PFDC: A Parallel Algorithm for Fast Density-based Clustering in Large Spatial Databases

Henning Meyerhenke
henningm@cs.uni-jena.de
Friedrich-Schiller-Universität Jena

Betreuer:	Prof. Dr. Hans-Dietrich Hecker, Friedrich-Schiller-Universität Jena
	Prof. Dr. Jörg-Rüdiger Sack, Carleton University, Ottawa
Art der Arbeit:	Studienarbeit
Fachbereich der GI:	0

Abstract. Clustering – the grouping of objects depending on their spatial proximity – is one important technique of knowledge discovery in spatial databases. One of the proposed algorithms for this is FDC [5], which uses a density-based clustering approach. Since there is a need for parallel processing in very large databases to distribute resource allocation, this paper presents PFDC, a parallel version of FDC. It has been implemented in C++ using MPI message passing to work on a variety of parallel platforms. Experiments show good speedup results of the proposed scheme.

1 Introduction

The processing of data from spatial databases – i.e. databases that store spatially referenced data such as geographic information systems – into information and real-world knowledge usually increases significantly the speed and quality of decisions based on this knowledge. Since the size of today's databases is typically too large for manual knowledge discovery, we need automated techniques for this. Clustering is one of them. It affiliates each object to a specific class such that similar objects (here in terms of location) belong to the same class, i.e. cluster. Clusters can then be associated with a specific meaning.

The sequential algorithm this work is based on is an enhancement of the sequential algorithm DBSCAN [3]. Both algorithms determine the affiliation of points to a specific cluster depending on their density, i.e. the number of points in their immediate neighbourhood. Because of this, they can meet most of the requirements for clustering in spatial databases (cf. [5]). Since experiments show that FDC outperforms DBSCAN significantly [5], it seems to be the tool of choice. But unlike DBSCAN [4], it has not been parallelized yet. However, large spatial databases with hundreds of thousand or millions of objects can be too large for processing these objects with only one computer in reasonable time or with limited memory size. This paper shows that FDC can be the basis for a parallel algorithm that makes use of its fast techniques and provides the advantages of parallel computing. After some introductory terminology from previous works, the algorithm is explained and experimental results of the implementation are given. The paper concludes with a short discussion of the results.

2 Density-based Clustering

Although the density-based notion can be generalized to other geometric objects in any dimension, this work is restricted to points in the plane for reasons of implementation. To allow for a better understanding, the most important terminology from [5] is presented in this section. It illustrates how the algorithm decides if a point belongs to a specific cluster or not.

Let *D* be a database with points, *Dist* a distance function (here: Euclidean), *Rad* a neighbourhood radius and *MinPts* a minimum threshold on the number of points in the neighbourhood.

Definition 1: A point $p \in D$ is *directly density-reachable* from a point $q \in D$ if
(i) $p \in N_{Rad}(q)$, where $N_{Rad}(q) = \{t \mid t \in D, Dist(q, t) \leq Rad\}$,
(ii) $| N_{Rad}(q) | \geq MinPts$.
We denote this by $p \leftarrow q$.

Definition 2: The binary relationship *density-linked* (denoted by $\leftrightarrow$) with respect to Rad > 0 and MinPts > 1 in D is defined as the following:
(i) $p \in D: p \leftrightarrow p$;

(ii) $p, q \in D$, $p \neq q$, then $p \leftrightarrow q$ if and only if $\exists$ chain $p = o_1, o_2, ..., o_n = q \in D$ such that $\forall$ i = 1, 2, ..., n-1 either $o_i \leftarrow o_{i+1}$ or $o_{i+1} \leftarrow o_i$ holds.

Note that $\leftrightarrow$ is an equivalence relationship.

Definition 3: Let $C_1, ..., C_m$ be the equivalent classes on D defined by $\leftrightarrow$ w.r.t. Rad and MinPts. Then holds:

(i) If $| C_i | > 1$, then C_i is a cluster;

(ii) if $| C_i | = 1$, then $o \in C_i$ is noise.

Theorem 1: Given a database D, the relationship $\leftrightarrow$ and the clustering parameters Rad and MinPts, the *clustering property* of every point $p \in D$ can be determined as follows:

(i) If $| N_{Rad}(p) | = 1$, p is a noise;

(ii) if $| N_{Rad}(p) | \geq$ MinPts, p is a *core point* of a cluster and all objects in $N_{Rad}(p)$ belong to the same cluster;

(iii) otherwise, let $T = \{o \mid o \in N_{Rad}(p), | N_{Rad}(o) | \geq \text{MinPts}\}$. If $T \neq \emptyset$, then p and all objects in T belong to the same cluster; else, p is a noise.

Given this, we need to compute the Rad-neighbourhood for every point p in the database. If either condition 1 or 2 of theorem 1 holds, we already have determined the clustering property of the point. Otherwise, we skip the point for the moment. Eventually, one of the two properties of condition 3 will be proven true: either p is found in the Rad-neighbourhood of a core point or it is noise.

3 Algorithm

The algorithm is divided into three parts. In the first stage, each processor essentially constructs a k-d tree on its part of the database. This is necessary because we assume that not all points loaded by a processor can be clustered at the same time in its main memory. Afterwards, the main task – determining the clustering property of theorem 1 for any point – is performed at the leaves of these k-d trees. Since different cluster IDs might have been assigned to the same cluster in the first place, merging of clusters is performed in a third step. In the remainder of this section, these stages are examined in more detail.

PFDC uses the idea of a parallel adaptive k-d tree as spatial data structure to process the points of the database. But unlike FDC, it also uses the idea of buffers, leading to a *guard k-d tree*:

For its construction, each processor loads at most $\lceil N/p \rceil$ points to its memory, where N is the total number of input points and p the number of processors. Then, all points are sorted along one dimension, say 0, with parallel sample sort and the load is balanced. This divides the plane in p distinct *root buckets*, one for each processor. It is necessary to communicate the *border points* with adjacent processors. These border points are at most 2 * Rad away from a bucket border (cf. figure 1). (We assume that all root buckets are at least 2 * Rad wide so that this communication step does not have to be done with more than the two adjacent processors – it could be done with more if necessary, though.)

Exchanging the border points with the adjacent processors is necessary to determine the clustering property of the own border points without communication during the clustering step. Furthermore, they are used for merging clusters between processors.

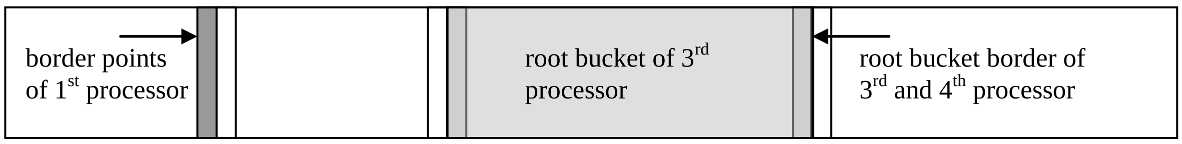
border points of 1st processor
root bucket of 3rd processor
root bucket border of 3rd and 4th processor

Figure 1: Division and communication of input data among processors

The actual clustering step will be performed at the leaves of the tree so that we can control the number of points stored in them to guarantee enough main memory. For good performance and speedup results, it is desirable that this step can be performed independently from data that is stored in another node of the tree (or even processor). This means that the leaves have to keep track of everything that is necessary to compute the clustering property of each point stored in them. We use the same idea as in the parallel communication step: Each node stores not only the points of its actual bucket, but also the points that lie in a buffer of width 2 * Rad around it. Then, all conditions of theorem 1 can be tested for all points in a leaf without the need for data from somewhere else. Figure 2a illustrates this: Every node stores a buffer so that all necessary points are accessible. The grey-shaded buffers are the ones necessary to construct node n_{26}, stored by the respective nodes that are shown with a thickly framed box in figure 2b.

Then, beginning only with its own root bucket and its buffers as the root of the local tree, each processor builds the described buffer (or guard) k-d tree sequentially in a sort-based fashion with alternating split dimensions (cf. [1] for details).

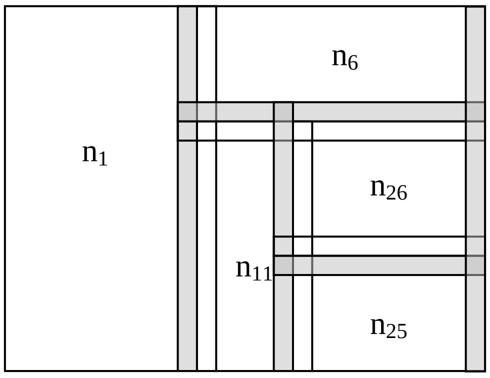

Figure 3a: Illustration of bucket and buffer building during the tree construction process

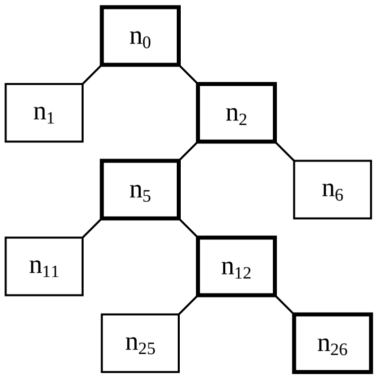

Figure 3b: Guard k-d tree corresponding to buckets in figure 2a

After the construction of the tree, points are clustered in the leaves as follows: Like in FDC, all points of the current leaf are mapped in equally sized grid cells of width and length Rad. For each point p, its Rad-neighbourhood is determined by querying the distances of points only from the neighbouring grid cells. For many points, this is much faster than the naive approach: compute for all points the distance to all other points in the leaf.

If $| N_{Rad}(p) | \geq$ MinPts, then p is a core point. It gets a new cluster ID, say P, which is also assigned to all points in $N_{Rad}(p)$ because these points belong to the same cluster. If it happens that a point $q' \in N_{Rad}(p)$ already has a cluster ID Q from another core point q, p and q are density-linked. Thus, they belong to the same cluster and P and Q are *equivalent*. These equivalencies happen in particular near leaf borders. To be able to merge these equivalencies, we store for each cluster ID its other equivalent cluster IDs. They are then treated in the third stage.

To get correct results, these equivalencies are merged twice: At first, sequentially by each processor, after clustering all its leaves. Transitive equivalencies (1 eq. 2, 2 eq. 3 $\Rightarrow$ 1 eq. 3) are resolved. Then, the minimum of their possible cluster IDs (in this case 1) is assigned to each point in the root node. This guarantees for each root node that all points belonging to the same cluster have the same cluster ID.

Equivalencies can also occur among processors if clusters span across root bucket borders. Hence, we need to merge clusters in parallel, too. It is done after all processors have finished their local merging. For this purpose, we communicate the unique point IDs of *merging candidates* and their cluster IDs computed so far. These merging candidates are all points – with some exceptions – that are not further away than Rad from the border of a root bucket. Points whose distance is between Rad and 2 * Rad, do not need to be communicated. If necessary, they will still be updated since they are in the neighbourhood of a merging candidate. After the communication step, equivalencies are noted as before and at the end sent to a “master processor”. This master merges the clusters as in the sequential steps and sends the results back to the other processors. They now assign the correct cluster ID to each of their points and write the results to disk.

4 Experiments

The program is implemented in C++ and uses the message passing library MPI. Hence, it runs (and has been tested) on different parallel hardware platforms and operating systems. However, the experiments presented here have been taken out on a SunFire 6800 with four nodes and operating system Solaris 8. Each node has 24 UltraSPARC III Cu processors with 900 MHz and 24 GB RAM.

The normal running times include loading the points to main memory, executing the algorithm and saving the results to disk. The times “without tree” (in italics) do not include the time for building the k-d tree to show that the clustering part can often be done with speedups near the theoretical optimum.

The four data sets, whose results are presented, are synthetic. They consist of 2^{17}, 2^{18}, 2^{19} and 2^{20} two-dimensional points respectively with ten to twenty clusters and about 10 % noise. For all experiments, Rad was set to 7. Table 1 shows the running times and speedups for these data sets as well as other variable parameters.

Number of processors	**1**	**2**	**4**	**8**	**Number of processors**	**1**	**2**	**4**	**8**
128k points (x, y) with x, y ∈ [0, 2000], MinPts = 9					*256k points (x, y) with x, y ∈ [0, 5000], MinPts = 14*				
Time total / s	32	17	9	5	Time total / s	62	38	21	10
Time without tree / s	*28*	*15*	*7*	*4*	*Time without tree / s*	*50*	*28*	*16*	*7*
Speedup total	1,00	1,88	3,56	6,40	Speedup total	1,00	1,63	2,95	6,20
Speedup without tree	*1,00*	*1,87*	*4,00*	*7,00*	*Speedup without tree*	*1,00*	*1,79*	*3,13*	*7,14*
512k points (x, y) with x, y ∈ [0, 5000], MinPts = 14					*1M points (x, y) with x, y ∈ [0, 5000], MinPts = 14*				
Time total / s	143	83	44	24	Time total / s	451	254	138	67
Time without tree / s	*117*	*61*	*33*	*18*	*Time without tree / s*	*392*	*201*	*109*	*55*
Speedup total	1,00	1,72	3,25	5,96	Speedup total	1,00	1,78	3,27	6,73
Speedup without tree	*1,00*	*1,92*	*3,55*	*6,50*	*Speedup without tree*	*1,00*	*1,95*	*3,60*	*7,13*

Table 1: Running times in seconds and speedups for four different data sets

We can see that most of the speedup results are good or even excellent, in particular if the construction of the tree is not considered. Two observations were made: The data set itself can cause varying speedups, because the fewer points are in the communicated buffers the better and vice versa. Hence, the point distribution and the borders of the root buckets are important, too. Secondly, other parameter settings have shown that the results can vary dramatically depending on Rad and MinPts.

5 Conclusion

Clustering in large spatial databases has been identified as a useful and sometimes challenging task. The shortage of resources such as memory and response time can be overcome by parallel computing. This paper has shown how the density-based clustering algorithm FDC can be parallelized in order to avoid the pitfalls mentioned above. The use of a guard k-d tree that “buffers” its nodes is a novel technique compared to FDC.

The parallel implementation uses message passing with standard MPI functions. This makes it run on different platforms, even on a simple network of workstations. The experiments show that the actual clustering can be performed with good speedups. They also show that the sort-based k-d tree used so far cannot compete with the clustering part in terms of speedup. An earlier implementation with the original sort-based scheme from [1] was even worse. Future work should concentrate on enhancing the k-d tree speedups – e.g. by using another k-d tree scheme – and on reducing communication in the parallel merging step.

Acknowledgement. I would like to thank Jörg-Rüdiger Sack and Doron Nussbaum for their support and valuable suggestions during my studies at Carleton University, Ottawa, Canada. Parts of this work have been carried out during that time.

References

[1] I. Al-furaih, S. Aluru, S. Goil, and S. Ranka: Parallel Construction of Multidimensional Binary Search Trees. In: IEEE Trans. on Parallel and Distributed Systems, vol. 11, no. 2, pp. 136-148, Feb. 2000.

[2] T. Cormen, C. E. Leiserson, R. L. Rivest: Introduction to Algorithms. MIT Press, 1990.

[3] M. Ester, H.-P. Kriegel, J. Sander, X. Xu: A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. Proc. 2nd Int. Conf. on Knowledge Discovery and Data Mining (KDD'96), pp. 226-231.

[4] X. Xu, J. Jaeger, H.-P. Kriegel: A Fast Parallel Clustering Algorithm for Large Spatial Databases. In: High Performance Data Mining. Ed. by Y. Guo and R. Grossmann, pp. 29-56. Kluwer Academic Publishers, 1999.

[5] B. Zhou, D. W. Cheung, B. Kao: A Fast Algorithm for Density-Based Clustering in Large Database. In: Proc. of the Third Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD '99), pp. 338-349. Springer-Verlag, 1999.