DB2 Parallel Edition

By Gilles Fecteau

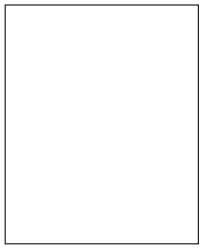
This article describes IBM's plans for a parallel DB2/6000 database server using the shared-nothing hardware model and function shipping. It also shows how this approach provides a scalable database server that meshes with the IBM POWERparallel™ architecture. The DB2 Parallel Edition will allow customers to grow databases with near linear performance improvement by adding nodes. The beta testing is now underway and availability will be announced later in 1994.

Parallel processing—concurrently executing two or more processors as a single unit—supports large UNIX-based applications. Most Relational Database Management Systems (RDBMSs) are being enhanced to support parallel processing to take advantage of low-cost workstations.

IBM began studying parallel technology in 1989 with prototypes developed by IBM Research. The IBM Research Division developed a prototype for a parallel database system using multiple PS/2 workstations and the OS/2® operating system. This prototype was demonstrated at COMDEX® in 1990.

Since then, IBM has been working on an architecture that allows DB2/6000 to run on several independently connected workstations managed as a single database by DB2/6000 parallel code. This product will operate on a range of hardware from LAN-connected RISC System/6000 systems to the IBM POWERparallel system.

Parallel query execution alone is not sufficient—large database users also need the ability to maintain very large databases. Achieving the goal of an effective parallel database means focusing on database management utilities and effective parallel queries.



Gilles Fecteau

Parallel Architectures

Several possible architectures can exploit multiple processors, large memory, and many disks. The most common architectures are as follows:

- **Shared nothing** uses multiple processors, each with its own memory and disk storage.
- **Shared disks** uses multiple processors, each with its own memory and shared disk storage.
- **Symmetric Multiprocessing (SMP)** uses multiple processors that have common memory and shared disk storage.

IBM is extending DB2/6000 to support the shared-nothing architecture. The database manager handles database requests from an application. It uses a communications link to coordinate the work of the multiple processors. This architecture was selected for two reasons:

- **Scalability**: It can scale to hundreds of processors.[1]
- **Portability**: Since it requires only a communications link between processors, its design can be ported to any platform that has communications. This is not true for SMP or shared disk. Although performance varies with the efficiency of the communication protocol, the high-speed switch in IBM POWERparallel systems (as well as many forms of UNIX sockets) ensures good performance.

The DB2 Parallel Edition will support SMP; in a parallel database configuration, each node could be an SMP.

Shared-Nothing

Figure 1 shows a DB2® parallel database running on multiple RISC System/6000s supporting a net-

[1] Dewitt and Gray. "Parallel Database Systems: The Future of High-Performance Database Systems," *Communications of the ACM* 35 (June 1992).

work of clients. A *data node* is the disk storage plus a portion of the database engine that manages the disk. In the DB2 parallel implementation, a database is stored across a network of processors that provide separate buffers, lock structures, logs, and disks for each processor. This prevents a cache contention problem that could result from all processors sharing one set of resources—as with an SMP implementation.

Because each processor has separate logs, applications are not limited to the I/O bandwidth of a single log and can perform recovery from failure in parallel.

Function Shipping

To minimize communication between processors, relational operators are executed on the processor containing the data whenever possible. Therefore, a central lock manager is not required as with most shared-disk implementations. This process of sending the work to the location of the data is known as *function shipping*.

Suppose that an `EMPLOYEE` table is distributed over multiple processors, and the following SQL statement is executed:

```
SELECT EMP_NO FROM EMPLOYEE
   WHERE SALARY>100000
```

The database manager on the *coordinating processor* (the processor from which the statement was issued) issues a request. This request (sent to every other processor in the group) is to select its subset of rows that have a `SALARY` value over 100,000. Each processor in the group then returns its answer set to the coordinating processor for final processing.

No central lock manager is needed because each processor initially extracts an answer from its own part of the `EMPLOYEE` table. At regular intervals, a *global deadlock detector* analyzes locks held to determine if a deadlock is present, then selects a victim to resolve the deadlock. This avoids sending thousands of lock requests between systems—a quick deadlock detection occurs every few seconds.

In *I/O shipping*—a less efficient alternative to DB2's function shipping—one or more of the group's processors are arbitrarily selected to run a query. All processors must send their data for a given query to the executing processors. Since all data pages of the `EMPLOYEE` table must be sent, more data movement occurs among processors than with function shipping. With function shipping, only the qualifying rows (employees with a salary over $100,000) are sent from the processor owning the data to the processor coordinating the query.

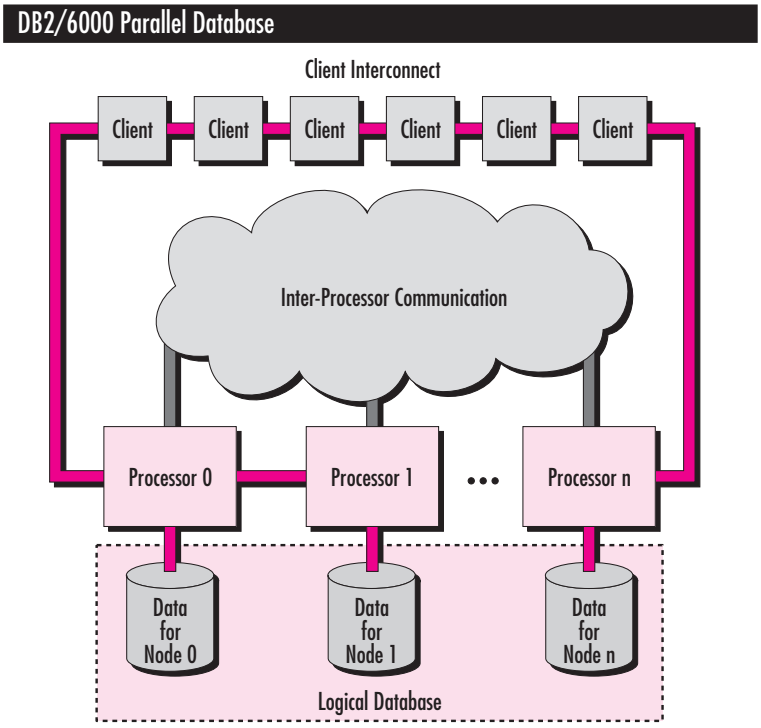


Figure 1. A parallel database environment

The system configuration for the I/O-shipping model differs from the function-shipping model.

- **I/O shipping**: Nodes that specialize in I/O must be configured with large numbers of disks, and other nodes without disks handle user queries.
- **Function shipping**: The configuration must provide a moderate number of disks on each processor, and the data must be partitioned over many processors.

The number of disks required for either model is generally the same for large databases from 50 GB to terabytes. I/O shipping may be less expensive for databases smaller than 50 GB, but these databases are not likely to use parallel technology.

DB2 Parallel Edition does not limit the function-shipping model to queries only. It can also perform database updates and run utilities. When a row is to be inserted into a table, it is sent to

the appropriate node where it is inserted (the appropriate node is determined by the hashing algorithm used for table partitioning). The index entries are updated and the row information is logged at the same node. Index maintenance, locking, and logging are distributed across processors.

Data Placement

For large databases, data placement can be complex and system administration can be difficult; therefore, appropriate Data Definition Language statements and administration utilities are needed to manage data partitioning. The DB2 parallel implementation is easier to administer than a non-partitioned database of the same size. For example, without partitioning, functions such as backup would take too long. Tools are provided for efficiently managing the large tables that occur in databases. The DB2 design ensures that database design decisions are separated from load-balancing decisions.

The DB2 Parallel Edition is easier to administer than a non-partitioned database of the same size.

Two important features of the DB2/6000 parallel database design that help in data partitioning and database administration are the partitioning key and nodegroups.

Partitioning Key

For a database that has many tables, an application developer can define a partitioning key for each table. Frequently joined tables should be partitioned on their respective join columns. The partitioning key information is specified as part of the `CREATE TABLE` statement, shown in Figure 2.

```
CREATE TABLE ACCOUNT (
        C_BRANCH INTEGER,
        CUST_NO INTEGER,
        CUSTOMER_NAME VARCHAR(50),
        LAST_DATE DATE,
        BALANCE DEC(8,2),
        .
        .
        .
        )
IN SMALLPOOL PARTITION BY
HASHING(C_BRANCH)
CREATE TABLE BRANCH (
             BRNO INTEGER NOT NULL,
             ADDRESS VARCHAR(200),
             .
             .
             .
             PRIMARY KEY(BRNO) )
```

Figure 2. Examples of CREATE TABLE statements with partitioning

The first statement specifies that the `ACCOUNT` table is partitioned on the `C_BRANCH` column; the second specifies that the `BRANCH` table is partitioned on the `BRNO` column. More than one column can be specified as the partitioning key. A system-defined hashing function is applied to the partitioning key value to determine in which processor a particular row will reside.

The `C_BRANCH` and `BRNO` columns were chosen as the partitioning keys for these two tables because they best suit the application. The suitability of these columns as partitioning keys remains valid, regardless of table size and the number of processors on which the tables are partitioned. To design tables, designers do not need prior knowledge of the configuration of the parallel system and the load on the system. Tuning for load balancing can be done after database definition time.

In the example in Figure 2, the processors on which the tables are partitioned are not specified directly. The `CREATE TABLE` statement has been extended to include an `IN <nodegroup name>` clause that provides this information. A *nodegroup* is an arbitrary name given to a set of nodes that will be used for tables. Tables in the same nodegroup will be partitioned over the same processors. The identifier `SMALLPOOL`, following the keyword `IN`, is the nodegroup name.

Nodegroup

In a shared-nothing hardware configuration, such as IBM's SPx family of parallel processors or LAN-connected workstations, each processor runs the equivalent of a single-node DB2/6000 database system. The database storage capabilities of each processor are the same as those provided by DB2/6000, including segmented table support that can implement tables of up to 64 GB. With the DB2 Parallel Edition, however, the storage capability of a system with n nodes is n times that of a uniprocessor DB2/6000 system.

Database tables can be defined across a set of nodes by first defining a nodegroup and then creating tables in it. A nodegroup can be created as follows:

```
CREATE NODEGROUP SMALLPOOL
   ON NODES (DBMACH1,DBMACH2,DBMACH6)
```

This statement defines a nodegroup called SMALLPOOL, consisting of three processors: DBMACH1, DBMACH2, and DBMACH6. Tables created in nodegroup SMALLPOOL are partitioned across these three processors. A nodegroup can contain one or more processors, and a processor can be a member of more than one nodegroup in the same database or across databases.

As the size of tables or the number of processors in the system increases, the ALTER NODEGROUP statement can be used to add processors to an existing nodegroup. After a processor is added to a nodegroup, data belonging to tables in the nodegroup can be redistributed using the rebalance utility.

The following characteristics make the DB2 Parallel Edition easy to manage:

- The application view of a table is separate from the physical placement of data.
- Data can be distributed over multiple disks to reduce I/O bottlenecks.
- The number of processors can be increased as the workload or size of the database increases.

Parallel Query Processing

The DB2 Parallel Edition generates a parallel execution strategy for all SQL statements using a cost-based relational database optimizer. After comparing several parallel execution strategies for each SQL statement, the cost-based optimizer selects the most efficient one. An SQL statement (such as SELECT, INSERT, UPDATE, or DELETE) is divided into several separate tasks. The *coordinator task* runs at the node where the application connects. It fetches input data from the application and returns the answer set to the application. Subordinate tasks, called *slave tasks*, perform the bulk of the activity required for the query and cooperate with each other when necessary. While there can be only one instance of a coordinator task for each application, there can be multiple instances of each slave task.

Since DB2 Parallel Edition does not impose any new restrictions on SQL statements, the investment made in existing applications remains. Generating a parallel execution strategy for a given SQL statement is automatic. A DB2/6000 application program does not have to be recompiled to exploit parallel execution. When the application program is bound to a parallel database, the appropriate parallel execution strategy is generated and stored, if required.

The optimization process of an SQL statement in a parallel DB2 Parallel Edition environment is based on two primary factors:

- **The distribution of the data across nodes**: DB2 Parallel Edition supports data partitioning by hashing the values of a set of columns across a set of nodes.
- **The cost of functions associated with different operations**: The DB2/6000 optimizer has cost formulas for the different tasks. New assumptions are added to account for parallel operations and messages (rows). The database manager generates the optimal parallel plan rather than having the best serial plan simply executed in parallel. The database manager performs this optimization without any external information.

The DB2 Parallel Edition generates a parallel execution strategy for all SQL statements using a cost-based relational database optimizer.

The repertoire of parallel strategies used by DB2 Parallel Edition includes parallel table scans and parallel index scans for tables; co-located, hashing redistributed, or broadcast joins for joins; and local and global aggregates, including the GROUP BY clause.

The full power of parallel processing is also used for other SQL constructs, such as subqueries, set operations (union/difference/intersect), and other operations such as UPDATE, INSERT, and DELETE.

Example of Parallel Queries

Using a database of ACCOUNT and BRANCH tables, this section describes three parallel execution strategies.

Parallel relation scan: The following query is divided into two task types: the coordinator task that returns the answer set to the application, and slave tasks (on a given partition of the ACCOUNT table) that select the matching rows and pipe them to the coordinator. The slave task has instances on all the nodes where the ACCOUNT table resides.

```
SELECT CUSTOMER_NAME FROM ACCOUNT
   WHERE BALANCE > 20000
```

Figure 3 shows the execution snapshot for this query. The data predicates are evaluated as soon as possible at the nodes where the data resides. This minimizes the amount of data exchanged using messages. Expressions and scalar functions are also "pushed down" and evaluated as soon as possible.

Parallel aggregation, including GROUP BY clause: This example shows how the DB2 Parallel Edition forces aggregation tasks to be

Figure 3. The execution snapshot for the parallel relation scan

executed at the site of the data, minimizing the sequential work at the coordinator task.

```
SELECT C_BRANCH, COUNT(*)
   FROM ACCOUNT
   WHERE BALANCE > 20000
     GROUP BY C_BRANCH
```

The execution strategy involves two task types:

- Slave tasks select matching rows on a given partition, group them according to C_BRANCH value, perform local aggregation (a local count for each group), and return the grouping column and the local count to the coordinator task.
- The coordinator task performs the global aggregation (a global count from the local count values) by merging the grouping values and local counts returned from the different slave tasks. It then returns the answer to the application.

Co-located joins: In the following example, both the ACCOUNT and BRANCH tables are partitioned on their joining columns.

```
SELECT BRANCH_NAME, CUSTOMER_NAME
  FROM ACCOUNT,BRANCH
  WHERE ACCOUNT.C_BRANCH = BRANCH.BRNO
  AND ACCOUNT.LAST_DATE > CURRENT DATE -
2 YEARS
```

One possible execution strategy consists of the following tasks:

- The slave task scans its partition of ACCOUNT table, applying the predicate. It then joins the resulting relation with its partition of BRANCH table. The slave task then sends the joined rows to the coordinator task.
- The coordinator task merges the results and sends the answer set to the application.

Typical decision-support applications include SQL queries that join relations in a usually predictable way. In those cases, it may be beneficial to partition the relations on their joining columns. This action can reduce the number of data exchange messages required for join processing.

Redirected join: If the ACCOUNT table is partitioned using the CUST_NO attribute rather than the C_BRANCH attribute, a co-located join strategy will generate incorrect results for the SQL query shown above. That is because a row of the ACCOUNT table may join with a row of the BRANCH table that resides on a different partition. Then, DB2 Parallel Edition could generate a redirected join strategy by hashing each selected ACCOUNT table row using the C_BRANCH value and redirecting them to the corresponding BRANCH table row location.

Broadcast join: DB2 Parallel Edition also uses a broadcast join where the selected rows of the ACCOUNT or BRANCH tables are broadcast to the nodes containing the partitions of the other table. This strategy is useful for an index-based join strategy or when the size of one of the joining tables is small.

Redistributed join: When neither the BRANCH nor the ACCOUNT table is partitioned on the joining columns, a redistributed join strategy would be used:

- ACCOUNT table rows are redirected by hashing on the C_BRANCH attribute.
- BRANCH table rows are redirected by hashing on the BRNO attribute.
- Redistributed partitions of ACCOUNT and BRANCH tables can then be joined locally.

One or more of the above join strategies can be viable, based on the data partitioning. The optimizer selects the strategy that minimizes the overall execution cost. The optimization strategy extends elegantly when more than two relations are being joined. For example, a multi-join query execution strategy might involve a mix of co-located, redirected, broadcast, and redistributed joins.

The parallel execution strategies in DB2 Parallel Edition are executed asynchronously. Coordinator involvement is restricted to initializing slave tasks and to collecting final result sets. In a multi-join query, there is no coordinator involvement between joins—all processing is driven by data flow between slave tasks.

Parallel Utilities

The DB2 Parallel Edition provides linear scaleup and speedup for all utilities. Two important utilities that help to manage a database are the data load and rebalance utilities.

Data Load

Data can be loaded into database tables using the DB2/6000 import utility or by using fast loading utilities such as Bridge Fastload or the DB2 optional fast-load program. The import utility loads data by performing `INSERT` operations. DB2 Parallel Edition provides an enhanced `INSERT` in which rows are batched before being sent to the destination processor. With the DB2 Parallel Edition `INSERT`, rows are batched by their destination processor and the batch is sent to that processor when the buffer is full. This new `INSERT` is available to most application programs that use SQL `INSERT` and coordinate restart points using `COMMIT`. It is implemented as an option at bind time. Most programs can use this option without any modification.

Another option for loading data is to directly create database files without using the `INSERT` mechanism. In the parallel database, the best approach is to partition the input data file based on the defined partition key values. This creates one file for each processor that will store the table. Next, each partition is loaded in parallel across the processors. An Application Programming Interface (API) that facilitates this process determines which processor will store a row of a table, based on its partition key value and its nodegroup name. This API can partition a data file into several files—one per processor.

The Rebalance Utility

A system-defined hashing function determines the partitioning key value of a row in a table and the processor on which the row is stored. The processor selected is from the nodegroup in which the table was created. A uniform hashing function distributes data evenly across the set of processors in the nodegroup. Data may need to be moved between processors when a new processor is added to a nodegroup, or when the data distribution across processors is not uniform (because of the skew in the data values in the partitioning key column). The rebalance utility helps to achieve a uniform distribution of data.

The rebalance utility is specified at the nodegroup level. Rebalancing a nodegroup results in the rebalancing of all tables within that nodegroup. Rebalance is an online operation—neither the database system nor the database need to be shut down. Database administrators can specify which data should be moved and where it should be moved.

The DB2 Parallel Edition provides linear scaleup and speedup for all utilities.

For example, suppose a nodegroup consists of 10 processors, numbered 1 to 10. Two processors, numbered 15 and 16 are added. The rebalance utility can move some data to take advantage of the new processors. Data can also be moved from processors 1 through 5 to processor 15, and data from processors 6 through 10 can be moved to processor 16. This can achieve a uniform distribution of data across all processors in the nodegroup.

In another example, suppose a nodegroup consists of four processors numbered 1 to 4. Processor 1 has 40% of the rows, and the other three each have 20%. The rebalance utility can move approximately 15% of the rows from processor 1 and send about 5% each to the other three.

A large rebalancing operation can be broken into smaller steps by rebalancing only part of the data each time. For example, for 50% of the rows to be moved from one processor to another, the move could be done in five separate rebalancing operations, each moving 10% of the data. If large tables need to be rebalanced, the work can be spread over several intervals of low activity.

Other Utilities

This section describes how several functions and utilities are executed in parallel.

Index creation: The parallel capabilities of DB2 Parallel Edition support the creation of

DB2 Parallel Edition can extend the capacity of an online transaction workload beyond a single RISC System/6000.

unique indexes, in which the key columns include the partitioning key columns or non-unique indexes. Creating indexes is done in parallel across processors. A local index is created for each partition of the table stored in a processor.

Reclustering utility (REORG): The REORG utility reclusters the rows of a table on disk. Clustering of each individual table partition at a processor is sufficient to exploit clustered index scan performance. The REORG operation, which executes in parallel across all processors, can provide linear speedup with several processors.

Backup/restore: Each processor uses the DB2/6000 backup utility to back up the segment of the database that is resident at that processor. The backup image includes the processor number. Not all processors have to be backed up at the same time. Backups can be taken in parallel across processors, but this method requires sufficient backup resources, such as tape drives at each processor. All logs must be available at each processor to recover the entire database to a consistent point in time.

DB2/6000 parallel functions interface with the IBM ADSM product that manages backups from multiple RISC System/6000 machines. Since the ADSM interface tracks backups and their associated logs, it enhances the manageability of DB2/6000 in parallel environments.

Forward recovery: Except for the resolution of the final in-doubt transactions, all processors can perform parallel forward recovery by reapplying log entries after a system failure or a restore from backup. DB2 Parallel Edition ensures that all nodes are recovered to the latest log.

Parallel Transaction Processing

DB2 Parallel Edition is a full-function parallel RDBMS. Since it is not limited to queries, it can be used to extend the capacity of an online transaction workload beyond a single RISC System/6000.

Combined with IBM's HACMP/6000, the DB2 Parallel Edition provides concurrent access to a database from all processors in an HACMP/6000 cluster of RISC System/6000s. The DB2 Parallel Edition function-shipping implementation is not limited to a single HACMP cluster—multiple clusters can be joined to support larger databases or higher throughput.

Conclusion

The IBM DB2/6000 parallel database server can efficiently manipulate large amounts of data by partitioning data over several nodes and executing queries in parallel. It provides the following advantages:

- Cost-based optimization
- The best parallel processing strategies
- Efficient, asynchronous execution of subtasks

DB2/6000 also provides efficient transaction-processing capabilities and a suite of parallel utilities for database management.

***Gilles Fecteau**, IBM Software Solutions Laboratory, 1150 Eglinton Avenue East, North York, Ontario M3C 1H7, Canada. Internet: gfecteau@vnet.ibm.com. Mr. Fecteau is one of the key designers of the parallel version of the DB2 workstation products (DB2/2™ and DB2/6000). He has been involved in several advanced technology efforts to bring parallel databases to market. Mr. Fecteau has a Bachelor of Engineering Science degree from Laval University in Quebec City, Quebec.*