Distributed and Parallel Databases, 7, 383–414 (1999)

Intensive Data Management in Parallel Systems: A Survey

M.F. KHAN
School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN 47907

RAY PAUL
Office of the Under Secretary of Defense, Washington, DC 20301

ISHFAQ AHMED
The Hong Kong University of Science & Technology, Clear Water Bay, Kowloon, Hong Kong

ARIF GHAFOOR
School of Electrical Engineering, Purdue University, West Lafayette, IN 47907

Recommended by: P. Valduriez

Abstract. In this paper we identify and discuss issues that are relevant to the design and usage of databases handling massive amounts of data in parallel environments. The issues that are tackled include the placement of the data in the memory, file systems, concurrent access to data, effects on query processing, and the implications of specific machine architectures. Since not all parameters are tractable in rigorous analysis, results of performance and bench-marking studies are highlighted for several systems.

1. Introduction

Current real-world applications demand database size and processing capabilities beyond the capacity of the largest and fastest transaction processing systems available today. Some leading applications include: handling of scientific data from satellites and space missions; processing the data for the human genome project; handling databases for national administration e.g. Social Security; handling of multimedia data; utilization of multidatabases spanning across corporations or other organizations; and collecting data and performing simulations for studying changes in the global climate.

The advent of affordable parallel computers has provided a hope for handling some of the problems generated by these and other potentially massive databases. Despite the existence of the hardware, however, it is far from clear how the power of this breed of parallel machines can be harnessed to solve the problems at hand. To date, most of the work on parallel machines has focussed on producing efficient algorithms for the solution of computationally intensive problems. Very little work has coupled this emphasis on solution of computationally challenging problems with the concomitant, and sometimes contradictory, demands placed by data-intensive applications, such as those found in massive databases.

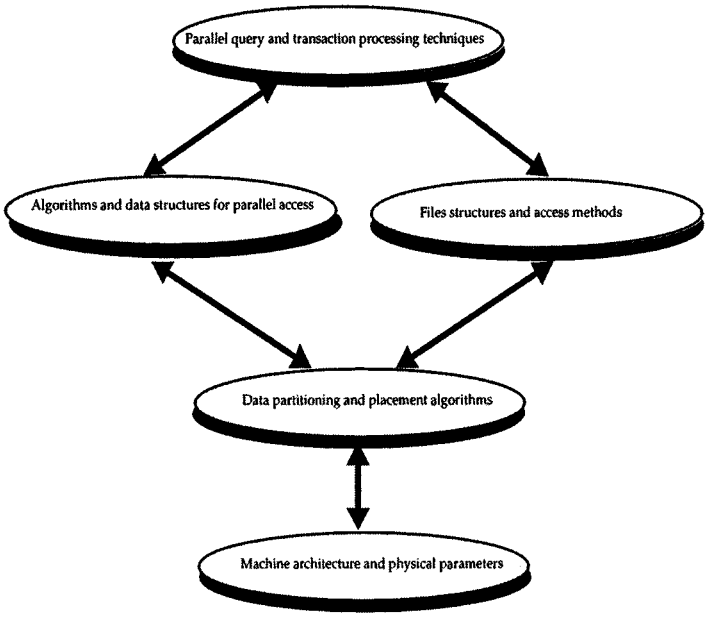

Figure 1. Layered semi-dependencies of components of a parallel database system.

In order to tackle the above problems we need a careful evaluation of the following factors, among others: hardware architecture, file structures and data layout schemes, data structures, and transaction processing models. Data layout refers to the manner in which data is distributed in the entire system, including disks, cache memory and processor memories. Data placement also concerns itself with the anticipated needs of access to different data objects, and the resulting effects on the communication channels.

Figure 1 lumps together relevant issues and shows a layered organization of parallel database systems. The organization of the current paper loosely follows the given layered organization.

Each of the higher levels in the layered organization depends on all the levels below it. The lowest level dealing with machine architecture issues determines what sort of file structures and file access methods are possible on a given processor interconnection, memory, and communication system. File organization influences the kind of data structures that can be built for manipulating data in these files. Algorithms are entwined with the data structures implementable on the underlying machine. Algorithm A may perform worse than algorithm B for given architecture, but algorithm B may give better results than algorithm A on other machine architectures. Other factors may influence the choice of data structures include the size of data set, reliability requirements, cost etc. In general, designers assume some kind of lower level layers when designing higher level layers, such as algorithms for concurrency control in database transaction processing systems. An example of such dependence is the design of main-memory systems, which allow the optimization of data-structures and algorithms to take advantage of the available main-memory and the lack of disk accesses.

The layered organization is not very rigid: often a machine may have some algorithms built in the hardware. An example of this is the parallel hardware sorter built in a database system [17].

Following the layered organization of Figure 1, rest of the paper is arranged as follows. We discuss machine level issues and architectural influences in the design of databases in Section 2. This includes discussion of main memory databases, as well as two orthogonal classifications of database architectures. Shared-everything etc. classification is communication based, whereas we look at data-flow issues in the section on processor-memory organization. Data placement issues make up Section 3. This includes discussion of partitioning of data for efficient parallel access. Section 5 discusses different ways of organizing files, since most of the large databases today are organized as files, techniques of this section can potentially produce large performance increases with little change in the underlying hardware. Section 6 summarizes different kinds of access methods for data segments smaller than files. Different kinds of search structures for concurrent access are also treated in this section. Section 7 looks at issues and algorithms for parallel query processing. It builds on top of the organization and access issues treated in earlier sections. Finally, the paper concludes by sampling a number of research and industry parallel database systems, and their salient features and innovations in this area.

2. Architectural issues in parallel databases

Von Neumann model of computation has severe limitations since it assumes a uniprocessor environment and has a bottleneck for accessing the memory. This has motivated the development of specific architectures for parallel databases systems.

The architecture of the parallel computer is critical in the selection of data layout schemes and the kind of algorithms that can be employed to solve database problems. In particular, we are concerned about the issues of data placement and the performance of parallel algorithms available with the given data structures.

A number of considerations prevail in the construction of parallel architectures for database processing use. These are described in the following.

A choice between *general architectures* and *special purpose architectures* has to be made. When an architectures is tailored to solve database related-problems, the systems are also known as *database machines*. Several database machines have been developed in order to handle massive amount of data or to have a high transaction processing rates. General purpose architectures refer to machines which are not specifically designed for any particular target application, but rather applications are built on top of them, via software and possibly some add-on hardware. In order to analyze and build parallel database systems, models can be developed which capture the element of parallelism present in these systems. There can be various types of parallelization in the database environments, such as inter-query, intra-query and intra-operation parallelization.

Granularity, which is a measure of the amount of computation in a software process, is another major design parameter for parallel machines. When the processing nodes are small in number, typically of the order of ten, with large word sizes and large memories, the architecture is called *coarse-grained*. Machines with several thousand processors, each with

small word size, and small memory, are termed as *fine-grained*. Architectures in-between are usually referred to as *medium-grained*.

Scalability of a parallel system determines its performance when the number of processors used for a given application is increased, or for a fixed number of processors, the problem size is increased. For very large database applications, scalability measures are an important factor for determining the suitability of an architecture.

Interconnection topology is another design issue in parallel architectures. In order for nodes to be able to work together on the same problem, communication is needed between them. Interconnection refers to the way different classes of computer modules are organized as a communication network. The modules can be processing node, memory modules, and I/O processors. Networks can be static or dynamic. Static networks use point-to-point direct connections which remain fixed during the program execution. Examples of static interconnections include *linear arrays*, *rings*, *trees*, *meshes*, and *hypercubes*. For a detailed treatment of how different topologies influence the design of parallel algorithms for those topologies, see [57]. Static networks are usually employed in distributed-memory message passing multicomputers while dynamic networks are usually used in shared-memory multiprocessors. In database applications, it is important to analyze the patterns of communication and then choose an interconnection network accordingly.

Memory Sharing refers to the way the programmer views the memory of the system. A shared memory system provides a single address space and a single coherent memory. A nonshared memory system, which employs distributed memory, can be viewed as a collection of processor memory pairs where communication is done using explicit message-passing. Shared-memory system can employ either centralized memory or a distributed memory. Examples of distributed shared memory include the DASH and KSR systems [9] and Cray T3D. Since the access to the memory (read/write) can be the major factor affecting the performance of a parallel database, this is an important consideration for database applications.

Control mechanisms are important architectural components, because they determine the timing and synchronization of execution of instructions of the programs.

Systems with implicit synchronization of instructions for the execution of operations are known as *SIMD* (*Single Instruction Multiple Data*). SIMD systems provide a single control for all the processors which execute the same instruction with different data. Examples of SIMD computers include CM-2, MasPar MP-1 nd MP-2, and DAP610.

Parallel processors that can execute different instructions at the same time are called *MIMD* (*Multiple instructions Multiple Data*). MIMD computers lack a global control unit, with each processor-memory pair executing instructions independently of each other. Synchronization is done explicitly by the programmer by using message-passing or semaphores, etc. Examples of MIMD computers include Thinking Machines CM-5, Intel Paragon, KSR-1, *n*Cube, etc. Parallel database systems have been developed on both SIMD and MIMD architectures [35].

Mapping of tasks and data within a machine greatly impacts the performance of the machine [1, 2, 15]. Mapping deals with the distribution of different tasks generated by an algorithm among the nodes of the system.

I/O Capabilities is another crucial factor for parallel processors. Current disks are very slow as compared to the CPU and the main memory. One proposed solution is to keep

the entire database in the main memory [58]. However, this is a very expensive solution and precludes its use for all but the smallest databases. Also the trend has been that CPU and main memory bandwidth has been increasing much faster than disk memory banwidth. Another solution is parallelization of I/O has been proposed as the solution to this problem [49, 70]. This can be achieved using disk arrays where the data is not placed on just a few high density disks. Rather, it is distributed over a large number of disks. This allows parallel access to different segments of data, thus increasing the effective bandwidth of I/O [101]. In order to alleviate the burden of I/O from the main processors, the use of special I/O processors is a common practice in parallel processors. For example, the Intel Paragon [46], which is an MIMD machine based on a 2D mesh topology, has arrays of I/O processors on the right and left edges of the mesh. Similarly, the Thinking Machines CM-5 [88] which is also an MIMD machine provides dedicated I/O processor. The number of I/O processors can be scaled with increasing number of processing nodes.

2.1. Main memory databases

As noted above, high cost of stable main memory[1] restricts memory resident systems to relatively small databases. Research is being undertaken in this area despite this fact, because the main memory bandwidths are increasing rapidly, as is the cost coming down [27, 53]. This trend promises wider application of main memory systems in the future. The high speed of data access in main memory databases allows some time-critical or real-time DBMS applications to run, which would not be possible on other architectures. A main memory transaction processing system require far fewer I/O operations than disk-based systems. The reason for this is that the only I/O performed in main memory transaction processing systems is to maintain the *durability* of the transactions. Whatever I/O operations remain to be performed show an order of magnitude increase in speed. This also decreases the context switching between different transaction threads running in parallel. As a result, cache flushes are reduced, producing a further increase in performance. Experiments establishing these results are given in [75].

2.2. Shared-everything architectures

In shared-everything architectures, all processors in the system have direct access to the main memory as well as all the disks. Thus the memory and the disks can be regarded as global or having single address space for all the processors in the machine.

The communication between the processors of a shared-nothing architecture may be through some fast interconnection network or through a single or multiple buses. The interconnection network or the bus can have considerable contention since all reads and writes to data objects must pass through them. This can cause a bottleneck for the I/O and inter-process communication in the system [69]. For this reason shared-everything systems tend to have a relatively small number of powerful processing nodes, in contrast with the *shared-nothing* systems (discussed below), which can have tens of thousands of nodes. The small number of nodes help control the proliferation of messages to manageable levels, thus avoiding clogging the communications channel. This is one reason why shared-nothing memory architectures tend to be favored in the recent systems, although [10] demonstrates efficient database

management systems using shared-everything architectures. In this study, the authors investigate the issues involved in using multiprocessors for transaction processing. A simulation model to study the behavior of shared-everything and shared-nothing is developed to study the effects of data contention and resource contention in both of these architectures. The effects of intra-query parallelism on both types of architectures under different operating conditions are quantified. The results of this simulation study show that shared-everything designs are able to provide efficient database systems under a variety of conditions.

The main benefit of shared-everything architectures is that programming them has many features that are similar to programming conventional, extant systems. This is because both shared-everything, and conventional uniprocessor systems usually employ a single data directory,[2] and a single global lock table for synchronization of data accesses. As a consequence of this, adapting existing database systems to the shared-everything architectures is usually simpler than other architectures e.g. non-shared memory [105]. This is a pragmatic advantage of shared-everything architectures.

The main problem in a shared-everything system with a centralized memory is that there can be contention on the memory. This can be overcome by providing memory access pipelining. If the shared memory is distributed, then a cache coherent mechanism must be provided for a consistent memory state.

Another persistent problem with shared-everything architectures is that they are difficult to scale up. Interference among the processors makes it difficult to increase the number of processors in such systems much beyond 32. With all the memory resources shared, it is required to have the bandwidth of the interconnection network greater than combined bandwidth of all the processors in the system. As the number of processors in the system increases, having such bandwidth of the interconnection network becomes more and more difficult, because of current technological constraints. One approach to address this problem is to have large private caches associated with the processors. It is shown in [87], however, that handling of these caches, which involves operations such as loading and flushing these caches, degrades the system performance considerably.

2.3. Shared-nothing architectures

In shared-nothing memory architectures, each processor has its main memory, and possibly a cache memory, as shown in figure 2. In addition, each processor may have its own disk. Thus each node may contain a local database. The nodes have direct access to a fast interconnection network for communication with other nodes. Each processor acts as server for the data contained in the local memory and disks attached to the processor. Many of the recent parallel databases use shared-nothing architectures. These include Bubba [18], Teradata, GAMMA [24], Tandem Nonstop SQL [86], and MEDUSA [102].

Advantages of shared-nothing architectures include exploitation of parallelism, thus increasing throughput.[3] Since the data in memories is local rather than global, interference among the processors is also minimal. The interconnection network is loaded less because only queries and their results are transmitted across it. On the other hand, shared-everything architectures necessitate moving of entire relations and data dictionaries in order to process queries. Since congestion on the interconnection network is low in shared-nothing systems,

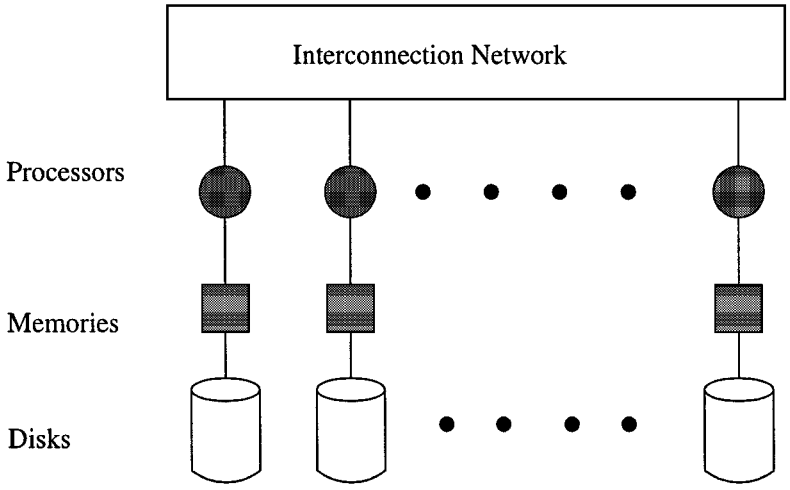

Figure 2. A schematic representation of shared-nothing architectures.

they can be scaled up more easily. Indeed, commercial systems have been developed with thousands of processors and having the shared-nothing architecture underlying the system. Such systems achieve near-linear speedup and can scaleup on a variety of transaction loads and operating conditions [27].

Fragmentation of data across the nodes allows maximal parallelism when a complex query is broken down into smaller queries. These smaller queries can be handled over the entire system, using appropriate parallel algorithms, as shown in studies such as [86].

The other advantage of these architectures is scalability [86]. Since the nodes don't have strong coupling among them, scaling up the system by adding more nodes is like adding more modules. Another factor which helps the scaling of shared-nothing architectures is the uniformity of such systems. Scaling and extensibility also imply that a large range of sizes of databases can be handled by the same hardware system with very easy modifications, if at all. Such systems are quite suitable for handling massive amounts of data, including multimedia data.

The main problem in a shared-nothing system is communication overhead which is mainly due to message latencies. This problem is solved using special purpose processors called hardware routers. The routers can not only reduce the message latencies but also relieve the main processor from performing message-passing.

2.4. Shared-disk architectures

In shared-disk architectures, each node has direct access to all the disks [65]. A shared-disk architecture is depicted in figure 3. In addition, each node also has a private memory bank of its own. Often, shared-disk architectures are treated as a variant of shared-all architectures. Harder et al. present the advantages for using shared disk architectures. It is noted that

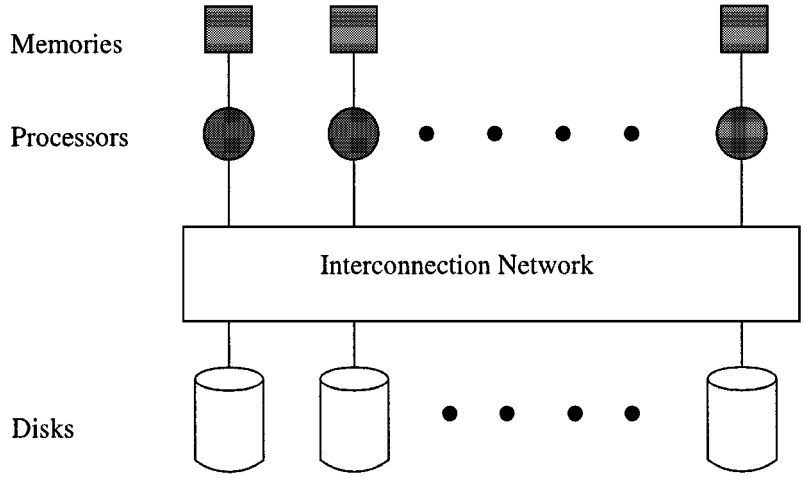

Figure 3. A schematic representation of shared-disk architectures.

in dealing with complex and large objects, partitioning of objects has negative effects on performance. The management of data and load-balancing problems are simpler to manage, because there are fewer data clusters. A common memory is used for buffer management, synchronization, and logging/recovery.

In practice, parallel machines also come in varieties where the nodes may be non-uniform. Later in the paper, we look at one such architecture in which a few dedicated I/O nodes have disks attached to them. Each I/O node controls a channel to some disks. The remaining nodes consist of just memory-processor pairs.

3. Data placement at the physical level

The distribution of data in memory is of concern in databases with large processing and data requirements. When the distribution of data over the disks or processors in not "even", it is said to have *data skew*. Significant amount of data skew in a system can overload certain processors, and reduce overall performance. One of the techniques used to control data skew in a system is the fragmentation of data. Fragmentation refers to the partitioning of the database into small blocks known as fragments, and spreading these out for efficient access.

To get maximal concurrency, data should be divided across many nodes of the parallel system. An incident advantage of such fragmentation of data is load-balancing of the system, i.e. the some nodes are not forced to become computational bottlenecks. This fragmentation of can be horizontal across relations, or vertical across attributes. Both types can be used to increase parallelism and load-balancing in the database. Horizontal fragmentation is very often referred to as *declustering*.

Horizontal fragmentation refers to the partitioning of a relation along its tuples. Thus, each fragment includes a subset of the tuples in the relation. In *vertical fragmentation*,

partitioning is carried out along the attributes of a relation. Each fragment contains a subset of the attributes of the relation, and the primary key of the relation.

3.1. Clustered vs. declustered data

To avoid system thrashing, data needs to be placed in such a manner that processing associated with the data is done, as much as possible, in the vicinity of the data. However, inter-query parallelization involves communication overheads, and a balance between the two has to be found for optimal performance [103].

Some parallel database systems apply a hash function to the relations and distribute the horizontal fragments across all the disks in a uniform manner. This is known as full declustering, and is used in GAMMA and NonStop SQL, among others. In such a system, exact match queries can be handled by a single disk node, and all other queries can proceed in parallel. Performance studies have been done for comparing fully declustered systems with systems where each relation is clustered on a single disk [61, 86]. The results indicate that declustering performs better under a wide range of condition. Another interesting result shows linear increase in performance in such systems, with the increase of the number nodes, up to 32 nodes. Full studies have yet to performed on much larger systems.

On the other hand, full declustering may cause problems in case of join operations. For example, it has been calculated that a binary join for a 1K-node machine could produce about 10K messages! The study in [86] also shows poor results for fully declustered systems in case of complex queries with joins. As a result, strategies with variable level of clustering are being sought.

In [21] one such scheme is presented, where the number of nodes across which a relation is declustered is determined by such parameters as the size of the relation and the expected frequency of access. The scheme is also dynamic, since the data may be reorganized as the contents and access patterns change.

Reduction of data-skew by application of hierarchical hashing schemes is proposed in [98]. The algorithm adds an extra scheduling phase to the previously proposed algorithms using hash and join phases. Hash partitions with largest skew elements are identified and split. After that each split portion is assigned to an optimal number of processors. This study shows that the performance of this algorithm is better than conventional algorithms, and performs better even when the amount of data skew is low.

3.2. Round-robin distribution of data

This is one of the simplest methods of data placement. Successive tuples in the database are placed on successive disks or fragments. The performance is excellent for the class of queries which necessitate sequential scanning of all the tuples in the relation. Queries which need associate access to data tend to run inefficiently on data distributed in round-robin fashion. One example of a query needing associative access is to retrieve all tuples with a given value for a certain field. Since such queries are very common in database usage, round-robin partitioning of data exacts a price in terms of performance.

3.3. Value-range partitioning of data

Clustering of tuples according to values in some fields can be useful in certain applications. For example, related tuples can be pre-fetched into cache memory in the expectation of their use in the near future. Thus bibliographic databases would do well by storing abstract information on the same pages as those of author information etc.

Value-range partitioning is a method of achieving this physical proximity of related data. Data with similar attributes is placed on the same disks or fragments. For example in an electronic phone book similar or identical last names are placed close to each other in physical memory. This can increase the efficiency of many kinds of database queries by pre-fetching of data, and in some cases obviates the need to search the entire database for certain kinds of tuples.

3.4. Data management in main memory

All the above schemes can be utilized in data placement in main-memory databases. However, there are few or no disk accesses, and the data accesses are several orders of magnitude faster than disk-based systems, and improvements in performance tend to be small. Often, there are inherent restrictions on the manner in which data can be placed physically in the main-memory. For this reason, most main-memory systems tend to be content with the simplest possible scheme for data placement.

4. Data organizations for parallel databases

Data structures which have been proposed for and employed in efficient access of massive data in parallel databases include linked lists, graph structures, Fibonacci heaps, and hash tables. The structures given below also depend on the efficiency of file access at some level. In a subsequent section, we also look at the effect of file structures upon the efficiency of data access. We look at some modifications of the simple file systems to increase efficiency of access.

Often a particular system will employ more than one of the above. For example, the nodes of a B-tree may point to text files on the disk. Hence a few attributes of the file which are used extensively in selection, sorting and processing of data are kept in the nodes of an efficient search structure (a B-tree), and actual data content is kept in plain, sequential files. Factors affecting the choice of the mix of data structures include the size of data sets, the difficulty of implementation, the effects of efficiency of the employed data structures in the overall system etc. We discuss some of these factors later in the paper, in the context of parallel machine architectures.

4.1. Concurrent operations on linked lists

Simplest of these are arrays of values and linked lists. Sorted arrays have $O(\log n)$ time for searching, deleting as well as inserting elements. Linked lists exhibit $O(1)$ insert time, and

$O(n)$ time for search and delete. In particular, we discuss *skip lists* [71], which generalize the idea of lists, including algorithms for concurrent access.

A simple linked list requires examining, in the mean case, half of its nodes when searching for an element. Skip lists improve on this by providing a mean case performance similar to that of binary search in sorted arrays.

If a sorted list has pointers to every other element in the list, one need examine only $\lceil n/2 \rceil + 1$ elements. Having pointers for every 2^ith node to 2^i nodes ahead implies only $\lceil \log n \rceil$ comparisons, and a double number of pointers. We get fast searching behavior, but insertions and updates are a problem.

Level of a node is the number of forward pointers in it. Skip lists assign each node a random level. A nodes ith forward pointer used to point to 2^{i-1} nodes ahead; now it is made to point to next node of level i or higher. This gives $\log n$ expected case performance, although the worst case performance is still bad i.e. $O(n)$. However, worst case performance happens very rarely, and no input sequence consistently produces the worst case. This behavior is similar to the thoroughly studied performance of quicksort by [79].

Advantages of using skip lists over B-trees include the following. Random balancing is easier than explicit balancing in B-trees. Skip lists are more space-efficient, requiring about 1.3 pointers per node in the average case. Skip lists require to keep less book-keeping information in each node. Finally, algorithms for search, delete, update etc. in skip lists are simpler than analogous algorithms for B-trees.

Disadvantages of skip lists as compared to B-trees include their relative infancy. B-trees, on the other hand, are thoroughly studied, tested and mature data structures. Hence there is an advantage in using widely available prefabricated software libraries built for B-trees. Skip lists have yet to gain general acceptance.

We study concurrent operations on linked lists in order to develop a concurrent framework for skip lists. Concurrent algorithms for searching have been written for unbalanced and balanced trees. The usual mechanism to enforce non-interference between different threads of execution has been the use of locks. Concurrent algorithms for rebalancing or deletion from B-trees are generally very complicated.

Concurrent updating of sorted linked lists can be performed in the following manner. The factors that must be met in any of the concurrency algorithms are the following: integrity of data structures; deadlock avoidance; and serializable schedules.

The simulations done with implementations of skip lists show almost linear speedup with the increase in the number of concurrent threads of execution. The number of locks blocked is proportional to the number of locks held, which is proportional to the ratio of concurrent writers to the elements in the data structure.

With these results, it seems that skip lists provide efficient concurrent algorithms for use in parallel databases. The algorithms tend to be simpler than the corresponding ones using B-trees.

4.2. Graph structures for parallel access

Trees are rooted graphs, and include B-trees, 2–3 trees and other balanced tree schemes. With suitable balancing techniques, we can have $\Theta(\log n)$ for search, insert and delete.

Extensive work on concurrent access to trees is reported in [7, 8, 32, 33, 36, 54–56, 74, 77].

It is desirable that independent processes access nodes of trees in a manner such that the data in a database, i.e. the contents of the tree nodes, remain in a consistent state throughout. Concurrent access algorithms for B-trees are given in [8, 55, 56, 77]; for AVL trees in [31]; for 2–3 trees in [32]; for binary search trees in [41, 54]. The different algorithms given cover a wide range of underlying processor and communication architectures, use different methods for actual implementation of the tree data structure, and use different definitions of concurrency control and consistency. They also often use different assumptions about the placement of data and type and distribution of workload [104]. Hence the performance figures derived analytically, or by simulation are not comparable for all of them. We provide some salient features of a few tree algorithms for concurrent access of data.

The primary operations of interest implemented by the algorithms are *search* and *insert*. Three algorithms are given in [8] for concurrent B-tree access. The approaches can be classified as conservative, optimistic, and a mix of both. In the conservative approach, nodes of the tree are allowed two kinds of locks: the read-lock and the exclusive-lock. A given node can obtain locks which are *compatible* with the locks that it obtained previously. Incompatible locks are delayed until the node in question releases the previous, conflicting locks. Read-locks do not conflict with each other, but exclusive locks conflict with read-locks, and other exclusive-locks. Hence reads can occur concurrently, but inserts exclude all other operations at the same node.

In the optimistic algorithm of [8], insert operations obtain *read-locks* on all of the internal nodes of the tree which lie on the path for the ultimate insert leaf-node, and a single exclusive-lock on the leaf-node where the data is to be inserted. The additional problem that this poses is that splits may occur on internal nodes, and they should be handled by special detection and recovery mechanisms. If a situation is detected where the leaf nodes are full, necessitating a split of the parent node to perform the insert, then the current process may cause a deadlock since a split also needs an exclusive-lock on the node. The solution proposed is to abandon the current process and re-insert, using the strictly conservative algorithm.

In the mixed algorithm, a third kind of lock, the *write-lock* is also used. The write-lock is compatible with the read-locks, but not with exclusive-locks. Furthermore, a write-lock may be converted into a exclusive-lock on request. Using write-locks on the chain of descent of an inserting process allows other inserting processes to descend simultaneously on the same chain.

Sibling tries have been proposed as a concurrent dynamic search structure [68]. Sibling tries support the regular operations on data, such as search, update, delete, and insert. Their advantage over B-trees is that the searches can begin at *any* node in the tree, and not just at the root node, as is the case in B-trees. Hence it may be very useful in highly concurrent shared memory systems, where the data, if organized as a tree, causes bottlenecks at the root node, and the nodes near the root. A further advantage is that many alternate routes are provided for each data point. The storage of the structure is proportional to the number of data items in the database, and independent of the number of processes that access the database concurrently.

Although theoretical bounds for such operations as search, insert etc. of nodes are better than tree algorithms, *Fibonacci heaps* are still mainly of theoretical interest, and very few

studies have considered the problems of concurrent access to Fibonacci heap nodes. *Relaxed heaps* [29] may be a better alternative to Fibonacci heaps in the context of parallel accesses to data.

4.3. Parallel hashing

Hashing is widely used in the organization of data, and often presents the most efficient solutions in practice for the storage and access of data. Hash tables practically provide $O(1)$ search and insert times. Worst case times can be as bad as $O(n)$, but this happens rarely in practice.

Since the data in parallel systems is mapped over many processor/memory pairs, an efficient scheme of distribution of data over the nodes of the entire system is required. Another criterion for efficient access is that the structure should allow parallel accesses to data while maintaining consistency. A table or array structure are very suitable for *multi-entry* data structures, since there are multiple points in the data structure through which access is guaranteed, rather than a single point. Examples of single-entry data structures are different kinds of trees for concurrent access, where all searches must start at the root node of the tree. This means that the root is *single-entry* for this particular search tree. A table where all the nodes are directly accessible is known as a *complete multi-entry* structure. As an example, a hash table which resolves collisions by chaining them to the same bucket is not a complete multi-entry structure, but a multi-entry structure. This is so because all the nodes on the hash chains after the header node are not directly accessible. A hash table with linear probing provides a complete multi-entry data structure but the number of entries in the table is limited by the table size. This highlights the fact that we can obtain some gain in the level of concurrency by imposing a limit on the number of entries that are handled. In parallel systems, it is highly desirable to implement complete multi-entry data structures in order to avoid potential bottlenecks at highly used entry points for data access.

In the implementation of hash tables, it is required to implement the operations of inserting, deleting, and searching keys. Usually the hash table is stored in a manner that makes the distribution of data even over the entire system. If possible, each data element is associated with a single processor. Linear probing is used for resolving collisions, since this reduces communication costs in the system. The performance of the hash table should take into account the variance in the type of load that a system may be subjected to, as well as the communication overheads. These factors make the performance analysis of parallel hash systems much harder than conventional systems. Simple analysis of parallel hashed data systems is provided in [99]. The simulation study in [99] shows that performance of a hash table with linear probing when the hash table is fully loaded is much worse than the performance of the hash table with 80% load when the only operation considered is *insert*. *Search* also has performance similar to insert. They also perform analysis and simulation of performance of a sorted array, and conclude that the average time complexity of three major operations *insert*, *delete*, and *search*, all favor the sorted array, as well as the actual results of the simulations. On the other hand, hash tables with load factors less than 80% tend to perform better than sorted arrays, both in complexity analysis and in the simulation studies.

5. File manipulation strategies

It is desirable to provide fast access and high bandwidth between the data stored on disks and the main memories associated with the processors. Files can be allocated as either *fixed blocks*, where all blocks are of the same size (e.g. UNIX), or *extent based systems*, where allocation of data is as a few large and variable chunks of disk space. Fixed block systems have the disadvantage of discontiguous allocation of data on disk, and an excessive amount of book-keeping data. Extent based systems may thus provide higher performance. Performance studies which indicate the superiority of certain allocation policies are reported in [80]. In particular, striping across disks with contiguous allocation showed 250% improvement over policies without disk striping and contiguous allocation.

5.1. Associative access to data

If the nature of data is non-static, as is the case when large number of updates are performed, or the nature of the system is unknown, provision should be made for even distribution of data (load-balancing) over the entire system via data reorganization. This load-balancing should not entail recompiling of the programs running on the system. This can be done by having associative access to the data in question.

A global index is kept indicating the placement of relations on the nodes. The index structure can be either B-tree based, or hash based. B-trees take more space, but range queries are more efficient. The global index is replicated on each node, so this may cause problems in scaling, because of consequent overhead [48].

As an example, we discuss a specific application platform for the use of associative access, namely the Connection Machine's CM-2. A variety of very large database applications have been attempted on the CM-2, which gives us an idea about massively parallel architectures. In general, there are no specialized I/O nodes. There are a large number of processor nodes (e.g. 2^{16} nodes). Data is stored in a distributed manner over the processors and thus can be operated upon in parallel at each node. Document retrieval, parsing and searching large text databases, and using the database as an associative memory are among the applications that have been studied in [93].

Algorithms are given where a single CM-2 is able to support 2000 simultaneous users doing searching and browsing 6 GB of free text database. Authors give empirical calculations establishing this result, without any performance studies. This reduces the reliability of the deduced figures, but the figures may still provide order-of-magnitude estimates of actual performance figures. A key improvement that is intended after this system is to add high speed multiple disk mass storage units to the system.

5.2. Partitioned signature files

Signature files are useful for associative retrieval on formatted on unformatted data files [34]. The major advantages of signature files over some other structures such as grid files or multi-dimensional hash structures are that the associative searches may be conducted over a

large number of dimensions and this number may even vary for different records within the same file. Auxiliary files, called *signature files*, contain database record abstractions called *signatures*. Extensions to signature files in order to increase the performance of signature file query processing are provided in [40]. Each data object can be considered to consist of many *object descriptors*. Object descriptors are partitions of the object. A word signature is obtained by hashing an object descriptor into a fixed-length bit vector. Then the signature for the object S is obtained by superimposing the signatures for the all the object descriptors for this object. To search a word in an object, a signature W of the word is obtained. Then the only objects that *may* have this word satisfy the following property:

$$\{S_i \vee (S_i \wedge W) = W\}$$

Partitioned signature files promise to be able to handle large amounts of data, be supported by parallel computer architectures, have moderate immunity from data skew, offer fault-tolerance to a limited degree, and support intra-query parallelism.

5.3. Clustered surrogate files

Clustered surrogate files [20] are used as an indexing scheme through a special data word, called the concatenated code word, or *CCW* for short. These CCW's constitute a *surrogate file*, which is small in size and simple to maintain through a small number of core operations. Considerable savings of time may be realized by performing related operations on the CCW surrogate files *before* performing them on the actual data files, which are often very large. Since the structure of surrogate files is compact and regular, mapping these files to different parallel architectures is not very complex. Further improvements in the CCW surrogate files may be obtained by *clustering* together of different CCW surrogate files. Then, a searching a subset of a surrogate file is often sufficient for a relational operation. [20] also develops parallel relational operations on clustered CCW surrogate files. These operations are useful in efficient organization and access to data in parallel database systems.

6. Parallel query processing

Parallel query processing is an essential part of constructing any parallel database system, and can account for important performance improvements. In this section we look at work in parallel algorithms for database query processing. Parallelization of query processing provides opportunities for *inter-query* parallelism, *intra-query* parallelism, as well as *intra-operation* parallelism.

In *inter-query* parallelism, different queries are executed in parallel on different processors. *Intra-query* parallelism involves the parallel execution of different sub-operations within the same query. *Intra-operation* parallelism refers to the even more fine-grained parallelism, where single operations within queries are distributed over more than one processor for concurrent execution.

We assume that the operations to be performed on partitioned data in a parallel database consist of the basic relational algebra operators, or their derivatives. These include selection, projection, union, set difference, cartesian product, intersection and various kinds of join operations performed on the database relations. Researchers have generally concentrated on select and join operations, since these are basic and heavily used primitives in database query processing.

Divide and conquer strategy is applied to break up operations with a large number of tuples into smaller chunks, assigning these chunks to different processors, and processing the chunks in parallel. Finally, some processing may be needed to combine the sub-results at different nodes to obtain the desired answer. The optimal breakup of larger tasks and the assignment of sub-tasks to different nodes depends on the nature of the query, the actual placement of data, and such factors as the communication overheads, limits of processor memories, other pending processing in the database etc. Thus, for example, computation on data is performed at or near the nodes which may hold the object data. This wisdom, borrowed from experience in distributed databases, is appropriate where the amounts of data are very large, and moving them around would incur undesirable overheads.

Nodes may be thought of as having streams of inputs, a processing engine for the incident streams of data, and resultant outgoing stream or streams of processed data and answers. For example if a query on relation A requires all tuples with an attribute satisfying a property, the relation A could be processed at two nodes, and the results combined. This may reduce the processing time by almost half in some cases, as compared with single node operation.

The above example also highlights the need for efficient *merge* operators in a parallel environment, so sub-results may be efficiently combined. Other such operators include sort and scan.

6.1. Parallel joins

Join is derived from the cartesian product of two or more relations. The most general form of join is called the θ*-join*. Given relations A and B, their join is represented as

$$R \bowtie_F B$$

where F is a formula known as the *join predicate*. Study of joins in relational databases is important, since joins are often the most expensive portions of most query processing involving computation. Hence performance gains here due to better algorithms and data handling techniques will ultimately reflect in the overall performance of the transaction processing system. It is for this reason that considerable research is being undertaken in an effort to improve the performance of parallel joins.

Conventionally, joins between relations A and B are performed by sorting A and B on the join attribute, and then merging the sorted relations to identify the target tuples. This procedure allows for many opportunities for parallelization.

In parallel systems, the sorting phase may be distributed over nodes such that there is relatively low amount of data skew and computational load is also spread over the nodes. Without such load-balancing, the speedup achieved is limited due to under-utilization of the resources, and extra overheads, such as intermediate disk saves etc.

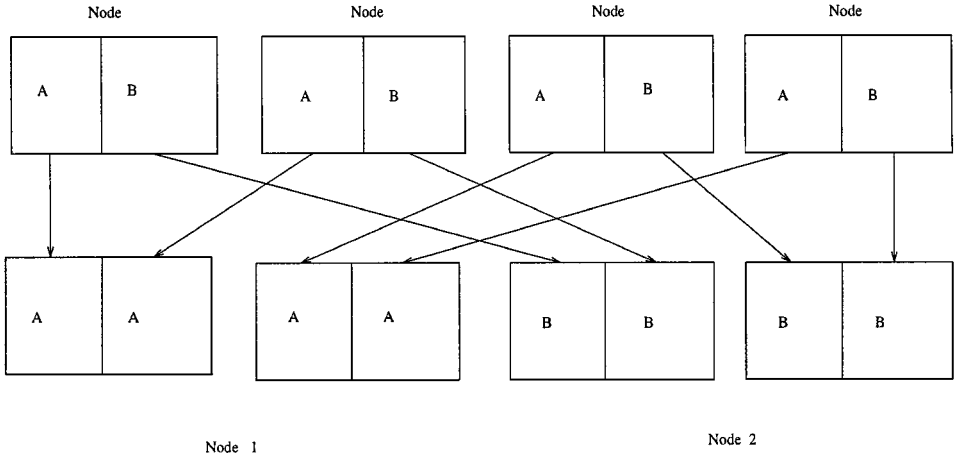

Figure 4. Example of parallel hash join.

An algorithm for more general kinds of than an equijoin has been proposed in [11]. To perform a join of relations A and B, their cartesian product is formed in parallel. All clusters of B are sent to all the nodes having clusters of A, joins are performed at all the A-nodes. The output from all the A-nodes is combined to give the final result. Since the entire relation B is copied at each of the A-nodes, we desire to have it as the smaller of the two. The actual performance also depends on the kind of communication network, and the algorithms employed at each node to perform the joins.

Another approach to performing the joins is the *hash-join*. In the join of relations A and B, the following steps are involved: (i) hash-partitioning A and B on the join attribute; (ii) performing join operations on each of pairs of partitions of A and B; and finally (iii) combining of the results. Figure 4 shows an example of two relations being joined in parallel. Compared with the sort-merge algorithm, it is required to partition both relations A and B. Also, distribution over nodes may not be good when the hash-partitions are very uneven. Variations of hash-partition algorithm are given in [25]. Studies such as [78] have shown that, overall, hash-join algorithms perform better than other join algorithms.

Algorithm for performing hash join for relations R and S is given below [67]. Relations R and S are fragmented into $R_1, \ldots, R_m$ and $S_1, \ldots, S_n$. Result fragments are denoted $T_1, \ldots, T_n$.

begin
  for i **in** $(1..m)$ **do**
    R_{ij} = apply hash function to R_i, j **in** $(1..n)$
    for j **in** $(1..n)$ **do**
      send R_{ij} to node storing S_j
    endfor
  endfor

for j **in** $(1..n)$ **do**
$R_j = \text{UNION}(R_{ij})$, i **in** $(1..m)$
$T_j = \text{JOIN}(R_j, S_j)$
endfor

end

It can be observed that the above algorithm for parallel hash join proceeds in two stages. In the first stage, the tuples of R with a particular hash value are sent only to those S fragments with tuples having the same hash value. In the second stage, the S fragments are joined with the R sub-fragments that are received at this particular S node.

An algorithm to increase the amount of parallelism in the hash-join algorithm by using *pipelining* is proposed in [95]. The hash-table for both relations A and B is formed. Whenever a tuple is produced from either relation, it is hashed to find the hash-key. Tuples in the other relation with the corresponding hash-key are compared with the new tuple. Any matching tuples are sent to the output stream. If one of the relations is exhausted, the tuples from the other relation are no longer inserted in the hash-table. This is so because only the first hash table is used in processing the join from now on i.e. it becomes like the non-pipelined version of the hash-join algorithm. Besides pipelining this algorithm also offers the advantage of being *symmetric* with respect to its operands. This eliminates the need to compare and order the operands before inputting them to the hash-join algorithm.

Most algorithms to select the best strategy in performing joins involving more than one relation first form an intermediate representation, called the *join-tree*. Optimizations are performed on the tree before feeding the tree-nodes to the join algorithms. This is done so in GAMMA [24], PRISMA [5, 95] and other parallel database systems. One may not always have the choice to select the shape of the tree, and the edges may have different costs, affecting the tree that is ultimately selected for performing the joins. GAMMA prefers linear trees with minimal total processing costs; [12] chooses to minimize the processing time on the longest path in the tree. The PRISMA database system combines the choice of tree with pipelining hash-join and distributing expensive operations over more processor nodes.

Algorithms have been proposed that perform joins relatively well even in the presence of data skew. One such algorithm is described in [98]. The output of the sort phase of the sort-merge algorithm is preprocessed before the join/merge phase. The largest skew elements are identified and are assigned to an optimal number of processors. This helps in load-balancing for the join phase. The algorithm is also reported to be *robust* with respect to data skew and the number of processors.

7. Case studies in parallel database systems

In this section we look at some past parallel database systems which have been developed in research and industrial environments. We discuss a number of distinctive features in different systems.

7.1. Case studies of memory resident systems

7.1.1. Integrated database processor (IDP). The IDP system was developed in Japan and is described in [89]. It is a main memory database machine. IDP processing is based on pipeline parallelism, rather than parallelism through multiple processors. Vectorization methods are used by which data in the RDMS internal pointer structures are dynamically rearranged into the vector form and processed using a pipelined vector processor. A 10-fold increase in speed is reported after the design changes of memory access methods and pipelining. The large factor of improvement underscores the importance of data organization and access methods in database processing.

7.2. Case studies in shared-everything systems

7.2.1. Processing on hypercube architectures. Join operations are an important part of database processing. Join processing on hypercube architectures is the key to implementing query engines on such architectures. For this reason, research has been undertaken on distribution of data on the disks, load balancing, as well as performance analysis of actual systems.

Implementation and performance of the join algorithm for relational databases on an *n*Cube computer system is studied in [6]. *Tuple balancing* is performed to obtain roughly even data distribution across parallel paths. This is achieved by nodes repeatedly exchanging tuple count information among themselves, and a node with larger number of tuples migrating data to one with lesser number. Relation compaction for saving disk space, and relation replication to allow quick recovery in case of failures, are also performed.

7.2.2. Teradata database machines. Teradata was a pioneer in producing several parallel database machines. The early models included DBC/1012 and P-20. The DBC/1012 was an MIMD machine based on shared-nothing architecture. P-90 used disk-arrays with multiple processor modules, providing fault tolerance and greater performance.

DBC/1012 utilized a broadcast interconnection network called the Ynet. This network had a bandwidth of 6 Mbits/s. The network did sort/merge operations on data, thus aiding in the performance improvement. Uses were connected to the DBC/1012 via InterFace Processor module boards (IFP's). An IFP was made up of a CPU, a channel interface controller, and a couple of high-speed Ynet interfaces, providing a degree of fault tolerance. Each IFP could handle up to 120 user sessions at a given time.

The data was distributed among Access Module Processors (AMP's) by using hashing. The primary index of the RDBMS was used as the key for this hashing. Normalization could also be applied to logical schema of the RDBMS to handle data skew. Future plans called for decrease in the level of granularity for hashing to enable further parallelism in disks attached to AMPs. Intra-query parallelism was supported by distributing queries among the AMPs. The different query plans could then be executed concurrently in the AMPs.

DBC/1012 uses hierarchical storage of data, including RAM, magnetic disk, optical disk, and external optical drive levels for data access and storage.

Future plans for DBC/1012 and P-90 computers included efficient designs to handle multimedia data, such as voice and image. It was expected that the linearly scalable database architectures could meet the challenge of massive amounts of data processing needed by such multimedia information services.

7.2.3. The Intel Concurrent File System (CFS). Intel's Concurrent File System (CFS) (figure 5) is of special interest since some performance measurements for it have been reported in the literature [14]. The techniques used in the CFS to increase I/O performance include the following. Large files are declustered over more than one disk. Some nodes are dedicated to I/O. These special I/O nodes have additional high-speed cache memories, permitting the caching and pre-fetching of files.

Some of the ideas such as those described in Section 5 on file organizations above, have been applied in the Intel Concurrent File System (CFS). CFS allows several read/write

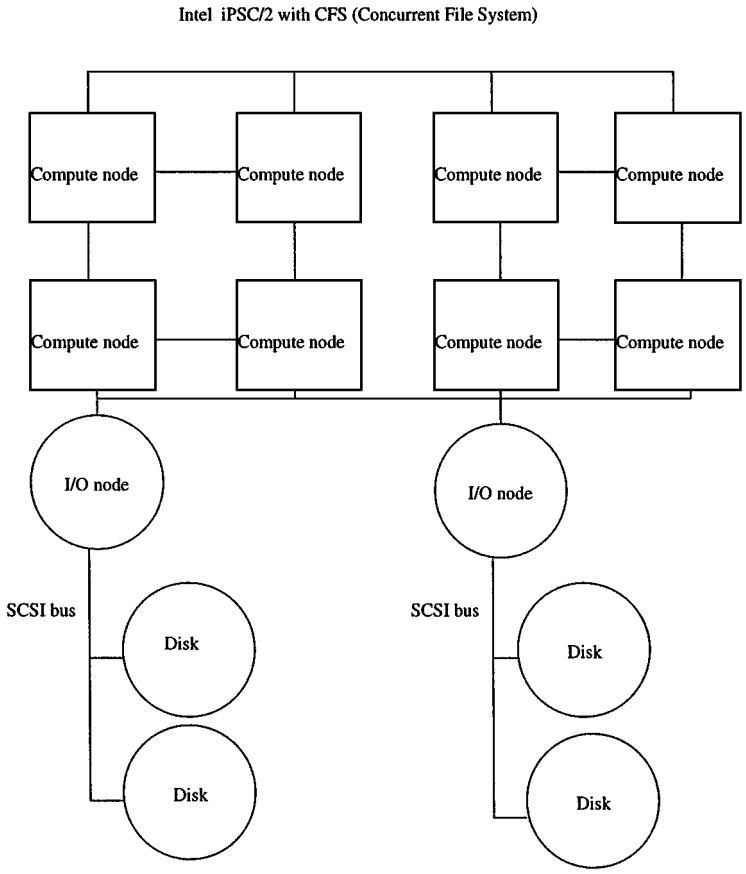

Figure 5. The intel concurrent file system.

operations in the same file to be performed in parallel. In CFS, all the disks of the system are treated as a single logical disk. This allows an application to access a large database from more than one client at the same time. Declustering is used to access file blocks. Primary advantage of this is that distribution of data over disks allows simultaneous transfer from them when data is required by independent concurrent processes. CFS has inbuilt high performance read and write calls for both synchronous operation, as in the UNIX operating system, as well as asynchronous operation, where I/O and data can proceed simultaneously.

CFS has been tested since about 1988. It reportedly achieves high data transfer rates from multiple I/O nodes, for very large file applications. Maximum sustained aggregate rate, *max_SAR*, is a measure of the rate of transfer of data obtained by summing the data rates of all the individual processors. The method used to measure the rate of transfer of data in CFS is max_SAR. The CFS study shows scalable high performance in CFS file system implemented on an Intel iPSC/2. Files are distributed evenly among disks. The factors that are tuned for maximal performance are: block size (this affects the efficiency of message passing); cache size at each I/O node; allocation of operations between compute nodes and I/O nodes to maximize concurrency of operations and minimize the overhead.

7.3. Case studies in shared-nothing systems

7.3.1. Bubba parallel database system. The goal of the Bubba project [13] was to provide high performance data access to large amounts of shared data. To this end, the system's desirable qualities include scalability, inexpensiveness, availability and ease of use. Based on the current limitations on bus technology, and desire for modular scalability, Bubba chose a "shared nothing" architecture.

Design issues are driven by the database requirements. Thus large amounts of data dictate processing of data wherever it may be located, in order to minimize data movement; multiple transaction processing loads dictate a need for a powerful programming language and full environment for run-time management of concurrent programs. High availability dictates fault tolerance through redundancy and recovery mechanisms.

The Bubba system consists of three kinds of nodes: the Interface Processors (IP), Intelligent Repositories (IR), and checkpoint-and-log IR's (CIR). Most processing is done in the IR's and IP's provide communication with the outside, and CIR's are used in recovery.

IR's may consist of more than a single processor. However, Bubba is designed to have small IR's so that the units of expandability are cheap, and also the failure of a single IR does not have large influence on the system. Another reason for keeping the IR's small is the simpler design within the IR's.

Two copies of data are kept online at all times, and a third copy is kept in the CIR's. Whenever faults or failures are detected in IR's, the contents of the faulted IR are recreated immediately. In order to facilitate this process, extra IP's, IR's and CIR's are kept accessible at all times.

In order to minimize data movement, data is placed statically, and programs are migrated to data rather than vice-versa. However, the data placement is reorganized periodically in order to improve overall performance. This reorganization is done transparently to the user processes.

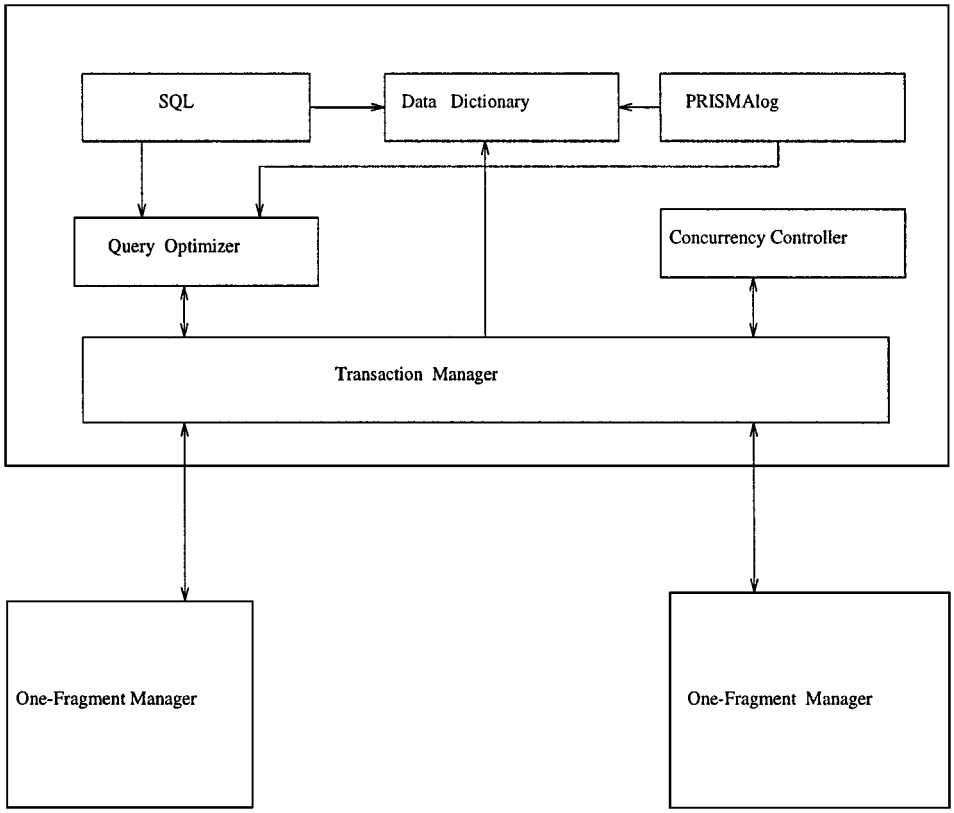

Figure 6. The PRISMA database architecture.

7.3.2. The PRISMA parallel database system. One of the research systems based on a shared-nothing architecture is PRISMA [5, 95] (figure 6). PRISMA runs on a multiprocessor system consisting of 100 nodes. Half of these nodes have their own disks. The nodes are connected to each other by communication processors. Each communication processor connects a node to 4 other nodes in the system, thus providing a relatively high bandwidth system. The database performs most of its functions in the main memory, and disks are used only for backups and recovery after failures. It consists of the components performing SQL and PRISMAlog parsing (i.e. interpreting database command languages), query optimization, transaction management, concurrency control, maintaining a data dictionary, as well as managers to control fragmentation and allocation of data objects. The kinds of parallelism employed include *multi-tasking, pipelining,* and *task-spreading*. Multi-tasking is used to breakup queries into sub-tasks, and then executing those sub-tasks in parallel whenever data dependence and data availability requirements allow it. More details about the parallel object-oriented language used, the hardware communication, and performance studies for this experimental main-memory, shared-nothing database system are given in [5].

7.3.3. The Connection Machine. The Connection Machine's CM-2 is a data parallel architecture. It has a total primary storage of 512 MB (composed of 256 Kbit chips), and 2 GB of memory (with 1 Mbit chips). Given a clock rate of 7 MHz, the data can be transferred to secondary storage at a rate of 45 GB/s. The primary storage has 64k ports, with a 1-bit processor for each port. The storage per processor is 8 KB, making a total of 512 MB of storage. The CM-2 model can also be equipped with hardware for accelerating floating point processing. This hardware allows 32 Connection Machine processors to share a floating point unit. The floating point unit consists of a single chip floating point multiplier and adder, with a few memory registers. It has been estimated that the peak performance achievable with a Connection Machine configured as such is in the vicinity of 2 Gegaflops/s.

The Connection Machine's memory is mapped into the memory of the host computer. The program to be executed, consisting of instructions, resides in the host computer. Instructions to be applied to the variables in the Connection Machine are sent to the micro-controller. The micro-controller decodes and executes the instructions for the Connection Machine.

The Connection Machine is also equipped with a hardware router, which selects the shortest path between the source and the destination of the message. Often, this routing can also be taken over by the application program, by using primitives built in the programming language.

When handling large amounts of data on the Connection Machine, we need to make some assumptions about the computational model in order to make the analysis simpler and manageable. Thus it is assumed that any given time of processing, the system consists of a collection of processors, each with its own physical memory. As far as possible, calculations are performed within a single processor. Inter-processor communication is needed, since very often the calculation may not be performed entirely at a single node. This involves cross-processor memory fetch operations. Such a fetch is performed asynchronously, and the second processor does not have knowledge about it. As an example, if the fetch is to a very large read-only table, this operation must be partitioned across processors in order to fit in memory. When processor p_i needs access to the table, it can compute in which processor p the data resides, and the local address of this data. The 2nd processor is identified via the first processor, and without the knowledge of the second processor. If a processor knows has information that a particular data element it has will be needed by another processor p_j, then processor p_i can immediately write this data to processor p_j. The data transfer is again asynchronous, and the second processor is unaware of this transfer. The only thing is that one of the data locations of processor p_j may have changed. One of the techniques used by the expecting processor to know the arrival of new data item is to set the memory to a special unlikely or impossible value, and then poll to find out if the value has changed. Exclusive access to data is assured by an atomic read-write command.

Methods for efficient utilization of processors for exhaustive tree searches, suitable for large parallel databases, are given in [36]. In SIMD Connection Machines, this is achieved by avoiding interprocess communication for computational purposes. Interprocess communication is, however, used for load balancing purposes. Because of the new dimensions introduced by communication costs in massively parallel machines, we see that mapping the problem to the specific architecture can pay off. An example of such pay-off is the mapping of the exhaustive tree search problem on the massively parallel SIMD Connection Machine.

Although a single problem cannot be measure of the difficulties involved in other search problems on such machines, it does give an indication of the excellent potential present in such machines for processing massive databases or complex queries requiring the search of large tree structures.

7.3.4. Gamma parallel database system. The design of the Gamma database machine and the techniques employed in its implementation are presented in [27]. Gamma is a relational database machine currently operating on an Intel iPSC2 hypercube with 32 processors and 32 disk drives. Gamma employs three key technical ideas which enable the architecture to be scaled to hundreds of processors. First all relations are horizontally partitioned across multiple disk drives enabling relations to be scanned in parallel. Second parallel algorithms based on hashing are used to implement the complex relational operators such as join and aggregate functions. Third dataflow scheduling techniques are used to coordinate multi-operator queries. By using these techniques it is possible to control the execution of very complex queries with minimal coordination. The design of the Gamma software and a thorough performance evaluation of the iPSCs hypercube version of Gamma is presented in [26]. The speedup results for selection and join queries are reported to be almost linear. This means that doubling the number of processors almost halves the response time. Scaleup, in which both hardware and work load are increased proportionally, is also investigated. The results show that the response time remains roughly constant with scaleup.

7.4. The GOLDRUSH megaSERVER

The GOLDRUSH system [94], developed by ICL, is a distributed store parallel processor system with upto 64 nodes. Objectives in the design of this parallel database server include robustness against failure of processors and disks, and the ability to manage the parallel machine as a single, centralized system. GOLDRUSH nodes consist of communications elements, processing elements, and management elements. Processing elements have two SCSI-2 connections, allowing upto 30 disks to be connected. Communications elements have a couple of FDDI connections. The PEs can each connect to about 50 GB of data storage. Most of the components in the system can be upgraded when better components become available. This allows the system to 'evolve' over time.

Data is mirrored in order to provide high availability. Disk crashes are transparent because processing can continue from a mirror. The data lock manager is also distributed, and lock information is mirrored in several PE. Hence there is robustness against element or process failure.

GOLDRUSH uses named sets of Elements for system management applications, including operations, capacity, configuration, and problems. Furthermore, there is provision for sets of disks and volumes. Management of sets of resources rather than particular pieces of hardware etc. is useful in providing failure transparency. For example, if an element fails, an alternative element replaces it internally, but the processes referring to the named set are oblivious to this change.

8. Commercial parallel database systems

Many major database vendors have developed versions of their products which employ parallel software and hardware for performance improvements. The performance benchmarks for specific parallel systems have been reported by the vendors. For example, recent results of a benchmark for a parallel version of Oracle DBMS running on a Sun E4000 server revealed that over 3000 concurrent users and a sustained processing load of over 700 processes a minute is achiveable. This is a vast performance improvement over single server systems.

Informix has developed parallel versions of its products which run on popular loosely-coupled symmetric multiprocessor (SMP) and massively parallel processor (MPP) systems running the UNIX operating system, including the following MPP platforms: Hitachi, IBM, ICL, NCR, Pyramid and Unisys, and the following clustered SMP platforms: Bull, Data General, Digital, HP, NCR, NEC, Sun, SGI and Sequent.

Informix's OnLine XPS is multithreaded database server designed to exploit the capabilities of loosely coupled or shared-nothing computing including clusters of SMP and MPP to deliver database scalability, manageability, and performance. OnLine XPS is optimized to support large-scale database environments for on-line transaction processing (OLTP), data warehousing, and other very large database (VLDB) applications. And OnLine XPS includes enhanced parallel SQL operations, high-availability capabilities, enterprise replication. OnLine XPS offers the ability to perform hash joins for performing join operations without indices on the join columns.

OnLine XPS takes a shared-nothing approach to managing data to reduce network I/O. Each node runs its own instance of the database including logging, recovery, and buffer management operations. Such an instance is called a co-server. Each co-server owns a set of disks and the partitions of the database that reside on these disks. Co-server may cooperate among them by exchanging subtasks for load distribution. Additional SMP nodes may also be added to scale up the database.

Data partitioning provides the basis for parallel execution of all SQL operations by partitioning large tables and indexes. Scans, joins, and sorts are distributed and executed across multiple CPUs and disks in parallel. Data partitioning minimizes I/O bottlenecks by allowing balanced I/O operations across all nodes and disks within a system. Control partitioning minimizes locking, logging, and schedule bottlenecks. Control is partitioned by utilizing function-shipping algorithms, data partitioning, and advanced query optimization within OnLine XPS. Requests are only sent to the node where the data resides. Each node manages the logging, recovery, locking, and buffer management for the database objects that it owns.

Queries are broken down into subqueries and executed in parallel (vertical parallelism or intraserver). These subqueries can be further broken down into subtasks to be executed in parallel on multiple co-servers (horizontal parallelism or interserver). The results of a particular scan can then be pipelined to a join subtask before the scans have been completed. Additionally, the results of the join can themselves be pipelined to other subtasks before the join is completed. Intraserver communications between subtasks that are located on the same SMP node take place via efficient shared-memory pointer passing. Interserver

communications between subtasks on different nodes are achieved using networked messages across a high-speed interconnect.

HP's Enterprise Parallel Server (EPS) has one GB/second high-speed fibre channel based system to system communication. EPS supernodes provide intraserver communication between subtasks that reside on the same SMP node via efficient shared memory pointer passing. The following benchmark results were reported on Informix XPS running on HP's Enterprise Parallel Server. A 100 GB database (350 GB total disk) was tested on an HP EPS20 cluster, using fibre channel interconnect technology, and nodes with four PA-RISC 7200 (100 Mhz) processors. The database was partitioned across eight Fast/Wide SCSI-2 disks on each node. Data was fragmented across disks using fragmentation techniques internal to XPS. No LVM disk striping was used. Doubling the number of nodes as well as the size of the database indicated database scalability of 95%. Doubling the number of nodes, but keeping the database size constant indicated a database speedup of 94%. In both cases, data load rates were found to be in excess of 2.2 GB/h/CPU, and thus a one node system with a total of four CPUs sustained a load rate of 8.8 GB/h. Thus the system demonstrated almost linear scalability in this test.

Sybase parallel databases (Sybase MPP) run on SMP, SMP cluster, and MPP platforms. They deploy multiple SQL Servers that work in unison to process queries, transactions, inserts, updates, and deletes in parallel, as well as parallel load, create index, backup and recovery. Sybase MPP has a message-based, shared-nothing architecture.

Sybase SQL Server 11 has achieved the following TPC-C benchmark results recorded for a mid-range UNIX-based symmetric multiprocessing (SMP) system, running on Digital Equipment Corporation's 64-bit AlphaServer 4100 5/400 system. SQL Server 11 attained 7,598 transactions per minute (tpmC) TPC-C benchmark on DEC Unix with four 400 MHz CPUs and 3 GB of memory. The configuration for this benchmark used the following Sybase features and modules: Logical Memory Manager, named caches, Data Slices, Soft Affinity, 64-bit support and fully symmetric networking. Sybase SQL Server 11 also achieved 14,739 transactions per minute (tpmC) running the TPC-C benchmark on HP's K460 system. Another benchmark test for Sybase SQL Server 11 achieved 18,438 transactions per minute (tpmC) on Sun Microsystems' Ultra Enterprise 6000 server utilizing 20 processors with the Solaris 2.5.1 operating environment.

Another recent parallel data server is described in [100].

We may expect continuous in the performance of such parallel systems. As such systems gain greater currency, competetion among vendors, reliable third party benchmarking, and employment of state of the art technology and economies of scale will further enhance the performance levels of parallel database systems.

9. Conclusions

It is clear from the foregoing sections that there are a large number of factors that warrant careful attention when designing parallel database systems for very large amounts of data. Among them are: the design of the physical machine architecture, data placement algorithms, efficient data structures and file manipulation strategies and parallel query processing methods.

Design of machine architectures incorporates the design of processors, memories, disks, and the interconnection networks. Architectural decisions affect the amount of different kinds of parallelism that can be obtained in shared-everything, shared-nothing or shared-disk systems. Shared-everything systems provide advantages of easier management since their programming is simple and adaptation of existing databases is relatively easier. However, the design of the interconnection network can affect the interprocessor communication times. Shared-nothing systems provide the benefits of scalability and lower communication costs. Potential for bottlenecks is also reduced in shared-nothing systems.

Selection of algorithms for database operations such as sorting, performing joins, projections etc. is critical in the performance of the system. Efficient algorithms are required for concurrent access to data structures such as lists, trees, heaps and hashing tables. Such data structures and file systems require deadlock-free serializability through locking or some other schemes in order to have prevent the corruption of data along with its availability. These file and data structures can then be utilized in parallelization of query processing, including inter-query parallelism, intra-query parallelism, and intra-operation parallelism.

One of the main issues in designing data partitioning and placement strategies is avoiding the data skew since significant amount of data skew in a system can cause inefficient utilization of processors and disks. Data fragmentation can be done using either horizontal or vertical fragmentation of the database relations. For maximal efficiency, distribution of data for load balancing has to be carefully weighed against the increased costs of communication and data recombination. In addition, combination of disk or tape striping large files across disks and contiguous allocation of selected data can provide major improvements over systems without data striping and contiguous allocation.

Further issues that need to be considered include the effects of operations such as reload, unload, and reorganization of data; heterogeneity issues; performance measurements for a mix of complex query work loads, and the relative advantages of parallel synchronous pipelining versus parallel asynchronous pipelining in the processing of database queries; data-specific effects such as those in multimedia data etc. These are open areas of research and it may be some time before definitive opinions are reached on them.

Notes

1. “Stable main memory” in the current context implies large enough main memory in the system so that the entire database can be loaded from the disk into the memory.
2. “Data directory” consists of meta-data about the data in the database e.g. access permissions for different data, alias names, sizes, locations etc.
3. “Throughput” in this context is generally measured in terms of transactions per second.

References

1. I. Ahmad and A. Ghafoor, “Semi distributed load balancing for massively parallel multicomputer systems,” IEEE Transactions on Software Engineering, vol. 17, no. 10, 987–1006, October 1991.

2. I. Ahmad, A. Ghafoor, and G. Fox, "Hierarchical scheduling of dynamic parallel computations on hypercube multicomputers," to appear in Journal of Parallel and Distributed Computing.
3. I. Ahmad, A. Ghafoor, and K. Mehrotra, "A decentralized task scheduling algorithm and its performance modeling for computer networks," in Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing, Dallas, Texas, December 1991.
4. W. Alexander and G. Copeland, "Process and dataflow control in distributed data-intensive systems," in Proceedings of the 1988 SIGMOD Conference, Chicago, June 1988.
5. P. Apers, B. Hertzberger, B. Hulshof, H. Oerlemas, and M. Kersten, "PRISMA, a platform for experiments with parallelism," Parallel Database Systems, Pierre America (Ed.), Springer-Verlag, 1990.
6. C.K. Baru, O. Frieder, D. Kandlur, and M. Segal, "Join on a cube: Analysis, simulation and implementation," Database Machines and Knowledge Base Machines, M. Kitsuregawa and H. Tanaka (Eds.), Kluwer, 1987.
7. F. Bastani, I. Iyengar, and I. Yen, "Concurrent maintance of data structures in a distributed environment," The Computer Journal, 1988.
8. R. Bayer and M. Schkoinick, "Concurrency of operations on B-Trees," Acta Informatica, vol. 9, no. 1, pp. 1–21, 1977.
9. G. Bell, "Ultracomputers, a teraflop before its time," Communication of the ACM, pp. 27–47, August 1992.
10. A. Bhide and M. Stonebraker, "A performance comparison of two architectures for fast transaction processing," in Proceedings of the Fourth International Conference on Data Engineering, Los Angeles, CA, 1988.
11. D. Bitton and J. Gray, "Disk shadowing," in Proceedings of the Fourteenth International Conference on Very Large Data Bases, Los Angeles, Calif., August 1988.
12. P. Bodorik and J.S. Riordon, "Heuristic algorithms for distributed query processing," in Proceedings of the First International Symposium on Databases in Parallel and Distributed Systems, Austin, TX, December 1988.
13. H. Boral and D. DeWitt, "Database machines: An idea whose time has passed? A critique of the future of database machines," in Proceedings of the 3rd International Workshop on Database Machines, August 1985.
14. R. Bordawekar, J. del Rosario, and A. Choudhary, "Design and evaluation of primitives for parallel I/O," in Proceedings of the Supercomputing '93 Conference, Portland, OR, November 1993.
15. N. Bowen, C. Nikolaou, and A. Ghafoor, "On the assignment problem of arbitrary process systems to heterogeneous distributed computing systems," IEEE Transactions on Computers, vol. 41, no. 3, pp. 257–273, March 1992.
16. A. Borr, "Robustness to crash in a distributed database: A non-shared memory multi-processor approach," in Proceedings of the 10th International Conference on Very Large Data Bases, Singapore, August 1984.
17. H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez, "Prototyping bubba: A highly parallel database system," in IEEE Transactions on Knowledge and Data Engineering, vol. 2, no. 1, March 1990.
18. J.C. Browne, A.G. Dale, C. Leung, and R. Jenevein, "Parallel multi-stage I/O architecture with self-managing disk cache for database management applications," in Proceedings of Database Machines: Fourth International Workshop, Bahamas, March 1985.
19. S. Ceri and G. Pelagatti, Distributed Databases: Principles and Systems, McGraw-Hill, 1984.
20. S.M. Chung, "Parallel relational operations based on clustered surrogate files," in Proceedings of the Third Symposium on the Frontiers of Massively Parallel Computation, College Park, MD, October 1990.
21. G. Copeland, W. Alexander, E. Boughter, and T. Keller, "Data placement in Bubba," in Proceedings of ACM-SIGMOD International Conference on Management of Data, Chicago, May 1988.
22. T.H. Cormen, C.E. Leiserson, and R.L. Rivest, Introduction to Algorithms, McGraw Hill, 1990.
23. E. DeBenedictis and J.M. Rosario, "Scalable I/O," nCUBE Technical Report, nCube-TR001-911015, October 15, 1991.
24. D. DeWitt, R.H. Gerber, G. Graefe, M.L. Heytens, K.B. Kumar, and M. Muralikrishna, "GAMMA—A high performance dataflow database machine," in Proceedings of the 12th International Conference on Very Large Databases, Kyoto, Japan, August 1986.

25. D. DeWitt and R. Gerber, “Multi processor hash-based join algorithms,” in Proceedings of the 11th International Conference on Very Large Databases, Stockholm, Sweden, August 1985.
26. D. DeWitt, et al., “The GAMMA database machine project,” IEEE Transactions on Knowledge and Data Engineering, vol. 2, no. 1, pp. 44–62, March 1990.
27. D. DeWitt and J. Gray, “Parallel database systems: The future of high performance database systems,” Communications of the ACM, June 1992.
28. D. Dias, B. Iyer, J. Robinson, and P. Yu, “Integrated concurrency-coherency controls for multisystem data sharing,” in Proceedings of the IEEE Transactions on Software Engineering, vol. 15, no. 4, April 1989.
29. J. Driscoll, H. Gabow, R. Sharairman, and R. Tarjan, “Relaxed heaps: An alternative to fibonacci heaps with applications to parallel computation,” Communications of the ACM, vol. 31, no. 11, pp. 1343–1354, November 1988.
30. N. Duppel, “Modeling and optimization of complex database queries in a shared-nothing system,” in Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing, Dallas, Texas, December 1991.
31. C.S. Ellis, “Concurrency in linear hashing,” ACM Transactions on Database Systems, vol. 12, no. 2, pp. 195–217, June 1987.
32. C.S. Ellis, “Concurrent search and insertion in AVL Trees,” in IEEE Transactions on Software Engineering, vol. C-29, no. 9, September 1980.
33. C.S. Ellis, “Concurrent search and insertion in 2–3 trees,” Acta Information, vol. 14, 1980.
34. C. Faloutsos and S. Christodoulakis, “Description and performance analysis of signature file methods for office filing,” ACM Transactions on Office Information Systems, vol. 5, no. 3, July 1987.
35. O. Frieder, “Multiprocessor algorithms for relational-database operators on hypercube systems,” IEEE Computer, pp. 13–28, Nov. 1990.
36. R. Frye and J. Myczkowski, “Exhaustive search of unstructured trees on the connection machine,” submitted to Journal of Parallel and Distributed Computing.
37. J. Fu and T. Kameda, “Concurrency control for nested transactions accessing B-Trees,” in Proceedings of the 8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, March 1989.
38. H. Garcia-Molina, R. Abbot, C. Clifton, C. Staelin, and K. Salem, “Data management with massive memory: A summary,” in Parallel Database Systems, Pierre America (Ed.), Springer-Verlag, 1990.
39. S. Ghandeharizadeh and D.J. Dewitt, “Hybrid-range partitioning strategy: A new declustering strategy for multiprocessor database machines,” in Proceedings of the Sixth International Conference on Data Engineering, February 1990.
40. F. Grandi, P. Tiberio, and P. Zezula, “Frame-sliced partioned parallel signature files,” in Proceedings 15th Annual International ACM SIGIR Conference, June 1992.
41. E. Haq and S. Zheng, “Parallel algorithms for balancing threaded binary trees,” in Proceedings of the Eight Annual International Phoenix Conference on Computers and Communications, Scottsdale, AZ, March 1989.
42. T. Harder, H. Schoning, and A. Sikeler, “Parallelism in processing queries on complex objects,” in Proceedings of the International Symposium on Databases in Parallel and Distributed Systems, Austin, TX, October 1988.
43. D.W. Hills and L.G. Steele Jr., “Data parallel algorithms,” Communications of the ACM, vol. 29, no. 12, pp. 1170–1183, 1986.
44. K.A. Hua and C. Lee, “Handling data skew in multiprocessor database computers using partition tuning,” in Proceedings of the Seventeenth International Conference on Very Large Data Bases, Barcelona, Spain, September 1991.
45. Y.-N. Huang and J.-P. Cheiney, “An effective algorithm for parallelizing hash joins in the presence of data skew,” in Proceedings of the Seventh International Conference on Data Engineering, Kobe, Japan, April 1991.
46. K. Hwang, Advanced Computer Architecture, Parallelism, Scalability, Programmability, McGraw Hill, 1993.
47. Intel, Paragon XP/S Product Overview Supercomputer Systems Division, Intel Corporation, Beaverton, OR, 1991.
48. S. Khoshafian and P. Valduriez, “Parallel query processing of complex objects,” in Proceedings of the Fourth International Conference on Data Engineering, Los Angeles, CA, February 1988.

49. M. Kim, "Synchronized disk interleaving," IEEE Transactions on Computers, vol. C-35, no. 11, November 1986.
50. M. Kitsuregawa, W. Yang, and S. Fushimi, "Evaluation of 18-stage pipeline hardware sorter," in Proceedings of the Third International Conference on Data Engineering, February 1987.
51. Y. Kiyoki, T. Kurosawa, K. Kato, and T. Masuda, "The software architecture of a parallel processing system for advanced database applications," in Proceedings of the Seventh International Conference on Data Engineering, Kobe, Japan, April 1991.
52. D. Kotz and C. Ellis, "Caching and writeback policies in parallel file systems," in Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing, Dallas, Texas, December 1991.
53. M.H. Kryder, "Data storage in 2000—trends in data storage technologies," IEEE Transactions on Magnetics, vol. 25, no. 6, November 1989.
54. H. Kung and P. Lehman, "Concurrent manipulation of binary search trees," ACM Transaction on Database Systems, vol. 5, no. 3, September 1980.
55. Y.-S. Kwong and D. Wood, "A new method for concurrency in B-Trees," IEEE Transactions on Software Engineering, vol. 8, no. 3, May 1982.
56. P. Lehman and S. Yao, "Efficient locking for concurrent operations on B-Trees," ACM Transactions on Database Systems, vol. 6, no. 4, December 1981.
57. F.T. Leighton, Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes, Morgan Kaufmann Publishers, 1992.
58. M.D.P. Leland and W.D. Roome, "The silicon database machine," in The Proceedings of the 4th International Workshop on Database Machines, Bahamas, March 1985.
59. K. Li and J.F. Naughton, "Multiprocessor main memory transaction processing," in Proceedings of the International Symposium on Databases in Parallel and Distributed Systems, Austin, TX, 1988.
60. W. Litwin, "Linear hashing: A new tool for file and table addressing," in Proceedings of the 6th International Conference on Very Large Data Bases, pp. 212–223, 1980.
61. M. Livny, S. Khoshafian, and H. Boral, "Multi-disk management," in Proceedings of the ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems, pp. 69–77, Banff, Canada, 1987.
62. D. Lomet and B. Salzberg, "Access method concurrency with recovery," in Proceedings of the ACM SIGMOD Conference, pp. 351–360, 1992.
63. D. Lomet and B. Salzberg, "Access methods for multiversion data," in Proceedings of the ACM SIGMOD Conference, pp. 315–324, May 1989.
64. R. Lorie, J. Daudenarde, G. Hallmark, J. Stamos, and H. Young, "Adding intra-transaction parallelism to an existing DBMS: Early experience," IEEE Data Engineering Newsletter, vol. 12, no. 1, March 1989.
65. C. Mohan and I. Narang, "Efficient locking and caching of data in multi-system shared disks transaction environment," IBM Research Report RJ 8301, 1991.
66. T. Ohmori, M. Kitsuregawa, and H. Tanaka, "Scheduling batch transactions on shared-nothing parallel database machines: Effects of concurrency and parallelism," in Proceedings of the Seventh International Conference on Data Engineering, Kobe, Japan, April 1991.
67. M. Ozsu and P. Valduriez, "Principles of distributed database systems," Prentice-Hall, 1991.
68. J. Parker, "Concurrent search structure," Journal of Parallel and Distributed Computing, vol. 7, no. 2, pp. 256–278, October 1989.
69. J.H. Patel, "Performance of processor-memory interconnections for multiprocessors," IEEE Transactions on Computers, vol. C-30, no. 10, pp. 771–780, October 1981.
70. D. Patterson, G. Gibson, and R. Katz, "A case for redundant arrays of inexpensive disks (RAID)," in Proceedings of the ACM SIGMOD Conference, pp. 109–116, Chicago, June 1988.
71. W. Pugh, "Skip lists. A probabilistic alternative to balanced trees," Communications of the ACM, vol. 33, no. 6, pp. 668–676, June 1990.
72. J.P. Richardson, H. Lu, and K. Mikkiilineni, "Design and evaluation of parallel pipelined join algorithms," in Proceedings of the ACM SIGMOD Conference, San Francisco, CA, June 1987.
73. J.R. Rose and L.G. Steele Jr., "C*: An extended C language for data parallel programming," Technical Report PL87-5, Thinking Machines Corporation, April 1987.
74. Y. Sagiv, "Concurrent operations on B*-trees with overtaking," Journal of Computer and System Sciences, vol. 33, no. 2, pp. 275–296, 1986.

75. K. Salem and H. Garcia-Molina, "System M: A transaction processing testbed for memory resident data," IEEE Transactions on Knowledge and Data Engineering, vol. 2, no. 1, pp. 161–172, March 1990.
76. B. Salzberg, "Concurrency in grid files," Information Systems Journal, vol. 11, no. 3, pp. 235–244, 1986.
77. B. Samadi, "B-Trees in a system with multiple users," Information Processing Letters, vol. 5, no. 4, pp. 107–112, 1976.
78. D.A. Schneider and D.J. DeWitt, "A performance evaluation of four join algorithms in a shared-nothing multiprocessor environment," in Proceedings of the ACM SIGMOD Conference, Portland, OR, June 1989.
79. R. Sedgewick, Algorithms, Addison-Wesley Publishing Company: Reading, MA, 1983.
80. M. Seltzer and M. Stonebraker, "Read Optimized file system designs: A performance evaluation," in Proceedings of the IEEE 7th International Conference on Data Engineering, 1991.
81. S. Seshadri and F.J. Naughton, "Sampling issues in parallel database systems," in Advances in Database Technology-EDBT'92, Vienna, Austria, March 1992.
82. D. Shasha and N. Goodman, "Concurrent search structure algorithms," ACM Transactions on Database Systems, vol. 13, no. 1, pp. 53–90, March 1988.
83. H.J. Siegel, Interconnection Networks for Large-Scale Parallel Processing: Theory and Case Studies, 2nd ed., McGraw-Hill: New York, 1989.
84. C. Stanfill and B. Kahle, "Parallel free text search on the connection machine," Communications of the ACM, vol. 29, no. 12, December 1986.
85. M. Stonebraker, "The case for shared-nothing," Database Engineering, vol. 9, no. 1, 1986.
86. The Tandem Database Group, "A benchmark of nonstop SQL on the debit credit transaction," in Proceedings of the ACM SIGMOD Conference, Chicago, 1988.
87. S. Thakkar and M. Sweiger, "Performance of an OLTP application on symmetry multiprocessor system," in Proceedings of the Seventeenth International Symposium on Computer Architecture, Seattle, Washington, May 1990.
88. Thinking Machine Corporation, The CM-5 Technical Summary, Cambridge, MA, 1991.
89. S. Torii, K. Kojima, S. Yoshizumi, A. Sakata, Y. Takamoto, S. Kawabe, and M. Takahashi, "Relational database system architecture based on a vector processing method," Information Sciences, vol. 48, no. 2, pp. 135–155, July 1989.
90. P. Valduriez, "Parallel database systems: Open problems and new issues," in Distributed and Parallel Databases, vol. 1, pp. 137–165, 1993.
91. C. Walton and A. Dale, "Data skew and the scalibility of parallel joins," in Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing, Dallas, Texas, December 1991.
92. C. Walton, A. Dale, and R. Jenevein, "A taxonomy and performance model of data skew effects in parallel join," in Proceedings of the Seventeenth International Conference on Very Large Data Bases, Barcelona, Spain, September 1991.
93. D. Waltz, "Applications of the connection machine," Computer, vol. 10, no. 1, pp. 85–97, January 1987.
94. P. Watson and G. Catlow, "The architecture of the ICL GOLDRUSH MegaSERVER," in Advances in Databases, BNCOD 13, C. Goble and J. Keane (Eds.), Springer, 1995, pp. 249–262.
95. A. Wilschut and P. Apers, "Pipelining in query execution," in Proceedings of the PARBASE-90 Conference, Miami, FL, March 1990.
96. A. Wilschut, P. Apers, and J. Flokstra, "Parallel query execution in PRISMA/DB," in Parallel Database Systems, Pierre America (Ed.), Springer-Verlag, 1990.
97. V. Winters, "Parallelism for high performance query processing," in Advances in Database Technology-EDBT'92, Vienna, Austria, March 1992.
98. J. Wolf, D. Dias, and J. Turek, "An effective algorithm for parallelizing hash joins in the presence of data skew," Proceedings IEEE 7th International Conference on Data Engineering, 1991.
99. I. Yen, D. Leu, and F. Bastani, "Hash table and sorted array: A case study of multi-entry data structures in massively parallel systems," in Proceedings of the Third Symposium on the Frontiers of Massively Parallel Computation, College Park, MD, October 1990.
100. J. Annen and M. Okumura, "Parallel data warehouse server," Fujitsu, vol. 50, no. 3, pp. 135–139, Fujitsu, Japan, 1999.

101. R. Brave, M. Kallahalla, P.J. Varman, and J. ScottVitter, “Competitive parallel disk prefetching and buffer management,” in Proceedings of the Fifth Workshop on I/O in Parallel and Distributed Systems, pp. 47–56, 1997.
102. G.M. Bryan, W.E. Moore, B.J. Curry, K.W. Lodge, and J. Geyer, “The MEDUSA project: autonomous data management in a shared-nothing parallel database machine,” in Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, p. 507, 1994.
103. G.A. Gibson, D.F. Nagle, K. Amiri, J. Butler, F.W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka, “A cost-effective, high-bandwidth storage architecture,” ACM SIGPLAN Notices (ACM Special Interest Group on Programming Languages), vol. 33, no. 11, pp. 92–103, November 1998.
104. M. Oguchi and M. Kitsuregawa, “Dynamic remote memory acquistion for parallel data mining on ATM-connencted PC cluster,” in Proceedings of the 1999 International Conference on Supercomputing, pp. 246–252, 1999.
105. T. Stricker, J. Stichnoth, D. O’Hallaron, S. Hinrichs, and T. Gross, “Decoupling synchronization and data transfer in message passing systems of parallel computers,” in Proceedings of the 9th ACM International Conference on Supercomputing, pp. 1–10, 1995.