

Parallel Hash Join Algorithms for Dynamic Load Balancing in a Shared Disks Cluster

Aekyung Moon¹ and Haengrae Cho²

¹ Software Robot Research Team, ETRI,
Gajung-dong, Yusong-gu, Taejon 305-350, Republic of Korea
akmoon@etri.re.kr

² Department of Computer Engineering, Yeungnam University,
Gyungsan, Gyungbuk 712-749, Republic of Korea
hrcho@yu.ac.kr

Abstract. Most of previous parallel join algorithms assume a shared nothing (SN) cluster, where each database partition is owned by a single processing node. While SN cluster can interconnect a large number of nodes and support a geographically distributed environment, it may suffer from poor facility for load balancing and system availability compared to a shared disks sharing (SD) cluster. In this paper, we first propose a dynamic load balancing strategy by exploiting the characteristics of SD cluster. Then we parallelize conventional hash join algorithms using the dynamic load balancing strategy. We also explore the performance of parallel join algorithms using a simulation model of SD cluster. The experiment results show that the proposed parallel join algorithms can achieve higher potential for dynamic load balancing with the inherent flexibility of SD cluster.

Keywords: Cluster computing, shared disks, hash join, load balancing.

1 Introduction

There are two primary flavors of cluster architecture designs: *shared nothing* (SN) and *shared disks* (SD) [7, 8]. In SN cluster, each node has its own set of private disks and only the owner node can directly read and write its disks. On the other hand, the SD cluster allows each node to have direct access to all disks. The SD cluster offers a number of advantages compared to SN cluster, such as dynamic load balancing and seamless integration. Furthermore, the rapidly emerging technology of storage area networks (SAN) makes SD cluster the preferred choice for reasons of higher system availability and flexible data access. The recent parallel database systems using the SD cluster include IBM DB2 Parallel Edition [2] and Oracle Real Application Cluster [13].

In this paper, we propose parallel hash join algorithms in the SD cluster. The hash join is very efficient for large relations since it divides the joining relation into several buckets whose join key hash value is different so that the matching cost is substantially reduced [5, 12]. Moreover, the hash join provides natural opportunity for parallel processing because of generating disjoint buckets. With

regard to this viewpoint, there have been considerable research on the parallel hash join [1, 3, 4, 9, 10]. However, most of previous parallel hash join algorithms assume SN cluster, and the SD cluster has received very little attention with respect to parallel query processing. Fortunately, many of the techniques developed for parallel query processing in SN cluster can be utilized for SD cluster as well. However, the fact that each node can directly access all data gives SD cluster a higher flexibility, especially a higher potential for dynamic load balancing, for parallel query processing compared to SN cluster.

This paper is organized as follows: Sect. 2 presents the problems of previous algorithms and Sect. 3 proposes the parallel hash join algorithms in SD cluster. The simulation model is developed in Sect. 4, and Sect. 5 analyzes the performance of the proposed algorithms. Sect. 6 concludes this paper.

2 Problem Definition

The hash join algorithm comprises *partition phase* and *join phase* [12]. Suppose there are two joining relations R and S , where the join attributes are JA_R and JA_S , respectively. The partition phase is to partition R and S into n disjoint buckets, $R_1, R_2, \dots, R_n$ and $S_1, S_2, \dots, S_n$. The join phase is to build hash tables for buckets from an inner relation, R , and probe the hash table using tuples of the corresponding buckets from an outer relation, S . Note that for any two tuples, r and s , where $r \in R_i$ and $s \in S_j$, if $i \neq j$ then $r.JA_R \neq s.JA_S$. In this way, R_i needs to be joined with S_i only. Thus,

$$R \bowtie S = \bigcup_{i=1}^{i=n} R_i \bowtie S_i \quad (1)$$

The effectiveness of parallel hash join algorithms depends upon the ability to equally divide the load among nodes. A factor which can impair the ability to parallelize join queries successfully is *the amount of data skew* present in the joining relations [1, 3]. Specifically, if some buckets are much larger due to data skew, the speed-up from parallel hash join algorithms may be severely limited because some nodes may be overloaded while others are underutilized.

To alleviate the data skew problem in SN cluster, the approach of data re-allocation or replication have been proposed [3]. However, both approaches may suffer from significant coordination and synchronization overhead. The idea of load balancing in SD cluster is to reduce the size of task allocation unit *smaller* than the bucket size. For example, Lu and Tan [6] proposed a task stealing approach, where an idle node can steal part of a bucket from an overloaded node that has not completed the join operation of a (big) bucket. To estimate the amount of bucket to be stolen, each node is assumed to share a global memory. Whenever a page is processed, the node updates the global memory - the number of pages of each bucket left and the number of pages of the result relation generated thus far. Note that this process should result in heavy message traffic if the global memory is not physically attached to each node, which is true in most commercial SD clusters [13].

3 Parallel Hash Join Algorithms

In this section, we first propose a task generation strategy for efficient dynamic load balancing. Then we propose three parallel hash join algorithms.

3.1 Task Generation Strategy

We assume a *task manager* that manages information for load balancing. The task manager is located in a specific node. Other nodes execute join operations and communicate to the task manager via message passing. We refer them as *join nodes*. The task manager has a task memory that stores the bucket information. The bucket information consists of five tuples: [bucket identifier (B_I), number of pages of a bucket (B_S), number of pages of a bucket to be processed at join phase (B_U), list of page identifiers of a bucket (P_B)].

The task manager determines the number of buckets at the partition phase. We try to set the bucket size nearly equal to the memory size to maximize parallelism with little overhead. When bucket overflow occurs due to data skew or limited memory size, the bucket is partitioned into several fragments. A *fragment* is a part of bucket that can be allocated entirely into memory. For each fragment, the task manager registers the following fragment information to the task memory: [fragment identifier (F_I), number of pages of a fragment (F_S), fragment allocation bit (F_A), list of page identifiers of a fragment (P_F)].

After the partition phase, the task manager generates the fragment information and allocates each fragment to a join node. The details are as follows:

1. A join node sends $[B_I, B_S, B_U, P_B]$ message for every bucket to the task manager after it finishes its partition phase. The task manager stores the bucket information into the task memory.
2. Then the task manager partitions each bucket into fragments. The number of fragments (N_F) is set to $\lceil B_U / |M| \rceil$, where $|M|$ is the number of pages that a join node can store in its memory. We assume that $|M|$ is equal for every join node. The size of each fragment ($F_S[i]$) is determined as follows:

$$F_S[i] = \begin{cases} \lceil B_U / N_F \rceil & i < N_F \\ B_U - \sum_{k=1}^{N_F-1} F_S[k] & i = N_F \end{cases} \quad (2)$$

For each fragment, the task manager registers $[F_I, F_S, F_A, P_F]$ into the task memory. At this time, F_A is set to '0' since the fragment is not allocated to any join node yet.

3. At the join phase, each join node requests next fragment to be joined. The task manager selects a fragment where $F_A = 0$, and sends $[B_I, F_I, F_S, P_B, P_F]$ message to the join node. The bucket information of B_I and P_B corresponds to the outer relation (S) and the other fragment information corresponds to the inner relation (R). F_A of the fragment is set to '1'.
4. After receiving the message, the join node reads every page of P_F from disk and creates a hash table for the fragment of R . Then the join node reads each page of P_B of S and probes its hash table of R .

Note that a join node sends a request message to the task manager for each fragment. This is not true in [6], where the message passing is done for each page. Since a fragment includes large number of pages, the proposed strategy can reduce the message traffic significantly. The potential drawback of the proposed strategy is that it cannot provide “task stealing”. That means at the final stage of join phase some nodes would execute join operations for the last fragments while other nodes are idle. However, we believe that this kind of load imbalance is not critical since a fragment can be stored entirely into the memory of a join node. Join operations on the fragment should not take long time.

It is worthy to compare the proposed strategy with the task allocation strategy of Oracle Real Application Cluster (ORAC) [13]. ORAC divides an input relation into several partitions, each of which has the same number of pages. Specifically, the size of a partition is determined by dividing the number of nodes into the number of pages of input relation. Then ORAC assigns a partition to each node. As a result, every node can process the same number of pages. However, data skew could still happen and each node would not be under the same load. This is because the pages may have different numbers of tuples. Some of the pages could be completely empty [13]. The proposed strategy can alleviate this problem by allocating each fragment to the node with the on-demand approach. Dynamic load balancing can be achieved as a result.

3.2 Parallel Hash Join Algorithms

In this section, we propose three parallel hash join algorithms in SD cluster. The join phase of each algorithm is equal as described in Sect. 3.1. So we describe the partition phase of each algorithm in detail.

Parallel Grace Hash Join (PGHJ). A node partitions its part of the joining relation into buckets and writes them into disk. Then the join node sends $[B_I, B_S, B_U, P_B]$ message for every bucket to the task manager. B_U is equal to B_S since buckets are not joined at the partition phase.

The number of buckets ($|B|$) may have an impact on the performance of the PGHJ. If $|B|$ is too large, B_S may be much smaller than the memory size of a join node ($|M|$). This means that large number of small fragments are created at the join phase; hence, the join phase takes longer time. If $|B|$ is too small, B_S will increase. While the bucket of inner relation is partitioned into smaller fragments, the bucket of outer relation should be accessed entirely for each fragment. As a result, small $|B|$ may result in more disk I/Os. With regard to this viewpoint, we set $|B|$ as $\lceil |R|/|M| \rceil$, where $|R|$ is the size of joining relation.

Parallel Simple Hash Join (PSHJ). $|B|$ is set to the number of join nodes. As a result, there is an one-to-one relationship between a bucket and a join node. The join node executes join operations for the related bucket during the partition phase. Specifically, at the partition phase, each join node partitions the joining relation into buckets in parallel. When the size of a bucket exceeds a page, the bucket page is sent to its related join node. If the joining relation is

an inner relation, the receiving node inserts tuples of the page into a hash table. Otherwise, the node probes the hash table for each tuple of the page.

During the tuple insertions, a hash table overflow may occur. The overflow area is written to disk. After the partition phase, the join node sends $[B_I, B_S, B_U, P_B]$ message for its related bucket to the task manager. For inner relation, B_U is set to the number of overflow pages and P_B includes the identifiers of overflow pages. The bucket information of outer relation is equal to that of the PGHJ.

Parallel Hybrid Hash Join (PHHJ). The PHHJ combines the PGHJ and the PSHJ. $|B|$ is set to $\lceil |R|/|M| \rceil$, and some buckets are joined at the partition phase. Specifically, each join node creates $|B|$ buckets. Among them, $|N|$ buckets ($B_1, \dots, B_{|N|}$) are joined at the related join node, where $|N|$ is the number of join nodes. The remaining buckets ($B_{|N|+1}, \dots, B_{|B|}$) are written to disk. Overflow processing is still done as described in the PSHJ.

After the partition phase, each join node sends $[B_I, B_S, B_U, P_B]$ message for every bucket to the task manager. If a bucket belongs to the inner relation, B_U is set to either the number of overflow pages or B_S , depending on whether the bucket is joined at the partition phase ($B_1, \dots, B_{|N|}$) or not ($B_{|N|+1}, \dots, B_{|B|}$).

4 Experiment Model

We evaluate the performance of parallel hash join algorithms by simulation. Fig. 1 shows the experiment model of an SD cluster. The model was implemented using the CSIM discrete-event simulation package [11].

We model the SD cluster consisting of a single manager node plus a varying number of join nodes, all of which are connected via a local area network. Disks are shared by every node. The model of each join node consists of a *partition process*, which partitions the joining relation into a number of buckets, and a *join process*, which executes join operations on buckets or fragments. The details of both processes capture the semantics of given hash join algorithm. The *resource manager* models CPU activity and provides access to the disks and the network. The manager node consists of *task manager*, which manages bucket and fragment information, and *query manager*, which allocates part of joining relation to each join node for parallel partitioning.

Table 1 shows the simulation parameters. Many of the parameter values are adopted from [6, 7]. We perform the experiments by varying the number of join nodes and the memory size of each join node. The CPU speed of each join node is assumed to be equal. Disk access time is drawn from a uniform distribution between 10 milliseconds to 30 milliseconds. To increase I/O performance at the partition phase, tuples of each relation are distributed among the disks in a round-robin fashion. The *network manager* is implemented as a FIFO server with 100 Mbps bandwidth. The CPU cost to send or to receive a message via the network is modeled as a fixed per-message instruction count plus an additional per-page instruction increment. Note that the protocol processing (i.e., CPU overhead) dominates the on-the-wire time for message passing.

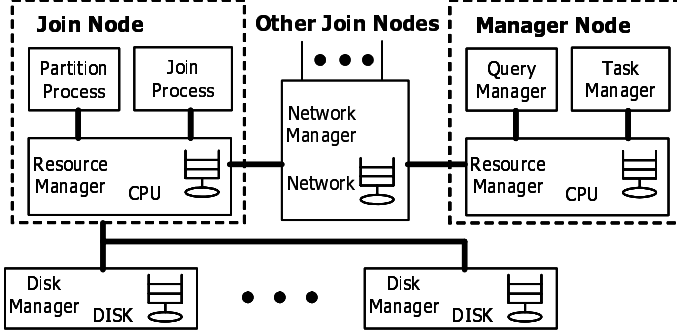


Fig. 1. Experiment model of an SD cluster

Table 1. Simulation parameters

System Configuration Parameters		
<i>CPU Speed</i>	Instruction rate of node CPU	1000 MIPS
<i>MemSize</i>	Per-node memory buffer size	40 Mbytes
<i>NetBandwidth</i>	Network bandwidth	100 Mbps
<i>NumNode</i>	Number of join nodes	4 ~ 40
<i>NumDisk</i>	Number of shared disks	16
<i>MinDiskTime</i>	Minimum disk access time	0.01 sec
<i>MaxDiskTime</i>	Maximum disk access time	0.03 sec
<i>RelSize</i>	Number of tuples per relation	10000 K
<i>DistinctTuples</i>	Number of distinct tuples	100 K
<i>PageSize</i>	Size of a page	8 Kbytes
<i>TuplesPerPage</i>	Number of tuples per page	50
System Overhead Parameters		
<i>FixedMsgInst</i>	Fixed number of instructions per message	20,000
<i>PerByteMsgInst</i>	Additional instructions per message byte	10,000 per page
<i>PerIOInst</i>	Number of instr. per a disk I/O	5000
<i>ReadTuple</i>	Number of instr. to read a tuple	500
<i>WriteTuple</i>	Number of instr. to write a tuple	500
<i>ProbeHashTbl</i>	Number of instr. to probe the hash table	200
<i>InsertHashTbl</i>	Number of instr. to insert a tuple to the hash table	100

For a relation R with a domain D distinct values (*DistinctTuples*), the i^{th} distinct join attribute value, for $1 \leq i \leq D$, has such number of tuples as given by the following Zipf-like distribution expression [6]:

$$\|D_i\| = \frac{\|R\|}{i^\theta \sum_{j=1}^D \frac{1}{j^\theta}} \quad (3)$$

where $\|R\|$ is the number of tuples of R and θ is the skew factor. When $\theta = 0$, the distribution becomes uniform. With $\theta = 1$, it corresponds to the highly skewed pure Zipf distribution.

The performance metric is *average response time* of join queries. The response time in second is measured as the difference when a join query is submitted and when every join result is returned. The time includes any time spent in the queue, disk I/O, and communication. A form of the batch mean method with 20 batches was used for the statistical analysis of simulation results.

5 Experiments and Results

This section explores the performance of parallel hash join algorithms using our simulation model of SD cluster. To evaluate the effect of task allocation unit, we implement two versions for each hash join algorithm: fragment allocation (PGHJ-F, PSHJ-F, PHHJ-F), and bucket allocation (PGHJ-B, PSHJ-B, PHHJ-B).

5.1 Experiment 1 – High Skew Workload

This workload models where the degree of data skew is high ($\theta = 1$ in Equation (3)). The number of join nodes is varied from 4 to 40. Fig. 2(a) shows the performance results of this workload.

For every algorithm, the fragment allocation strategy outperforms the bucket allocation strategy. This is because the bucket allocation strategy suffers from some big buckets when the degree of data skew is high. On the other hand, at fragment allocation strategy, a big bucket is partitioned into smaller fragments and each fragment can be joined at a number of join nodes in parallel. Due to its high potential of parallelism, the fragment allocation strategy performs better as the number of nodes increases. This property is well matched with the trend of ever-increasing scalability of parallel systems.

The PSHJ exhibits substantial performance improvements as the number of nodes increases. Note that at PSHJ the number of buckets is equal to the number of nodes. As the number of nodes increases, more buckets are created and thus the size of each bucket is reduced; hence, the amount of hash table overflow decreases. When there are small number of nodes, the PGHJ outperforms the PSHJ due to large bucket size of PSHJ. However, the PGHJ is outperformed by the PSHJ as the number of nodes increases, because at PSHJ a decreasing fraction of joining relation is written back to disk. Note that the amount of disk write at PGHJ is constant independent of the number of nodes. The PHHJ performs similar to the PGHJ at small number of nodes, while performs similar to the PSHJ at large number of nodes. Since the bucket size of PHHJ is determined according to the memory size like the PGHJ, the join execution time per bucket does not increase significantly at small number of nodes. Furthermore, at large number of nodes, the PHHJ can join most buckets without writing back to the disk like the PSHJ.

It is important to point out that the trends observed in Fig. 2(a) and its general shape are almost identical to the performance results in [12] for single node versions of the same algorithms and in [10] for SN clusters. It demonstrates that each of the algorithms parallelizes well in SD cluster. Furthermore, it serves

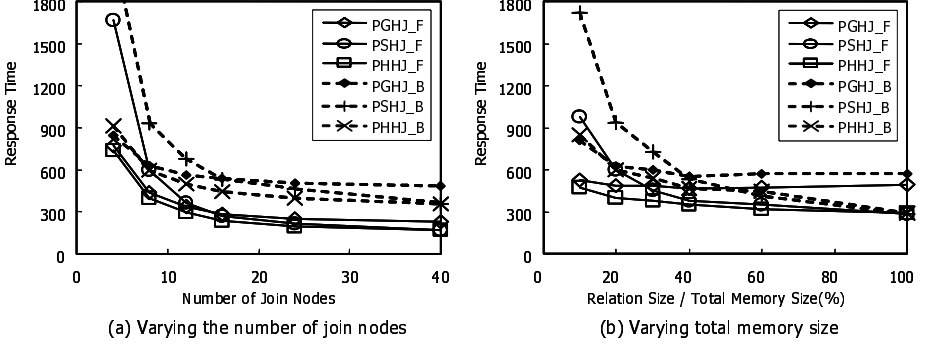


Fig. 2. Experiment results at high skew workload

to verify that our parallel implementation of each algorithm in SD cluster was done in a fair and consistent fashion.

We also explore the effect of memory size of each join node where the degree of data skew is high. The number of join nodes is set to 8, while the total memory size of all join nodes is varied from 10% of joining relation to 100%. Fig. 2(b) shows the experiment results. The performance of the bucket allocation strategy downgrades rapidly as the size of memory decreases, especially when the percentage of total memory relative to the size of joining relations is small. This is because the bucket allocation strategy suffers from the frequent bucket overflows at the small memory size. On the other hand, in the fragment allocation strategy, the bucket overflow area is partitioned into smaller fragments and the join nodes can process the fragments in parallel.

When the joining relation fits entirely in memory (at 100%), both PHHJ and PSHJ perform in the same manner as expected. As the size of memory decreases, the PSHJ degrades rapidly because it repeatedly reads and writes the same data. The PGHJ is relatively insensitive to the memory size. This occurs because the PGHJ does not use the extra memory for join queries, and thus increasing memory simply reduces the number of buckets. However, at PHHJ, increasing memory from 50% to 100% allows the algorithm to stage every joining bucket into memory.

5.2 Experiment 2 – Uniform Workload

This workload models where values of join attributes are uniformly distributed, and thus data skew is not present ($\theta = 0$ in Equation (3)). Fig. 3 shows the performance results of this workload by varying the number of join nodes and by varying the total memory size.

Unlike the high skew workload, the task allocation strategy does not result in significant performance difference between join algorithms. This is because the size of buckets are almost equal to each other in the uniform workload. Hence, in the bucket allocation strategy, the load of each join node can be balanced similar to the fragment allocation strategy.

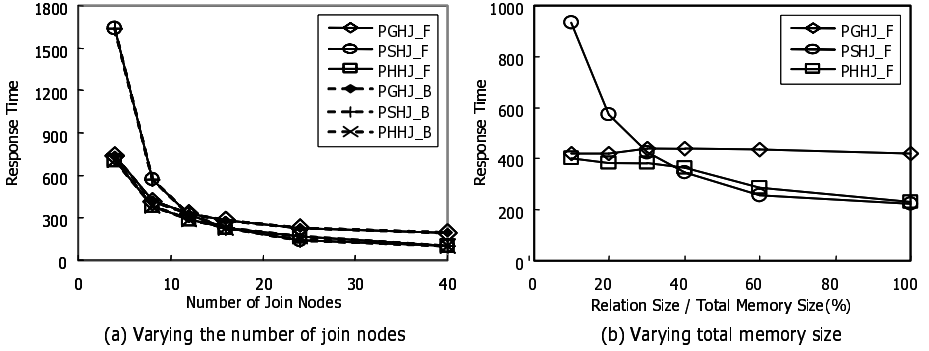


Fig. 3. Experiment results at uniform workload

Every hash join algorithm performs better compared to the high skew workload in Fig. 2. The reason also comes from the equal bucket sizes in the uniform workload. Note that there are some big buckets in the high skew workload. As we described in Sect. 5.1, join nodes that process the big buckets are overloaded and thus they delay the total response time. While the fragment allocation strategy may reduce the response time by partitioning the big buckets of inner relation into fragments, each fragment should be joined against the big bucket of outer relation. This means that the big buckets of outer relation may be accessed repeatedly for each fragment by the corresponding join node. As a result, the degree of performance improvement is limited somewhat.

The performance graphs exhibit similar pattern to that in the high skew workload. An interesting observation is that the PSHJ outperforms the PHHJ when the number of nodes is 16 and 24 in Fig. 3(a). This is due to the number of fragments. At PSHJ, fragments are created when a bucket overflow occurs and the size of buckets are nearly equal to each other in the uniform workload. This means that the number of fragments at PSHJ is always a multiple of the number of nodes in the uniform workload; hence, loads of join operations can be completely balanced. This feature may not hold on PHHJ since the number of buckets is determined with the memory size of join node.

6 Concluding Remarks

Most of previous parallel join algorithms assume SN cluster, and the SD cluster has received very little attention with respect to parallel query processing. However, the fact that each node can directly access all data gives SD cluster a higher flexibility, especially a higher potential for dynamic load balancing, for parallel query processing compared to SN cluster. With regard to this viewpoint, we first propose a fragment based task allocation strategy that can distribute load of join operations evenly into the participating join nodes. Then we parallelize the conventional hash join algorithms for SD cluster using the fragment based task allocation strategy.

We have explored the performance of parallel hash join algorithms under a wide variety of database workloads and system configurations using a simulation model of SD cluster. The considerable results obtained from the experiments can be summarized as follows. First, when the join attribute is skewed, the fragment based task allocation strategy exhibits substantial performance improvements, compared to the bucket based task allocation strategy. Next, as the number of nodes increases, the fragment based task allocation strategy performs better due to its higher potential of parallelism. This property is well matched with the trend of ever-increasing scalability of cluster systems. Finally, the PHHJ outperforms other hash join algorithms in most experiments.

Acknowledgements

The work of Prof. Cho was supported by the University ITRC Project.

References

1. Bamha, M., Exbrayat, M.: Pipelining a Skew-Insensitive Parallel Join Algorithm. *Parallel Processing Letters* 13(3) (2003) 317-328
2. DB2 Universal Database for z/OS Version 8 - Data Sharing: Planning and Administration. IBM SC18-7417-01 (2004)
3. Heal, A., Yuan, A., El-Rewni, H.: Dynamic Data Reallocation for Skew Management in Shared Nothing Parallel Databases. *Distributed and Parallel Databases* 5(3) (1997) 271-288
4. Imasaki, K., Nguyen, H., Dandamudi, S.: Performance Comparison of Pipelined Hash Joins on Workstation Clusters. *LNCS* 2552 (2002) 264-278
5. Lu, H., Ooi, B., Tan, K.: Query Processing in Parallel Relational Database Systems. IEEE Computer Society Press (1995)
6. Lu, H., Tan, K.: Dynamic and Load-Balanced Task Oriented Database Query Processing in Parallel Systems. *LNCS* 580 (1992) 357-372
7. Ohn, K., Cho, H.: Cache Conscious Dynamic Transaction Routing in a Shared Disks Cluster. *LNCS* 3045 (2004) 548-557
8. Ranade, D.: Shared Data Clusters. John Wiley, Inc. (2002)
9. Schikuta, E., Kirkovits, P.: Cluster Based Hybrid Hash Join: Analysis and Evaluation. In: *Proc. IEEE Conf. Cluster Computing* (2002) 461-466
10. Schneider, D., DeWitt, D.: A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment. In: *Proc. ACM SIGMOD Conf.* (1989) 110-121
11. Schwetman, H.: CSIM18 Simulation Engine. Mesquite Software, Inc. (1996)
12. Silberschatz, A., Korth, H.F., Sudarshan, S.: Database System Concepts (4th ed.). McGraw Hill (2002)
13. Vallath, M.: Oracle Real Application Clusters. Elsevier Digital Press (2004)