In Proc. 7th Intl. Conf. on High Performance Computing and Networking HPCN Europe, April 12-14, 1999, Amsterdam, The Netherlands, LNCS 1593, Springer Verlag, pp.884–894.

Java as a Basis for Parallel Data Mining in Workstation Clusters

Matthias Gimbel, Michael Philippsen, Bernhard Haumacher,
Peter C. Lockemann, and Walter F. Tichy

Universität Karlsruhe, Department of Computer Science
Am Fasanengarten 5, D-76128 Karlsruhe, Germany
gimbel@ira.uka.de
http://wwwipd.ira.uka.de/RESH/

Abstract. The exploitation of hidden information from large datasets by means of data mining techniques suffers from long response times. We address this problem by using the processing power of workstation clusters and have studied the performance of OLAP queries as a first step towards a portable data mining platform.
The results of our study suggest that with the availability of parallel workstation clusters that are equipped with high performance communication networks, fine-grained and communication-intensive parallelizations of queries are promising – even though they are considered too costly in traditional database systems.
The paper describes our Java framework for parallel OLAP-type query execution, necessary optimizations to the standard Java implementation, and analyzes the performance of non-standard parallel execution schemes on a workstation cluster.

1 Introduction

The need for and the benefits of data mining have been commonly accepted by the business and scientific communities. There is no dearth either of preprocessing techniques, mining methods, and postprocessing facilities. Unfortunately though, there are still huge difficulties to exploit the potential of data mining.

One difficulty is the processing time. It can take hours or even days for an algorithm to produce a result. Data mining is computationally expensive due to the massive amounts of data and due to the complexity of knowledge discovery algorithms. Even worse, because of the explorative nature of the knowledge discovery process (the preprocessing steps and the learning algorithms offer many parameters to experiment with), there exists a strong interest to make knowledge discovery an interactive process – with according performance requirements.

One answer to the performance demands is the use of parallelism. In traditional data base systems coarse-grained concurrency has been demonstrated to be appropriate for inter- and intra-transaction parallelism. It is an open question whether this design is well-suited for the different nature of data mining, where the algorithmic nature of of data exploration seems to call for fine-grained

parallelism. Indeed, we suspect that parallelism in data mining covers the entire spectrum from coarse to fine granularity, where the former is more suited to the early preprocessing stages which handle vast amounts of data, whereas the latter is better suited when it comes to the complex processing of data.

Since networks of workstations with high-performance communication hardware have made significant progress, not only do they become an alternative to expensive parallel machines but at the same time they promise the desired continuous spectrum in data mining granularity. On the other hand they are a fairly new concept that may require new answers to parallelism in data processing.

As a rough first approach one may equate the building blocks of data mining with OLAP[1] queries. Therefore, in order to explore the impact of parallelism in networks of workstations, we have studied the performance behavior of OLAP queries on this platform. We have developed a Java-framework for query execution with various execution strategies. The results are promising: since latency and bandwidth of current networks of workstations are no longer the main problem, fine-grained parallelism has an important place in data mining.

At first glance, the choice of Java may sound surprising. However, we targeted a portable solution and we expected to benefit from the optimized versions of Java's object serialization and RMI (remote method invocation) that have been developed along our way. Moreover, the results provide further directions in the development of the Java platform as well as parallel query processing and data mining techniques in general.

The paper is organized as follows: First, we give a short survey of related work. In Section 3 we describe the execution of OLAP-Queries. Section 4 introduces JavaParty, the parallel Java infrastructure we have used, and its central performance features, i.e. fast serialization and RMI. In Section 5 we discuss the implementation of our execution engine in JavaParty. The benchmarks of Section 6 suggest that non-standard query parallelization techniques might work.

2 Related Work

Parallel database systems are in broad commercial use today and have been the subject of research for several years [6, 5, 15]. The way queries are executed in these systems still mainly depends on the underlying machine architecture. Three types of parallel architectures can be distinguished:

- Shared everything (shared memory)
- Shared disk (each processor has its own memory, but they have shared access to the data on disk)
- Shared nothing (SN, each processor has its own memory and disk. Networks of workstations belong to this class)

Whereas fine-grained, communication-intensive techniques (e.g. pipelining between nodes) are employed mainly in shared-memory systems [1, 9] this work

[1] OLAP is a business term and stands for Online Analytical Processing, in contrast to OLTP (Online Transaction Processing)[11].

suggests that it can be advantageous to use them as well on SN-architectures with high performance communication networks. Furthermore, on SN architectures the static allocation of the data on disk typically drives the execution of the query [13, 10]. We suggest that it may not be necessary to tie allocation of data and execution too tightly if the network is sufficiently fast.

The architectures described above have foremost been studied for traditional query processing which tends to be I/O-bound. More balance between processing and I/O needs is observed in data warehousing. Although those applications differ from data mining in that they are more static and rely heavily on preplanning, data mining could benefit from approaches to speed up data warehousing. Here much work is done in the fields of data and index structures [3] as well as in developing parallel system designs [11]. Indeed there has been first work towards extendig parallelism to data mining [7]. However, the approach to simply map the data analysis steps of a data mining algorithm to SQL-statements executed by a database system can lead to insufficient performance [16]. Hence, a closer coupling of parallel mining algorithms and parallel DBMS seems reasonable. In this environment we concentrate on the various granularities of parallelism.

3 Execution of Data Mining and OLAP-Queries

OLAP-queries are different from traditional OLTP transactions because they touch large parts of the data, are complex (i.e., they employ many expensive operators), and unpredictable (i.e. the part of the data, which will be used in the further process and this process are determined on short notice). Consequently, classical performance-enhancing techniques, e.g. indexes, play a minor role. The efficiency of query execution is the main factor. In the remainder of this section we describe the execution of OLAP queries both in a sequential and in a parallel environment and identify the degrees of freedom and the critical parameters.

3.1 Sequential Execution

The following type of SQL query frequently occurs in OLAP tasks:

SELECT $Att_1, \ldots, Att_k$ FROM $R_1, \ldots, R_l$
WHERE $Cond_1, \ldots, Cond_m$ GROUP BY $Att_1, \ldots, Att_n$

The query produces tuples with attributes $Att_1, \ldots Att_k$ (that can also be aggregates such as SUM(), COUNT(), etc.) from a given set of tables $R_1, \ldots, R_l$. The WHERE-clause is used to mask data items and to combine items of several tables. The GROUP BY-clause specifies attributes on which the items are grouped before being aggregated. Figure 1a shows a simple operator tree, that combines and processes data from two input tables. When the query is executed, the data flows through this tree. The data is accessed from two tables in the physical storage, then the selection operators (π) filter out the data specified by the WHERE-clause. The join operator ($\bowtie$) combines the resulting data from the two tables. Finally, an aggregation (agg) is performed on the joined data.

Here no use is made of the inherent parallelism (pipelines, independent branches), this parallelism is just used for sequential re-ordering of the execution.

3.2 Parallel Execution

In a parallel environment the inherent parallelism can be exploited. There even exists a new potential for parallelism due to the distribution of the data. However, parallel query processing is more complex. It is even more difficult to achieve performance that scales well with growing numbers of processors.

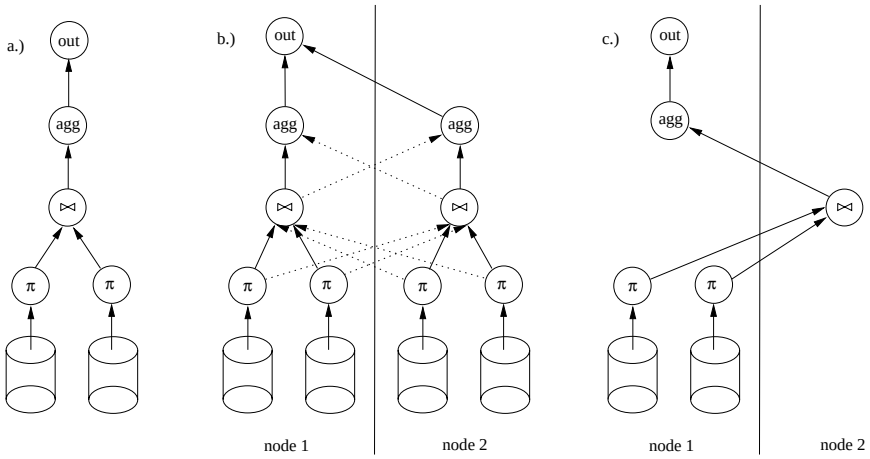

Fig. 1. Query trees for sequential and parallel execution

When operators are duplicated to multiple nodes, one has to consider global-scope operators. These operators have to correlate tuples they have seen earlier with those currently received and therefore need a complete history regarding the partition of the data they process. E.g. in case of a join, corresponding tuples from the joined relations have to "meet" within the same operator node in order to produce the correct result. In general, data is redistributed during query execution: Preceding operators have to send their tuples to the appropriate instance of the global-scope-operator according to the attribute values this operator uses to correlate the tuples. The resulting execution scheme (Figure 1b) is typical for parallel databases on SN-architectures and rests on the assumption that communication cost is the dominating factor: Each node keeps a part of the data on its disk, but possesses a full set of operators, and therefore communication between nodes (dotted lines) is employed only when necessary.

3.3 Alternative Parallelization Strategies

In clusters of workstations with high performance communication networks, the underlying assumptions no longer hold. Inter-node communication is much better (lower latency and higher bandwidth) so that one could expect that there are new degrees of freedom to be exploited for optimized parallelization strategies:

- The *degree of parallelism*: Under the traditional strategy the degree of parallelism is restricted by the growing cost of data redistribution and, hence, data communication. Moreover, it is heavily influenced by the static distribution of the raw data across the nodes. Once communication cost is low, both are no longer the limiting factors: The number of nodes should be determined by the needs of the algorithms, and initial distribution of the data may be arbitrary. Instead of replicating the operators to all nodes, the number of parallel operators performing an action can be chosen by the optimizer.
- The *parallelism paradigm* can be chosen dynamically, ranging from pure data parallelism to the extreme use of pipelining, where every operator is located on a different node and the data is piped through the nodes. Pipelining may work if less performance is lost by extensive communication than is gained by increased parallelism. An example for the moderate use of pipeline parallelism is shown in Figure 1c: The file-I/O, the selection, and the final aggregation are performed on one node, the expensive join is performed on the other node. Compared to the SN-approach, twice the amount of data is sent over the network.

4 Optimized JavaParty as Distributed Environment

While our long-term goal is to place OLAP and data mining activities within a continuum of function and data granularity, the present studies explore the suitability of networks of workstations and Java for parallel OLAP processing. The software platform we used is the JavaParty system for transparent parallel and distributed programming in Java. This section briefly introduces JavaParty and presents the central features that allow JavaParty to perform efficiently.

4.1 Standard JavaParty

JavaParty [12, 14] is a programming layer on top of Java that transparently adds remote objects purely by declaration, and avoids exposing the programmer to sockets, RMI, and message passing libraries. JavaParty code is preprocessed into Java code with RMI hooks, both are then compiled by regular Java and RMI compilers into platform independent and secure ByteCode.

JavaParty extends Java with a new class modifier `remote`. By this modifier, the programmer can distinguish between objects that are local and objects that may be instantiated on a remote node. Since Java's threads are implemented by means of objects as well, the programmer can create remote threads that run on remote processors. JavaParty implements Java's object semantics, i.e., the programmer has the impression of writing regular multi-threaded Java programs. The source code size does not change when moving from Java to JavaParty, but JavaParty programs are portable between single-processor workstations, shared memory parallel computers, and distributed memory platforms. Since the topology and the number of processor nodes of the underlying parallel computer is completely transparent to JavaParty programs, they automatically

adapt to changing configurations. The JavaParty preprocessor and the rest of the JavaParty environment are 100% pure Java and freely available [12].

4.2 Reducing the Cost of Remote Object Access in JavaParty

Being built on top of Java's Remote Method Invocation, JavaParty's performance depends on an efficient RMI. Unfortunately, in current Java implementations RMI is too slow for high performance computing since a remote method invocation between two workstations that are connected through 100 MBit-Ethernet takes about 4.6 milliseconds (one object with 32 `int` values as argument, JDK 1.1.7, two 500 MHz Digital Alphas).

In order to plan for improvements, we have analyzed where this time is spent: About one third of the time is spent in serialization, one third is spent within the current implementation of RMI, and one third is spent in the communication network. Hence, we have optimized all three parts: by an optimized serialization, an optimized RMI implementation, and by using the ParaStation communication network to couple the workstations. We now achieve a remote method invocation that takes about 350 microseconds (with the same object as argument).

Better Serialization. We have discussed the individual problems of Java's object-serialization and built an optimized version of it, see [8]. For several benchmarks our serialization outperforms the official serialization by a factor of up to 35.

Better RMI. We have re-designed and re-implemented RMI. Similar to the official RMI design, we have three layers (stub/skeleton, reference, and transport). In contrast to the official version however, our design features clear interfaces between the layers. This has two essential advantages. First, a performance advantage: The layers communicate by means of method invocation instead of passing around an object which describes a remote invocation. Hence, in our design a remote method invocation requires just two additional method invocations at the interfaces between the layers. No costly helper object creation is needed. The second main advantage is that alternative transports can be used. Whereas the official RMI is not designed to work with non-TCP/IP-networks, the clean interface between reference layer and transport layer allows for implementations that can work with high performance communication hardware.

For the ParaStation Network (see below) we have implemented a packet based transport that directly uses the hardware communication ports. We exploit the fact that packets are guaranteed to be delivered in order. Moreover, since either the whole cluster of workstations is working or not, there is no need to protect against network errors, e.g., temporary unavailability of some nodes, connection failure, etc. Therefore, the ParaStation transport is very slim.

ParaStation Network. ParaStation [17] is a communication technology for connecting off-the-shelf workstations into a supercomputer. The current ParaStation system is based on Myrinet [2], fits into a PCI slot, employs technology used in

massively parallel machines, and scales up to 4096 nodes. ParaStation's user-level message passing software preserves the low latency of the communication hardware by taking the operating system out of the communication path, while still providing full protection in a multiprogramming environment. On the Alpha platform, ParaStation achieves end-to-end (process-to-process) latencies as low as 15 μs and a sustained bandwidth of more than 50 Mbyte/s per channel.

5 OLAP Queries in JavaParty

5.1 System Design

Figure 2 shows the general design of our parallelization framework.

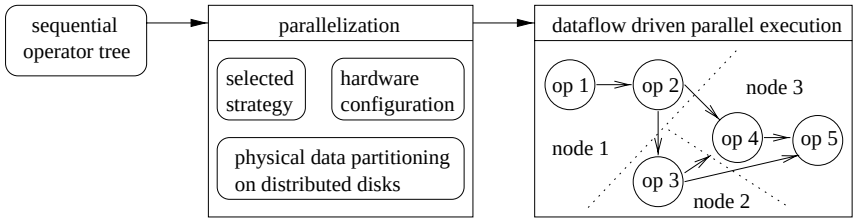

Fig. 2. OLAP queries in JavaParty

The query statement provided by the user is parsed and translated into an operator tree. This tree is then parallelized according to three factors: First, the static distribution of the data on the distributed disks must be taken into account. Second, the configuration of the network of workstations must be considered: How many nodes are available. Finally, the parallelization can work according to alternative parallelization strategies, as discussed in section 3.3.

From these input factors, a parallel execution scheme is generated that is driven by dataflow. The operators of the parallel execution tree are distributed to and instantiated on the available nodes. The operators are connected to implement the required dataflow. Figure 2 shows an example where five operators are distributed to three nodes.

5.2 System Implementation in JavaParty

For the implementation, we made use of JavaParty's remote objects. Every operator (select, join, etc.) is implemented as a remote object. These objects can be distributed and instantiated on the network of workstations at will. To instantiate a select operator on node `n`, the corresponding code looks like:

```
DistributedRuntime.SetTarget(n);
node = new jSelectNode();
```

The operator nodes contain an array of references to their successors in the dataflow graph. They are initialized after node creation. All the operators are subclasses of the general operator class `jNode` shown below. This class is an active class since it has a `run()` method that is executed by a single thread per object. The `run()` method executes an infinite loop that processes the data arriving in the input buffer `inbuff`.

```
remote class jNode implements Runnable{
    jNode[] Succs;      // Successor nodes
    byte[][] inbuff;    // receive buffers

    void run(){
        while(true)
            processTuple();
    }
    void processTuple(){...} // read, process and send to successor
    void sendTuple(data){    // called to send data
        Succs[k].put(data);
    }
    synchronized void put(data){...} // take data and buffer it
}
```

The `sendTuple()` routine is used to push results to the successor nodes. The index `k` is calculated according to the redistribution scheme (see section 3). To coordinate multiple senders, the `put`-method is declared `synchronized`.[2]

Note, that the data is transfered by means of a simple remote method invocation (`Succs[k].put()`). This approach has several benefits compared to the use of a dedicated socket for each connection: (1) It is much simpler, (2) flow control can be performed more easily, (3) it consumes fewer resources, since there is no need for a special thread object per incoming connection, and (4) optimizations of remote method invocation result in improved performance of OLAP queries.

6 Experimental Results

6.1 Benchmark Setup

The measurements were performed on a Cluster of 8 Alpha-workstations running Digital Unix with 500MHz clock rate connected with the ParaStation Network. We used the JDK 1.1.6 for our experiments. As our test dataset we used the 1-GB-TPC-D Dataset [4]. This dataset is part of a standard decision support benchmark and provides large sample datasets from a typical retail environment. The two tables used here hold data from 1.5 Mio. orders and 150000 customers and have the following structure (key attributes are emphasized):

```
table ORDER (Orderkey, Custkey, Orderstatus, Totalprice, Orderdate,
             Orderpriority, Clerk, Shippriority, Comment)
```

[2] We do not show the complete code to keep it simple. Special care must be taken to avoid buffer overflow etc.

```
table CUSTOMER (Custkey, Name, Address, Nationkey, Phone, Acctbal,
                Mktsegment, Comment)
```

Each query was repeated several times; we took the arithmetic mean of the execution times.

6.2 Enhanced Communication

In the first experiment we measured the effects of optimized JavaParty implementation. As a basis for this test we used Query 1, see Fig. 3.

This very resource-intensive query that is well-suited for SN-architectures lists all nationkeys, the number of orders placed by customers in that nation, the total volume and the average price per order for all orders placed before 1997.

Query 1

```
SELECT NATIONKEY,
COUNT(NATIONKEY),
SUM(TOTALPRICE),
AVG(TOTALPRICE)
FROM ORDER, CUSTOMER
WHERE ORDER.CUSTKEY =
CUSTOMER.CUSTKEY
AND ORDERDATE < 1997-01-01
GROUP BY NATIONKEY
```

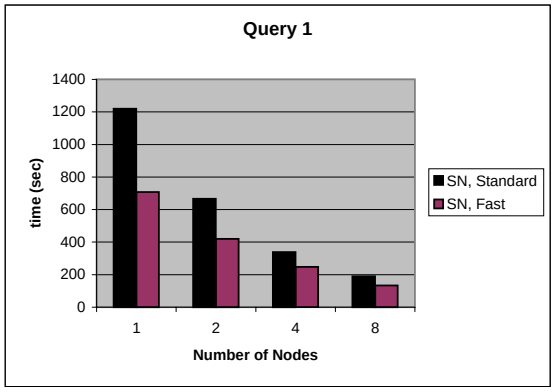

Fig. 3. The impact of fast Serialization and RMI

We ran an implementation of this query based on a SN-approach on 1, 2, 4, and 8 nodes and compared the execution times before/after the optimization of JavaParty. As depicted in Figure 3, we see a performance gain of up to 45%, on 8 nodes we have an improvement of 33%. For more nodes, the improvement is less prominent. The reason is that under the classic shared-nothing-strategy, the amount of local communication is reduced, when more nodes are added. (The amount of data that remains on the same node during redistribution is anti-proportional to the number of nodes to which the data is spread.) Since our optimized version of RMI takes special care of local communication, this optimization must have less effect for more nodes.

6.3 Alternative Parallelization Strategy

To compare a fine-grained pipeline-based execution strategy with a traditional SN-approach, we've studied the performance of Query 2, see Table 1.

The data is skewed, since it only has five different values for the priority. Due to that fact, only 5 nodes can be used for the aggregation, the others do only file-I/O and selection. That means that the scaling is pretty good until it comes to more than 5 nodes.[3] For two nodes, the pipeline-based implementation used an execution scheme similar to that shown in Figure 1c: The file-I/O is performed on node A, the selection is performed on node B, and the final aggregation is performed on node A again. So there is no local pipelining. The data is sent over the network to the selection. The result is sent back to the aggregation node. For 4 and 8 nodes we took proportionally more operator nodes of each type (node splitting), but the placement scheme was the same: half the nodes perform file-I/O and aggregation, the other half performs the selection. Again there is no local pipelining; all the data is communicated to and from the selection nodes.

Query 2

```
SELECT ORDERPRIORITY,
COUNT(ORDERPRIORITY)
FROM ORDER
WHERE ORDERDATE < 1997-01-01
GROUP BY ORDERPRIORITY
```

nodes	SN	Pipeline	**Improvement**
1	269 s	269 s	**0%**
2	181 s	172 s	**5%**
4	119 s	103 s	**14%**
8	95 s	71 s	**25%**

Table 1. Different execution strategies

Table 1 shows the benchmark results. With a growing number of nodes, the improvement that comes with the pipelining strategy increases. This has two reasons: First, the SN-approach suffers from growing redistribution costs with more nodes (as explained in the previous section). Second, the SN-approach is more affected by data skew. Hence, the pipeline strategy pays off even for just 2 nodes. Our pipeline-based strategy outperforms the traditional SN-approach by 25% on eight nodes, even though more than twice the amount of data is communicated.

7 Conclusion

We presented our Java-framework for the parallel execution of OLAP queries on networks of workstations. Our results show Java(Party) as a promising approach for parallel data mining on this platform. Moreover, the results demonstrate, that communication costs are no longer the only crucial factor in parallel query-optimization for this platform. Therefore, alternative parallelization strategies should be studied to achieve efficient execution of data mining tasks.

In the future, we will investigate fine-grained parallelism for object-relational query-execution and the coupling with parallel data mining algorithms.

[3] In this example, the problem could be solved by a two-phase `GROUPBY`, but in other redistribution cases, e.g. joins, that will not help.

Acknowledgments

We would like to thank the JavaParty team, especially Christian Nester, Daniel Lukic, and Joachim Blum for their work on problems caused by porting the JavaParty environment to ParaStation.

References

1. B. Bergsten, M. Couprie, and M. Lopez. DBS3: A parallel data base system for shared store (synopsis). In *Proc. Parallel and Distr. Inf. Sys.*, San Diego, CA, January 1993.
2. Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jarov N. Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.
3. Chee-Yong Chan and Yannis E Ioannidis. Bitmap index design and evaluation. In *Proceedings of the SIGMOD International Conference on Management of Data*, SIGMOD Record. ACM Press, 1998.
4. Transaction Processing Council. http://www.tpc.org/dspec.html.
5. David DeWitt and Jim Gray. Parallel database systems: The future of high-performance database systems. *Comm. of the ACM*, 35(6):85–98, June 1992.
6. D.J. DeWitt, R.H. Gerber, G.Graefe, M.L.Heytens, K.B.Kumar, and M. Muralikrishna. Gamma-a high performance dataflow database machine. In *12th Conference on Very Large Data Bases (VLDB)*, pages 228–237, Kyoto, Japan, August 1986.
7. A.A. Freitas and S.H. Lavington. *Mining very large Databases with parallel processing.* Kluwer Academic Publishers, 1998.
8. Bernhard Haumacher and Michael Philippsen. More efficient object serialization. In *International Workshop on Java for Parallel and Distributed Computing*, Puerto Rico, April 12–16 1999.
9. Wei Hong. Exploiting inter-operation parallelism in XPRS. In *Proceedings of the SIGMOD International Conference on Management of Data*, volume 21-2 of *SIGMOD Record*, pages 19–28, New York, NY, USA, June 1992. ACM Press.
10. Informix dynamic server v7.3. White paper, Informix Corp., 1998.
11. W.H. Inmon, Ken Rudin, C.K. Buss, and R. Sousa. *Data Warehouse Performance.* Wiley Computer Publishing, New York, USA, 1998.
12. JavaParty. http://wwwipd.ira.uka.de/JavaParty.
13. Oracle7 server. scalable parallel architecture for open data warehousing. White paper, Oracle Corp., 1995.
14. Michael Philippsen and Matthias Zenger. JavaParty: Transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, November 1997.
15. P.Valduriez. Parallel database systems: Open problems and new issues. *Distributed and parallel Databases*, 1(2):137–165, April 1993.
16. Sunita Sarawagi, Shiby Thomas, and Rakesh Agrawal. Integrating association rule mining with relational database systems: Alternatives and implications. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 27(2), 1998.
17. Thomas M. Warschko, Joachim M. Blum, and Walter F. Tichy. ParaStation: Efficient parallel computing by clustering workstations: Design and evaluation. *Journal of Systems Architecture*, 44:241–260, December 1997. Elsevier Science Inc., New York.