IMPLEMENTATION AND PERFORMANCE EVALUATION OF THE PARALLEL RELATIONAL DATABASE SERVER SDC-II

Takayuki Tamura
Minoru Nakamura[a]
Masaru Kitsuregawa

Institute of Industrial Science,
The University of Tokyo
7-22-1, Roppongi, Minato-ku,
Tokyo 106, Japan.
{tamura,kitsure}@tkl.iis.u-tokyo.ac.jp

Yoshihisa Ogawa

Fujitsu Limited
1015, Kamikodanaka, Nakahara-ku,
Kawasaki 211, Japan.

Abstract -- *This paper presents the implementation and performance evaluation of the SDC-II, the Super Database Computer II. The SDC-II is a highly parallel relational database server, which consists of eight data processing modules interconnected by two networks, where each module contains up to seven processors connected by two busses and four disk drives. The SDC-II employs several key techniques to efficiently support join-intensive queries: a sophisticated parallel hash join algorithm named ‘‘the bucket spreading parallel hash join algorithm,’’ an efficient data passing mechanism in hardware and software, and an intelligent interconnection network which has an ability to generate a flat bucket distribution and to provide almost conflict free routing. The evaluation results obtained through the join queries with skewed data, show that the SDC-II works quite efficiently under various conditions attaining a high degree of parallelism.*

1 Introduction

The wide spread adoption of relational database systems has brought very large databases and very large transaction logs. Though the log data have been usually discarded, some users begin to use them to analyze the behavior of their customers and to provide new business plans. Queries issued to the systems in such anaylses are ad-hoc and intensively use heavy relational operators such as join, aggregation, and order-by. Thus very high performance relational database systems which can efficiently scan a large amount of data and have sophisticated processing algorithms for heavy operators are needed to build decision support systems of practical use.

Considerable research on parallel database processing has been done in the last decade, and the effectiveness of parallelization in improving performance has been proved through the research [4, 7, 9, 2, 5, 16]. Based on the results, several vendors are now shipping commercial parallel DBMS products running on commercial parallel platforms[1]. While research interests and efforts have concentrated on the parallelism exhibited in speedup and scaleup characteristics, the efficiency of fundamental I/O operations tended to be ignored. Few papers present analyses on I/O performance including disk transfer and network communication on real parallel machines. Most of such machines have serious bottlenecks in disk drives, busses, or interconnection networks, resulting in ill-balanced configurations for large database applications. In addition, dynamic behavior of the internal states of the systems, such as buffer consumption, processor workloads, and network traffic based on run time trace information, has been also disregarded.

On the other hand, because the data parallelism exploited by hash-based parallel join algorithms is sensitive to the skew of the input data distribution, a number of skew handling methods have been also proposed [13, 17, 10]. However, these techniques are not yet implemented nor considered in commercial products. Moreover, except for [6], almost all evaluations of the effect of data skew and skew handling in parallel hash join operations have been done by simulation, not on real database machines.

This paper presents the implementation and performance evaluation of the SDC-II, the Super Database Computer II, to describe dynamic behavior of the system and analyses of our skew handling mechanism in more detail than former research.

The first prototype of the SDC (SDC-I) was finished in 1991 [11, 8]. One of the key features of the SDC is its use of the bucket spreading parallel hash join algorithm, which is robust in the presence of data skew[13]. This algorithm enables the system to achieve a high degree of scalability under various data distributions. In addition, we designed a special I/O handling mechanism to support efficient data transfer eliminating operating system overhead. Because most database applications are I/O bound, we have directed our efforts to performance improvement

---

[a] Currently at Mitsubishi Electric Corporation.

of disk accesses and network communication rather than that of computations. It consists of two data processing modules interconnected by a network, where each module contains five MC68020 microprocessors and two SCSI 8 inch disk drives. Though the SDC-I showed much higher performance than other parallel database machines in those days, we could not prove the scalability of the system due to its limited number of modules.

As a solution to the problem, we started development of the SDC-II. The SDC-II consists of eight data processing modules, where each module employs seven MC68040's and four SCSI 3.5 inch disk drives with dedicated I/O processors for the disks and the networks. Furthermore, we introduced additional hardware into the interconnection network to assist the bucket spreading hash join algorithm. It enables the network to automatically distribute the data load equally among the modules, forming a flat bucket distribution. At the same time, the network is able to reduce the conflict rate to almost zero without introducing topological redundancy. This is one of the factors that guarantees system scalability.

The remainder of this paper is organized as follows. In section 2, we describe the parallel relational algorithms used by the SDC-II. Section 3 and section 4 present the implementation of the SDC-II hardware and software, respectively, and describe how they work during the operations. The performance evaluation of join queries with skewed and non-skewed data is described in section 5. Section 6 presents a brief explanation on the complex query execution. Section 7 concludes this paper.

2 Bucket Spreading Hash Join Algorithm

The parallelism which resides in relational query processing on the shared nothing architecture can be classified into three categories: inter-query, intra-query, and intra-operator parallelism. In inter-query or intra-query parallelism, several queries or several operators within a query are processed in parallel at different processing nodes, respectively. However, in ad-hoc query intensive environments, these parallelisms may suffer from bottlenecks in some queries or operators, due to the larger size of the operand relations and the higher complexity of the queries. In contrast, algorithms employing intra-operator parallelism can relax these bottlenecks by declustering the operand relations among all processing nodes and running multiple copies of the operator at all processing nodes to exploit the data parallelism.

The hash based join algorithms are well suited for intra-operator parallelism and have been widely adopted for parallel join operations. The parallel hash join algorithm first partitions both operand relations into disjoint subsets called buckets, by applying a hash function to the join attribute of each tuple. Since each bucket consists of the tuple which map to the same hash value, joining the two relations results in joining each pair of the corresponding buckets from the two relations. Each join operation can be performed in parallel amongst the processing nodes.

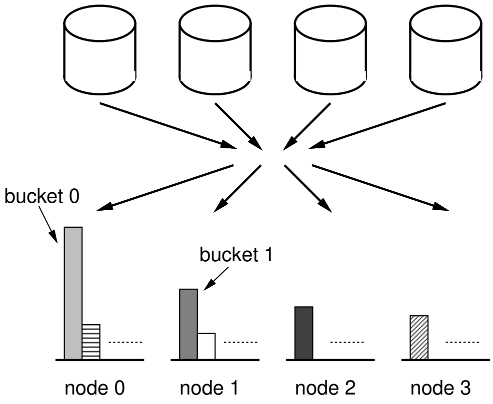


Figure 1: Static bucket assignment in conventional algorithms

In conventional algorithms such as GRACE and Hybrid, each bucket generated by the hash function is located at a certain processing node following a statically determined bucket-to-node mapping (figure 1). Then buckets are loaded from the local disks of each processing node to build the hash tables without inter-node communication. Although they work well when the bucket sizes are uniformly distributed, the distribution is hardly predictable and cannot always be guaranteed to be uniform for real world queries. Once the distribution fluctuates, then bucket loading times, hash table creation times, and hash table probing times will vary among the processing nodes. Moreover, certain buckets may not fit in main memory, incurring significant performance deterioration since the overflowed buckets require recursive hash partitioning. In such cases, the total execution time is limited by the slowest processing node. Thus, the conventional hash join algorithms become very fragile in the presence of bucket size skew.

Our solution to the problem is adding new features to the conventional algorithms: a bucket spreading strategy and a dynamic bucket assignment strategy. The enhanced algorithm is called *the Bucket Spreading Hash Join* (BS hash join)[13]. Its outline is as follows (figure 2). Let the smaller relation be *R* and the larger be *S*.

i) Split Phase: Each processing node applies the hash function to each tuple of its portion of the relation *R*. The number of buckets is set to a large value to avoid bucket

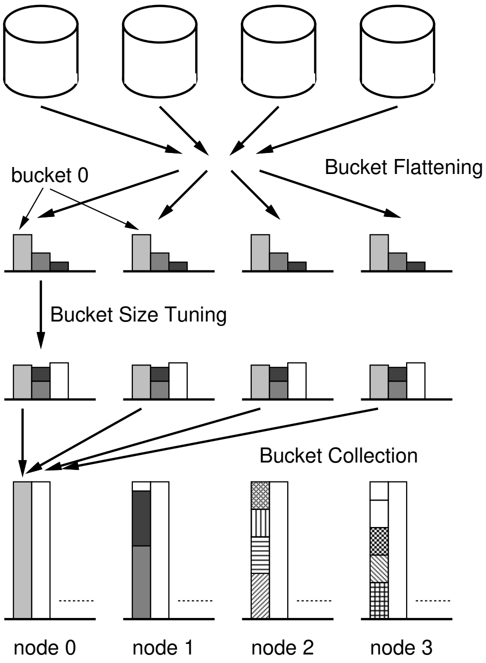


Figure 2: Dynamic bucket assignment in Bucket Spreading Hash Join

overflow. Then the tuples are sent out over the interconnection network so that each bucket is distributed equally among the processing nodes. That is, the fragments of each bucket will be almost equal in size among the processing nodes (*the bucket spreading strategy*). This can be done using software, by determining the destination nodes of the tuples of each bucket in a round robin manner. However, we employ specialized network hardware to realize a flat distribution with high efficiency and low congestion. This is called *the Bucket Flattening Omega Network*, and is described in the following section.

ii) Scheduling Phase: When all processing nodes finish reading the tuples from their disks, they examine the size of the buckets. The size of the buckets will be distributed, but due to the large number of buckets, many small buckets will have been generated. These smaller buckets are clustered together to form larger buckets, which are able to fit in main memory. This process is called bucket size tuning[12], and uses several heuristics. Since the fragments of a bucket have almost equal size, not all of the processing nodes need to be involved in this phase. Instead, a certain coordinator can perform the clustering with its local information. The coordinator also determines the assignment of the buckets to the processing nodes at this phase (*the dynamic bucket assignment strategy*).

If some buckets exceed main memory by themselves in spite of the above overflow avoidance technique, the buckets are split recursively and distributed again. However, this process can be done in parallel at all the processing nodes, thus the performance degradation is not so serious as the conventional algorithms.

iii) Join Phase: Based on the number of clustered buckets, different strategies are chosen. If there is only one bucket assigned to each processing node, the hash table is built on the assigned processing node by collecting the fragments of the bucket from all processing nodes. Then the join process proceeds as in the Simple hash join. The relation S is read out from the disks and each tuple is sent to the destination node directly according to its hash value. The hash table on the processing node is probed with the tuple to produce result tuples.

On the other hand, if more than one bucket are assigned to each processing node, the relation S is split in the same way as in the split phase above. The fragments of a bucket from relation R are collected together on the corresponding processing node to build its hash table. Then the bucket fragments from the relation S are also collected to probe the hash table to produce the results. This process repeats until there are no more buckets.

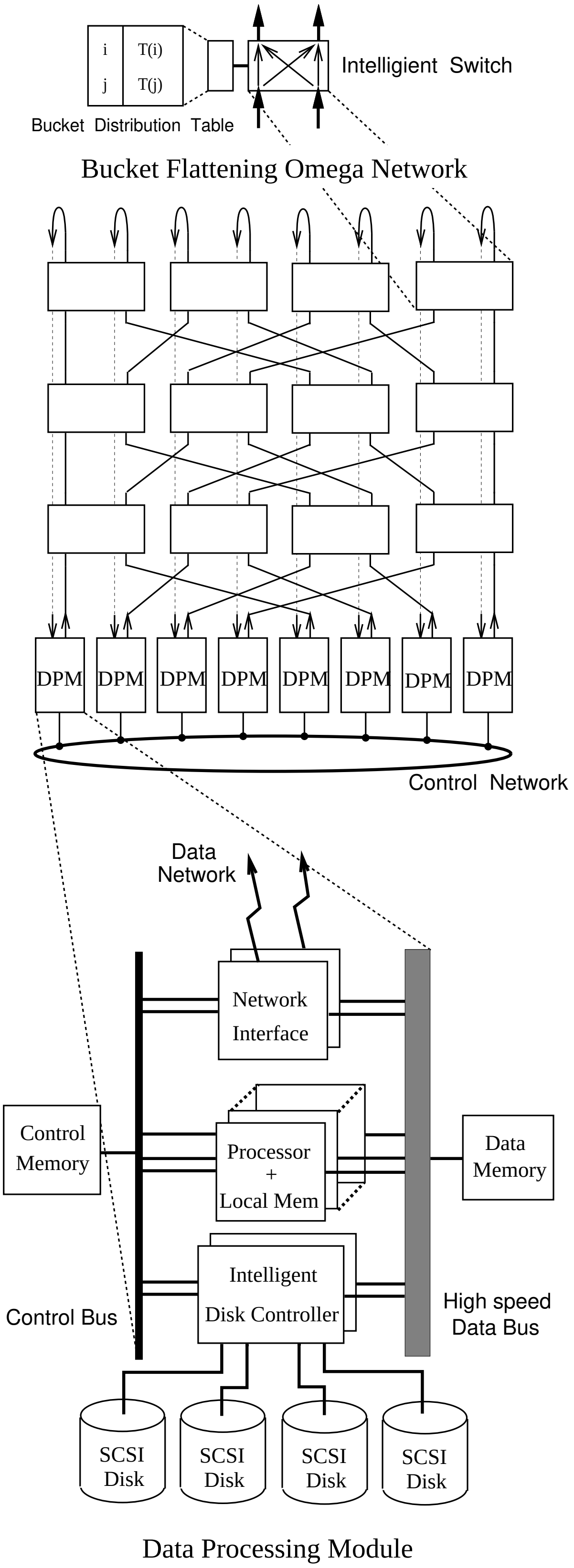
In the BS hash join algorithm, the bucket size tuning and the collection of buckets over the interconnection network may seem to incur an extra cost. However, this contributes to diminish the incidence of bucket (hash table) overflowing significantly and obtains high efficiency in the utilization of disk space and main memory space.

3 Architecture of the SDC-II

3.1 Global Architecture

Instead of employing a cluster of commercial workstations, we designed and built the SDC-hardware from scratch in order to obtain higher performance. As shown in figure 3, the SDC-II globally employs a distributed memory architecture where eight processing nodes are connected through two kinds of interconnection network: the data network (DNet) and the control network (CNet). The data network offers high speed channels for data transfers, while the control network handles control information and supports communication with the front-end machine. Two data networks are provided to divide the load of each network especially in the multi-join operations. The data network also provides the bucket flattening support for the BS hash join algorithm.

Each processing node is called a data processing module (DPM), which has a bus coupled multiprocessor architecture. It contains up to six MC68040 microprocessors



Bucket Flattening Omega Network

Data Processing Module

Figure 3: Architecture of the SDC-II

operating at 25 MHz as data processors (DP), and one MC68040 as a control processor (CP). The CP manages all the activities of the DPM, such as initialization and synchronization of all DPs. It also contains the interface for the 10 Mbps Ethernet employed as the control network. Dedicated I/O processors are also provided for each of two disk controllers (DC), MC68340 microcontroller at 16MHz, and two data network interfaces (NI), MC68020 microprocessor at 25MHz. Each disk controller controls two Fujitsu M2624S 3.5 inch disk drives connected to individual SCSI busses. The I/O processors are responsible only for DMA channel setup. Data transfer itself is done by the DMA hardware using the FIFO buffer on each board.

The processors are connected through two common busses, one for high speed data transfer (HBus, 100 MBytes/sec) and the other for handling inter-processor communication and mutual exclusion (CBus). The HBus is used solely for bulk data transfer. The local address space of each processor is mapped to a certain area of the global address space and accessible from the other processors through the CBus. In addition, the location monitor is mapped into this area, which enables the other processors to send a signal to the processor by raising an interrupt. Shared memory is also composed of two portions: 32 MBytes of data memory (DM) for the raw tuple data and 2 MBytes of control memory (CM) for storing the control data structures such as page headers, hash table entries, and memory consumption statistics for each staging buffer. Since the database processing applications show poor temporal access locality, caching the shared memory data does not necessarily lead to performance improvements. Thus we omitted the cache consistency mechanism from the busses, resulting in a simplified bus implementation. In contrast, each processor executes the codes stored in its local memory, which can be cached in the processor's I-cache.

3.2 Bucket Flattening Omega Network

The data network of the SDC-II was designed to offer architectural support for the BS hash join algorithm. In the split phase of the BS hash join algorithm, each bucket is fragmented among the DPMs to form a flat bucket distribution. As described before, this can be accomplished without any hardware support. However, the effective network bandwidth will be fairly degraded due to the congestion caused by the random scattering of packets. The additional functionality introduced into the data network can solve this problem by determining the destination of each packet adaptively with the network hardware. Because the details of the algorithm and the implementation of the network are beyond the scope of

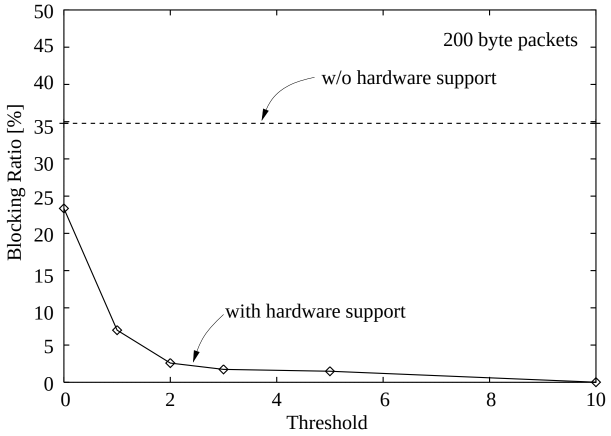


Figure 4: Blocking avoidance in the bucket flattening omega network

this paper, we describe their outlines here.

We employ an omega topology (figure 3), where each switch can be set to either of two states, straight or crossed. Thus the data network is also referred to as the bucket flattening omega network. For avoiding the bottlenecks with the centralized control method, we focus on balancing the number of tuples flowed through each output port of a switch. This is accomplished by keeping track of the difference between the number of tuples output to one port and the other port for each bucket. Whenever a tuple of some bucket leaves the switch, the value of the difference corresponding to the bucket is updated depending on which output port is chosen. Let $M(b)$ denote the difference for bucket b, and b_0 and b_1 the bucket IDs of the tuples which arrived at the two input ports. When two tuples arrived at each port at the same time, the destinations are determined by the sign of $M(b_0) - M(b_1)$. If one tuple arrived at input port 0 and both output ports are free, the destination is determined by the sign of $M(b_0)$. In both cases, the tuples are never blocked. However, if one tuple arrived at input port 0 and output port 0 is occupied by the tuple previously received from input port 1, blocking can occur. If the sign of $M(b_0)$ directs the tuple to output port 0, the tuple has to be blocked at the switch. To decrease the blocking ratio, we introduced a parameter to bias the value of $M(b)$, at the expense of increased skew. That is, when there is a possibility of blocking, each switch tests $M(b) \pm T$ instead of just $M(b)$. T is a threshold value, and a larger T reduces the likelihood of blocking.

We implemented the switching unit with FPGA (field programmable gate array) and SRAM chips. The FPGA realizes the logic portions for a 2×2 crossbar, an adder/subtracter, a memory interface, and sequence controllers, occupying approximately 6,200 gates in the Xilinx LCA XC4010-5 chip. Two SRAM chips (32 KWord $\times$ 16 bit) are used to store the values of $M(b)$ and parameters such as the threshold T. The switch is operating at 12.5 MHz and each of the two ports is 16 bit wide, thus the maximum data transfer rate is 25 MBytes/sec ignoring routing overhead cycles.

Figure 4 shows the measured blocking ratio of the implemented bucket flattening omega network enabling and disabling the support for the flat bucket distribution under a heavy load. Here, four DPMs are involved and each of them sends tuples with randomly distributed bucket IDs at a maximum rate. The horizontal axis denotes the threshold value, which does not affect the software-only method. The blocking ratio represents a relative amount of the blocking cycles occupied in the whole cycles elapsed. The switching unit contains a blocking cycle counter, which enables the measurement of the blocking ratio. Even with the threshold of 0, the bucket flattening method shows a low blocking ratio than a normal destination based method. However, as the threshold increases, the blocking ratio decreases rapidly, and it is almost zero with the threshold larger than 3. Thus, the bucket flattening omega network can avoid congestion almost completely during the split phase of the BS hash join algorithm, attaining a high effective bandwidth.

4 System Software of the SDC-II

While computation dominates in scientific applications, I/O dominates in database processing. The system software of the SDC-II is based on a data driven processing model, in which the sequence of processing is determined by the flow of data, instead of the flow of instructions represented in the program. A similar principle can be found in the event driven model, which is realized in various window and simulation systems. However, these systems intend to handle independent asynchronous events which arise randomly, while the data driven processing system assumes burstness of data transfer, which gives the system an opportunity to take advantage of data buffering.

Figure 5 depicts an abstract view of data movement in the SDC-II, where the hatched circles represent fixed-length pages filled with tuples and white circles empty pages. The filled pages are put in FIFO queues located in shared memory. Each queue, called a buffer, corresponds to a logical I/O stream, and is prepared according to the operation being executed. Actually, the buffers are located in the control memory, and the page headers including pointers to the data bodies on the data memory are put in them. The data processors, represented by four boxes at the center of the figure, keep scanning all input buffers to fetch the next page when they are idle. When

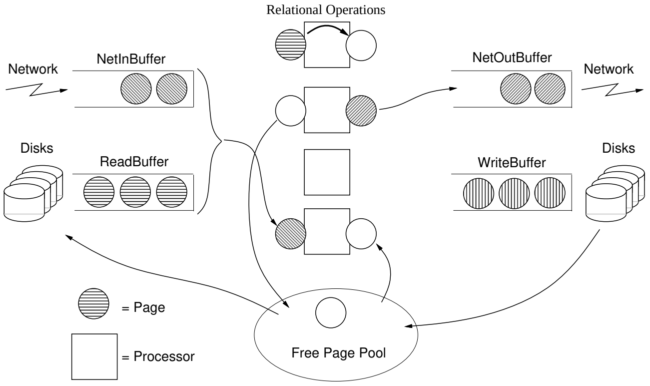


Figure 5: I/O model of the SDC-II

a processor finds a page in any buffers, it dequeues the first page, performs the operations specified by the source buffer repeatedly on each tuple in the page, produces the result tuples in a new page allocated from the free page pool, and finishes with enqueueing the result page into the appropriate output buffer and releases the source page to the free page pool. Thus, in the SDC-II, the processes never perform the input operation voluntarily, but the input streams themselves invoke the processes instead. Because the processes need not to be suspended/resumed, the use of interrupts and context switchings is avoided. The overhead of each I/O operation is kept low enough for the data processors to endure a high I/O transfer rate. On the other hand, the I/O processors, including the disk controllers and the network interfaces, are responsible for interrupt handling of the physical devices. In addition, neither the data processors nor the I/O processors have the permanent pages, but the pages flow among them.

There are several advantages brought by the data driven processing model. First, because the I/O processors are never forced to place the input data in any location specified by the data processors, they can completely avoid the use of temporary buffers and the extra copy operations over them. The I/O processors allocate data area dynamically from the free page pool, and pass it to the data processors by putting its header in the input buffer. The allocation of free pages is very cheap, since they are fixed-size and maintained in a linked list. Second, because the I/O processor can determine when to initiate a data transfer autonomously, prefetching of disk data can be efficiently implemented. In fact, each disk controller issues a read command directing the disk drives to read the entire capacity of the relation, instead of repeatedly issuing read commands with the block size at each completion of a block read. The data streams from the disks are split into pages on-the-fly, and put into the disk read buffers by the disk controllers. When the disk controllers detect that the memory is almost exhausted or the read buffer becomes full because of a slow down in the consumer processes, they send abort signals to the disk drives.

As for the output operations, the disk controllers and the network interfaces behave differently. The network interfaces have to send packets immediately to the physical networks, multiplexing all the logical streams. They scan the network output buffers to get outgoing packets filled in pages, just like the data processors do for the input buffers. In contrast, the disk controllers try to defer the write operations as much as possible. Thus the disk controllers usually pay no attention to the disk write buffers, unless the amount of free memory is too low. This implements half of the dynamic destaging mechanism [12], where bucket destaging is deferred as long as the aggregate buckets fit within main memory. Once the bucket size exceed the main memory size, the largest bucket is picked up and the tuples belonging to that bucket are destaged to the disks, deallocating the occupied memory space for use by incoming tuples.

When a query is issued to the system, the front-end machine compiles and schedules the query. Currently, however, this phase is done manually, since the compiler tools are not yet ready. Then the execution plan is sent to the coordinator process running on a particular DPM. It broadcasts the plan to each DPM control process running on the control processor of each DPM. The DPM control process is responsible for all the activities occurring in the DPM. It sets up the I/O buffers in the shared memory, then initiates the data processors and the I/O processors. Once execution starts, the processors perform their operations without any centralized control. However, there are several barrier synchronization points such as the transition from the build phase to the probe phase.

5 Performance Evaluation

Now we present a performance evaluation of the SDC-II. The two parallel hash join algorithms, GRACE and BS, are compared varying the data skew of input relations. Table 1 shows the configuration of the SDC-II system used for the tests.

We use the following equi-join query.

```
insert into result
select * from R, S
where R.joinkey = S.joinkey
and R.selkey < x
```

This query is a derivation of the *JoinAselB* query in the Wisconsin Benchmark[3]. While two input relations have the same attributes and the same number of tuples

Table 1: Configuration of the SDC-II system under test

max. DPMs	6
max. data networks	1
max. data processors / DPM	4
disk drives / DPM	4
staging memory size / DPM	32 MBytes
page size	8 KBytes
disk drive	
disk drive type	Fujitsu M2624S
formatted capacity	550 MBytes
sustained transfer rate	2.4 MBytes/sec
average seek time	12 ms
data network	
topology	8 port Omega network
max transfer rate	25 MBytes/sec at each port

Table 2: Parameters used in the single join evaluation

parameter	description
N	number of DPMs
P	number of processors / DPM
$\|R\|,\|S\|$	size of relations R, S [tuples] ($\|R\| = \|S\|$)
B	number of buckets
L	tuple length [bytes]
z	bucket size skew (decay factor of the Zipf-like function)
s	selectivity [%]
j	joinability [%]

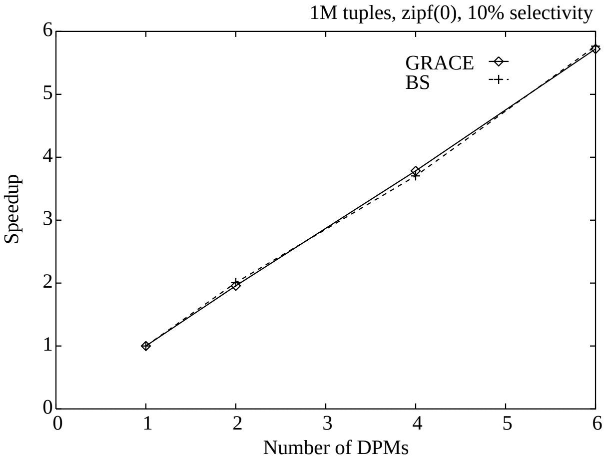


Figure 6: Speedup for GRACE and BS

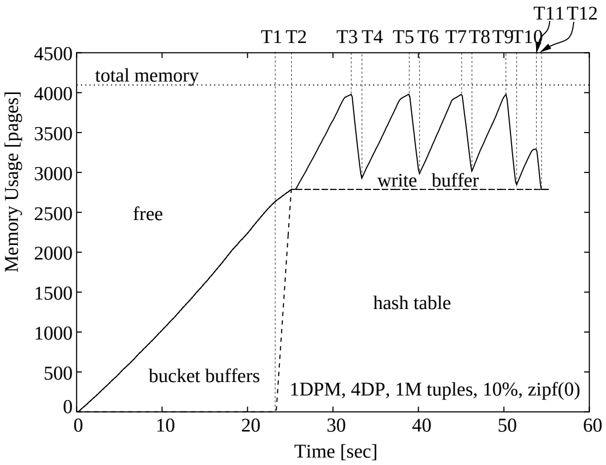


Figure 7: Memory usage trace for GRACE

as in the benchmark, the join attribute now has a non-uniform distribution to cause skew in the resulting bucket sizes. The selection attribute still follows the uniform distribution, which makes control over the selectivity factor very straightforward. Each relation is randomly declustered among all of the DPMs' disks. Because the advantage of declustering the relations based on their join attribute values is not always available in an ad-hoc query environments, we did not test such cases. Similarly, we assumed no index on the selection attribute. A fragment of the relation is stored in a segment of contiguous blocks in a certain disk on a certain DPM, and accessed sequentially during the scan operation.

We assume that the size of the buckets follows a Zipf-like distribution, which is specified as follows.

$$|R_i| = \frac{|R|}{i^z \cdot \sum_{b=1}^{B} \frac{1}{b^z}} \quad (1 \le i \le B)$$

where $|R_i|$ denotes the number of tuples in the i-th bucket of relation R, B the number of buckets, and $|R|$ the number of tuples in relation R which satisfy the selection predicate in the query.

Table 2 summarizes the parameters used in the following performance evaluation.

5.1 Speedup Evaluation

First, we examine the speedup characteristics of the SDC-II. In this test, each DPM contains four data processors, while the number of DPMs varies from 1 to 6. The number of buckets is proportional to the number of DPMs, keeping the number of buckets assigned to each DPM constant (i.e., $B = 12 \times N$). We use the GRACE and the BS algorithms. Other parameters are fixed: $|R| = |S| = 1000000$, $L = 208$, $B = 12$, $s = 10$ %, $j = 100$ %, and $z = 0$. Figure 6 shows the resulting speedup curve. The system shows almost linear speedups for both algorithms.

Even when the number of DPMs is one, all the buckets (about 20 MBytes) fit within main memory. Thus we can take advantage of the dynamic destaging method. That is, at the end of the split phase of relation R, no bucket buffers will have been flushed to disks. This enables all the buckets to be clustered into one bucket, and to form a single hash table without incurring any additional I/O. As the number of DPMs increases, each fragment of a relation stored in a DPM gets smaller. Thus, the scenario of query execution is not affected. With a larger relation size or selectivity, the performance for a small number of DPMs would appear relatively low, since the total amount of memory in such cases is not sufficient to hold all the buckets within memory, buckets would need to be flushed to the disks.

Figure 7 shows the transition of memory consumption for one DPM case traced using a software monitoring method during execution. From time 0 to $T1$ (23.250 sec), the relation R is loaded and split into buckets consuming free pages. At $T1$, the split phase ends up with generating 2636 pages of bucket buffers. From $T1$ to $T2$ (25.125 sec), a hash table is built from them without I/O. Then relation S is loaded to probe the hash table. Since most of main memory is already used by the hash table, the amount of free memory reaches a memory low threshold as the results are accumulated in the write buffer (at $T3$ = 32.125 sec, $T5$ = 38.875 sec, $T7$ = 45.000 sec, and $T9$ = 50.250 sec). This causes the disk controllers to abort the read operation of S and to switch to writing the results. When sufficient free memory is regained (at $T4$ = 33.375 sec, $T6$ = 40.125 sec, $T8$ = 46.250 sec, and $T10$ = 51.500 sec), the controllers restart the read operation autonomously. At $T11$ (53.750 sec), all the results are generated, and the join operation finishes by flushing the results at $T12$ (54.375 sec). Since all the buckets fit in memory, the total amount of I/O is $|R|L + |S|L + sj \cdot |R| \cdot 2L = 457.6$ MBytes, and will take 47.7 seconds to complete without any overhead. Thus, the overhead and the idle times are kept low throughout the operation. This convinces us that the data driven execution model of the SDC-II works quite efficiently.

5.2 Scaleup Evaluation with Skewed Data

Next, we examine the scaleup characteristics, where the size of the source relations is proportionally increased as the number of DPMs increases. The GRACE and BS algorithms are tested with Zipf(0) and Zipf(3) distributions, while the number of DPMs varies from 1 to 6. Other parameters used in the tests are: $P = 4$, $|R| = |S| = 1000000 \times N$, $L = 208$, $B = 12 \times N$, $s = 10$ %, and $j = 100$ %. Though the distributions of buckets are denoted as Zipf(0) and Zipf(3), this does not mean that $|R_i| (1 \leq i \leq B)$ follows the Zipf-like distribution for $B = 12 \times N$ buckets with $1000000 \times N$ tuples. Instead, $|R_{((i-1) \bmod 12)+1}|$ follows the Zipf-like distribution for 12 buckets with 1000000 tuples. Thus, the size of the largest bucket is about 17 MBytes without respect to the number of DPMs, and does not exceed the main memory size. For the GRACE algorithm, IDs of the buckets assigned to the n-th DPM $(1 \leq n \leq N)$ are given by $n + j \cdot N \; (0 \leq j < 12)$.

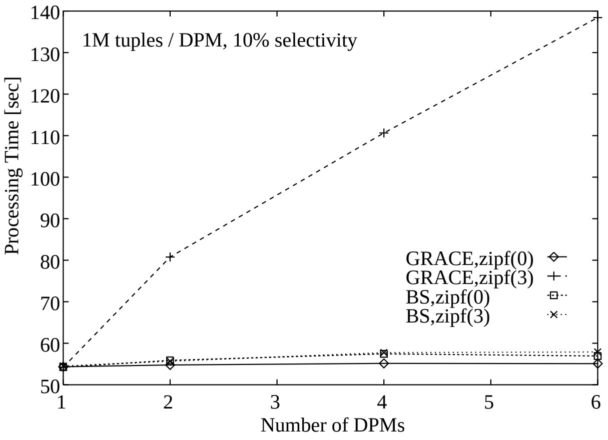


Figure 8: Scaleup for GRACE and BS

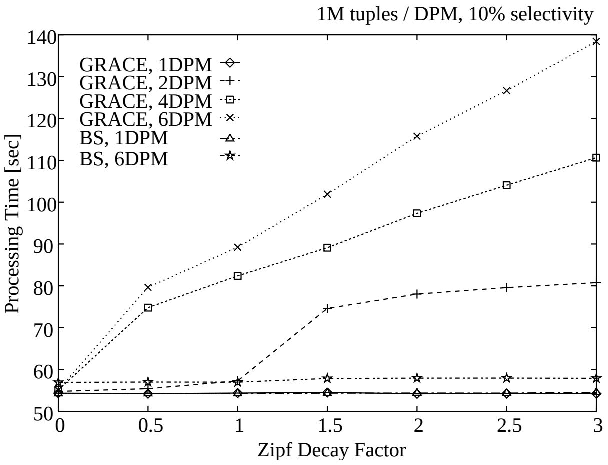


Figure 9: Data skew vs. processing time

Figure 8 shows the obtained scaleup curve. With a flat data distribution (Zipf(0)), both GRACE and BS show good scaleup. Though BS suffers from the extra cost of collecting fragments of each bucket to the corresponding

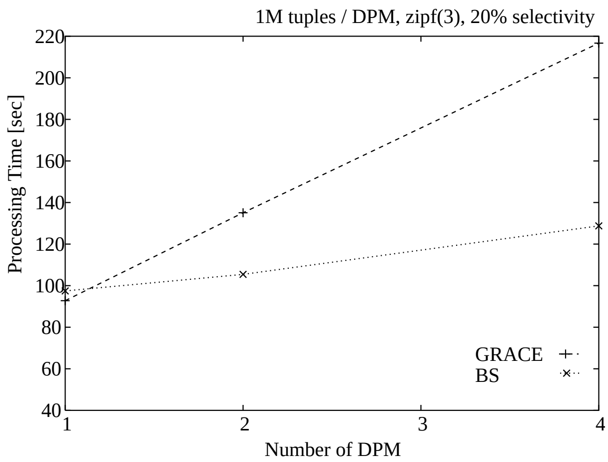


Figure 10: Scaleup for bucket overflow

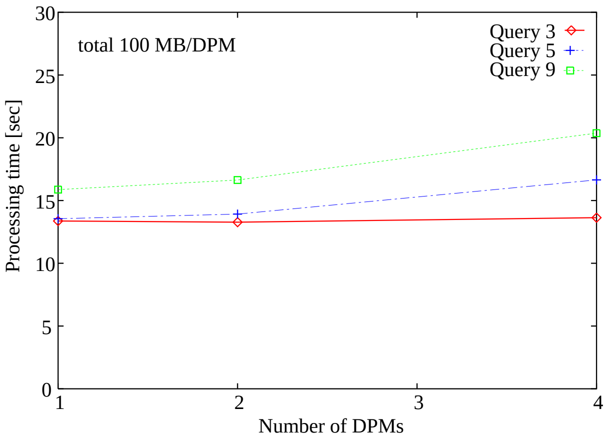


Figure 11: Scaleup of the draft TPC-D queries

DPM over the network, the increase in the processing time is negligible. Even with a fairly skewed data distribution (Zipf(3)), BS retains the same scaleup. On the other hand, GRACE is degraded seriously with skewed data as the number of DPMs increases. Though the total relation size (about 20 × N MBytes) is less than the total main memory size (32 × N MBytes), the GRACE algorithm cannot prevent the larger buckets from concentrating on a particular DPM, causing bucket destaging on the DPM (in this case, DPM 0).

Figure 9 shows the relationship between data skew and processing times. Similar parameters as in the above experiment are used. The GRACE algorithm is very sensitive to data skew even with two DPMs. In contrast, BS can obtain the same performance even with data skew though some degradation is observed.

5.3 Evaluation of Bucket Overflow Solutions

In the previous tests, even the largest buckets of the Zipf(3) distribution can fit in memory. Here, we double the selectivity factor to cause bucket overflow. With 20% selectivity, the largest buckets become more than 33 MBytes, slightly larger than main memory. In this case, the number of overflowing buckets equals to the number of DPMs. The buckets which are generated through recursive partitioning are assumed to have the same distribution as the original bucket. After the first recursive partitioning, the largest buckets will be less than 28 MBytes, requiring no more partitioning on them.

Figure 10 shows the scaleup curve under bucket overflow. In the BS algorithm, since all the buckets are distributed equally among the DPMs, all DPMs can participate in partitioning their portions of the overflowed buckets. Thus, performance degradation is not so serious as the GRACE algorithm, in which most of the DPMs are idle while a certain DPM is obliged to process a heavy load alone.

Consequently, the BS algorithm can avoid bucket overflow in most cases, but even if it fails, it can also resolve the problem efficiently.

6 Further Experiments

Though the experiments in the previous section are based on rather simple queries to show the basic characteristics of the SDC-II system, realistic queries tend to be more complex. So, our current work includes processing of such queries. Here we present a brief explanation of the experiments with multi-join queries[14], which are extracted from the draft version of the TPC Benchmark^{TM}D [15]. We tested 3 out of the 17 queries, query 3, 5 and 9, which includes two joins, three joins, and four joins respectively. We compiled and scheduled these queries manually, to produce right-deep multi-join execution trees.

Figure 11 shows the scaleup curves for the three queries. The size of all relations is 100 MBytes for one DPM, and is proportional to the number of DPMs. The results show fairly good performance because the creation of intermediate relations are completely avoided due to the right-deep execution methods. However, we couldn't attain perfect flat scaleup curves for the queries 5 and 9. Because the number of tuples outgoing to/incoming from the network increases as the number of DPMs increases, the processor load gets higher when many DPMs are involved. Thus, these queries contain some processor bound portions. This fact implies that more attention must be paid for the processor bottleneck when processing complex queries. However, we omit the details here because of the page limit.

7 Conclusions

We presented the implementation of the SDC-II highly parallel SQL server and examined its performance on hash join operations with various data distributions. Synergy of sophisticated parallel join algorithm, dedicated parallel hardware, and specially tailored system software enables the SDC-II to attain high scalability as well as high performance even with skewed data.

Bucket overflow is a barrier for the hash based parallel join processing to guarantee the perfectness of its algorithm. The bucket spreading hash join algorithm is not only able to avoid bucket overflow in most cases, but even if it fails, it can also resolve the problem efficiently exploiting the data parallelism. The data driven process model of the SDC-II system software handles the flow of data with little overhead. In I/O bound application, computation time overlaps completely with I/O time, resulting in an almost ideal performance. The hardware of the test system was built from scratch utilizing commodity devices as much as possible except for the dedicated disk controllers and the interconnection networks. The disk controller can fully take advantage of prefetching and write buffering by determining when to start reading or start writing autonomously. The network has an ability to determine the destinations of input tuples adaptively based on the tuple distribution information, realizing a flat bucket distribution and conflict free routing.

The performance evaluation results convinced us that our approach is effective and promising. The modular architecture gives the ability to flexibly scale the system and the bucket spreading hash join algorithm allows an almost linear increase in performance.

Acknowledgement

The hardware implementation of the SDC-II was jointly performed with Fujitsu Limited. We would like to thank everyone who worked on the SDC-II. In particular, we would like to express our gratitude to Mr. Kaneda and Mr. Hirosue for helping to make the SCD-II a reality.

References

[1] C. Baru, G. Fecteau, et al., *DB2 parallel edition*, IBM Systems Journal, Vol.34, No.2, 1995.

[2] B. Bergsten, M. Couprie, and P. Valduriez, *Prototyping DBS3, a shared-memory parallel database system*, In proc. of PDIS, 1991.

[3] D. Bitton, D.J. DeWitt, and C. Turbyfill, *Benchmarking database systems - a systematic approach*, In proc. of VLDB, 1983.

[4] D.J. DeWitt, S. Ghandeharizadeh, D.A. Schneider, A. Bricker, H. Hsiao, and R. Rasmussen, ''The Gamma database machine project,'' *Transactions on Knowledge and Data Engineering*, 2(1):44--62, 1990.

[5] D.J. DeWitt and J. Gray, ''Parallel database systems: The future of high performance database systems,'' *Communication of the ACM*, 35(6):85--98, 1992.

[6] D.J. DeWitt, J.F. Naughton, D.A. Schneider, and S. Seshdari, *Practical skew handling in parallel joins*, In proc. of VLDB, pp. 27--40, 1992.

[7] G. Graefe, *Encapsulation of parallelism in the volcano query processing system*, In proc. of ACM SIGMOD, pp. 102--111, 1990.

[8] S. Hirano and M. Kitsuregawa, *A high performance parallel I/O model and its deadlock prevention/avoidance technique on the Super Database Computer (SDC)*, In proc. of IEEE HICCS, 1993.

[9] W. Hong and M. Stonebraker, *Optimization of parallel query execution plans in XPRS*, In proc. of PDIS, pp. 218--225, 1991.

[10] K.A. Hua and C. Lee, *Handling data skew in multiprocessor database computers using partition tuning*, In proc. of VLDB, pp. 525--535, 1991.

[11] M. Kitsuregawa, S. Hirano, M. Harada, M. Nakamura, and M. Takagi, *The Super Database Computer (SDC): Architecture, algorithm and preliminary evaluation*, In proc. of IEEE HICCS, pp. 308--319, 1992.

[12] M. Kitsuregawa, M. Nakayama, and M. Takagi, *The effect of bucket size tuning in the dynamic hybrid GRACE hash join method*, In proc. of VLDB, pp. 257--266, 1989.

[13] M. Kitsuregawa and Y. Ogawa, *Bucket Spreading Parallel Hash: A new parallel hash join method with robustness for data skew in Super Database Computer (SDC)*, In proc. of VLDB, pp. 210--221, 1990.

[14] D.A. Schneider and D.J. DeWitt, *Tradeoffs in processing complex join queries via hashing in multiprocessor database machines*, In Proc. of VLDB, pp. 469--480, 1990.

[15] TPC, ''TPC Benchmark^{TM} D (Decision Support), Working Draft 6.0'', Transaction Processing Performance Council, 1993.

[16] A.N. Wilschut, J. Flokstra, and P.M.G. Apers. *Parallelism in a main-memory DBMS: The performance of PRISMA/DB*, In proc. of VLDB, pp. 521--532, 1992.

[17] J.L. Wolf, D.M. Dias, P.S. Yu, and J. Turek. *An effective algorithm for parallelizing hash joins in the presence of data skew*, In proc. of DE, pp. 200--209, 1991.