Mining of Association Rules in Very Large Databases: a Structured Parallel Approach

P. Becuzzi, M. Coppola, and M. Vanneschi

Dipartimento di Informatica, Università degli Studi di Pisa, Italy
{becuzzi,coppola,vannesch}@di.unipi.it

Abstract Newer and newer parallel architectures being developed raise a strong demand for high-level and programmer-friendly parallel tools. We show some results regarding mining of association rules, a well-known Data Mining algorithm, which we ported from sequential to parallel within the PQE2000/SkIE environment. The main goals achieved are the low effort spent in parallelizing the code, the machine independence of the application produced, source code portability and performance portability. Here we report test results for the same parallel program on three different architectures.

1 Introduction

During the last years the problem of extracting valuable knowledge from bigger and bigger amounts of automatically collected data has become of greater industrial relevance. In the field of Data Mining new techniques and tools are being developed [3, 7, 11] to cope with the growing size of modern databases, and parallel computing is widespreadly seen as the main way to High Performance Data Mining. This sharpens the demand of software engineering techniques and environments for programming parallel computers: the industrial need for short time-to-market of application software conflicts with the typically very long development time on today's parallel platforms.

The aim of the PQE2000 project [10] is to transfer the research results in the field of parallel architectures, structured parallel programming tools and strategic applications to the industry. Knowledge Data Discovery is definitely one field in which the technological transfer is needed.

This work is part of a joint project for High Performance Tools for Tax Fraud Detection, carried on by the PQE2000 project, the Italian Ministry of Finance and SOGEI SpA [2]. The first step in developing such an applications is the selection of a set of Data Mining kernels as ordinary and parallelization case studies. In this paper we show some results regarding the classic problem of mining of association rules, a well-known Data Mining algorithm. We ported

* Computing resources for this research were provided both by C3P center of C.N.R. (STRIDE project, co-founded by the EC) and by Italian M.U.R.S.T. ("Design Methodologies and Tools of High Performance Systems for Distributed Applications" – MOSAICO project).

from sequential to parallel the code for finding the *frequent sets* of a database of transactions, working within the PQE2000/SkIE environment.

The main goals achieved are the low effort spent in parallelizing the code, the machine independence of the application produced, source code portability and performance portability. Here we report test results for the same parallel program on three different architectures.

The next section contains the basic definition of our subproblem and an overview of the sequential and parallel algorithms involved. Section three defines our target, the quick development of a portable application, introduces to the SkIE programming environment and to the parallel systems that we used. Section four describes the structure of the prototype we have realized, discussing various design issues. Their impact on performance is explained through section five. The last section evaluates the experiment with respect to the initial target, and presents future research and development directions.

2 The Frequent Itemsets problem

The problem of discovering simple association rules in databases has been proposed back in 1993. We'll refer to [3], chapter 12, for exact definitions and a thorough theoretical background regarding the generation of association rules. It's enough to say that the rules are easily produced once the subproblem of finding *frequent sets* in the data is solved. It's the most heavy sub-task, most of the efforts in the literature being addressed to its efficient solution.

Given a set $\mathcal{I} = \{i_1, \ldots, i_m\}$ of items, we will call *transaction* a pair T of a unique identifier and a set of items; a *k-itemset* will be a simple set of k items. Our database $\mathcal{D}$ will be a set of transactions. Itemsets can contain each other, and be contained in a transaction. The parameters of the problem are the set of attributes $\mathcal{I}$, the database $\mathcal{D}$ and a fixed real number $0 < s < 1$, the *minimum support*. The support $S(X)$ of an itemset $X \subseteq \mathcal{I}$ is the fraction of transactions in $\mathcal{D}$ that contain that itemset; if $S(X) > s$, then we say that X is *frequent*, or has (at least) minimum support. We want to find all itemsets that are frequent in the given database $\mathcal{D}$, and compute their support counts.

The itemsets inherit a natural order by inclusion from being subsets of the set of attributes $\mathcal{I}$. They form a lattice structure, and since the minimum support property for a set implies the same holds for all its subsets, the frequent sets are the union of a class of sub-lattices: we want to discover this structure. Several approaches are viable, most of them are reviewed in [11, 12] together with a theoretical background. The one we have chosen for our experiment is the bottom-up exploration done by the Apriori algorithm. The key idea is that since supersets of non-frequent itemsets are non frequent we can build up the set of frequent sets levelwise, instead of exploring the whole power set of $\mathcal{I}$.

From the set L_k of frequent k-itemsets we build a candidate set of $(k+1)$-itemsets C_{k+1}, using as heuristic filter the property that for each element $x \in C_{k+1}$, all k-itemsets in x belong to L_k. The size of C_{k+1} can be much less than that of $(\mathcal{I})^{k+1}$. A linear scan of the database is required at each pass to extract

L_k from C_k, and as much passes as the length of the biggest frequent itemset. Pruning candidates using information gathered during the previous pass is not effective at the second one: $C_2 = L_1 \times L_1$ is often roughly the same as $\mathcal{I} \times \mathcal{I}$. Of course the algorithm requires temporary data structures holding information about L_k and C_k: so the amount of memory needed can be quite large.

We regard the parallel solutions of Apriori as belonging to one of three main classes outlined in [1]. **Count Distribution** solutions replicate the candidate set at each node, and partition the database among the processors. The itemset counting is distributed, global communications are needed at each step of Apriori. **Data Distribution** parallelizations spread the candidate set over the nodes, regardless of its structure. Partitioning a big database requires "local" transaction data to travel to most or all of the processors at each pass. **Candidate Distribution** programs partition the candidate set trying exploit its structure, so that the database can be accordingly partitioned in order to minimize or avoid subsequent communications, and/or load imbalancing.

The three approaches deal in different ways with the problem of keeping in the local memory part of the transaction data and the intermediate counts. A sequential algorithm that addresses the same problem is the **Partition** scheme.

It was introduced in [7] to allow efficient use of the main memory, reducing the I/O burden due to making repeated scans of the data in the main loop of Apriori. Here the computation is divided in two consecutive *phases*, and the database is partitioned.

Phase I computes frequent itemset information $\{\forall k, {}^jL_k\}$ separately for each slice j of the data, working with the minimum support specified. It's easy to prove that a globally frequent k-itemset must belong to at least one ${}^jL_k$. Hence the family of sets $\mathcal{H} = \{k > 0 \,|\, \cup_j \, ({}^jL_k)\}$ must contain all of the frequent k-itemsets, and is an upper approximation of the solution of the problem.

Phase II has a simpler structure and lower computational load. It extracts the solution computing itemset counts with a linear scan of the whole data.

Parallel implementations of Partition are akin to Count parallelizations, as they scale up almost linearly with DB size, but cannot easily handle very low minimum support (very large L_k and C_k sets) because of the memory requirements. Some previous results about this kind of parallelization of Apriori can be found in [6].

The statistical properties of the data can vary among partitions: too much different ${}^jL_k$ lead to a very coarse approximation of the true global results. The problem is known as *data skew*, and increases memory usage and computational load of phase II. Sequential Partition usually avoids this pitfall, since to minimize the overall workload the partitions in phase I are kept as big as possible. The parallel implementation however has to find a tradeoff between the total amount of computation and the degree of exploitable parallelism, bounded by the number of partitions. We just mention that data skew can be solved by hashing the database unless the partition size is too small with respect to data size and minimum support.

3 Target and environment of the experimentation

Parallelizing a non trivial, data-intensive sequential code is a severe test of the efficiency of a programming environment. Our main requirements were:
1) no limits imposed on database size to be dealt with; **2)** high reuse of sequential code, with more efforts in modularization than in writing ad-hoc code; **3)** ease of integration with other algorithms and within more complex application; **4)** to exploit as much as possible the portability of code and performance; **5)** to shorten as much as possible the development time of the program.

Some of the conditions were easily fulfilled by working with SkIECL, the coordination language of our programming environment. It allows very simple exploitation of parallelism between code modules, as well as seamless substitution of sequential modules with semantically equivalent, parallel ones.

We started from an "off-the-shelf" Apriori implementation of average performance. The original structure has then been changed into a partition scheme. The choice of a parallel partition scheme comes up from the two main facts that Parallel Partition schemes scale well with respect to increasing database size and processor number, and that they can be easily combined with other parallelization schemes by substituting the Apriori module in the first phase. This is even easier when using a structured coordination language.

We didn't spent much time in tuning the algorithm, the main effort in modifying the sequential code was to make it fully modular, so to allow the different processes to share the same high-level library code for the operations, just exchanging data when needed. Having this done, it was straightforward to select the functions to parallelize, insert the corresponding modules into parallel constructs and starting to run tests.

The structure and features of the SkIE programming environment can't be described in full here. We refer to some of the literature about the environment and the compilation techniques [9, 10], just outlining here the basis of the approach and the technical details that are strictly necessary to the analysis of our prototype. SkIECL is a coordination language, designed to allow easy parallel composition of high-level modules written in different languages. A parallel application is developed by integrating (possibly existing) program modules in a SkIECL framework. Parallelism can be expressed only by means of *parallel constructs* or *skeletons*, which are machine-independent and fully compositional. Every communication detail is handled by the compiler, as well as some forms of compile-time optimisation that are based on profiling data and expected performance models.

Our sequential code is mostly C++ routines that call some class methods defined in the sequential version of the application. The classes were compiled to a standard linked library, while the code of the main algorithm was encapsulated using some of the SkIECL constructs. Apart from the seq construct, that simply defines the interfaces of a sequential module, we used the constructs pipe (expresses the pipeline composition of the contained modules), farm (functional replication of the contained *worker* construct over a stream of independent tasks, with load-balancing) and comp (used to express functional composition of two pipe skeletons without parallel execution). The first two skeleton are tailored to stream parallelism, while the comp in its general form is designed to express data parallel computations.

The very beginning of the development relied upon an heterogeneous cluster of LINUX machines over ordinary Ethernet. We have then been able to make extensive tests over three parallel architectures, all of them using MPICH, of those supported by the SkIECL compiler:

CS-2 — a QSW Computing Surface 2 with 24 processing nodes, (2 ×130Mhz HyperSPARC processors, 128Mb RAM) with a proprietary fat-tree network. This machine is in fact the MIMD portion of one of the PQE2000 prototypes.
LINUX — a cluster of LINUX CPUs made up of 10 Pentium II (266Mhz, 128 Mb RAM), that uses Fast-Ethernet and a crossbar hub as communication network.
SMP — a SMP workstation (4 Ultra-SPARC 250Mhz, 256Mb RAM).

4 Structure of the prototype

When designing a parallel version of the Partition scheme, keeping the data inside the local memory of each processor turns out to reduce the global communications to just two: the first one at the end of the first phase, to globally compute the approximation of the result set, the second one at the end of the algorithm, to exchange counting information for the itemsets.

We see in Fig.1 the overall structure of the prototype. It is a comp of the two Partition phases, which do not compute in parallel, each one realized as a pipe of modules, with the slower ones functionally replicated inside farm constructs. The sequential modules 1–3 realize Phase I of the Partition algorithm, modules 4–6 Phase II. There you see also support processes (d,c) automatically generated by the compiler to act as data dispatchers and collectors inside each farm construct.

The database is read by process 1, and flows as a stream of partitions through the pipeline of following stages. Module 2, which is functionally replicated, applies Apriori to each partition in its incoming stream, producing a stream of hash-tree structures holding local results. Module 3 collapses this stream, computing the upper approximation $\mathcal{H} = \{k > 0 | \cup_j \ ({}^j L_k)\}$ of the result.

Phase II starts with reading again the database and producing a stream of partitions; the verification work is done in module 5, also replicated by a farm, and $\mathcal{H}$ is sent to all the worker processes as initialization data. Module 6 collects the stream of itemset support counts and sums them to produce the final results. It also produces the association rules using the sequential algorithm.

In principle there is no need to exploit the same degree of parallelism or use the same partition size in both phases, but in our case using a different number of workers for the two farm is useless. Even centralized reading of the data inside modules 1 and 4 is not required by the algorithm. In the beginning, we decided to do no preprocessing or off-line partitioning of the data on local disks.

In our view, repeatedly mining the data (like in meta-learning algorithms) would be carried out through some data-mart support, with data filtering parameters that may change each run. Having a single input point for the whole

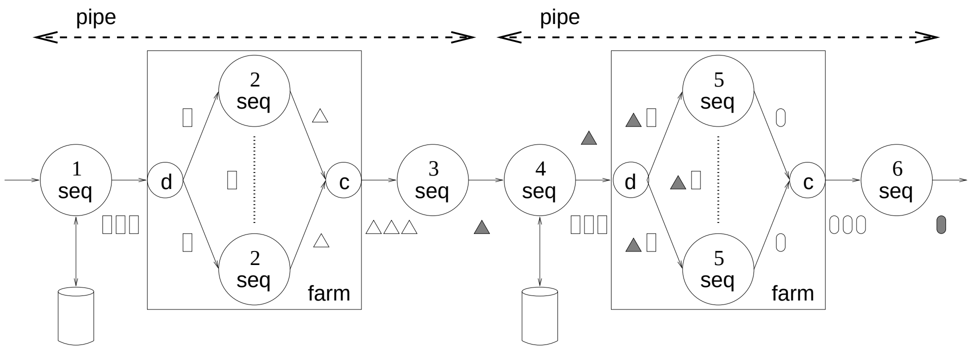

Figure1. Block structure of the prototype

algorithm makes allowance for filter and interface functions, and is not necessarily a performance loss if the data come from a separate database server.

We know that the exploitable parallelism degree in a farm is bounded by the ratio between communication/scheduling time and computation time of the worker nodes. Since some of the data structures may become quite large, we must check that the communication network does not congest.

This is not a sharp bound for systems with a high performance interconnection network (CS2), or with communication happening efficiently in memory (small SMP like the one we used). The test results on these architectures are measured with the centralized file read module that we have just described.

However, on the LINUX cluster the many software layers between the MPI API and the Ethernet, and this being a slower device, can create a bottleneck. We changed our code to distribute only partition references, and load the data in modules 2 and 5 from local disks. The data has not been partitioned: each node has a copy of the whole file, and the load distribution remains dynamic. This kind of code changes are made easy and quick by the high level parallel language, requiring only limited editing of the source code and its SkIE interface definitions. The net effect on the LINUX cluster has been to cut down the completion times nearly by one third, down to the figures shown in the following. We are now planning to use the very same code on the CS-2 at first to exploit its local disks, then to develop an interface to the parallel file system.

5 Performance analysis

The test files used to measure the performance of the program were generated with the Quest synthetic data generation code, used in the literature as a common reference. An explanation of the underlying data model can be found in [3]. In this paper we show results obtained from files having average transaction length of 20 items, and frequent itemsets of average size of 6. These parameters provide a heavy combinatorial load to the Apriori module, hitting the weak point of simple count/partition parallel schemes. Our test databases are made up of 1, 4 and 12 million transactions, and are about 90, 360 and 1100 Mbytes in size.

Results over the CS-2 are shown in Fig. 2. We must keep in mind that direct comparison between CPUs built with different technologies is not fair. For our purposes, we can say that the three architectures used are ordered by the computation/communication time ratio, with the LINUX (the fastest CPUs and the slowest network) having the lowest value, and the SMP the highest.

In Fig.3b the fall of the speed-up curves for the 90Mbyte file at 2% shows that raising the parallelism degrees the sequential workload (reading the data and computing the set of rules) begins dominating the completion time.

The CS2 and SMP are less vulnerable to this effect, as is clear by the comparison of the two tests with centralized reading (the ones with a *), and from the first tests on the SMP, Fig.3a. The SMP results are amazingly close to ideal speedup limit for four processors: the high speed of inter-node communication can partly explain the figures, but more tests are needed to evaluate the influence of various other factors, like the interaction between the database size and the amount of memory used for disk buffering.

In the general setting, we can reach substantially better results by removing the two sequential bottlenecks of reading the data, and using bigger databases.

Our view is confirmed by the results: we can see the effect of removing the centralization in Fig.3b for 90 and 360 Mbytes files at 2%. Fig.4 shows a smooth behaviour of the program, and that the speedup increases with the computational load (more data and/or lower support).

Apart from the results taken at 0.5%, we note that the program reaches an asyntotic behaviour, because the speedup does not change significantly (Fig.4b). The 0.5% minimum support value is not directly comparable to the other results: the speedup is even higher, but we had to remove the sequential generation of association rules. The intermediate data were becoming too big, and the sequential part too slow, meaning that rule generation needs to be parallelized as well at this point.

We point out that the limit on the exploitable parallelism comes from the nature of the problem: for too small partitions no Partition scheme is effective. We ran tests and verified that very small partitions slow down the program as expected, but then the performance is almost unaffected until the data structures exceed the main memory size.

We conclude that communications and I/O do not impose severe restrictions on this parallelization, and more performance improvement can come from substitution of the algorithm inside module 2 and from the introduction of some degree of parallelism in the association rule generation of module 6.

6 Evaluation of the results and future work

To evaluate the re-engineering costs of the parallelization, consider first that the original sequential C++ code was about 2900 lines; the modular version is about 3550 lines, with code mainly added to dump the status of some objects.

The main file of the final prototype consists of 350 lines of parallel code (SkIECL and C++) replacing the original sequential main file of 220 lines.

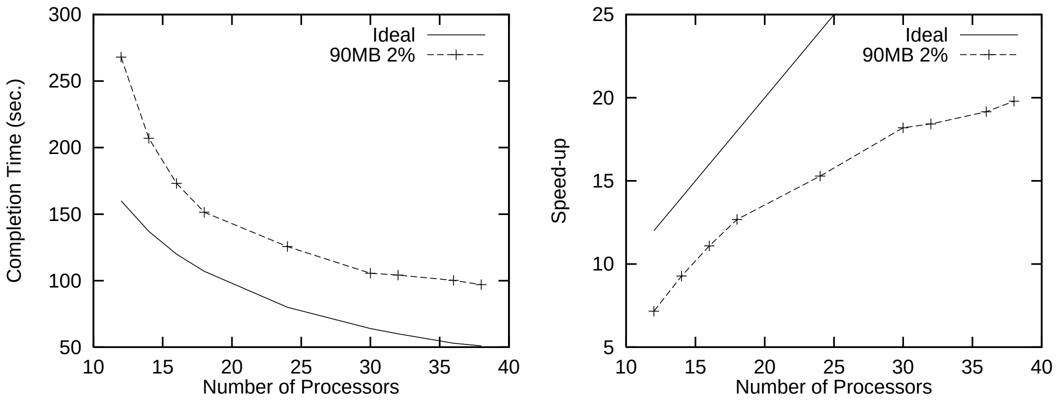

Figure2. CS-2 parallel completion time and speed-up, T20.I6.D1M

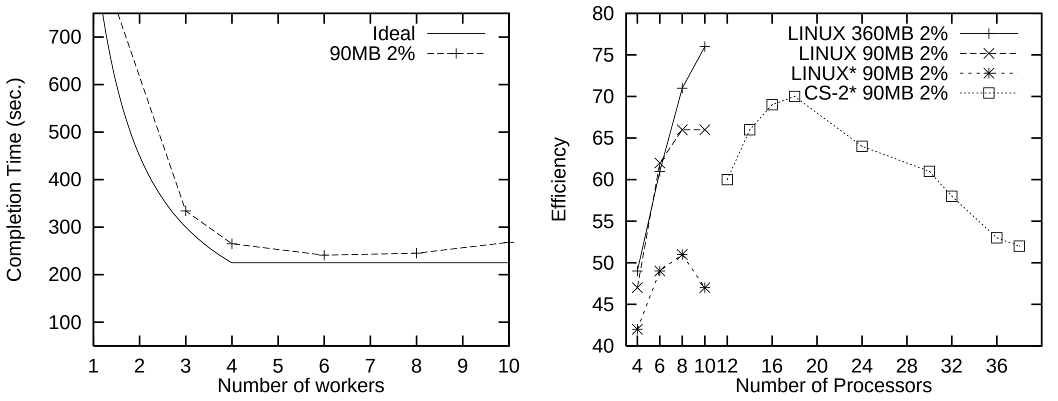

Figure3. (a) SMP completion time versus workers, up to 4 processors; (b) program efficiency over LINUX and CS-2 with support set at 2%.

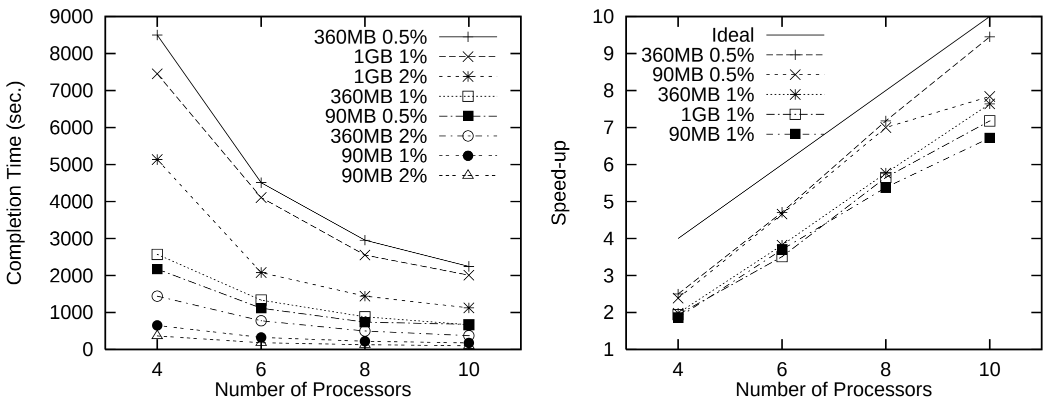

Figure4. LINUX parallel completion time and speed-up, T20.I6.D1M / 4M / 12M

Relative to the original code size, the percentage of reused sequential code is over 90%, while new code has been written that is 22% (sequential part) and 12% (parallel). The overall size of the source code has grown by less than 25%. The amount of time used for development and testing is up to now about three man-months, with two months spent on the sequential code, and one more to test and try the different parallel solutions.

Devised a global structure for the program, we started with the parallelization of the slowest portion of the code (phase I), moving to more complex solutions (farm in phase II, distributed I/O) as early as it was needed to improve performance. This process is incremental, because the effort spent in parallelizing one section is never wasted by changes in other modules.

We have reached the goal of quickly producing a compact, reasonably efficient parallel code that can be both practically used and further improved along the same path. No bound has been forced on the size of the database. The limit of the developed application is only the size of the intermediate data, as every Partition or Count parallelization of Apriori, but the underlying structured model allows us to substitute Apriori with a more scalable and/or parallel algorithm.

Compared to the small effort required by parallelizing the code, the results are good: we succeeded in parallelizing an existing high-level application code with no deep insight of its structure, achieving good speedup and scalability over the original program.

There are results in the field which outperform our sequential code, like in [7], where the Partition algorithm is introduced. Nevertheless, if we compare with [12] we discover that our prototype on all the three machines outperforms the algorithm presented on dataset with the same structure. The CS-2 and LINUX have a better completion time, while the SMP can reach nearly the same performance using 4 processor instead of 32. Notably, the sequential algorithm on the CS-2 is nearly 3 times slower than the sequential Par-MaxClique, but the parallel version is nearly twice as fast when using 32 processors.

From the test results is clear that the same source code is of practical use from small, low-cost cluster of PC's to real parallel machines. This saving of programming efforts has no comparison within the literature, with the parallel programs being architecture-tailored (not portable) or written using communication libraries. See for instance how much attention is due to implementation details, like communication cost analysis or synchronizations, in [1] or [8]. Besides, the good results maintained across different architectures are also an evidence that structured parallel environments can promote restructuring of existing programs instead of development of new ones from scratch.

We plan to complete the test set for the algorithm with much larger databases, and to improve the behaviour with respect to decreasing minimum support for the itemsets. Most of the optimisations devised could benefit from the availability of a virtual shared memory support, to scale-up the size of data structures beyond the limit of a single node memory. We could tune up the sequential Apriori code; a better solution would be to completely replace it with another algorithm. New, more efficient algorithms exist that cut down the average computational complexity of finding the frequent itemsets, for instance see [4] and [11]. The newer algorithms, which are closely related to the Candidate Distribution, as well as the ones that exploit only Data Distribution allow us to handle very large candidate sets, and lower the minimum support for the search. They can be used together with the partition approach to process huge databases, and can also be mixed with a Count Distribution scheme just like in [5].

References

[1] R. Agrawal and J.C. Shafer. Parallel mining of association rules: Design, implementation and experience. *IEEE Transactions on Knowledge and Data Engineering*, 8(6), December 1996. IBM Research Report RJ 10004, January 1996.

[2] P. Becuzzi, M. Coppola, D. Laforenza, S. Ruggieri, D. Talia, and M. Vanneschi. Data analysis and data mining with parallel architectures: Techniques and experiments. Technical report, project "Parallel Intelligent Systems for Tax Fraud Detection", December 1998.

[3] U. M. Fayyad, G.Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors. *Advances in Knowledge Discovery and Data Mining*. AAAI press / MIT press, 1996.

[4] D. Gunopulos, H. Mannila, R. Khardon, and 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, May 1997, Tucson, Arizona*, pages 209–216, New York, 1997. ACM Press.

[5] E.H. Han, G. Karypis, and V. Kumar. Scalable parallel data mining for association rules. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, volume 26,2 of *SIGMOD Record*, pages 277–288, New York, May13–15 1997. ACM Press.

[6] Andreas Mueller. Fast sequential and parallel algorithms for association rule mining: A comparison. Technical Report CS-TR-3515, Dept. of Computer Science, Univ. of Maryland, College Park, MD, August 1995.

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

[8] T. Shintani and M. Kitsuregawa. Hash based parallel algorithms for mining association rules. In *PDIS '96: 4th Int. Conf. on Parallel and Distributed Information Systems*, pages 19–30, Los Alamitos, Ca., USA, December 1996. IEEE Computer Society Press.

[9] M. Vanneschi. Heterogeneous HPC Environments. In David Pritchard and Jeff Reeve, editors, *Euro-Par '98 Parallel Processing*, volume 1470 of *LNCS*, pages 21–34, Southampton, UK, September 1998. ACM / IFIR, Springer-Verlag.

[10] M. Vanneschi. PQE2000: HPC tools for industrial applications. *IEEE Concurrency: Parallel, Distributed & Mobile Computing*, 6(4):68–73, Oct-Dec 1998.

[11] M.J. Zaki. *Scalable Data Mining for Rules*. PhD thesis, University of Rochester, Rochester, New York, 1998.

[12] M.J. Zaki, S.Parthasarathy, and M.Ogihara. Parallel algorithms for discovery of association rules. In *Data Mining and Knowledge Discovery*, volume 1. Kluwer Academic Publishers, 1997.