Università di Pisa

Dipartimento di Informatica

Technical Report: TR-01-06

High-Performance Data Mining with Skeleton-based Structured Parallel Programming

Massimo Coppola Marco Vanneschi

May 8, 2001

ADDRESS: Corso Italia 40, 56125 Pisa, Italy. TEL: +39 050 2212700 FAX: +39 050 2212726

High-Performance Data Mining with Skeleton-based Structured Parallel Programming

Massimo Coppola Marco Vanneschi

May 8, 2001

{coppola, vannesch}@di.unipi.it

Abstract

We show how to apply a Structured Parallel Programming methodology based on skeletons to Data Mining problems, reporting several results about three commonly used mining techniques, namely association rules, decision tree induction and spatial clustering. We analyze the structural patterns common to these applications, looking at application performance and software engineering efficiency. Our aim is to clearly state what features a Structured Parallel Programming Environment should have to be useful for parallel Data Mining. Within the skeleton-based PPE SkIE that we have developed, we study the different patterns of data access of parallel implementations of Apriori, C4.5 and DBSCAN. We need to address large partitions reads, frequent and sparse access to small blocks, as well as an irregular mix of small and large transfers, to allow efficient development of applications on huge databases. We examine the addition of an object/component interface to the skeleton structured model, to simplify the development of environment-integrated, parallel Data Mining applications.

1 Introduction

In recent years the process of Knowledge Discovery in Databases (KDD) has been widespreadly recognized as a fundamental tool to improve results in both the industrial and the research field. Parallel computing is a key resource in enhancing the performance of applications and computer systems to match the computational demands of the Data Mining (DM) phase for huge electronic databases. The exploitation of parallelism is often restricted to specific research areas (scientific calculations) or subsystem implementation (database servers) because of the practical difficulties of parallel software engineering. Parallel applications for the industry have to be (1) efficiently developed and (2) easily portable, characteristics that traditional low-level approaches to parallel programming lack. The work of our research group has been directed to address the issue of parallel software engineering and shorten the time-to-market for parallel applications. The use of Structured Parallel Programming (SPP) and high-level Parallel Programming Environments (PPE) are the main resources in this perspective. The structured approach has been fostered and supported by several research and development projects, which resulted in the P3L language and the SkIE PPE [1, 2, 3].

Here we present our analysis of a significant set of DM techniques, which we have ported from sequential to parallel with SkIE. We report our experiences [4, 5, 6, 7] about the problems of association rule extraction, classification and spatial clustering. We have developed three prototype applications by restructuring sequential code to structured parallel programs. The SPP approach of the SkIE coordination language is evaluated against the engineering and performance issues of these I/O and computationally intensive DM kernels. We show that the improvements in program design and maintenance do not impair application performance. We also examine object-oriented additions to the skeleton programming model in order to ease huge data management in remote and mass memory. The next-generation PPE, called ASSIST [8], will provide a common interface to access external libraries, servers, shared data structures as remote objects and grid computational resources. We study the application of shared objects as a tool to simplify the implementation of out-of-core classification algorithms. The common implementation schemes that emerge, as well as the performance results that we show, sustain the validity of a structured approach for DM application. It is also well recognized in the literature [9, 10] the need for a tighter integration of high performance DM systems with the support for management, filtering and selection of data. We believe that the SPP approach and the availability of standard interfaces within the PPE will simplify the development of integrated parallel KDD environments.

The next section explains the basics of SPP models, giving an overview of the field, of our research and of the SkIE PPE, as well as a short comparison of the computer architectures we ran our tests on. Sec. 3 draws the general framework of sequential and parallel DM, and contains some general definitions. Sec. 4 examines the first prototype, parallel partitioned Apriori. Definitions of the problem and the algorithm, a summary of closely related works, description of the parallel structure and analysis of the test results are reported. The same organization of the matter is given to sec. 5, about parallel clustering with DBSCAN, and sec. 6, which describes a parallel C4.5 classifier employing a shared object abstraction. In sec. 7 conclusions are drawn, and an object interface to external, shared mechanisms is proposed on the grounds of the experiences made.

2 Structured Parallel Programming

The software engineering problems we mentioned in the introduction follow from the fact that most high performance computing technologies do not fully obey to the principles of modular programming [11]. Our work on parallel programming models and languages has led us to the concept of parallel coordination language. This is the main feature of the PPE SkIE. The *constructs* of the language allow to express the global structure of the program, providing a high level description that is machine independent. All the low level details of communication and parallelism management are left to the language support. The interaction through clearly defined interfaces makes independent the implementation of different modules. The parallel coordination is based on the *parallel skeleton* programming model [12, 13]. Skeleton-based models have many powerful features, like compositionality, performance models and semantic-preserving transformations allowing to define optimization techniques. *Structured parallel programming* within the parallel coordination approach aims at merging these advantages with the of ease of interfacing and reusing existing application code. The *compositional* nature of skeletons allows to nest parallel modules inside each other, developing complex parallel structures from the simple, basic ones. The concept of module interface eases the interaction among different sequential *host languages* (among them C/C++, Fortran, Java), and with the environment. The properties of the underlying skeleton model can be exploited for global optimizations, while retaining the existing sequential tools and optimizations for the purely sequential modules. The SkIE-CL coordination language provides the user

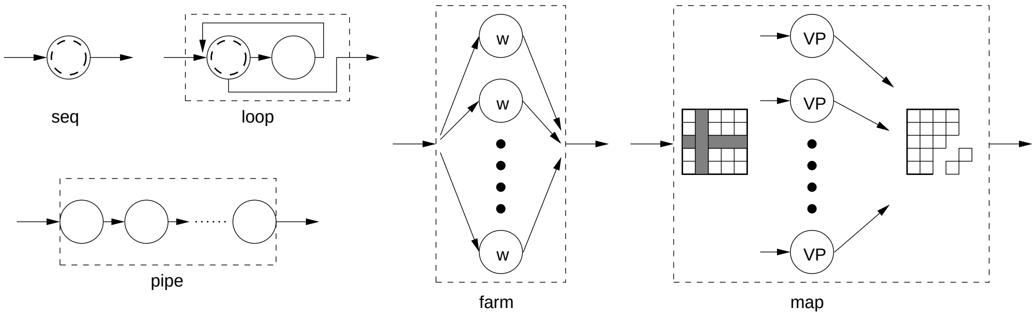

Figure 1: The parallel skeletons in SkIE.

with a set of parallel skeletons whose graphical representations are shown in figure 1. The general semantic of the skeletons is data-flow like, with packets of data we call *tasks* streaming between the interfaces of linked program modules. The SkIE skeletons are a subset of those studied in the literature. Here we describe the semantics of the ones that we'll use in the paper.

The simplest skeleton, the seq, is a mean to encapsulate sequential code from the various host languages into a modular structure with well-defined interfaces. Pipeline composition of different stages of a function is realized by the pipe skeleton. The independent functional evaluation over tasks of a stream, in a load-balancing fashion, is expressed through the farm skeleton. The *worker* module contained is seamlessly replicated and its copies operate on subsets of the input stream. The loop skeleton is used to define cyclic, possibly interleaved, data-flow computations in which tasks have to be repeatedly evaluated until they satisfy the condition expressed by a sequential *test* module. The map skeleton is used to define data-parallel computation on a portion of a single data structure, which is distributed to a set of virtual processors (VP) according to a decomposition rule. The tasks output by the VP are recomposed to a new structure which is the output of the map.

2.1 Implementation issues and performance portability

The templates used to implement the skeleton semantics in SkIE are parametric nets of processes which the compiler instantiates and maps on the target parallel machine. Appropriate techniques are used to hide communication latency and overheads. The current implementation uses static templates, which are optimized at compile time.

Performance portability across parallel platforms is a feature of SPP-PPEs. The results shown in the rest of the paper come from tests we have made on four parallel machines belonging to different architectural classes. Low-cost clusters of workstations and full-fledged parallel machines are represented, which differ in the memory model and relative performance of computation, communication and mass memory support. The first and more generally available platform is Backus, a cluster of 16 LINUX workstations (**COW**) connected by a Fast Ethernet crossbar switch. The **CS-2**, from QSW, is a multiprocessor architecture with distributed memory, dual-processor nodes and a fat tree network. The Cray **T3E** is a massively parallel processor (MPP) with non-uniform access shared memory supported in hardware. The last architecture in our list is a 4 CPU symmetric multiprocessor (**SMP**) with uniform memory access (UMA). This kind of parallel architecture is not scalable, but is often used as a building block for larger, distributed memory clusters.

On the one hand, we might put on a line the various platforms ordered by the raw computing performance of their CPUs. We would find the CS-2, the SMP, the COW and then the T3E, which is the fastest. On the other hand, the computation to communication bandwidth ratio has a more profound impact on parallelism exploitation. If we look at it, the COW is outweighted by the true parallel machines, and the SMP clearly offers the fastest communications. Finally, I/O speed and scalability are key factors for DM applications. The CS-2 and the COW have local disks in each node in addition to the shared network file system. The distributed I/O capabilities, even if not a parallel file-system, allow for implementing more scalable applications. The fastest local I/O is on the COW, followed by the CS-2, while the SMP is sometimes impaired by a single mass memory interface. Sustained, irregular and highly parallel I/O on the T3E, in the configuration we used, leads to high latency and insufficient bandwidth.

3 Data Mining and integrated environments

The goal of a DM algorithm is to extract from the input an optimal description of the data, within a fixed model space and obeying to a model cost measure. Each model description language defines a model space, for which one or more DM algorithms exist. A number of interacting factors determine the quality and usefulness of the results: selection and preprocessing of the raw input data, choice of the kind of model, choice of the algorithm to run, tuning of its execution parameters. In order to select the best combination, the KDD process involves repeated execution of the DM step, supervised by human experts and meta-learning algorithms. Fast DM algorithm are essential as well as the efficient coupling between them and the software managing the data. This problem is mostly clear for Parallel DM, where the I/O bandwidth and communications are two balancing terms of parallelism exploitation [14].

To exploit parallelism at all levels, from the algorithm down to the I/O system, thus removing any bottleneck, a higher degree of integration has already been advocated in the literature [15, 9]. Ideally, parallel implementations of the DM algorithms, the file system, DBMS, and Data Warehouse should seamlessly cooperate with each other and with visualisation and meta-learning tools. Some high-performance, integrated systems for DM are already being developed for the parallel [10] and the distributed settings [16]. Other works like [17] concentrate on requirements for the data transport layer in parallel and distributed DM. We want to address the system integration issues through the use of a PPE. Besides simplifying software development, a PPE should provide standard interfaces to conventional and parallel file systems and more complex services.

We look at data as being in tabular form. Each row of the table is a data item, while the columns are the various attributes of the object. The stored objects may be points, sets of related measurements or fields extracted from a database record. The attributes can be integer, real values, labels or boolean values. Using *market basket analysis* as a practical example, each "object" is a commercial transaction in a store. In the case of *clustering*, data are usually points in a space R^a, each row being a point and each of the a attributes a spatial coordinate value. In the rest of the paper, $\mathcal{D}$ is the input database, N is its number of rows, a the number of attributes, or columns. Assuming the attributes are of bounded size, the overall input size is $O(Na)$. When appropriate, the number of rows in a horizontal partition is n, and M is the amount of memory for each processor. p is the degree of parallelism.

DM algorithms use the input data to build a solution in the model space, but in some cases the intermediate data may be even larger than the input. DM techniques usually have to partition the input or the intermediate representation of the solution to exploit parallelism or manage the workload over the available resources. Algorithms are said to do horizontal partitioning when dividing data according to rows. Vertical partitioning is to divide according to columns, breaking the records. Either of the two may be suited to a particular method for I/O and algorithmic reasons. Of course, beside these two simple schemes, other parallel organizations come from the coordinate decomposition of the input data and the structure of the search space [14].

4 Apriori Association Rules

The problem of association rule mining (ARM), which has been proposed back in 1993, has its classical application in *market basket analysis*. From the sell database we want to detect rules of the form $AB \Rightarrow C$, meaning that a customer that buys together objects A and B also buys object C with some minimum probability. We refer the reader to the complete description of the problem given in [18], as we concentrate our attention on the computationally hard subproblem of finding *frequent sets*.

In the ARM terminology the database $\mathcal{D}$ is made up of transactions (the rows), each one consisting of a a unique identifier and a number of boolean attributes. The attributes are called *items*, and a *k-itemset* contained in a transaction is a set of k items which are true in the transaction. The support $\sigma(X)$ of an itemset X is the proportion of transactions that contain X. Given $\mathcal{D}$, the set of items $\mathcal{I}$, and a fixed real number $0 < s < 1$, called *minimum support*, the solution of the frequent set problem is the collection $\{X | X \subseteq \mathcal{I}, \sigma(X) \geq s\}$ of all itemsets that have at least the minimum support. The support information of the frequent sets can be used to infer all the valid association rules in the input. The power set $\mathcal{P}(\mathcal{I})$ of the set of items has a lattice structure naturally defined by the set inclusion relation. A level in the lattice is a set of all itemsets with equal number of elements. The minimum support property is preserved over decreasing chains: $(\sigma(X) > s) \wedge (Y \subseteq X) \Rightarrow \sigma(Y) > s$. Figure 2a shows a very small example with the frequent sets put in evidence in the lattice. Computing the support count for a single itemset requires a linear scan of $\mathcal{D}$. The database is often in the order of Gbytes, and the number of potentially frequent itemsets, $2^{|\mathcal{I}|}$, usually exceeds the available memory. To efficiently work out the frequent sets, their structure and properties have to be exploited.

The easiest way to classify the algorithms for ARM is actually according to their lattice exploration strategy. Sequential and parallel solutions differ in the way they arrange the exploration and distribute the data structures to minimize computation, I/O, memory requirements, and in the fraction of the lattice they explore that is not part of the solution. In the following we will restrict ourselves to the Apriori algorithm, described in [18], and its direct evolutions.

Apriori builds the lattice level-wise and bottom-up, starting from the 1-itemsets and using as a pruning heuristic the fact that non-frequent itemsets cannot have frequent supersets. From each level L_k of frequent itemsets, a set of candidates C_{k+1} is derived. The support for all the candidates is verified on the data to extract the next level of frequent itemsets L_{k+1}. Apriori is a breakthrough w.r.t. the naive approach, but some issues raise when applying it to huge data. A linear scan of $\mathcal{D}$ is required for each level of the solution. The underlying assumption is that the itemsets in C_k are much less than all the possible k-itemsets, but this is often false for $k = 2, 3$, because the pruning heuristic doesn't apply well. Computing the support values for C_k becomes quite hard if C_k is large. A review of several variants of sequential Apriori, which aim at correcting these problems, is given in [19]. A view on the theoretical background of the frequent itemset problem and its connections to other problems in machine learning can be found in [20].

4.1 Related work on parallel association rules

We studied the Partitioned variant of Apriori introduced in [21], which is a two-phase algorithm. The data is horizontally partitioned into blocks that fit inside the available memory, and Apriori is applied separately to each block, with the same relative value of s. The union of the frequent sets for all the blocks is a superset of the true solution. The second phase is a linear scan of $\mathcal{D}$ to compute the support counts for all the elements in the approximate solution. We obtain the frequent sets with only two I/O scans. Phase II is efficient, and so the whole algorithm, if the approximation built in phase I is not too coarse. This holds as long as the blocks are not too small w.r.t. $\mathcal{D}$, and the data distribution is not too skewed.

The essential limits of the partitioned scheme is that both the intermediate solution and the data have to fit in memory, and that too small a block size causes data skew. The clear advantage for the parallelisation is that almost all work is done independently on each partition.

Following [22], we can classify the parallel implementations of Apriori into three main classes, *Count*, *Data* and *Candidate Distribution*, according to the interplay of the partitioning schemes for the input and the C_k sets. We have applied the two phase partitioned scheme without the vertical representation described in [21], using a sequential implementation of Apriori as the core of the first phase. Count Distribution solutions horizontally partition the input among the processors, and use global communications once at each level to compute the candidate support. Although it is more asynchronous and efficient, the parallel implementation of Partitioned Apriori asymptotically behaves like Count Distribution w.r.t. to the parameters of the algorithm. It is quite scalable with the size of $\mathcal{D}$, but cannot deal with a huge candidate sets or frequent sets, i.e. it is not scalable with lower and lower values of the s support parameter.

4.2 The skeleton structure

The structure of the Partitioned algorithm is clearly reflected in the skeleton composition we have used, which is shown in fig. 2b. The two phases are connected within a pipe skeleton. Since there is no parallel activity between them, they are in fact mapped on the same set of processors. The common internal scheme of the two phases is a three-stage pipeline. The first module within the inner pipe reads the input and controls the computation. The second module is a farm containing p seq workers running the Apriori code. The third module is sequential code performing a stream-reduction to compute the sum of the results. In phase I the results are hash-tree structures containing all the locally frequent sets, and they are summed to a hash tree containing the union of the local solutions. Phase II has a simpler code in the workers, and the results are arrays of support counts that are added together to compute the global support for all the selected itemsets.

Since we initially wanted to test the application without assuming the availability of parallel access to disk-resident data, we used sequential modules to interface to the file system and to distribute the input partitions to the other modules. On the COW, where local disks were available and the network performance was inadequate to that of the processors, we also implemented distributed I/O in the workers (fig. 2b) by replicating the data over all the disks and retaining the farm for its load balancing characteristics.

4.3 Results

The Partitioned Apriori we realized in SkIE is a very good example of the advantages of structured parallel programming. A sequential source code has been restructured in a modular parallel application, whose code is less than 25% larger and reuses 90% of the original. The development times were also quite short, as reported in [4]. The test results of figure 3 - 5 are consistent over a range of different architectures.

We used the synthetic dataset generator from the Quest project, whose underlying model is also explained in [18], choosing average frequent sets of

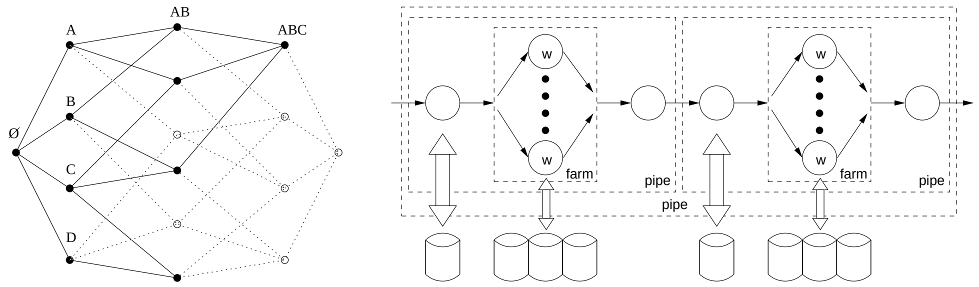

Figure 2: (a) The lattice of frequet sets. (b) Skeleton structure of the prototype.

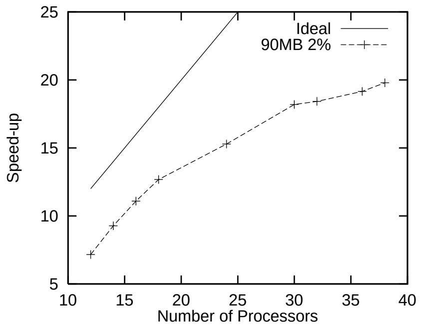

Figure 3: Speed-up on the CS-2.

6 items and a transaction length of 20 out of 1 thousand items. With these parameters, huge C_k sets are generated even for a high value of the minimum support. For a number of transactions of $N = 1, 4, 12$ millions, datasets of 90, 360 and 1100 Mbytes are produced.

The CS-2 architecture already shows a good behaviour with a small dataset and low load, see fig. 3. By comparison, because of the slower communications the COW has a lower efficiency, see the two (*) curves in fig. 5a. A better performance is obtained by removing the I/O bottleneck, thus increasing the computation to communication ratio and shortening the startup times w.r.t. the overall computation. In fig. 5a we find the efficiency results with distributed I/O on the COW. The speedup graphs in fig.5b also show that the application is scalable on the COW at higher computational loads. The same application runs on the SMP (fig. 4a), where the almost ideal speedup of 3.8 is reached with $p = 6$. The T3E results of fig. 5c with $s = 2\%$ don't look satisfying. A profiling of the running times has shown that the problem is in the interaction with the file server, which is a high fixed overhead that becomes less severe at higher workloads, as the behaviour for

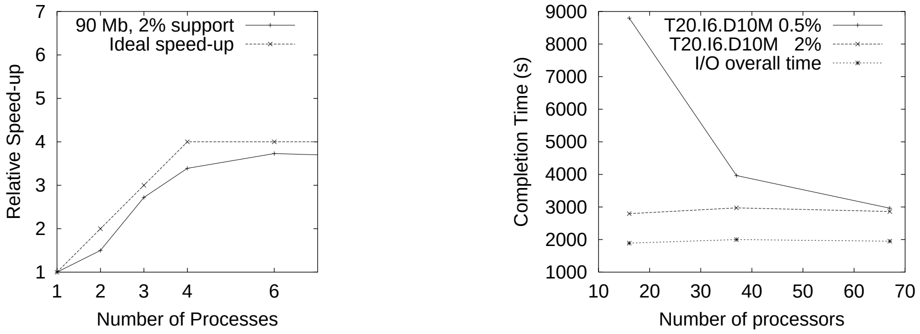

Figure 4: (a) SMP speed-up. (b) T3E completion time, 10M transactions.

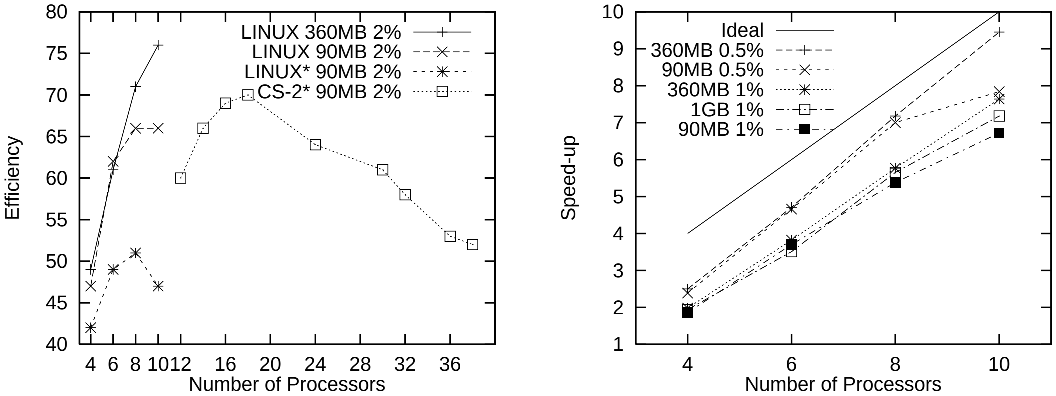

Figure 5: (a) program efficiency over LINUX and CS-2 with support set at 2%. (b) LINUX parallel speed-up w.r.t. to parallelism and varying load.

$s = 0.5\%$ shows.

5 DBSCAN Spatial Clustering

Clustering is the problem of grouping input data into sets in such a way that a similarity measure is high for objects in the same cluster, and elsewhere low. In Spatial Clustering the input data are seen as points in a suitable space R^a, and discovered clusters should describe their spatial distribution. Many kinds of data can be represented this way, and their similarity in the feature space can be mapped to a concrete meaning, e.g. for frequency spectral data to the similarity of two real-world signals. A high dimension a of the data space is quite common and can lead to performance problems [23]. Usually, the spatial structure of the data has to be exploited by means of appropriate index structures, to enhance the locality of data accesses. DBSCAN is a density-based spatial clustering technique [24], whose parallel form we recently studied in [7]. Density-based clustering identifies clusters from the density of objects in the feature space. In the case of DBSCAN, computing densities in R^a means counting points inside a given region of the space.

The key concept of the algorithm is that of *core point*. Given two user parameters ϵ and *MinPts*, a core point has at least *MinPts* other data points within a neighborhood of radius ϵ. A suitable relation can be defined among the core points, that allows us to identify dense clusters made up of core points, to assign non-core points to the boundary of neighboring clusters, or to mark points as noise. To assign cluster labels to all the points, DBSCAN repeatedly searches for a core point, then explores all the cluster it belongs to. The situation is exemplified in fig.6b, where a cluster of three core points and three boundary points is found. The process is much alike a graph visit, where connected points are those closer than ϵ, and the visit recursively explores all reached core-points. When a point in the cluster is considered as a *candidate*, its neighborhood points are counted. If they are enough, the point is labelled and its neighbors are then put in the candidate queue.

DBSCAN holds the whole input set inside the R*-Tree spatial index structure [25]. Data are kept in the leaves of a secondary memory tree with an ad-hoc directory organization and algorithms for building, updating and searching the structure. Given two hypothesis that we'll detail in the following, the R*-Tree can answer to spatial queries (what are the points in a given region) with time and I/O complexity proportional to the depth of the tree, which is $O(\log N)$. For each point in the input there is exactly one neighborhood retrieval operation, so the expected complexity on DBSCAN is $O(N \log N)$.

The first hypothesis needed is that almost all regions involved in the queries are small w.r.t. the dataset, hence the search algorithm needs to examine only a small number of leaves of the R*-Tree. We can assume that the ϵ parameter is not set to a neighborhood radius comparable to the whole dataset. The second hypothesis is that a suitable value for ϵ *exists*. It is well known that all spatial data structures loose efficiency as the dimension a of the space grows, in some cases already for $a > 10$. The R*-Tree can be easily replaced with any improved spatial index that supports range queries, but for a high value of a this could not lead to an efficient implementation anyway. It has been argued in [23], and it is still matter of debate, that for higher and higher dimensional data the concept of neighborhood of fixed

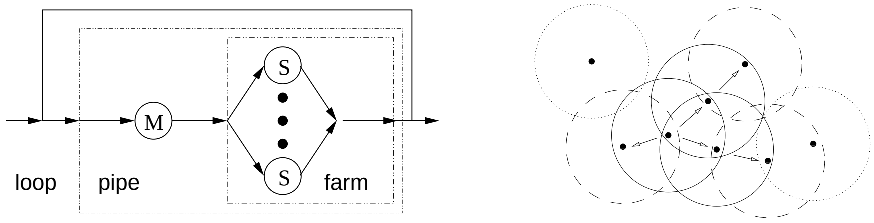

Figure 6: (a) The skeleton composition for parallel DBSCAN. (b) The cluster exploration strategy for $MinPts = 3$.

radius progressively looses its meaning for the sake of spatial organization of the data. As a consequence, for some distributions of the input the worst-case performance of *good* spatial index structures is that of a linear scan of the data [26]. For those cases where applying spatial clustering to huge and high-dimensional data produces useful results, but requires too much time, parallel implementation is the practical way to speed up DBSCAN.

5.1 Parallel structure

The region queries to the R*-Tree are the first issue to address to enhance DBSCAN. The method definition guarantees that the shape of clusters is invariant w.r.t. the order of selection of points inside a cluster, so we have chosen a parallel visit strategy, with more independent operations on the R*-Tree at the same time. A *Master* process executes the sequential algorithm, demanding all the neighborhoods retrievals to a *Slave* process. By relaxing the ordering constraint on the answers from the R*-Tree, two kinds of parallelism are exploited in this scheme: pipelining (pipe) between Master and Slaves, and independent parallelism (farm) among several Slave modules, each one holding a copy of the R*-Tree. We see the resulting structure in figure 6a.

This is a proper data-flow computation on a stream of tasks, with a loop skeleton used to make the results flow back to the beginning. The answers from the Slaves are in the form (*candidate point, number of neighbors, set of neighbors*). Two factors make the structure effective in decoupling and distributing the workload to the slaves, that the Master selects single points of the input (doing no spatial searches), and the Slaves do not need accurate information about the cluster labelling done in the Master. All the Slaves need to search the R*-Tree structure, which is actually replicated. While in the sequential algorithm no already labelled point is inserted again in the visit queue, the process of oblivious parallel expansion of different parts of a cluster may repeatedly generate the same candidates. The Master process checks this condition when enqueuing candidates for the visit, but this parallel overhead is too high if all the neighboring points are returned each time a region query is made.

Two filtering heuristics are used in the Slaves to prune the returned set of points. Neighbors of non-core points do not become candidates for the current cluster, so only their number is returned. Previously returned points, on the other hand, are surely present in the visit queue or already labelled, so they are not sent again to the Master. We let each Slave maintain information about the points already returned by previous answers. We have verified, for the degree of parallelism in the tests, that pruning based on local information only is enough to avoid computation and communication overheads in the Master.

5.2 Results and analysis

The results reported are from the COW platform, using up to ten processors. The data are from the the Sequoia 2000 benchmark database, a real world dataset of 2-d geographical coordinates. DBSCAN was originally evaluated in [24] using samples from that database, D1 and D2, which hold 10% and 20% of the data. We also used the whole dataset (D3, 62556 points) and a scaled-up version (D4, 437892 points), made up of seven partially overlapping copies of the original dataset. The DBSCAN parameters were $MinPts = 4$ and $\epsilon \in \{20000, 30000\}$. We report tests with $p \in \{6, 8\}$, the parallel degree being the number of slaves. Efficiency is computed w.r.t. the resources really used, i.e. $p + 1$.

The parallel visit scheme we have implemented is quite effective. The performance gain on the two small files does not compensate the parallel overhead, but speedup and efficiency are consistently high when there is enough computation. We have also verified, by setting $\epsilon = 100000$, that a higher cost of the R*-Tree retrieve leads to nearly optimal efficiency even for file D1. This can easily be the case when the R*-Tree is out-of-core, or the dimension of the data space is high for the spatial index chosen.

Summing up, the parallel structure used is effective for high loads and large datasets which are impractical to deal with using the sequential algorithm. The D3 and D4 datasets, although small enough to be loaded in memory, require up to 8 hours of sequential computation. On the contrary, the efficiency of the parallel implementation rises with the load, thus

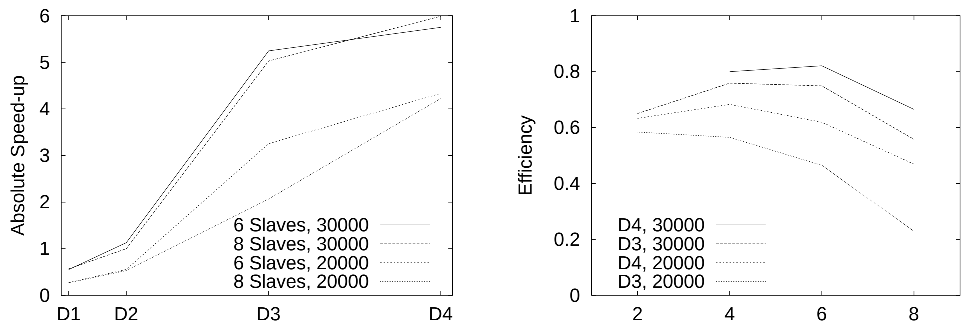

Figure 7: (a) Parallel DBSCAN speedup vs dataset, $p \in \{6, 8\}$ and $\epsilon \in \{20000, 30000\}$. (b) Efficiency vs parallelism, for the D3, D4 files and $\epsilon \in \{20000, 30000\}$

ensuring a good behaviour of the parallel DBSCAN even when the data is out-of-core. It is also possible to devise a shared secondary memory tree with all the Slaves reading concurrently and caching data in-memory when possible. Further investigation is needed to evaluate the limits of the heuristics used in pruning results, and possibly devise better ones, for a higher degree of parallelism. The parallel DBSCAN, like its sequential counterpart, is also general w.r.t. the spatial index used. It is thus easy to exploit any improvement or customization of the spatial data structure that may fasten the computation.

6 C4.5 Classification

Given a set of objects with assigned class label, the classification problem requires building a model of the class label in term of the characteristics of the objects, to understand their relation and eventually predict the class of new, unlabelled data. Many classifiers are based on induction of decision trees, and they rely on a common basic scheme. Here we recall the concepts and definition used in Quinlan's C4.5 classifier [27], which is the starting point of our work. We start from a tabular database $\mathcal{D}$ of *cases* with a attributes, each attribute being either *categorical* (discrete, finite domain) or *continuous*, i.e. real-valued. One of the categorical attributes is distinguished as the *class* of the case.

A *decision tree* T is a model of the data which describes the behaviour of the class attribute in terms the values of the others. The leaves of the tree are class-homogeneous subsets of the input. Each interior *decision* node is associated to a test on the values of the attributes, and has a child subtree for each outcome of the test. Starting from the root, each node is associated to a subset of $\mathcal{D}$. The test in the decision nodes define a partition of the cases among the child subtrees. Hence a path from the root to any leaf defines a series of tests that all cases in that leaf verify.

C4.5 decision nodes test a single attribute. A child node is made for each different value of a categorical attribute, while boolean tests of the form $x \lesseqgtr \textit{threshold}$ are used for continuous attributes. The algorithm is divided into two main phases, the building phase and the pruning and evaluation one. The former is the most time consuming, that has been parallelized. The building phase proceeds top-down, at each decision node choosing the test by exhaustive search over all the attributes. For each attribute, a cost function, the Information Gain (IG), is evaluated over the node data. The overall search is greedy, because it is based only on local evaluation and no backtracking is done. The algorithm has basically a Divide and Conquer (D&C) structure, except for a detail that we will discuss later. Building a node requires operating on the partition of the input that is associated to that node. The tree itself is a compact knowledge model, but the data partitions can be as large as the whole input. Ensuring efficiency and locality of data accesses is the main issue in building the decision tree.

Assuming that the data fit in memory, to evaluate the IG for a categorical attribute A histograms are computed of the couples (class, A) in the current partition, which require $O(n)$ operations per column. For the continuous attributes, to compute the IG we need the class column to be sorted according to the attribute. The cost of repeated sorting ($O(n \log n)$ operations) accounts for most of the C4.5 running time. Partitioning the data according to the selected attribute then requires a further sorting step of the whole partition. When data doesn't fit in memory, the above complexity results are in terms of I/O operation and virtual memory page faults, and the in-core algorithm quickly becomes unusable. External-memory algorithms and memory-hierarchy aware parallel decompositions are needed to overcome this limitation.

6.1 Related work on parallel classifiers

Several different parallel strategies for classification have been explored in the literature. Three of them can be considered as basic paradigms which are combined and specialized in the real algorithms. *Attribute* parallelism vertically partitions the data and distributes calculation over different columns.

Data parallelism uses horizontal partitioning of the data, and uses coordinate computation from all the processors to build each node. *Task* parallelism is the independent classification of separate nodes and subtrees. Each of these forms has its variants, which may use replicated or partitioned data structures, do static or dynamic load-balancing and computation grain optimization.

We will concentrate on the works based on the same C4.5 definition of information gain. Much of the research effort has been spent to avoid sorting the partitions to evaluate the IG, and to split the data using a reasonable number of I/O operations or communications. A commonly used variation is to keep the columns in sorted order, vertically partitioning the data. The drawback is that horizontal partitioning is done at each node split. Additional information and computations are needed to split the columns following the test on the selected attribute while keeping them sorted. Binary splits require some extra processing to form two groups of values from each categorical attribute, but simplify dealing with the data and make the tree structure more regular.

Many existing parallel algorithms address the problem of repeated sorting this way. The parallel implementations of the sequential SLIQ classifier [28] have either in-core memory requirements or communication costs which are $O(N)$ for each node. The SPRINT parallel algorithm [29] lowers the memory requirements and fully distributes the input over the processors, but still requires hash tables of size $O(N)$ to split the larger nodes. Joshi *et al.* [30] pointed out that this amount of communication per processor makes SPRINT still inherently unscalable. Their ScalParC uses a breadth-first level-synchronous approach in building the tree, together with a custom parallel hashing and communications scheme. ScalParC is memory-scalable and has a better average split communication cost, with a worst-case $O(N)$ per-level.

Our research has focused on developing a structured parallel classifier based on a D&C formulation. Instead of balancing the computation and communications for a whole level, we aim at a better exploitation of the locality properties of the algorithm. A similar approach is those in [31]. They propose as general technique for D&C problems a mixed approach of data parallel and task parallel computation. Substantially, at first all the nodes above a certain size are computed in a data-parallel fashion by all the processors. The smaller nodes are then classified using a simple task parallelisation. The problem of locality exploitation has been addressed also in [32] with a *Hybrid Parallelisation*. A level-synchronous approach is still

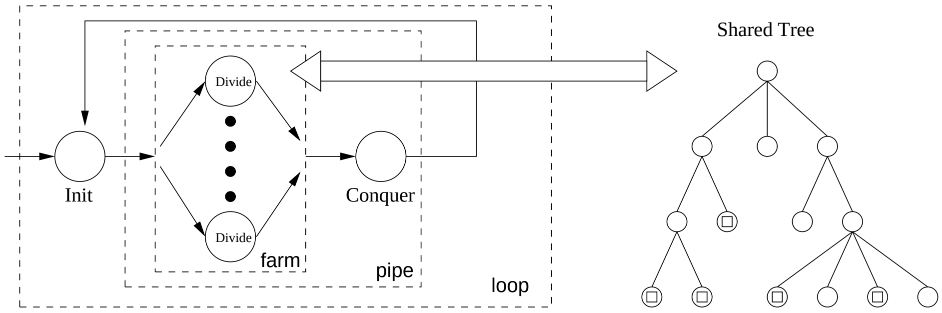

Figure 8: The C4.5 parallel structure.

used, but as the amount of communications exceeds the estimated cost of data reorganization, the available processors are split in two groups that operate on separate sets of subtrees.

6.2 Structured Parallelisation

The C4.5 algorithm is divide and conquer in its nature, but for the threshold calculation. As already noticed in [33], the exact thresholds are not needed during the building phase. We started from a task parallelisation approach. Each node classification operation is a task, which generates as sub-tasks the input partitions for the child nodes. To throttle the computation grain size, a single task computation may expand a node to a subtree of more than one level, and return as subtasks all the nodes in the frontier of the subtree. We already verified in [5] that if threshold calculation for continuous attributes is delayed until the pruning phase, the D&C computation can be exploited in a SkIE skeleton structure by means of application-level parallel policies. Removing the overhead proportional to N, the task computations become $O(n \log n)$ operation and much less irregular. It is then effective to schedule the tasks in size order, giving precedence to large tasks that generate more parallelism.

The skeleton structure in figure 8 implements the recursive expansion of nodes by letting tasks circulate inside a loop skeleton. A pipeline of two stages expands each task. The anonymous workers in the farm skeleton expand each incoming node to a separate subtree. The second stage in the pipe is a sequential *Conquer* process coordinating the computation. The template underlying the farm skeleton takes care of load-balancing, so its efficiency depends on the available parallelism an the computation to communication ratio.

In a previous version, all the input data were replicated in the workers, to make them anonymous, and the Conquer module was keeping locally the decision tree structure. Tree management, and the need to explicitly communicate partitioning information through the interfaces of all the modules, were severe bottlenecks for the program. We have designed a Shared Tree (ST) library, an implementation of a general tree object in shared memory, used to represent the decision tree T. Since data locality follows the evolution of the decision tree, the input is hold inside the ST, over the frontier of the expanding tree, and is immediately accessible from each process in the application. C4.5 is a D&C algorithm with a very simple conquer step, which simply consists in merging the subtrees back into T. All the operations required by the algorithm are done in the sequential workers of the farm. They access the shared structure to fetch their input data, they create the resulting sub-tree and store back the data partitions on its frontier.

The Conquer module is still present to apply the task *selection policy* we previously mentioned. A simple priority queue is used to give precedence to larger tasks, leading to a data-driven expansion scheme of the tree, in contrast to the depth-first scheme of sequential C4.5 and to the level-synchronous approach of ScalParC [30]. We also use a task *expansion policy.* We made a choice similar to that of [31] in distinguishing the nodes according to their size. In our case we balance the task communication and computation times, which influence dynamic load-balancing, by using three different classes of tasks. The base heuristic is that *large* task are expanded of one level only to increase available parallelism, *small* ones are fully computed sequentially, and intermediate ones are expanded to incomplete subtrees up to a given number of nodes and within computation time bounds. The actual limits were tuned following the same experimental approach described in our previous work [5]. For these tests, large task have more than 2000 cases (4% of the data), small ones less than 50 (0.1%), and sequential computation bounds are 1 second and 70 nodes. The input is the file *Adult* from the UCI Machine Learning Repository.

The results shown in figure 9 are good for a task parallelisation of C4.5. Next we have to move toward a full out-of-core computation. The key point in making the application scalable is to exploit parallelism in the processing of large data partitions. Our line of research differs from that of [31] because we aim at developing a general support for operating on large objects. It has been already shown in [34] that exploiting remote memory over high-speed networks can be faster than swapping to virtual memory. To operate on data that is outside the local memory we want to exploit collective parallel operations on huge data, as well as external memory algorithms [35] for

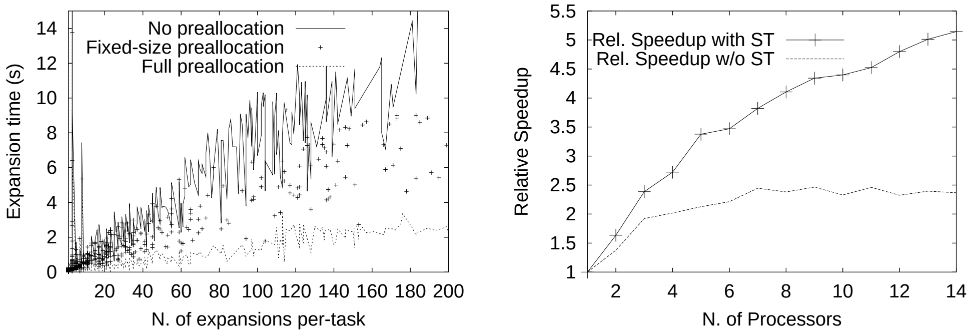

Figure 9: (a) Per-task completion time vs number of nodes in the subtree, with and without allocation optimization. (b) Relative Speedup with and w/o the ST.

a single processor. The application programmer sees these operations as methods of objects, and the run-time support will take care of selecting the appropriate implementation of the methods from time to time.

A first step has been accomplished with the implementation of the Shared Tree library. We see in 9b that it allows to reduce the centralized work and achieve a better scalability for the task parallelisation. Highly dynamic irregular problems stress all the components of the memory system, and the implementation of shared objects is still in its experimental stage. There is no space here to fully describe such a support, which is better outlined in [6]. Fig 9a shows the enhancement in the task computation time obtained by introducing a simple preallocation support to the dynamic shared memory handler.

7 Conclusive Remarks and Issues for PPE Enhancements

We have presented a set of commonly used sequential DM algorithms which were restructured to parallel by means of the SkIE parallel programming environment. The good reuse of application code and the ease of the conversion confirm the validity of the approach w.r.t. software engineering criteria. The parallel applications produced are nevertheless efficient and scalable. Performance results have been shown over different computer architectures, with low-level issues demanded to the environment support, and application tuning turned into parameter specification for high-level, clearly understandable user-defined policies.

In the three DM application we have seen there are apparently different access patterns: large block reads intermixed with long computations (Apriori partitioned), frequent small data accesses with poor locality (DBSCAN), data intensive computation with unpredictable reading and writing of a great amount of data, no chances to do static optimizations (C4.5). Actually, if we think about larger databases, all these data will eventually be pushed out-of-core. The R*-Tree used in DBSCAN should be shared and stored in mass memory, and the training data for C4.5 should not be limited by the size of the shared memory. Exploiting the local memories and the shared memory for caching, and applying external memory techniques where appropriate would make the structure of programs complex and unhandy.

In the modular, integrated view of the SPP - PPE, the explicit management of the interaction with parallel file systems and data bases, for instance, is an obstacle to portability. To simplify and make modular the interface to shared resources, we are studying the use of the object and component models. The experiments reported with a Shared parallel Tree data type show that this allows to improve the performance without sacrificing the advantages of the structured approach. There are currently some groups working to merge the object [36] and component [37] programming models with parallel programming. While in the former work some basic parallel computational patterns are recasted as object classes and design patterns, the latter underlines the gain in code reuse by using standard interface definition languages (IDL).

Our attention is to the integration issues, to implement the objects is such a way that it is based on the experiences gained so far, and it exposes uniform interfaces both toward the application programmer and to the surrounding system environment. Of course, a common way of accessing shared data and I/O services is a starting point to add interfaces to several standard technologies extensively used in the field of KDD. Parallel file systems, DBMS, CORBA services should transparently be used as data sources and destinations, both at the module interface level and from within the module code.

References

[1] M. Vanneschi, Heterogeneous HPC Environments, in: D. Pritchard, J. Reeve (Eds.), Euro-Par '98 Parallel Processing, Vol. 1470 of LNCS,

Springer, 1998, pp. 21–34, invited talk.

[2] M. Vanneschi, PQE2000: HPC Tools for Industrial Applications, IEEE Concurrency: Parallel, Distributed & Mobile Computing 6 (4) (1998) 68–73.

[3] B. Bacci, M. Danelutto, S. Pelagatti, M. Vanneschi, SkIE : A heterogeneous environment for HPC applications, Parallel Computing 25 (13–14) (1999) 1827–1852.

[4] P. Becuzzi, M. Coppola, M. Vanneschi, Mining of Association Rules in Very Large Databases: a Structured Parallel Approach, in: Euro-Par'99 Parallel Processing, Vol. 1685 of LNCS, Springer, 1999, pp. 1441–1450.

[5] P. Becuzzi, M. Coppola, S. Ruggieri, M. Vanneschi, Parallelisation of C4.5 as a Particular Divide and Conquer Computation, in: Rolim et al. [38], pp. 382–389.

[6] G. Carletti, M. Coppola, Structured Parallel Programming and Shared Objects: Experiences in Data Mining Classifiers, submitted for publication (2001).

[7] D. Arlia, M. Coppola, Experiments in Parallel Clustering with DB-SCAN, to appear in the Proceedings of the Euro Par '01 conference, LNCS (2001).

[8] M. Vanneschi, ASSIST: the ASI-PQE 2000 Parallel Programming Environment, Tech. rep., ASI / PQE 2000 program (Jan. 2001).

[9] W. A. Maniatty, M. J. Zaki, A Requirement Analysis for Parallel KDD Systems, in: Rolim et al. [38], pp. 358–365.

[10] G. Williams, I. Altas, S. Bakin, P. Christen, M. Hegland, A. Marquez, P. Milne, R. Nagappan, S. Roberts, The Integrated Delivery of Large-Scale Data Mining: the ACSys Data Mining Project, in: Zaki and Ho [39], pp. 24–54.

[11] D. B. Skillicorn, D. Talia, Models and languages for parallel computation, ACM Computing Surveys 30 (2) (1998) 123–169.

[12] D. B. Skillicorn, Foundations of Parallel Programming, Cambridge University Press, 1994.

[13] J. Darlington, Y. Guo, H. W. To, J. Yang, Skeletons for Structured Parallel Programming, in: Proc. of the 5th SIGPLAN Symp. on Principles and Practice of Parallel Programming, 1995, pp. 19–28.

[14] D. Skillicorn, Strategies for Parallel Data Mining, IEEE Concurrency 7 (4) (1999) 26–35.

[15] M. J. Zaki, Parallel and Distributed Data Mining: An Introduction, in: Zaki and Ho [39], pp. 1–23.

[16] S. Partharasarty, S. Dwarkadas, M. Ogihara, Active Mining in a Distributed Setting, in: Zaki and Ho [39], pp. 65–82.

[17] S. Bailey, E. Creel, R. Grossman, S. Gutti, H. Sivakumar, A High Performance Implementation of the Data Space transfer Protocol (DSTP), in: Zaki and Ho [39], pp. 55–64.

[18] U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, R. Uthurusamy (Eds.), Advances in Knowledge Discovery and Data Mining, AAAI press / MIT press, 1996.

[19] A. Mueller, Fast sequential and parallel algorithms for association rule mining: A comparison, Tech. Rep. CS-TR-3515, Dept. of Computer Science, Univ. of Maryland, College Park, MD (Aug. 1995).

[20] D. Gunopulos, H. Mannila, R. Khardon, H. Toivonen, Data mining, hypergraph transversals, and machine learning (ext. abstract), in: PODS '97. Proc. of the 16th ACM Symposium on Principles of Database Systems, 1997, pp. 209–216.

[21] A. Savasere, E. Omiecinski, S. Navathe, An efficient algorithm for mining association rules in large databases, in: U. Dayal, P. Gray, S. Nishio (Eds.), VLDB '95: Proc. of the 21st Int. Conf. on Very Large Data Bases, Zurich, Switzerland, Morgan Kaufmann Publishers, Los Altos, CA, 1995, pp. 432–444.

[22] R. Agrawal, J. Shafer, Parallel mining of association rules: Design, implementation and experience, IEEE Transactions on Knowledge and Data Engineering 8 (6).

[23] K. Beyer, J. Goldstein, R. Ramakrishnan, U. Shaft, When Is "Nearest Neighbor" Meaningful?, in: C. Beeri, P. Buneman (Eds.), Database Theory - ICDT'99 7th International Conference, LNCS, 1999, pp. 217–.

[24] M. Ester, H.-P. Kriegel, J. Sander, X. Xu, A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise, in: Proceedings of KDD '96, 1996.

[25] N. Beckmann, H.-P. Kriegel, R. Schneider, 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, 1990, pp. 322–331.

[26] S. Bertchold, D. A. Keim, H.-P. Kriegel, The X-Tree : An Index Structure for High-Dimensional Data, in: Proc. of the 22nd International Conf. on Very Large Data Bases, 1996, pp. 28–39.

[27] J. Quinlan, C 4.5: Programs for Machine Learning, Morgan Kaufmann, San Mateo, 1993.

[28] M. Mehta, R. Agrawal, J. Rissanen, SLIQ: A fast scalable classifier for data mining, in: Proceedings of the 5th Int.l Conference on Extending Database Technology, 1996.

[29] J. Shafer, R. Agrawal, M. Mehta, SPRINT: A Scalable Parallel Classifier for Data Mining, in: Proceedings of the 22nd VLDB Conference, 1996.

[30] M. V. Joshi, G. Karypis, V. Kumar, ScalParC: A New Scalable and Efficient Parallel Classification Algorithm for Mining Large Datasets, in: Proceedings of 1998 International Parallel Processing Symposium, 1998.

[31] M. K. Sreenivas, K. AlSabti, S. Ranka, Parallel Out-of-Core Divide-and-Conquer Techniques with Application to Classification Trees, in: Proceedings of the International Parallel Processing Symposium (IPPS/SPDP), Puerto Rico, 1999, pp. 555–562.

[32] A. Srivastava, E.-H. Han, V. Kumar, V. Singh, Parallel Formulations of Decision-Tree Classification Algorithms, Data Mining and Knowledge Discovery: An International Journal 3 (3) (1999) 237–261.

[33] J. Darlington, Y. Guo, J. Sutiwaraphun, H. W. To, Parallel Induction Algorithms for Data Mining, in: Advances in intelligent data analysis: reasoning about data IDA'97, Vol. 1280 of LNCS, 1997, pp. 437–445.

[34] M. Oguchi, M. Kitsuregawa, Using Available Remote Memory for Parallel Data Mining Application, in: 14th International Parallel and Distributed Processing Symposium, 2000, pp. 411–420.

[35] J. S. Vitter, External Memory Algorithms and Data Structures: Dealing with MASSIVE DATA, Draft, http://www.cs.duke.edu/~jsv (Jan. 2000).

[36] D. Goswami, A. Singh, B. R. Preiss, Using object-oriented techniques for realizing parallel architectural skeletons, in: Matsuoka et al. [40], pp. 130–141.

[37] B. Smolinski, S. Kohn, N. Elliott, N. Dykman, Language Interoperability for High-Performance Parallel Scientific Components, in: Matsuoka et al. [40], pp. 61–71.

[38] J. Rolim, et al. (Eds.), Parallel and Distributed Processing, Vol. 1800 of LNCS, Springer, 2000.

[39] M. J. Zaki, C.-T. Ho (Eds.), Large-Scale Parallel Data Mining, Vol. 1759 of LNAI, Springer, 1999.

[40] S. Matsuoka, R. Oldehoeft, M. Tholburn (Eds.), Computing in Object-Oriented Parallel Environments, 3rd Int.l Symp., ISCOPE 99, Vol. 1732 of LNCS, Springer, 1999.