number of maintenance events dominate the overall runtimes of the algorithms.

Zhang and Lin [5] proposed a histogram technique based on the clustering paradigm. A histogram must be reconstructed if too many updates occur, and there are usually a large number of updates at each timestamp, which makes histogram maintenance lack of efficiency.

Kalnis et al. [13] presented three algorithms to discover moving clusters from historical trajectories of moving objects. They treat a moving cluster as a sequence of spatial clusters that appear in consecutive snapshots of the object movements. Such moving clusters can be identified by comparing clusters at consecutive snapshots, the cost of which can be very high.

Jensen et al. [6] proposed a scheme capable of incrementally clustering moving objects in two-dimensional Euclidean space, which extended the concepts of CF and CF tree in Birch and employed a notion of object dissimilarity considering object movement across a period of time. Their experiments show this scheme performs significantly faster than traditional methods which frequently rebuild clusters.

Lai and Heuer [7] developed an approach dynamically maintaining a small set of MMCs, and they obtain global clusters by clustering these representative MMCs with traditional clustering algorithms for static data sets. Rosswog and Ghose [14] consider the situation that the moving objects intersect the space occupied by objects from another cluster and extend the distance measure to a function of the position history of the objects so as to improve the accuracy of traditional data clustering algorithms on spatio-temporal data sets.

III. PRELIMINARIES

In this section, we first introduce the model of MMC and some concepts about it, and then we describe the structure of the dynamic grid used in our approach and define some notions associated with it.

A. Model of MMC

In this paper, we consider moving objects in two-dimensional (2D) Euclidean space, which can be easily extended to higher dimensions. We define the *minimum update interval* (*minUI*), which denotes that the velocity of all the moving objects can be treated as constant during this interval. We assume that each moving object can transmit its new position and velocity to the server at the beginning of each *minUI*.

Each moving object can be represented as (*oid*, *p*, *v*, *t*), where *oid* is the unique ID of this object, *p* is the position of this object at time *t* and *v* is the velocity at time *t*. Both *p* and *v* are 2D vectors. During each *minUI*, the position of a object is a linear function of time and at time *t1*, it can be computed as *p(t1) = p + v(t1 - t)*, where *t1>t.*

Definition 1. The center of a MMC including *N* objects is of the form (*P*, *V*), where $P = (\sum_{i=1}^{N} p_i) / N$ and $V = (\sum_{i=1}^{N} v_i) / N$. *P* and *V* are position and velocity of the center respectively.

A MMC can be represented as (*cid*, *objn*, *objid*, *center*, *cf*, *t*), where *cid* is the ID of this MMC, *objn* is the number of moving objects in this MMC, *objid* is a list which contains the ID of objects in this MMC, *cf* is the clustering feature (CF) of this MMC and *center* is the center at time *t*. The CF of a MMC is defined as follows.

Definition 2. The CF of a MMC including *N* objects is of the form (*SP*, *SP*^{2}, *SV*, *SV*^{2}, *SPV*), where $SP = \sum_{i=1}^{N} p_i$, $SP^2 = \sum_{i=1}^{N} p_i^2$, $SV = \sum_{i=1}^{N} v_i$, $SV^2 = \sum_{i=1}^{N} v_i^2$ and $SPV = \sum_{i=1}^{N} p_i v_i$.

Claim 1. The CF at time t_{now} ($t_{now} > t$) can be maintained incrementally as follows [6]:
$CF' = (SP + SV(t_{now}-t), SP^2 + 2SPV(t_{now}-t) + SV^2(t_{now}-t)^2, SV, SV^2, SPV + SV^2(t_{now}-t))$.

Claim 2. When an object (*oid*, *p*, *v*, *t*) is inserted or deleted from the MMC, its CF can be modified as
$CF' = (SP \pm p, SP^2 \pm p^2, SV \pm v, SV^2 \pm v^2, SPV \pm pv)$.

Definition 3. The average radius *R*(*t*) of a MMC is the average Euclidean distance (ED) between its member objects and its center. It can be computed as

$$R(t) = \sqrt{\frac{1}{objn} \sum_{i=1}^{objn} D^2(p_i(t), p_c(t))} = \sqrt{\frac{1}{objn} (SP^2 - \frac{(SP)^2}{objn})} \quad (1)$$

where *D* (p_i (*t*), p_c (*t*)) is the ED between object *i* and the center.

According to [6], the average radius of a MMC at time *t1* can be updated based on the CF given at time *t* (*t1*>*t*) as

$$R(t1) = \sqrt{(a\Delta t^2 + b\Delta t + c) / objn}, \quad (2)$$

Where $a=SV^2-(SV)^2/objn$, $b=2(SPV-SPSV/objn)$, $c=SP^2-(SP)^2/objn$ and $\Delta t = t1 - t$.

B. Structure of Dynamic Grid

We employ the dynamic grid (DG) similar to that used in [8]. The spatial area is divided into square grids and the grids have a hierarchical structure. The first level of DG is the root and denotes the whole area, and the grids at the bottom level of DG are leaves. Each grid has four children at its lower level and each child corresponds to one quadrant of the parent grid. The children are numbered from 0 to 3. Fig. 1 illustrates the first three levels of a DG and Fig. 1(b) indicates the numbered children of a grid. Actually, the structure of DG is a quadtree, in which each tree node corresponds to a grid in DG and each tree level corresponding to a level in DG. In the following context, we no longer distinguish between grid and quadtree node.

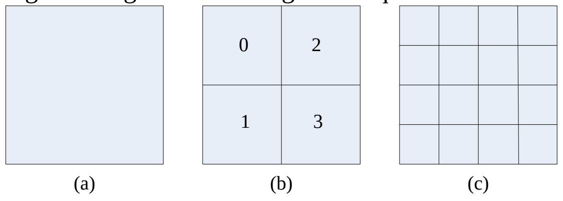

Figure 1. The first three levels of a DG

(a) the first level; (b) the second level; (c) the third level

we can use a 6-tuple (*Rcoord*, *height*, *cnum*, *cidlist*, *children*, *neighblist*) to represent a grid, where *Rcoord* is the relative coordinate of the grid, *height* is the height of the grid, *cnum* is the number of MMCs in this grid, *cidlist* is the list which contains the ID of MMCs in this grid, *children* is a list containing the pointers of the grid's children and *neighblist* is a list consisting of the pointers of the grid's neighbors.

Definition 4. The relative coordinate (RC) of a grid in DG can be represented by the form (x, y, l), where (x, y) is the RC of the upper left corner of the grid, and l is the length of this grid which is not larger than the threshold D_m (D_m will be defined in later section). The RC of the root is (0, 0, L), where L is the length of the whole area. If the RC of a grid is (x, y, l), then the RCs of its four children can be computed as ($2x$, $2y$, $l/2$), ($2x$, $2y+1$, $l/2$), ($2x+1$, $2y$, $l/2$) and ($2x+1$, $2y+1$, $l/2$) in order of children numbers.

Definition 5. The neighbors of a grid in DG mean grids adjacent to it. As shown in Fig. 2, a grid G has at most 8 neighbors which are numbered from 0 to 7 (e.g. Fig. 2(a)). The root of DG has no neighbor and other grids have at least 3 neighbors (e.g. Fig. 2(b)).

By employing RC, given the RC of a grid, we can easily compute the RCs of its neighbors. For example, in Fig. 2(a), if the RC of G is (x, y, l), then the RC of its neighbor 3 is (x-1, y, l), and the RC of its neighbor 7 is (x+1, y+1, l).

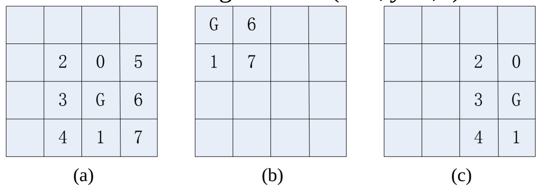

Figure 2. Neighbours of grid G

Definition 6. Each grid in DG has two possible states: expanded and unexpanded. If the subtree of a grid contains no MMCs, this grid is unexpanded. Otherwise, it is expanded.

Definition 7. Each level of DG has two possible states: expanded and unexpanded. As long as a grid at this level is expanded, this level is expanded. Otherwise, it is unexpanded.

Definition 8. The height of a grid is equal to the number of expanded levels in its subtree except for itself. Thus the height of an unexpanded grid is 0 and the height of an expanded grid is equal to the maximum height of its children plus 1.

IV. MAINTENANCE OF DYNAMIC GRID

The DG in our approach is dynamically maintained since MMCs may continue leaving a grid and then entering another grid. This section introduces some important algorithm about DG, including initialization, insertion, deletion, division and combination.

A. *Initialization*

We construct the initial DG with h levels (an initial quadtree with h levels), where h can be estimated from the capability of each grid c and the total number of MMCs n, that is, h satisfies $4^{h-1}c \geqslant n$. For each grid in DG, we initialize its *cnum* and *height* with 0, set its *cidlist* empty, compute its *Rcoord* as Definition 4 and initialize its *neighblist* according to Definition 5.

We employ a hash table *gridhash* which maps each MMC ID to the pointer of the grid it belongs to, so that given the ID of a MMC, we can fast locate the grid it belongs to. Moreover, we store the pointer of each grid into a table, in which each grid pointer can be quickly accessed by the level it belongs to and the RC of the grid.

B. *Insertion and Division*

We decide which grid a MMC belongs to by checking which "minimum" expanded grid its center is in. The minimum expanded grids denote the expanded grids with the minimum length and the minimum length *minl* can be computed as $minl = maxl / 2^{maxh}$, where *maxl* and *maxh* are the length and the height of the root in DG respectively.

After finding the belonging grid, the MMC will be inserted into this grid. To insert a MMC with ID *cid* into grid G, we modify *cnum* of G and all the ancestors of G, add *cid* to *cidlist* of G, modify the hash table *gridhash* and check whether G needs to be divided.

```
Divide(G)
Input: G (Rcoord, height, cnum, cidlist, children, neighblist)
       is a grid to be divided
1   G.height++
2   modify the height of all the ancestors of G if neccesary
3   for each MMC with the ID cid in G
4       decide cid belongs to G.children[ch] via cid.p
5       add cid into G.children[ch].cidlist
6       G.children[ch].cnum++
7       modify the hash table gridhash
8   clear G.clidlist
9   for each child ch of G
10      if G.children[ch].cnum exceeds the grid capacity
11         Divide(G.children[ch])
end Divide.
```

Figure 3. Division algorithm for DG

In order to limit the number of MMCs in one grid, we set the grid capability c. If the number of MMCs in a grid exceeds c, this grid will be "divided", which means the MMCs in this grid will be redistributed to its children (at its lower level). The division algorithm is shown in Fig. 3. To divide a G, we first modify the height of G and all the ancestors of G according to Definition 8, and then for each MMC in G, we decide which child of G it belongs to and add its ID to the *cidlist* of this child. Meanwhile, we modify *cnum* of this child and *gridhash*. When the redistribution is over, we check whether the children of G should be divided.

C. Deletion and Combination

When a MMC leaves a grid G, we should delete it from G. In order to delete the MMC *cid* from grid G, we modify *cnum* of G and all the ancestors of G, delete *cid* from *G.cidlist*. If G does not contain any MMC now, we proceed to check the parent of G p. If p contains no MMC, we combine the children of p.

It's not complex to combine the children of p. We just need set the height of p to 0 and modify the height of all the ancestors of p. Then, if the parent of p (if existed) contains no MMC now, the combination will repeat and probably so on up to the root.

V. Clustering Based on Dynamic Grid

This section will present our clustering scheme which is based on DG. We first introduce the distance metric and data structures used in our approach, and then we describe the details of the main algorithms.

A. Distance Metric

We utilize the distance metric with weight value proposed in [6] and as the *minUI* is short, we simplify this distance metric that we just consider three timestamps and our distance metric can be defined as

$$DM(O_1,O_2)=\sum_{i=1}^{3} w_i D^2(p_1(t_i),p_2(t_i)) \tag{3}$$

where t_1 is the current time t, $t_2 = t + minUI / 2$, $t_3 = t + minUI$, $D(p_1(t_i), p_2(t_i))$ is the ED between objects O_1 and O_2 at time t_i, and w_i ($0 < w_i < 1$) is the weight value at time t_i which satisfies $w_1 \geqslant w_2 \geqslant w_3$ and $w_1 + w_2 + w_3 = 1$.

Accordingly, the distance metric applying to an object O and a MMC C is

$$DM(O,C)=\sum_{i=1}^{3} w_i D^2(p_O(t_i),p_C(t_i)) \tag{4}$$

where p_O is the position of O and p_C is the center position of C. Also, the distance metric can be applied to two MMCs C_1 and C_2 as follows

$$DM(C_1,C_2)=\sum_{i=1}^{3} w_i D^2(p_1(t_i),p_2(t_i)) \tag{5}$$

where p_1 and p_2 are the center positions of C_1 and C_2 respectively.

B. Data Structures and Initialization

Two data structures are needed: the event queue Q and the hash table *MMChash*. Q stores future split events <t, *cid*> in ascending order of t, where t is the split time and *cid* is the ID of the MMC. *MMChash* maps each object ID to the MMC it belongs to, so that given the ID of an object, we can fast locate the MMC it belongs to.

During the initialization, we set the MMC capability C, which represents the maximum number of objects a MMC can contain, and we define the threshold R_m to represent the maximum average radius of a MMC. Also, we set the threshold D_m to denote that if the distance between an object and a MMC according to (4) exceeds D_m, this object cannot be inserted into the MMC. Then we construct the initial MMCs by the algorithm of object insertion introduced later.

C. Object Insertion and Deletion

Since the length of each grid is not larger than D_m, to insert an object O, we just need search MMCs in G and the neighbors of G instead of searching all the preexisting MMCs. As shown in Fig. 4, we first find the grid G O belongs to, and then search MMCs that are not full in G and the neighbors of G to find the nearest MMC to O according to (4). If the distance between the nearest MMC and O is larger than D_m, we create a new MMC for O. Otherwise, we try to insert O into this MMC. We compute the virtual CF of this MMC after absorbing O according to Claim 2 and then compute the virtual radius. If the virtual radius is larger than R_m, which indicates the insertion will lead this MMC into split, the insertion is failed and we create a new MMC for O. Otherwise, we update this MMC, modify *MMChash*, check whether this MMC changes its belonging grid, update the split event about this MMC in Q and the insertion is successful.

```
InsertObj(O)
Input: object O
1  find the grid G object O belongs to
2  search MMCs which are not full in G and G.neighblist
3  if the distance between O and the MMC cid nearest to it
   is larger than $D_m$
4      create a new MMC for O
5  else
6     compute the virtual CF and virtual radius vr of cid
      absorbing O
7     if $vr \geqslant R_m$
8        create a new MMC for O
9     else
10       cid.objn++
11       update cid.cf and cid.center
12       add O into cid.objid
13       modify the hash table MMChash
14       if cid changes the grid it belongs to
15          insert cid into the new grid
16          delete cid from the old grid
17       update the split event about cid
end InsertObj.
```

Figure 4. The algorithm of object insertion

To delete an object O from a MMC is a reverse process of the object insertion. We just need remove O from *objid* of the MMC and update *objn*, *center*, *cf* of this MMC.

D. Split

The split of a MMC occurs when its average radius exceeds R_m. According to (2), the split time is when $R(t) = R_m$. For simplicity, we consider $R^2(t)=R_m^2$, that is, $(a\Delta t^2+b\Delta t+c)/objn=R_m^2$. It's a quadratic equation about Δt and the solution is

$$\Delta t=\begin{cases}(-b+\sqrt{b^2-4a(c-R_m^2 objn)})/(2a), & a\neq 0\\ (R_m^2 objn-c)/b, & a=0\end{cases} \tag{6}$$

According to [6], from current time *t* to *t* + *minUI*, the split time of a MMC during *minUI* can be determined in the process: if $R^2(t) > R_m^2$, the split time is the current time. Otherwise, if $R^2(t+minUI) \le R_m^2$, there is no need to split during *minUI*, and if not, the split time can be computed according to (6).

As a MMC often splits many times during *minUI*, after its split, we need know whether it will split again during *minUI*, so we compare its average radius at current time and at the end *of minUI* with R_m. This algorithm is shown in Fig. 5.

```
willSplit(cid, endtime)
Input: MMC cid and  the end time of the interval endtime
Output: true if split will happen, otherwise false
1    if the average radius of cid at current time is
     larger than $R_m$
2        return true
3     else
4        compute the average radius of cid at endtime
5        if it is larger than $R_m$
6           return false
7        else
8           return true
end willSplit.
```

Figure 5. The algorithm deciding whether the split will happen

```
Split(cid, starttime,endtime)
Input: MMC cid, split time starttime and
         the end time of the interval endtime
Output: A list of the deleted objects ID idlist
1    update cid.cf, cid.center to starttime
2    while willSplit(cid, endtime) is true
3       for each object O in cid
4          compute DM(O, cid) and record the ID of the
           object with the maximum DM
5       delete the object with the maximum DM from cid
6       add the ID of this object into idlist
7    return idlist
end Split.
```

Figure 6. Split algorithm

To avoid multiple splits of a MMC during *minUI*, we propose a new approach to handle the split event. When a MMC with the ID *cid* splits at time *t*, we first compute the distance between each object in *cid* and the center of *cid* according to (4). Then we delete the object with the maximum distance from *cid* and check whether *cid* will split again during *minUI*. If it will, we repeat the process above. Otherwise, the split ends and we check whether *cid* changes its belonging grid. The algorithm is shown in Fig. 6.

After the split, we check whether *cid* can be merged. Then we build a new MMC *newcid* for the deleted objects. If the average radius of *newcid* is less than R_m, we check whether it can be merged with other MMC except *cid*. Otherwise we just add the split event about *newcid* into *Q*.

E. Merge

The merge operation first searches for a MMC for merging. The search process is similar to that in the object insertion algorithm. To find a MMC that can be merged with MMC *cid*, we get the grid *G* containing *cid* via the hash table. Then we search the MMCs that have enough space to absorb *cid* in *G* and the neighbors of *G*. If the MMC that can absorb *cid* without split during *minUI* exists, it's what we want. Otherwise, we choose the MMC which have the latest split time after absorbing *cid*. The details of this algorithm are shown in Fig. 7.

```
FindMMC(cid, endtime)
Input: MMC cid and the end time of the interval endtime
Output: MMC $cid_1$ to be merged with cid
1     G = gridhash(cid)   //G is the belonging grid of cid
2     for each MMC $cid_1$ except cid in G and G.neighblist
3         if $cid_1$.objn + cid.objn ≤ C   //C is MMC capacity
4            vcf = $cid_1$.cf + cid.cf
5            compute the split time during the interval according to vcf
6             if the merged MMC will not split during the interval
7                return $cid_1$
8             else
9                record $cid_1$ with the latest split time
10    return $cid_1$
end FindMMC.
```

Figure 7. The algorithm of finding a MMC for merging

After finding the MMC cid_1, we merge it with *cid*. We first add all the object IDs which are in cid_1*.objid* into *cid.objid* and modify the hash table *MMChash*. Then we update *objn*, *cf* and *center* of cid. At last, we update the split event about *cid* in *Q* and remove the MMC cid_1.

VI. EXPERIMENTS

This section presents the results of our experiments. We first introduce the experiment settings and data, and then compare our approach with other algorithms.

A. Experiment Settings and Data Preparation

All the experiments are conducted on an Intel Core 2 Quad 2.66 GHz PC with 3.25 GB RAM. We use synthetic data sets generated by our data generator. The whole space is a square space of size 32768×32768 units. Objects start at a random position in the space with random velocity from 0 to 5. We set *minUI* to 10 seconds and at the end of each *minUI*, we randomly choose parts of the objects and randomly change their velocity within 0 to 5. The total continuous clustering time is set to 1000 seconds.

We compare our grid-based approach (GridCMO) with other three approaches not based-on grid: the algorithm that handles split by removing the farthest objects (RECMO), the one using the split algorithm in [6] (DVCMO) and another one using the same split algorithm as our approach but not based-on grid (CMO). We conduct the experiments on 7 data sets with different size and run each algorithm 50 times on each data set.

B. Clustering Time

We compare the clustering speed of the four methods. The average clustering speed of the four algorithms is shown in Fig. 8.

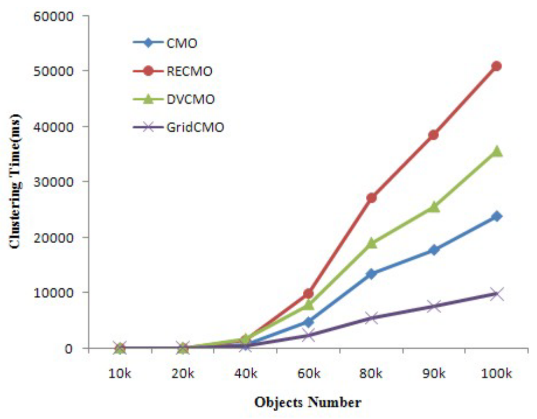

Figure 8. Average clustering speed

As can be seen from Fig. 8, GridCMO and CMO are faster than RECMO and DVCMO, which validates the efficiency of our split algorithm. GridCMO is the fastest one. This suggests that our dynamic grids accelerate the clustering. Moreover, the gaps between GridCMO and the other three methods are increasing as the data size increases, which indicates the dynamic grids employed in our approach are efficient especially for large data sets.

C. *Average Radius*

We proceed to compare the average radius of the MMCs obtained by the four methods, which can measure the conpactness of MMCs generated by these algorithms. The results are shown in Fig. 9.

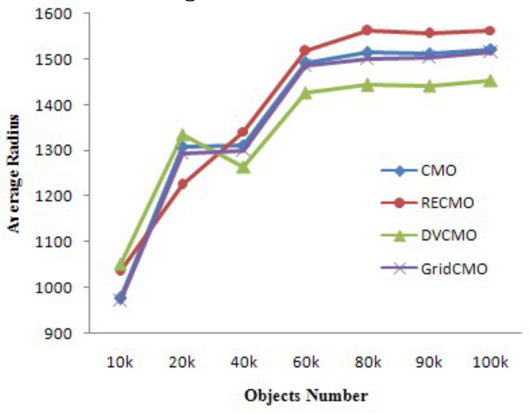

Figure 9. Avarage radius

As can be seen from Fig. 9, DVCMO has the minimum average radius because it handles the split by dividing the MMC into two parts with smaller radiuses, while the other three algorithms handle the split by deleting some “extreme” objects and aim to maintain a MMC as long as possible. GridCMO and CMO have the similar average radiuses which are close to that of DVCMO. This indicates that our split algorithm can keep the compactness of MMCs. RECMO has the maximum average radius because it only removes one object from the MMC each time it handles the split.

VII. CONCLUSION

This paper proposes an efficient grid-based approach for continuous clustering of moving objects. We develop an efficient split algorithm to handle the split of clusters, which avoids multiple splits of one cluster during a period of time. Also, we employ dynamic grids to narrow the searching area when merging clusters. The experimental evaluation has been conducted and validates that both our split algorithm and the dynamic grids accelerate the clustering as well as keep the compactness of the clusters. Our future work aims at applying the clustering scheme in the real world.

ACKNOWLEDGMENT

This work was supported by China 973 Fundamental R&D Program (No.2005CB321903) and China 863 High-tech Program (No.2009AA043305).

REFERENCES

[1] J. Macqueen, “Some Methods for Classification and Analysis of Multivariate Observations,” Proc. Fifth Berkeley Symp. Math. Statistics and Probability, pp. 281-297, 1967.

[2] T. Zhang, R. Ramakrishnan, and M. Livny, “BIRCH: An Efficient Data Clustering Method for Very Large Databases,” Proc. ACM SIGMOD Int’l Conf. Management of Data (SIGMOD ’96), pp. 103-114, 1996.

[3] M. Ankerst, M. Breunig, H.P. Kriegel, and J. Sander, “OPTICS: Ordering Points to Identify the Clustering Structure,” Proc. ACM SIGMOD Int’l Conf. Management of Data (SIGMOD ’99), pp. 49-60, 1999.

[4] Y. Li, J. Han, and J. Yang, “Clustering Moving Objects,” Proc. 10th ACM SIGKDD Int’l Conf. Knowledge Discovery and Data Mining (KDD ’04), pp. 617-622, 2004.

[5] Q. Zhang and X. Lin, “Clustering Moving Objects for Spatio-Temporal Selectivity Estimation,” Proc. 15th Australasian Database Conf. (ADC ’04), pp. 123-130, 2004.

[6] C.S. Jensen, Dan Lin, and Beng Chin Ooi, “Continuous Clustering of Moving Objects,” IEEE Transl. on Knowledge and Data Engineering, vol. 19, pp. 1161-1174, 2007.

[7] Chih Lai and E.A.Heuer, “Efficiently maintaining moving micro clusters for clustering moving objects,” Proc. 3rd IEEE Int’l Conf. on System of Systems Engineering (SoSE '08), pp. 1-6, 2008.

[8] W. Wang, J. Yang, and R. Muntz, “Sting: A Statistical Information Grid Approach to Spatial Data Mining,” Proc. 23rd Int’l Conf. Very Large Data Bases (VLDB ’97), pp. 186-195, 1997.

[9] R. Ng and J. Han, “Efficient and Effective Clustering Method for Spatial Data Mining,” Proc. 20th Int’l Conf. Very Large Data Bases (VLDB ’94), pp. 144-155, 1994.

[10] S. Guha, R. Rastogi, and K. Shim, “CURE: An Efficient Clustering Algorithm for Large Databases,” Proc. ACM SIGMOD Int’l Conf. Management of Data (SIGMOD ’98), pp. 73-84, 1998.

[11] G. Karypis, E.-H. Han, and V. Kumar, “Chameleon: Hierarchical Clustering Algorithm Using Dynamic Modeling,” Computer, vol. 32, no. 8, pp. 68-75, Aug. 1999.

[12] S. Har-Peled, “Clustering Motion,” Discrete and Computational Geometry, vol. 31, no. 4, pp. 545-565, 2003.

[13] P. Kalnis, N. Mamoulis, and S. Bakiras, “On Discovering Moving Clusters in Spatio-Temporal Data,” Proc. Ninth Int’l Symp. Spatial and Temporal Databases (SSTD ’05), pp. 364-381, 2005.

[14] J. Rosswog and K. Ghose, “Accurately clustering moving objects with adaptive history filtering,” Proc. 24th Int’l Symp. Computer and Information Sciences (ISCIS 2009), pp. 657-662, 2009.