in: Proc. 5th Int. PARLE Conf. (ParallelArchitectures and Languages Europe), Munich
Springer-Verlag, Lecture Notes inComputer Science 694, June 1993, pp. 622-633

On the Performance of Parallel Join Processing in Shared Nothing Database Systems

Robert Marek
Erhard Rahm
University of Kaiserslautern, GERMANY

Abstract: Parallel database systems aim at providing high throughput for OLTP transactions as well as short response times for complex and data-intensive queries. Shared nothing systems represent the major architecture for parallel database processing. While the performance of such systems has been extensively analyzed in the past, the corresponding studies have made a number of best-case assumptions. In particular, almost all performance studies on parallel query processing assumed single-user mode, i.e., that the entire system is exclusively reserved for processing a single query. We study the performance of parallel join processing under more realistic conditions, in particular for multi-user mode. Experiments conducted with a detailed simulation model of shared nothing systems demonstrate the need for dynamic load balancing strategies for efficient join processing in multi-user mode. We focus on two major issues: (a) determining the number of processors to be allocated for the execution of join queries, and (b) determining which processors are to be chosen for join processing. For these scheduling decisions, we consider the current resource utilization as well as the size of intermediate results. Even simple dynamic scheduling strategies are shown to outperform static schemes by a large margin.

1 Introduction

Parallel database systems are the key to high performance transaction and database processing [DG92]. These systems utilize the capacity of multiple locally distributed processing nodes interconnected by a high-speed network. Typically, fast and inexpensive microprocessors are used as processors to achieve high cost-effectiveness compared to mainframe-based configurations. Parallel database systems aim at providing both high throughput for on-line transaction processing (OLTP) as well as short response times for complex ad-hoc queries. Efficient query processing increasingly gains importance due to the wide-spread use of powerful query languages and user tools. Next-generation database applications for engineering, VLSI design or multi-media support will lead to substantially increased query complexity [SSU91]. Since these complex queries typically access large amounts of data or/and perform extensive computations, in general the response time goal can only be achieved by employing parallel query processing strategies [Pi90]. Furthermore, performance should scale with the number of nodes: adding processing nodes ideally improves throughput for OLTP or response times for complex queries linearly.

Most research and development efforts on parallel database systems have concentrated on so-called *shared nothing* architectures [St86, DG92]. Shared nothing systems consist of multiple functionally homogenous processing nodes or processing elements (PE). Each PE comprises one or more CPUs and a local main memory, and runs local copies of application and system software like operating system and database management system (DBMS). Cooperation between PE takes place by means of message passing over a high-speed network. The characteristic feature of shared nothing systems is that the database is partitioned and distributed among all nodes so that every PE "owns" one partition. If a transaction (query) needs access to data owned by another node, a sub-transaction is started at the respective owner PE to access the remote data. In this case, a distributed two-phase commit protocol is also to be executed to guarantee the all-or-nothing property of the transaction [MLO86, ÖV91]. Existing shared nothing systems supporting parallel transaction processing include the products Tandem NonStop SQL [Ta89, EGKS90] and Teradata's DBC/1012 [Ne86] as well as several prototypes including Bubba [Bo90], Gamma [De90], EDS [WT91] and PRISMA/DB [WFA92]. With the exception of Tandem, these systems represent database machines (back-end systems) dedicated to database processing. The database operations or DML (Data Manipulation Language) statements submitted to the back-end system may originate directly from the end-user (ad-hoc queries) or from application programs running on workstations or mainframe hosts.

With respect to parallel transaction processing, we distinguish between inter- and intra-transaction parallelism. *Inter-transaction parallelism* refers to the concurrent execution of multiple independent transactions on the same database. This kind of parallelism is already supported in centralized DBMS (multi-user mode), e.g., in order to overlap I/O delays to achieve acceptable

system throughput. To improve response time, intra-transaction parallelism is needed either in the form of inter-DML or intra-DML parallelism. *Inter-DML parallelism* refers to the concurrent execution of different DML statements (queries) of the same transaction. However, the degree of parallelism obtainable by inter-DML parallelism is limited by the number of database operations per transaction as well as by precedence constraints between these operations. Furthermore, the application programmer would have to specify inter-DML parallelism by means of adequate language features.

As a result, current parallel database systems support intra-transaction parallelism only in the form of *intra-DML parallelism*[1]. Relational database systems with their descriptive and set-oriented query languages (e.g. SQL) have made possible this kind of parallelism [DG92]. Intra-DML parallelism is implemented by the DBMS query optimizer, completely transparent for the database user and application programmer. For each query, the optimizer determines an (parallel) execution plan specifying the basic operators (e.g. scan, selection, join, etc.) to process the operation. The optimizer may support two types of intra-DML parallelism: inter- and intra-operator parallelism. *Inter-operator parallelism* refers to the concurrent execution of different operators in an execution plan, while *intra-operator parallelism* aims at parallelizing a single operator. In both cases, parallel processing is largely influenced by the database allocation. In particular, the database should be allocated such that operators or sub-operators on disjoint database portions can be processed in parallel on different PE. Typically, this is achieved by horizontally partitioning relations among several PE.

Despite the fact that several parallel database systems have been benchmarked and numerous performance studies on parallel query processing have been conducted (see section 2), we feel there is a strong need for further performance evaluations. This is because previous benchmarks and performance studies mostly assumed a number of best-case conditions that have an overriding effect on performance. One of the most questionable assumptions is the sole consideration of single-user experiments in most studies, frequently without even making this assumption explicit. Our research focus is to study the performance of parallel database systems under more realistic conditions and to identify shortcomings of current query processing approaches. The next step then is to develop better query processing strategies that work well under ideal and realistic conditions.

For this purpose, we have developed a comprehensive simulation system of a generic shared-nothing database system. In a previous paper, we have already presented performance results using this simulation model for the debit-credit workload as well as for real-life workloads represented by database traces [MR92]. For the present paper, we have extended our simulation model to study the performance of complex database queries, in particular join queries. In relational database systems, joins occur frequently and are the most expensive operations to execute, especially on large relations. We investigate join performance in single- as well as in multi-user mode. Our multi-user experiments clearly demonstrate the need for dynamic query processing and scheduling algorithms that take the current system state into account. Important scheduling decisions that should dynamically be drawn include determination of how many and which processors should be used for join processing. These decisions should be based on the size of intermediate results and current processor utilization. Our experiments show that even simple dynamic strategies outperform static schemes by a large margin.

The next section provides a survey of related performance studies. Sections 3 and 4 briefly describe our simulation system and the workload parameters, respectively. In section 5 we present and analyze simulation results for different configurations and scheduling strategies. Finally, we summarize the major findings of this investigation.

2 Related Work

Most benchmarks and performance studies of parallel database systems either concentrated on throughput for OLTP workloads or response time experiments for complex queries. For simple OLTP workloads such as debit-credit, it was shown that transaction rates can be linearly improved with the number of nodes [Ta88, Bo90, MR92]. The use of intra-transaction parallelism

1. In the case of ad-hoc queries there is only a single DML statement per transaction. Hence, intra-transaction parallelism is equivalent to intra-DML (intra-query) parallelism.

was found to be similarly effective with respect to decreasing the response time of complex queries, both in benchmarks [EGKS90, De90, WFA92] as well as in many analytical and simulation studies. However, the majority of the studies on intra-transaction parallelism is based on best-case assumptions like single-user mode, uniform data distribution, uniform load balancing, etc. Recently, researchers have begun to relax some of the uniformity assumptions by considering the effects of different forms of "data skew" [WDJ91, DG92]. However, these studies still assume single-user mode. This also holds for performance studies of different parallel join strategies (e.g. [SD89, Pa90]) and of schemes for processing N-way joins by means of inter-operator parallelism [SD90, MS91, CYW92].

While the single-user studies provided many significant insights, it is imperative to evaluate the effectiveness of intra-transaction parallelism in multi-user mode, i.e., in combination with inter-transaction parallelism. Assuming that a large system with hundreds of processors is exclusively reserved for processing a single query is clearly unrealistic since it would result in very poor cost-effectiveness. Furthermore, single-user operation would prevent meeting the throughput requirements for OLTP transactions. One problem with supporting multi-user mode ist that the current system load may significantly vary during query execution thus making dynamic scheduling strategies necessary. [GW89] already demonstrated that considerable performance gains can be realized by choosing dynamically among multiple query plans - depending on both system load and the size of intermediate results. However, they restricted their considerations to two alternative query plans (either B-tree scan and index nested loops join or file scans and hash join) and did not consider parallelization issues.

3 Simulation Model

Our simulation system models the hardware and transaction processing logic of a generic shared nothing DBMS architecture. The system has been implemented using the discrete event simulation language DeNet [Li89]. Our system consists of three main components: *workload generation*, *workload allocation* and *processing subsystem* (Figure 1). The workload generation component models user terminals and generates work requests (transactions, queries). The workload allocation component assigns these requests to the PE of the processing subsystem where the actual transaction processing takes place. In this section, we summarize the implementation of these components.

3.1 Workload Generation and Allocation

Database Model

Our database model supports four object granularities: database, partitions, pages and objects (tuples). The database is modeled as a set of partitions that may be used to represent a relation, a fragment of a relations or an index structure. A partition consists of a number of database pages which in turn consist of a specific number of objects. The number of objects per page is determined by a blocking factor which can be specified on a per-partition basis. Differentiating between objects and pages is important in order to study the effect of clustering which aims at reducing the number of page accesses (disk I/Os) by storing related objects into the same page. Furthermore, concurrency control may now be performed on the page or object level. Each relation can have associated clustered or unclustered B^*-tree indices.

We employ a horizontal data distribution of partitions (relations and indices) at the object level controlled by a relative distribution table. This table defines for every partition P_j and processing element PE_i which portion of P_j is allocated to PE_i.This approach models range partitioning and supports full declustering as well as partial declustering.

Workload Generation

Our simulation system supports heterogeneous workloads consisting of several transaction (query) types. In this paper we restrict ourselves to queries (transactions with a single DML statement), in particular join queries. Query types may differ in the structure of their operator trees, referenced relations, selection predicates etc. The simulation system is an open queuing model and allows definition of an individual arrival rate for each query type.

The join queries studied in this paper use three basic operators: *scan*, *sort* and *join*. These operators can be composed to query trees representing the execution plan for a query. The *scan* of a relation A using a predicate P produces a relational data output stream. The scan reads each tuple t of R and applies the predicate P to it. If P(t) is true, then the tuple is added to the output stream.

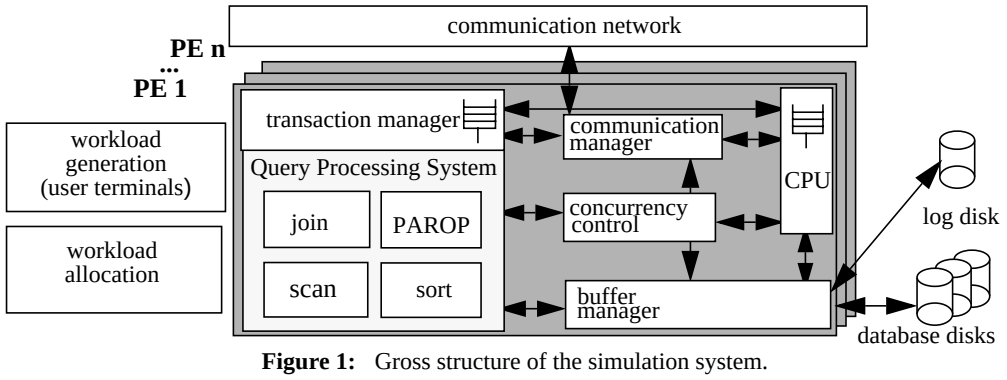


Figure 1: Gross structure of the simulation system.

We support relation scans as well as index (B^*-tree) scans. The *sort* operator reorders its input tuples based on an attribute sort criteria. The *join* operator composes two relations, A and B, on some join attribute to produce a third relation. For each tuple t_a in A, the join finds all tuples t_b in B whose join attribute values are equal to that of t_a[2]. For each matching pair of tuples, the join operator inserts a tuple built by concatenating the pair into the output stream.

For our study we have implemented a representative parallel join strategy based on hash partitioning. It applies a hash function on the join attribute to partition both input relations (scan output relations) to a specific number of join processors (dynamic data redistribution). This hash partitioning guarantees that tuples with the same join attribute value are assigned to the same join processor. This approach has the advantage that it offers a high potential for dynamic load balancing since the number and selection of join processors constitute dynamically adjustable parameters. For local join processing we have implemented a sort-merge algorithm. At each join processor the input relations are first sorted on the join attribute. The sorted relations are then scanned and matching tuples are added to the output stream. The complete join result is obtained by merging the results of the distributed local joins.

In the query graphs of our model, parallelism is expressed by means of a so-called *parallelization* meta-operator (PAROP). This operator does not perform any semantic data transformations. Instead it implements inter- as well as intra-operator parallelism and encapsulates all parallelism issues[3]. In particular, the PAROP operator comprises two basic parallelization functions: a *merge* function which combines several parallel data streams into a single sequential stream, and a *split* function which is used to partition or replicate the stream of tuples produced by a relational operator.

We employ the PAROP operator to parallelize scan and join operators. For this purpose, PAROP operators are inserted into the query trees. With respect to intra-operator parallelism several strategies can be chosen to allocate parallel suboperations to processors. For scan operators, the processor allocation is always based on a relation's data allocation. For join operators, on the other hand, we support several static and dynamic allocation alternatives, e.g. random allocation or based on the PE's CPU utilization. More details will be given in section 5 together with the simulation results.

The example in Figure 2 illustrates the use of the PAROP operator. The query specifies that a join is to be performed between relations A and B and the result is to be printed. Relation A is partitioned into three fragments A_0, A_1, A_2 (residing on disjoint PE) and relation B into two fragments B_0, B_1. The two lower PAROP operators specify that the scan operations are parallelized according to this fragmentation. Furthermore, they indicate that the output streams of the local scans are to be split onto two join processors (according to some split function). Before the local joins are processed, the input streams have to be merged. The final PAROP operator specifies that the local join results are sent to and merged at an output (print) node.

2. We only consider equi-joins in this paper.
3. A similar operator-based parallelization model has been implemented in the Volcano prototype [G90].

Workload Allocation:
Two forms of workload allocation have to be distinguished. First, each incoming transaction (query) is assigned to one PE (acting as the coordinator for the transaction) according to a placement strategy. Our simulation system supports different placement strategies, in particular a random allocation or the use of a routing table[4]. The second form of workload allocation deals with the assignment of suboperations to processors during query processing. As mentioned above, this is performed according to the chosen strategy for parallel query processing.

3.2 Workload Processing

The processing component models the execution of a workload on a shared nothing system with an arbitrary number of PE connected by a communication network. Each PE has access to private database and log files allocated on external storage devices (disks). Internally, each PE is represented by a transaction manager, a query processing system, a buffer manager, a concurrency control component, a communication manager and a CPU server (Figure 1).

The transaction manager controls the (distributed) execution of transactions. The maximal number of concurrent transactions (inter-query parallelism) per PE is controlled by a multiprogramming level. Newly arriving transactions must wait in an input queue until they can be served when this maximal degree of inter-transaction parallelism is already reached. The query processing system models basic relational operators (sort, scan and join) as well as the PAROP meta-operator (see above).

Execution of a transaction starts with the BOT processing (begin of transaction) entailing the transaction initialization overhead. The actual query processing is performed according to the relational query tree. Basically, the relational operators process local input streams (relation fragments, intermediate results) and produce output streams. The PAROP operators indicate when parallel sub-transactions have to be started and perform merge and split functions on their input data streams. An EOT step (end of transaction) triggers two-phase commit processing involving all PE that have participated during execution of the respective transaction. We support the optimization proposed in [MLO86] where read-only sub-transactions only participate in the first commit phase.

CPU requests are served by a single CPU per PE. The average number of instructions per request can be defined separately for every request type. To accurately model the cost of query processing, CPU service is requested for all major steps, in particular for query initialization (BOT), for object accesses in main memory (e.g. to compare attribute values, to sort temporary relations or to merge multiple input streams), I/O overhead, communication overhead, and commit processing.

For concurrency control, we employ distributed strict two-phase locking (long read and write locks). The local concurrency control manager in each PE controls all locks on the local partition. Locks may be requested either at the page or object level. A central deadlock detection scheme is used to detect global deadlocks and initiate transaction aborts to break cycles.

Database partitions can be kept memory-resident (to simulate main memory databases) or they can be allocated to a number of disks. Disks and disk controllers have explicitly been modelled as servers to capture I/O bottlenecks. Disks are accessed by the buffer manager component of the

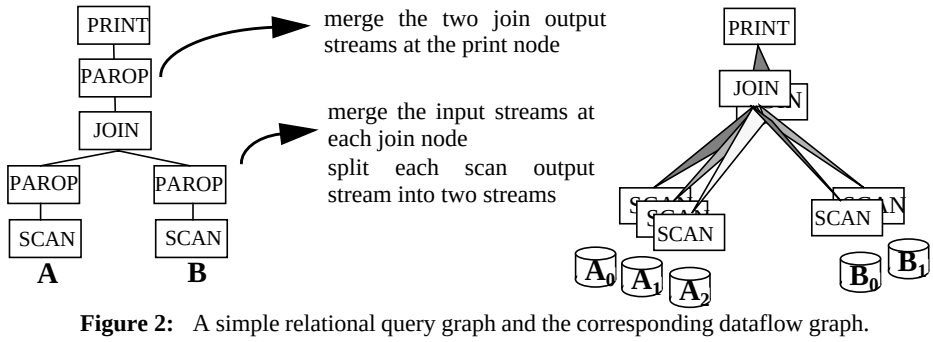


Figure 2: A simple relational query graph and the corresponding dataflow graph.

4. The routing table specifies for every transaction type T_j and processing element PE_i which percentage of transactions of type T_j will be assigned to PE_i.

associated PE. The database buffer in main memory is managed according to a global LRU (Least Recently Used) replacement strategy.

The communication network provides transmission of message packets of fixed size. Messages exceeding the packet size (e.g. large sets of result tuples) are disassembled into the required number of packets.

4 Workload Profile and Simulation Parameter Settings

Our performance experiments are based on the query profile and database schema of the Wisconsin Benchmark [Gr91]. This benchmark has extensively been used for evaluating the performance of parallel database systems [EGKS90, De90, WFA92]. Although the Wisconsin Benchmark constitutes a single-user benchmark, we use it also for multi-user experiments.

Table 1 shows the major database, query and configuration parameters with their settings. Most parameters are self-explanatory, some will be discussed when presenting the simulation results. The join queries used in our experiments correspond to the Wisconsin *joinABprime* query [Gr91], but we support selections on both input relations. Each query performs two scans (selections) on the input relations A and B and joins the corresponding results. The *A* relation contains 1 million tuples, the *B* relation 100.000 tuples. The selections on *A* and *B* reduce the size of the input relations according to the selection predicate's selectivity (percentage of input tuples matching the predicate). Both selections employ indices (B^*-trees), clustered on the join attribute. The join result has the same size as the scan output on B. Scan selectivity is varied between 0.1% and 10%, thus yielding join result sizes between 100 and 10.000 tuples.

The number of processing nodes is varied between 10 and 80, the number of join processors between 1 and 80 depending on the experiment. Both database relations are partitioned into an identical number of fragments and allocated on disjoint PE. Two declustering strategies are studied in the experiments with each relation allocated to either half of the PE or to a third of the PE.

The parameters for the I/O (disk) subsystem are chosen so that no bottlenecks occurred (sufficiently high number of disks and controllers). The duration of an I/O operation is composed of the controller service time, disk access time and transmission time. The parameter settings for the communication network have been chosen according to the EDS prototype [WT91].

Configuration	settings	Database/Queries	settings
number of PE (#PE)	10,20,30,60,80	**relation A:**	(200MB)
CPU speed per PE	20 MIPS	#tuples	1.000.000
		tuple size	200 bytes
avg. no. of instructions:		blocking factor	40
BOT	25000	index type	(clustered) B^*-tree
EOT	25000	storage allocation	disk
I/O	3000	allocation to PE	1..#PE/2 (1..#PE/3)
send message (8 KB)	5000	**relation B:**	(20MB)
receive message (8 KB)	10000	#tuples	100.000
scan object reference	1000	tuple size	200 bytes
join object reference	500	blocking factor	40
sort n tuples	n $\log_2$(n) * 10	index type	(clustered) B^*-tree
		storage allocation	disk
buffer manager:		allocation to PE	#PE/2 + 1..#PE
page size	8 KB		(#PE/3 + 1..2#PE/3)
buffer size per PE	250 pages (2MB)	**join queries:**	
		access method	via clustered index
		input relations sorted	FALSE
disk devices:		scan selectivity	0.1%-10% (varied)
controller service time	1 ms (per page)	no. of result tuples	100-10000 (varied)
transmission time per page	0.4 ms	size of result tuples	400 bytes
avg. disk access time	15 ms	degree of parallelism	
		for join:	1-80 PE(varied)
communication network:		arrival rate	single-user,
packet size	128 bytes		multi-user (varied)
avg. transmission time	8 microsec	query placement	random (uniformly
			over all PE)

Table 1: System configuration, database and query profile**.**

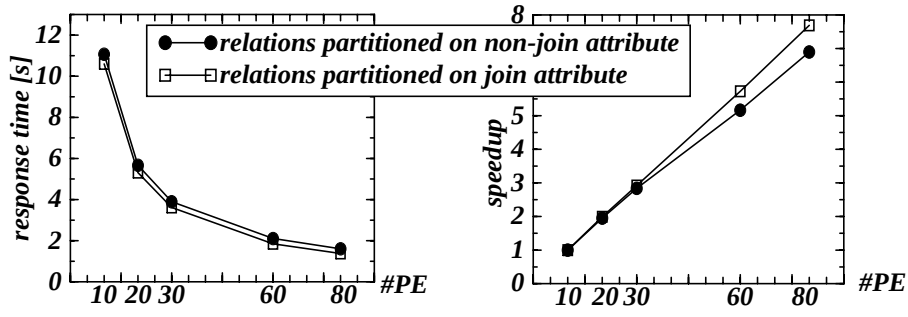


Figure 3: Influence of parallel query processing on response time and speedup.

5 Simulation Results

Our experiments concentrate on the performance of parallel join processing in single-user and multi-user mode. The single-user experiments have been performed to validate our simulation system and to clarify the differences to the multi-user results. The base experiment described in section 5.1 analyses scalability of our join strategy in single-user mode. In sections 5.2 and 5.3 we investigate join performance for different degrees of intra-query parallelism in single-user and multi-user mode, respectively. Additionally, the performance impact of the size of intermediate results is analysed. Finally, we compare the performance of four workload allocation alternatives for parallel join processing in multi-user mode (5.4).

5.1 Base Experiment

The base experiment measures response time and response time speedup[5] of our parallel join strategy for the parameter settings of Table 1. The number of processing nodes (#PE) is varied between 10 and 80. The input relations A and B are both partitioned into #PE/2 fragments and allocated on disjoint nodes. The queries' join operators are executed on the PE holding relation A. Thus, both scan operators as well as the join operator are processed on #PE/2 nodes using intra-operator parallelism. Both scans are supported by indices and select 10% of the their input tuples. Two cases are considered depending on whether or not the join attribute corresponds to the partitioning attribute of the relations. If the relations are partitioned on the join attribute, only the small relation B needs to be redistributed among the A nodes performing the joins. Otherwise, both relations are redistributed according to a hash function on the join attribute.

Figure 3 shows the obtained response time and speedup results. As expected, response times are better when the relations are partitioned on the join attribute because of the reduced communication overhead. Still, the query response time is significantly reduced in both cases as more PE are added for query execution[6]. For both query types, we observe a near-linear response time speedup (speedup factors of 6.8 and 7.7 on 80 PE). This is favored by the large relation sizes and the considerable I/O overhead for accessing the database files on disks. So approximately 57% of the query response time is due to disk I/O for 80 PE (I/O activity occurs not only for the scan, but also during the join phase since the temporary relations could not always be kept in main memory). Since query execution is comparatively expensive, even 80 PE could be employed effectively. We also conducted the base experiment for memory-resident fragments. In this case the speedup values were considerably lower, in particular for the joins on non-partitioning attributes. This is because the communication overhead for redistributing the relations is more significant when no I/O delays occur.

Even for a disk-based database allocation, perfect linear speedup cannot generally be achieved over the entire range of processing nodes. This is because start-up costs for the distributed execution of the (scan and join) operators increase with the number of PE involved, while the total

5. *Response time speedup* measures the improvement of complex query response times as more PE are added to execute the query. For N PE, the speedup is obtained by dividing the response time for the base case (10 PE in our experiments) by the response time result for parallel execution on N PE (N > 10) with the same database size [Gr91].

6. [De90] observed basically the same behaviour when running similar join queries on the Gamma database machine.

amount of useful work remains the same. Furthermore, the communication overhead for redistributing the scan output among the join processors increases quadratically with the number of processors. Therefore, the ratio between start-up and communication overhead, and the amount of useful work per PE deteriorates when the number of PE grows, thereby limiting the effectiveness of parallel query processing. This is a general trade-off of parallel query processing and has been quantified in several previous studies [Bo90, DGSB90, MR92]. In the following experiments, these effects will be more pronounced than in the base experiment.

5.2 Degree of Join Parallelism in Single-User Mode

In this and the next subsection we study join performance for different degrees of intra-transaction parallelism and intermediate result sizes. For this purpose, we vary the number of join processors as well as the selectivity of the scan operators. For these experiments we use a constant system size of 80 PE and a declustering of both relations across 40 disjoint PE. Thus scan overhead for a given selectivity factor remains unchanged for the different configurations so that performance differences are due to join processing. The number of join processors is varied between 1 and 80 and the join PE are chosen at random.

Figure 4 shows the resulting response time and speedup results for different scan selectivities in single-user mode. We observe that increasing the number of join processors is most effective for

scan selectivity	number of join PE					
	1	10	20	30	60	80
10 %	6177	2082	1837	1797	1767	**1736**
1 %	677	281	**264**	266	279	286
0.1 %	182	**149**	151	158	172	178

response time [ms]

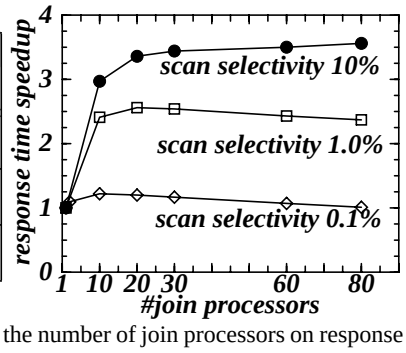


Figure 4: Influence of the size of intermediate results and the number of join processors on response time and speedup in single-user mode.

"large" joins, i.e. for high scan selectivity (10%). In this case, response times could continuously be improved by increasing the degree of intra-operator parallelism, although only slightly for more than 20 join processors. For small joins (selectivity 0.1%) response times improved only for up to 10 join processors. This is because the work per join processor decreases with the degree of intra-operator parallelism, while the communication overhead for redistributing the data increases. Thus even for large joins and despite single-user mode, comparatively modest speedup values are achieved. Of course, this is also influenced by the fact that the scan portion of the response times is not improved when increasing the number of join processors.

In the response time table of Fig. 4, the minimal response times are printed in bold-face to indicate the "optimal" degree of intra-query parallelism (minimum response time point P_{mrt}). In single-user mode when the entire system is at the disposal of a single query, the optimal degree of parallelism is solely determined by rather static parameters such as the database allocation, relation sizes and scan selectivity. Thus the query optimizer can determine the number of join processors without considering the current system state (no need for dynamic load balancing).

5.3 Degree of Join Parallelism in Multi-User Mode

For the multi-user experiment, we varied the arrival rate for our join query. The resulting response time results for different degrees of join parallelism and 1.0% and 0.1% scan selectivities are shown in Figure 5. The results show that multi-user mode significantly increases query response times. Furthermore, the effectiveness of join parallelism increasingly deteriorates with growing arrival rates. This is mainly due to increased CPU waits, because CPU requests (for communication as well as for object references) of concurrent queries have to be served by a limited number of processors. An important observation is that the optimal degree of join parallelism (P_{mrt}) for single-user mode does not yield the best response times in multi-user mode. In fact, for multi-user mode the optimal degree of join parallelism depends on the arrival rate and thus on the current system utilization. The higher the system load, the worse the single-user P_{mrt} point performs and the lower the optimal multi-user P_{mrt} becomes. This is because the communication overhead in-

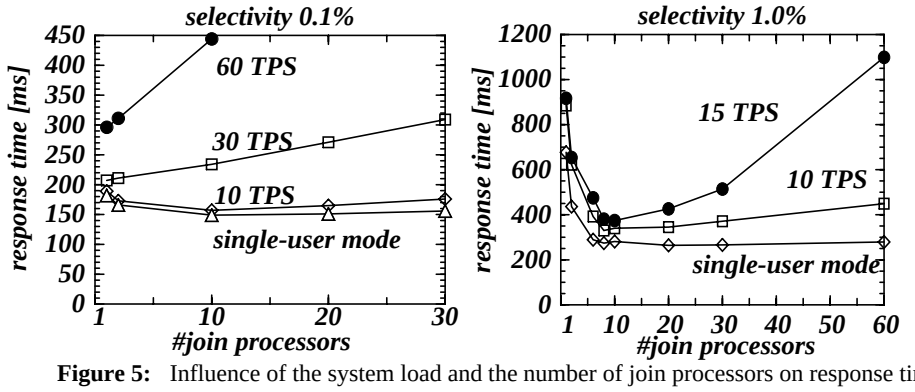


Figure 5: Influence of the system load and the number of join processors on response time.

creases with the number of join processors which is the less affordable the more restricted the CPU resources are.

The differences between single-user and multi-user results are particularly pronounced for small joins (0.1% selectivity). For an arrival rate of 60 TPS, join execution on a single join processor achieved here the best response time. In this case, the single-user P_{mrt} of 10 results in a response time that is 50% higher than for the multi-user P_{mrt} of 1. For 30 TPS, join parallelism also did not result in any response time improvement but only introduced unnecessary communication overhead thereby limiting throughput. In the case of 1% scan selectivity, join parallelism was more effective since more work has to be executed by the join processors. However, Fig. 5 shows that a good degree of parallelism is difficult to find since it is confined to a small range. In single-user mode, on the other hand, more than the optimal number of join processors did not significantly increase response times. For 1% selectivity, the multi-user P_{mrt} differs from the single-user P_{mrt} as well (for 15 TPS, the multi-user P_{mrt} is 10 rather than 20).

Our experiment clearly demonstrates the need of dynamic load balancing and scheduling for parallel join processing in multi-user mode. The optimal degree of intra-query parallelism has to be chosen according to both the size of intermediate results and the current system load. In the next experiment we study the performance of a dynamic workload allocation strategy that selects the join processors based on the current system utilization.

5.4 Processor Allocation of Join Operator

While the processor allocation of scan operators is determined by the data distribution, there is more freedom for allocating parallel join operators. This is because the join is not performed on base relations but on intermediate data that can be distributed dynamically. Hence a join operator may be executed on any PE permitting a wide range of allocation strategies. In our last experiment, we study the performance of the following four join operator allocation strategies:

- **Strategy 1 “Minimize Data Transfer”:**
  This strategy tries to minimize the communication cost for data transfers by allocating the join operators to those PE owning most of the data needed for join processing. The degree of join parallelism and the selection of join processors are determined by the data distribution. For our join query, strategy 1 means that the join operators are allocated on the processors holding fragments of the larger relation A.
- **Strategy 2 “Assign Join Operators to the Processors With Minimal Work for Scan”:**
  This strategy aims at balancing the load by assigning join operators to processors where no scans have to be performed. If all processors hold fragments of the input relation, the join operators are assigned to those nodes with the smallest fragments.
- **Strategy 3 “Random”:**
  This strategy does not care about any information on the database distribution or query profile. It tries to avoid that certain nodes become overloaded by simply distributing the join operators across all available PE at random.
- **Strategy 4 “Keep Overall CPU Utilization Balanced”:**
  This strategy uses global information on the processing nodes’ CPU utilization. The basic idea is to keep the overall CPU utilization balanced in order to avoid CPU bottlenecks. The join operators are assigned to those PE which currently offer the lowest CPU utilization[7].

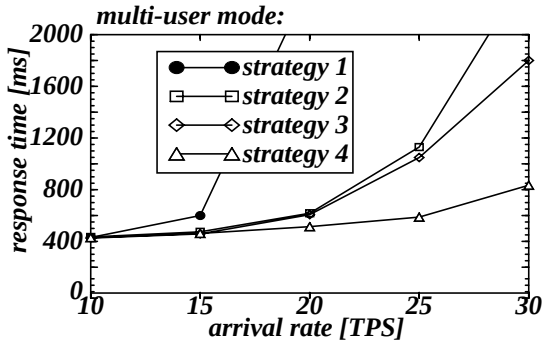
Alternatives 1 to 3 represent *static strategies* since they do not consider the current system state; operator allocation is only based on static information such as the database distribution (strategies 1 and 2) or the number of PE (strategy 3). Strategy 4 is a dynamic approach since it considers the current CPU utilization for workload allocation.

single-user mode:

join allocation	responsetime [ms]
strategy 1	378
strategy 2	411
strategy 3	395
strategy 4	400

response time [ms]

multi-user mode:



join allocation	arrival rate				
	10 TPS	15 TPS	20 TPS	25 TPS	30 TPS
strategy 1	20 50 10 0	30 75 15 0	39 97 20 0	----------	----------
strategy 2	23 29 09 31	35 44 15 46	46 57 18 63	56 70 22 76	65 80 26 89
strategy 3	22 37 18 11	33 55 28 16	44 74 35 23	55 88 47 30	64 95 59 38
strategy 4	23 23 23 23	34 35 34 33	45 46 45 44	56 57 56 55	66 67 66 65

cpu utilization [%] (global, nodes 1..20, 21..40, 41..60)

Figure 6: The influence of the workload allocation strategy and the system load on response time and processor utilization.

Using these strategies, we performed single-user as well as multi-user experiments on a 60 node shared nothing system. To provide some alternatives for operator allocation, we determined the data distribution as follows: relation A is distributed across the nodes 1 to 20 and relation B across nodes 21 to 40. Nodes 41 to 60 do not hold any data. Both scans select 1% of the relations tuples. To facilitate a comparison between the different allocation strategies, we employ a fixed degree of join parallelism in this experiment by always using 20 join processors. Strategy 1 uses the A-holding PE (1-20) for join processing, while strategy 2 selects the nodes 41-60 as join processors since they have no scan operations to perform. Strategies 3 and 4 may employ any PE of the system for join processing. The selection of the 20 join processors occurs at random (strategy 3) or based on the current CPU utilization (strategy 4).
Figure 6 plots response time results for single-user and multi-user mode. Furthermore, the average processor utilization for the multi-user experiments is shown. Each entry in this table consists of four numbers indicating the average CPU utilization of all PE, of the A-holding PE 1-20, of the B-holding PE 21-40, and of PE 41-60, respectively.

Single-user performance
As expected, the best single-user response times are achieved by strategy 1which minimizes communication overhead. Strategy 2 yields the highest response time since the joins are performed on nodes not holding any data leading to the highest communication and cooperation overhead. Strategies 3 and 4 offer response times in between of strategies 1 and 2. This shows that in single-user mode there is no need for dynamic load balancing since all PE have a low CPU utilization. This also explains the low differences between the four strategies (< 10%) indicating that in single-user mode selection of the join processors is less important than finding the right degree of join parallelism (section 5.2).

7. For this purpose, we assume that information on the global CPU utilization is available when the join operators are allocated. This can be achieved by maintaining such information either at a designated PE or by periodically broadcasting it among all PE. The overhead for exchanging this information can be kept small by piggy-backing it to other messages used for query processing.

Multi-user performance

As the system load increases, the performance impact of the different allocation strategies becomes more visible. The average query execution time raises rapidly with increasing arrival rates, particularly in the case of the static strategies. Strategy 1 which performed best in single-user mode exhibits the lowest performance in multi-user mode. This strategy caused substantially higher response times and lower throughput than the other schemes. Throughput is limited to about 20 TPS since this strategy only uses 40 of the 60 PE. For this arrival rate, the A-holding nodes are completely overloaded (97%) thus leading to drastically increased response times. This result underlines that in multi-user mode limiting the communication overhead is by far less important than load balancing.

Strategies 2 and 3 achieved better performance since they use all processors thus supporting about 30 TPS. However, as the table on CPU utilization reveals there are still significant load imbalances with these two static strategies. In particular, with strategy 2 the B-holding nodes are underutilized so that the other nodes become overloaded at 30 TPS. Strategy 3 (random) is slightly better than strategy 2 since it spreads the join work equally among all processors. This strategy however, suffers from the load imbalances due to the different degrees of scan activity on the different nodes. Here, the A-holding nodes become overloaded first thus limiting throughput.

The dynamic workload allocation strategy 4 clearly provided the best throughput and response time results. This strategy avoids local bottlenecks by assigning the join operators to the PE with the lowest CPU utilization. As a result, resource utilization is kept balanced among all nodes and response time raises very slowly with increasing arrival rates. This also supports a higher throughput than 30 TPS. Thus, the dynamic load balancing strategy is capable of satisfying both short response times by utilizing intra-query parallelism as well as high throughput.

Although strategy 4 outperformed the static strategies, we observed an interesting phenomenon in our experiments which is inherent to dynamic load balancing strategies. We found out that strategy 4 tends towards assigning two consecutive queries' joins to the same processors, since the impact of the first query's activation on resource utilization does not appear immediately and since the information on CPU utilization is updated only periodically. Therefore, queries based on the same information about resource utilization will be assigned to the same processing nodes, thus impeding each other. By taking this effect into account, the dynamic strategy 4 can be further improved, e.g., by estimating changes in the resource utilization due to an assignment decision.

6 Summary

We have presented a simulation study of parallel join processing in shared nothing systems. In contrast to previous studies, we focussed on the performance behavior in multi-user mode since we believe this will be the operating mode where parallel query processing must be successful in practice. Multi-user mode means that only limited resources are available for query processing and that both response time and throughput requirements must be met. This necessitates dynamic scheduling strategies for assigning operators during query processing.

In contrast to scan operations, parallel join strategies offer a high potential for dynamic load balancing. In general, a join is not performed on base relations but on derived data obtained by previous scan operations. These intermediate results are dynamically redistributed among several join processors to perform the join in parallel. The number of join processors (degree of join parallelism) and the selection of these processors represent dynamically adjustable parameters.

Our experiments demonstrated that effectively parallelizing join operations is much simpler in single-user than in multi-user mode. In single-user mode the optimal degree of join parallelism is largely determined by static parameters, in particular the database allocation, relation sizes and scan selectivity. Determining where the join operators should be executed is also unproblematic since all processors are lowly utilized in single-user-mode. Thus, the join processors can also be selected statically so that communication overhead is minimized.

In multi-user mode, the optimal degree of join parallelism depends on the current system state and is the lower the higher the nodes are utilized. Using the optimal single-user degree of join parallelism in multi-user mode is therefore not appropriate and was shown to deliver sub-optimal performance (up to 50% higher response times in our experiments). Our results demonstrated that selection of the join processors must also be based on the current utilization in order to achieve both short response times and high throughput. Even a simple load balancing strategy based on

the current CPU utilization was shown to clearly outperform static strategies. The best workload allocation strategy in single-user mode achieved the worst performance in multi-user mode. Thus, balancing the load is more important for selecting the join processors in multi-user mode than minimizing the communication overhead.

In future work, we will study further aspects of parallel query processing in multi-user mode that could not be covered in this paper. In particular, we plan to investigate dynamic scheduling strategies for mixed workloads consisting of different query and transaction types [RM93]. Furthermore, we will consider the impact of data contention (lock conflicts) and data skew on the performance of parallel query processing.

7 References

CYW92 Chen, M.; Yu, P.; Wu, K. 1992: Scheduling and Processor Allocation for Parallel Execution of Multi-Join Queries. *Proc. 8th IEEE Data Engineering Conference*, 58-67.

Bo90 Boral, H. et al. 1990: Prototyping Bubba: A Highly Parallel Database System. *IEEE Trans. on Knowledge and Data Engineering 2*(1), 4-24.

De90 DeWitt, D.J. et al. 1990: The Gamma Database Machine Project. *IEEE Trans. on Knowledge and Data Engineering 2*(1), 4-62.

DG92 DeWitt, D.; Gray, J. 1992: Parallel Database Systems: The Future of High Performance Database Processing. *Communications of the ACM* 35(6), 85-98.

EGKS90 Englert, S., Gray, J., Kocher, T., Shath, P. 1990: A Benchmark of NonStop SQL Release 2 Demonstrating Near-Linear Speedup and Scale-Up on Large Databases. *Proc. ACM SIGMETRICS Conf.*, 245-246.

GW89 Graefe, G; Ward, K. 1989: Dynamic Query Evaluation Plans. *Proc. 1989 SIGMOD Conf.*, 358-366.

G90 Graefe, G. 1990: Volcano, an Extensible and Parallel Query Evaluation System. University of Colorado at Boulder, Department of Computer Science.

Gr91 Gray, J. (Editor) 1991: The Benchmark Handbook. Morgan Kaufmann Publishers Inc.

Li89 Livny, M. 1989: DeNet Users's Guide, Version 1.5. Computer Science Department, University of Wisconsin, Madison.

MR92 Marek, R.; Rahm, E. 1992: Performance Evaluation of Parallel Transaction Processing in Shared Nothing Database Systems. *Proc. 4th Int. PARLE Conference*, LNCS 605, Springer, 295-310.

MLO86 Mohan, C., Lindsay, B., Obermarck, R. 1986: Transaction Management in the R* Distributed Database Management System. *ACM TODS* 11 (4), 378-396.

MS91 Murphy, M.; Shan, M. 1991: Execution Plan Balancing. *Proc. 1st Int. Conf. on Parallel and Distributed Information Systems*.

Ne86 Neches, P.M.1986: The Anatomy of a Database Computer - Revisited. *Proc. IEEE CompCon Spring Conf.*, 374-377.

ÖV91 Özsu, M.T., Valduriez, P. 1991: Principles of Distributed Database Systems. Prentice Hall.

Pa90 Patel, S. 1990: Performance Estimates of a Join. In: Parallel Database Systems (*Proc. PRIMSA Workshop*), Lecture Notes in Computer Science 503, Springer Verlag, 124-148.

Pi90 Pirahesh, H.et al. 1990: Parallelism in Relational Data Base Systems: Architectural Issues and Design Approaches. In *Proc. 2nd Int.Symposium on Databases in Parallel and Distributed Systems*, IEEE Computer Society Press.

RM93 Rahm, E.; Marek, R. 1993: Analysis of Dynamic Load Balancing for Parallel Shared Nothing Database Systems. Techn. Report, Univ. of Kaiserslautern, Dept. of Comp. Science, Febr. 1993.

SD89 Schneider, D.A., DeWitt, D.J. 1989: A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment. *Proc. ACM SIGMOD Conf.*, 110-121.

SD90 Schneider, D.A., DeWitt, D.J. 1990: Tradeoffs in Processing Complex Join Queries via Hashing in Multiprocessor Database Machines. *Proc. 16th Int. Conf. on Very Large Data Bases*, 469-480.

SSU91 Silberschatz, A.; Stonebraker, M.; Ullman, J. 1991: Database Systems: Achievements and Opportunities. *Communications of the ACM* 34(10), 110-120.

St86 Stonebraker, M. 1986: The Case for Shared Nothing. *IEEE Database Engineering* 9(1), 4-9.

Ta88 The Tandem Database Group 1988: A Benchmark of NonStop SQL on the Debit Credit Transaction. *Proc. ACM SIGMOD Conf.*, 337-341.

Ta89 The Tandem Database Group 1989: NonStop SQL, A Distributed, High-Performance, High-Availability Implementation of SQL. Lecture Notes in Computer Science 359, Springer-Verlag, 60-104.

WDJ91 Walton, C.B; Dale A.G.; Jenevein, R.M. 1991: A Taxanomy and Performance Model of Data Skew Effects in Parallel Joins. *Proc. 17th Int. Conf. on Very Large Data Bases*, 537-548.

WT91 Watson, P., Townsend, P. 1991: The EDS Parallel Relational Database System. In: Parallel Database Systems (*Proc. PRIMSA Workshop*), Lecture Notes in Computer Science 503, Springer-Verlag, 149-168.

WFA92 Wilschut, A.; Flokstra, J.; Apers, P. 1992: Parallelism in a Main-Memory DBMS: The performance of PRISMA/DB. *Proc. 18th Int. Conf. on Very Large Data Bases*, 521-532.