Experiments in Parallel Clustering with DBSCAN

Domenica Arlia and Massimo Coppola

Università degli Studi di Pisa, Dipartimento di Informatica
Corso Italia 40, 56125 Pisa, Italy
coppola@di.unipi.it

Abstract. We present a new result concerning the parallelisation of DBSCAN, a Data Mining algorithm for density-based spatial clustering. The overall structure of DBSCAN has been mapped to a skeleton-structured program that performs parallel exploration of each cluster. The approach is useful to improve performance on high-dimensional data, and is general w.r.t. the spatial index structure used. We report preliminary results of the application running on a Beowulf with good efficiency.

1 Introduction

The goal of Clustering consists in grouping data items into subsets that are homogeneous according to a given notion of similarity. It is well known [1] that clustering techniques are poorly scalable w.r.t. the amount of data, and to the number of data attributes. There is still an open debate [2] about *effectiveness* of distance measures for clustering data with very high dimensionality. Even if new data structures have recently been developed (like the X-Tree or the M-Tree) the performance of spatial index structures asymptotically degrades to that of a linear scan for high dimensional, ill distributed data.

There are relatively few results about improving practical use of clustering by means of parallel computation, and to the best of our knowledge, little has been done about parallel clustering with spatial access methods. Most of the research about spatial indexes is done in the Database community, and it has to face different and more complex problems like concurrent updates, which are outside the scope of our present work.

On the contrary, we will mainly address the performance of the retrieval operation, assuming that the data are already properly stored. Our contribution is a parallel implementation of the DBSCAN clustering algorithm [3], which is a new achievement. We aim at lowering computation time for those cases where density-based spatial clustering takes too long but it is still appropriate.

The main research line of our group is aimed at high-level structured languages in Parallel Programming Environments (PPE), to enhance code productivity, portability and reuse in parallel software engineering. The coordination language of our programming environment SkIE provides a set of parallel skeletons which express the basic forms of parallelism and encapsulate modules of

R. Sakellariou et al. (Eds.): Euro-Par 2001, LNCS 2150, pp. 326–331, 2001.

sequential code. The overall software architecture and the philosophy of the language are more extensively described in other papers, see [4]. Essentially, parallel skeletons are higher-order functionals used to express communication and synchronisation semantics among program modules. Sequential and parallel semantics are thus separated by module interface definitions.

Data Mining (DM) algorithms are an interesting source of problems of dynamic nature involving large data structures. We started a new research, summarized in [5], by looking at the interplay among the design of DM applications and the design and implementation of a PPE.

Here we report the results achieved so far with the DBSCAN algorithm. We started form the sequential source and turned the key functionalities into separate sequential modules. We have devised a parallel cooperation scheme among these modules, explained in §3. We finally mapped this high-level structure to a composition of skeletons in the coordination language of SkIE. We propose a refinement of the simple parallel scheme to control parallel overhead, and report test result for the parallel application.

2 Problem Definition

DBSCAN is a density-based spatial clustering algorithm. The original paper [3] fully explains the algorithm and its theoretical bases, which we briefly summarize in the following. By density-based we mean that clusters are defined as connected regions where data points are dense. If density falls below a given threshold, data are regarded as noise. Given a set of N points in R^d, DBSCAN produces a flat partition of the input into a number of *clusters* and a set of *noise* points. The density threshold is specified by choosing the minimum number *MinPts* of points in a sphere of radius ϵ. As a basic definition, a **core point** is a point whose ϵ-neighborhood satisfies this density condition. Clusters are non-empty, maximal sets of core points and surrounding boundary points. The high level structure of DBSCAN is a linear search for unlabelled core points, each new core point starting the ExpandCluster procedure (Fig. 1a). ExpandCluster is the working core of the algorithm, and its key operation is a spatial query, the operation of retrieving all the points belonging to a given region of the space. It can be shown that clusters are invariant w.r.t. the order of point selection in ExpandCluster, and that the overall complexity is $O(N \cdot r())$, where $r()$ is the complexity of the neighborhood retrieval.

A characteristic of spatial clustering is that spatial index structures are essential to enhance the locality of data accesses. The first implementation of DBSCAN uses a R*-Tree [6] to hold the whole input set. The R*-Tree is a secondary memory tree for d-dimensional data. It can answer to spatial queries with complexity proportional to the tree depth, $O(\log N)$ in time and I/O accesses. With this assumption, the authors of DBSCAN report a time complexity of $O(N \log N)$. As stated in the introduction, if the data distribution is unknown, spatial index structures need $O(N)$ search time for large values of N and d. This is also true for the R*-Tree when too large or dense regions are queried.

```
ExpandCluster (p, Input_Set, ϵ, MinPts,
                    ClusterID)
label(p, ClusterID)
put p in a seed queue
while queue is not empty
    extract c from the queue
    retrieve the ϵ-neighborhood of c
    if (there are at least MinPts neighbours)
        foreach neighbour n
            if (n is labelled NOISE)
                label n with ClusterId
            if (n is unlabelled)
                label n with ClusterId
                put n in the queue
```

```
Master :
while (there are pending results)
    get {p, s, set_n} from the result queue
    if (s > MinPts)
        foreach point n ∈ set_n
            if (n is labelled NOISE)
                label n with ClusterID
            if (n is unlabelled)
                label n with ClusterID
                put n in the candidate queue

Slave :
forever
    get point c from the candidate queue
    set_n = ϵ-neighborhood( c )
    put {c, #set_n, set_n} in the result queue
```

Fig. 1. (a) Pseudo-code of ExpandCluster — (b) Parallel decomposition.

```
farm retrieve in(candidate p) out(neighb n)
    Slave in(p) out(n)
end farm

pipe body in(neighb a) out(neighb b)
    Master in(a) out(candidate p)
    retrieve in(p) out(b)
end pipe

loop dbscan in(neighb i) out(neighb o)
          feedback(i=o)
    body in(i) out(o)
    while test in(o) out(bool cont)
end loop
```

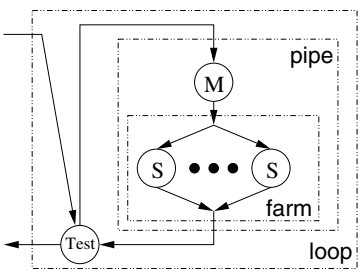

Fig. 2. The skeleton structure of parallel DBSCAN and its template implementation

3 The Parallel DBSCAN

We first addressed the performance of region queries. Region queries account for more than 95% of the computation time even when the R*-Tree fits in memory and contains two-dimensional data. Here we describe a simple replication approach, which has non obvious consequences on the behaviour of the algorithm. Our current target is to trade-off computational resources to increase the bandwidth of the slowest part of DBSCAN. We made no efforts yet in designing parallel access to the R*-Tree. Sequential DBSCAN is general w.r.t. the data structure, and I/O bandwidth is a more immediate limitation than data size.

By relaxing the visit order constraints, we have rewritten ExpandCluster as the composition of two separate processes (Fig. 1b), which actually interact through asynchronous, unordered channels. The parallel visit can be proven correct w.r.t. the sequential one, and can concurrently execute the queries.

The Master module performs cluster assignment, while the Slave module answers neighborhood queries using the R*-Tree. Reading spatial information is decoupled from writing labels, and the Slave has a pure functional behaviour. Having restructured the algorithm to a Master-Slave cooperation, we must ex-

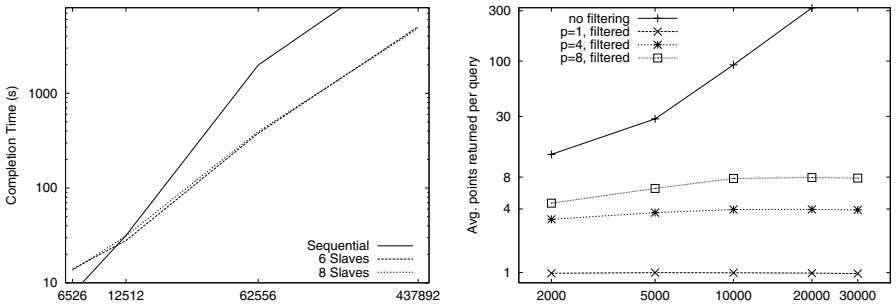

Fig. 3. (a) Completion times ($p = 1, 6, 8$) w.r.t. data size, with $\epsilon = 30000$, log/log scale. (b) Effect of filtering on query answers, $p = 1, 4, 8$, for file *ca* and varying ϵ

press its structure using the skeletons of our language. There is pipeline parallelism between Master and Slave, and functional independent replication can be exploited among multiple Slaves. In SkIE, the two skeletons pipe and farm respectively declare these two basic forms of parallelism. The outer loop skeleton in Fig. 2 expresses a data-flow loop, with back flow of information (the query answers) and a sequential module to test program termination.

Unlike the sequential DBSCAN, points already labelled are returned again and again from the slaves. Labels are kept and checked in the Master, which can quickly become a bottleneck, as the upper curve in Fig. 3b shows. This parallel overhead comes from the complete separation of labelling and spatial information. We reduce the overhead by introducing partially consistent information in the Slaves. The Slaves maintain local information used to discard redundant results, by ⓐ returning the set of neighbours for core points only and ⓑ never sending again a previously returned point. The filtering rule ⓐ alone has negligible effect, but is needed for the correctness of rule ⓑ. We see in Fig. 3b that the average number of points returned per query now approaches the degree of parallelism. This is also the upper bound, since no point is sent more than once by the p Slaves.

The program has been tested on a Beowulf class cluster of 10 PCs, with two samples (6K, 12K points), the full *ca* dataset (62K points) used in [3], and a scaled-up version of the data comprising 437 thousand points. More details can be found in [5]. This data size (the R*-Tree is up to 16 Mbytes) still allows in-core computation. Nevertheless the speedup (up to 6 with 8 slaves) and efficiency are good. From the completion times (Fig. 3a) we can see that a parallelism degree $p = 6, 8$ is the most we can usefully exploit on our architecture with in-core input datasets. We expect that the additional overhead for the out-of-core computation of a larger input set will raise the amount of available parallel work.

4 Conclusions

The program structure we have devised has several advantages. It exploits the modularity of the sequential application, so it helped in reengineering existing code to parallel with minimal effort. It is easily described and implemented using the skeleton coordination patterns. The additional code is simple and general: both sequential DBSCAN and our parallel implementation will still produce identical results when replacing the R*-Tree with another spatial index structure. At present time we duplicate the spatial data structure to gain computation and I/O bandwidth. Saving disk space will be addressed in a later stage.

We deal with the parallel overhead of the simple Master-Slave decomposition through a distributed filtering technique. This solution has been tested for in-core data size with high computational load, showing large saving of computation and communication, and a consistently good speedup w.r.t. the sequential algorithm.

We still have to verify the scalability results for larger input sets, forcing the R*-Tree to be actually stored out-of core. Secondary memory sharing by parallel file system can reduce or avoid the amount of data replication. Comparison of the R*-Tree with newer spatial data structures is also needed, and more complex filtering heuristics can be devised to further reduce the parallel overhead at higher parallelism degree, by exploiting locality and affinity scheduling among the queries. Finally, we are studying the extension of parallel DBSCAN results to OPTICS [7], a DBSCAN-based automatic clustering methodology which relies on a specific visit order in the ExpandCluster procedure.

Acknowledgments

We wish to thank Dr. Jörg Sander and the authors of DBSCAN for making available the source code.

References

1. Daniel A. Keim and Alexander Hinneburg. Clustering techniques for large data sets–from the past to the future. In *Tutorial notes for ACM SIGKDD 1999 international conference on Knowledge discovery and data mining*, pages 141–181, 1999. 326
2. K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When Is "Nearest Neighbor" Meaningful? In C. Beeri and P. Buneman, editors, *Database Theory - ICDT'99 7th International Conference*, volume 1540 of *LNCS*, pages 217–235, January 1999. 326
3. Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Proceedings of KDD '96*, 1996. 326, 327, 329
4. B. Bacci, M. Danelutto, S. Pelagatti, and M. Vanneschi. SkIE : A heterogeneous environment for HPC applications. *Parallel Computing*, 25(13–14):1827–1852, December 1999. 327

5. Massimo Coppola and Marco Vanneschi. High-Performance Data Mining with Skeleton-based Structured Parallel Programming. Technical Report TR-06-01, Dipartimento di Informatica, Università di Pisa, May 2001. 327, 329
6. N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *Proc. of the ACM SIGMOD International Conf. on Management of Data*, pages 322–331, 1990. 327
7. M. Ankerst, M. M. Breunig, H.-P. Kriegel, and J. Sander. OPTICS: ordering points to identify the clustering structure. In *Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, pages 49–60, 1999. 330