

Chapter 1

Introduction

Parallel databases are database systems that are implemented on parallel computing platforms. Therefore, *high-performance query processing* focuses on query processing, including database queries and transactions, that makes use of parallelism techniques applied to an underlying parallel computing platform in order to achieve high performance.

In a Grid environment, applications need to create, access, manage, and distribute data on a very large scale and across multiple organizations. The main challenges arise due to the volume of data, distribution of data, autonomy of sites, and heterogeneity of data resources. Hence, *Grid databases* can be defined loosely as being data access in a Grid environment.

This chapter gives an introduction to parallel databases, parallel query processing, and Grid databases. Section 1.1 gives a brief overview. In Section 1.2, the motivations for using parallelism in database processing are explained. Understanding the motivations is a critical starting point in exploring parallel database processing in depth. This will answer the question of *why* parallelism is necessary in modern database processing.

Once we understand the motivations, we need to know the objectives or the goals of parallel database processing. These are explained in Section 1.3. The objectives will become the main aim of any parallel algorithms in parallel database systems, and this will answer the question of *what* it is that parallelism aims to achieve in parallel database processing.

Once we understand the objectives, we also need to know the various kinds of parallelism forms that are available for parallel database processing. These are described in Section 1.4. The forms of parallelism are the techniques used to achieve the objectives described in the previous section. Therefore, this section answers the questions of *how* parallelism can be performed in parallel database processing.

Without an understanding of the kinds of parallel technology and parallel machines that are available for parallel database processing, our introductory discussion on parallel databases will not be complete. Therefore, in Section 1.5, we introduce various parallel architectures available for database processing.

Section 1.6 introduces Grid databases. This includes the basic Grid architecture for data-intensive applications, and its current technological status is also outlined.

Section 1.7 outlines the components of this book, including parallel query processing, and Grid transaction management.

1.1 A BRIEF OVERVIEW: PARALLEL DATABASES AND GRID DATABASES

In 1965, Intel cofounder Gordon Moore predicted that the number of transistors on a chip would double every 24 months, a prediction that became known popularly as Moore's law. With further technological development, some researchers claimed the number would double every 18 months instead of 24 months. Thus it is expected that the CPU's performance would increase roughly by 50–60% per year. On the other hand, mechanical delays restrict the advancement of disk access time or disk throughput, which reaches only 8–10%. There has been some debate regarding the accuracy of these figures. Disk capacity is also increasing at a much higher rate than that of disk throughput. Although researchers do not agree completely with these values, they show the difference in the rate of advancement of each of these two areas.

In the above scenario, it becomes increasingly difficult to use the available disk capacity effectively. Disk input/output (I/O) becomes the bottleneck as a result of such skewed processing speed and disk throughput. This inevitable I/O bottleneck was one of the major forces that motivated parallel database research. The necessity of storing high volumes of data, producing faster response times, scalability, reliability, load balancing, and data availability were among the factors that led to the development of parallel database systems research. Nowadays, most commercial database management systems (DBMS) vendors include some parallel processing capabilities in their products.

Typically, a parallel database system assumes only a single administrative domain, a homogeneous working environment, and close proximity of data storage (i.e., data is stored in different machines in the same room or building). Below in this chapter, we will discuss various forms of parallelism, motivations, and architectures.

With the increasing diversity of scientific disciplines, the amount of data collected is increasing. In domains as diverse as global climate change, high-energy physics, and computational genomics, the volume of data being measured and stored is already scaling terabytes and will soon increase to petabytes. Data can

be best collected locally for certain applications like earth observation and astronomy experiments. But the experimental analysis must be able to access the large volume of distributed data seamlessly. The above requirement emphasizes the need for Grid-enabled data sources. It should be easy and possible to quickly and automatically install, configure, and disassemble the data sources along with the need for data movement and replication.

The Grid is a heterogeneous collaboration of resources and thus will contain a diverse range of data resources. Heterogeneity in a data Grid can be due to the data model, the transaction model, storage systems, or data types. Data Grids provide seamless access to geographically distributed data sources storing terabytes to petabytes of data with proper authentication and security services.

The development of a Grid infrastructure was necessary for large-scale computing and data-intensive scientific applications. A Grid enables the sharing, selection, and aggregation of a wide variety of geographically distributed resources including supercomputers, storage systems, data sources, and specialized devices owned by different organizations for solving large-scale resource-intensive problems in science, engineering, and commerce. One important aspect is that the resources—computing and data—are owned by different organizations. Thus the design and evolution of individual resources are autonomous and independent of each other and are mostly heterogeneous.

Based on the above discussions, this book covers two main elements, namely, parallel query processing and Grid databases. The former aims at high performance of query processing, which is mainly read-only queries, whereas the latter concentrates on Grid transaction management, focusing on read as well as write operations.

1.2 PARALLEL QUERY PROCESSING: MOTIVATIONS

It is common these days for databases to grow to enormous sizes and be accessed by a large number of users. This growth strains the ability of single-processor systems to handle the load. When we consider a database of 10 terabyte in size, simple processing using a single processor with the capability of processing with a speed of 1 megabyte/second would take 120 days and nights of processing time. If this processing time needs to be reduced to several days or even several hours, parallel processing is an alternative answer.

$$\begin{aligned}
 10 \text{ TB} &= 10 \times 1024 \times 1024 \text{ MB} = 1,048,576 \text{ MB} \\
 10,048,576 \text{ MB} / 1 \text{ MB/sec} &\approx 10,048,576 \text{ seconds} \\
 &\approx 174,760 \text{ minutes} \\
 &\approx 2910 \text{ hours} \\
 &\approx 120 \text{ days and nights}
 \end{aligned}$$

Because of the performance benefits, and also in order to maintain higher throughput, more and more organizations turn to parallel processing. Parallel machines are becoming readily available, and most RDBMS now offer parallelism features in their products.

But what is parallel processing, and why not just use a faster computer to speed up processing?

Computers were intended to solve problems faster than a human being could—this is the reason for their being invented. People continue to want computers to do more and more and to do it faster. The design of computers has now become more complex than ever before, and with the improved circuitry design, improved instruction sets, and improved algorithms to meet the demand for faster response times, this has been made possible by the advances in engineering. However, even with the advances in engineering that produce these complex, fast computers, there are speed limitations. The processing speed of processors depends on the transmission speed of information between the electronic components within the processor, and this speed is actually limited by the speed of light. Because of the advances in technology, particularly fiber optics, the speed at which the information travels is reaching the speed of light, but it cannot exceed this because of the limitations of the medium. Another factor is that, because of the density of transistors within a processor; it can be pushed only to a certain limit.

These limitations have resulted in the hardware designers looking for another alternative to increase performance. Parallelism is the result of these efforts. Parallel processing is the process of taking a large task and, instead of feeding the computer this large task that may take a long time to complete, the task is divided into smaller subtasks that are then worked on simultaneously. Ultimately, this divide-and-conquer approach aims to complete a large task in less time than it would take if it were processed as one large task as a whole. Parallel systems improve processing and I/O speeds by using multiple processors and disks in parallel. This enables multiple processors to work simultaneously on several parts of a task in order to complete it faster than could be done otherwise.

Additionally, database processing works well with parallelism. Database processing is basically an operation on a database. When the same operation can be performed on different fragments of the database, this creates parallelism; this in turn creates the notion of *parallel database processing*.

The driving force behind parallel database processing includes:

- Querying large databases (of the order of terabytes) and
- Processing an extremely large number of transactions per second (of the order of thousands of transactions per second).

Since parallel database processing works at the query or transaction level, this approach views the degree of parallelism as coarse-grained. Coarse-grained parallelism is well suited to database processing because of the lesser complexity of its operations but needs to work with a large volume of data.

1.3 PARALLEL QUERY PROCESSING: OBJECTIVES

The primary objective of parallel database processing is to gain performance improvement. There are two main measures of performance improvement. The first is *throughput*—the number of tasks that can be completed within a given time interval. The second is *response time*—the amount of time it takes to complete a single task from the time it is submitted. A system that processes a large number of small transactions can improve throughput by processing many transactions in parallel. A system that processes large transactions can improve response time as well as throughput by performing subtasks of each transaction in parallel.

These two measures are normally quantified by the following metrics: (i) speed up and (ii) scale up.

1.3.1 Speed Up

Speed up refers to performance improvement gained because of extra processing elements added. In other words, it refers to running a given task in less time by increasing the degree of parallelism. Speed up is a typical metric used to measure performance of read-only queries (data retrieval). Speed up can be measured by:

$$\text{Speed up} = \frac{\text{elapsed time on uniprocessor}}{\text{elapsed time on multiprocessors}}$$

A *linear speed up* refers to performance improvement growing linearly with additional resources—that is, a speed up of N when the large system has N times the resources of the smaller system. A less desirable *sublinear speed up* is when the speed up is less than N . *Superlinear speed up* (i.e., speed up greater than N) is very rare. It occasionally may be seen, but usually this is due to the use of a suboptimal sequential algorithm or some unique feature of the architecture that favors the parallel formation, such as extra memory in the multiprocessor system.

Figure 1.1 is a graph showing linear speed up in comparison with sublinear speed up and superlinear speed up. The resources in the x -axis are normally measured in terms of the number of processors used, whereas the speed up in the y -axis is calculated with the above equation.

Since superlinear speed up rarely happens, and is questioned even by experts in parallel processing, the ultimate goal of parallel processing, including parallel database processing, is to achieve linear speed up. Linear speed up is then used as an indicator to show the efficiency of data processing on multiprocessors.

To illustrate a speed up calculation, we give the following example: Suppose a database operation processed on a single processor takes 100 minutes to complete. If 5 processors are used and the completion time is reduced to 20 minutes, the speed up is equal to 5. Since the number of processors (5 processors) yields the same speed up (speed up = 5), a linear speed up is achieved.

If the elapsed time of the job with 5 processors takes longer, say around 33 minutes, the speed up becomes approximately 3. Since the speed up value is less than the number of processors used, a sublinear speed up is obtained.

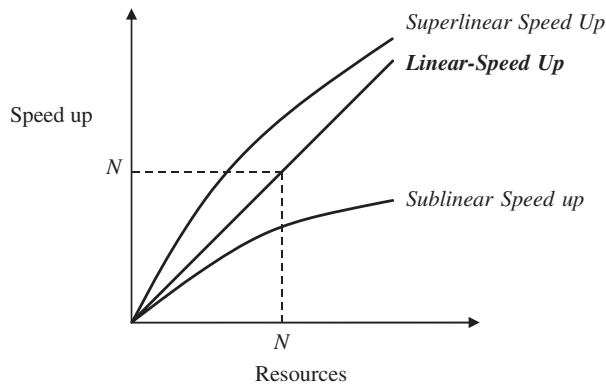


Figure 1.1 Speed up

In an extremely rare case, the elapsed time of the job with 5 processors may be less than 20 minutes—say, for example, 16.5 minutes; then the speed up becomes 6. This is a superlinear speed up, since the speed up (speed up = 6) is greater than the number of processors (processors = 5).

1.3.2 Scale Up

Scale up refers to the handling of larger tasks by increasing the degree of parallelism. Scale up relates to the ability to process larger tasks in the same amount of time by providing more resources (or by increasing the degree of parallelism). For a given application, we would like to examine whether it is viable to add more resources when the workload is increased in order to maintain its performance. This metric is typically used in transaction processing systems (data manipulation). Scale up is calculated as follows.

$$\text{Scale up} = \frac{\text{uniprocessor elapsed time on small system}}{\text{multiprocessor elapsed time on larger system}}$$

Linear scale up refers to the ability to maintain the same level of performance when both the workload and the resources are proportionally added. Using the above scale up formula, scale up equal to 1 is said to be linear scale up. A sub-linear scale up is where the scale up is less than 1. A superlinear scale up is rare, and we eliminate this from further discussions. Hence, linear scale up is the ultimate goal of parallel database processing. Figure 1.2 shows a graph demonstrating linear/sublinear scale up.

There are two kinds of scale up that are relevant to parallel databases, depending on how the size of the task is measured, namely: (i) transaction scale up, and (ii) data scale up.

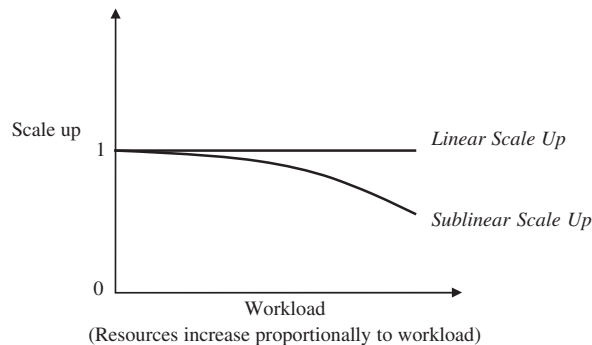


Figure 1.2 Scale up

Transaction Scale Up

Transaction scale up refers to the increase in the rate at which the transactions are processed. The size of the database may also increase proportionally to the transactions' arrival rate.

In transaction scale up, N -times as many users are submitting N -times as many requests or transactions against an N -times larger database. This kind of scale up is relevant in *transaction processing systems* where the transactions are small updates.

To illustrate transaction scale up, consider the following example: Assume it takes 10 minutes to complete 100 transactions on a single processor. If the number of transactions to be processed is increased to 300 transactions, and the number of processors used is also increased to 3 processors, the elapsed time remains the same; if it is 10 minutes, then a linear scale up has been achieved (scale up = 1).

If, for some reason, even though the number of processors is already increased to 3 it takes longer than 10 minutes, say 15 minutes, to process the 300 transactions, then the scale up becomes 0.67, which is less than 1, and hence a sublinear scale up is obtained.

Transaction processing is especially well adapted for parallel processing, since different transactions can run concurrently and independently on separate processors, and each transaction takes a small amount of time, even if the database grows.

Data Scale Up

Data scale up refers to the increase in size of the database, and the task is a large job whose runtime depends on the size of the database. For example, when sorting a table whose size is proportional to the size of the database, the size of the database is the measure of the size of the problem. This is typically found in *online analytical processing (OLAP)* in *data warehousing*, where the fact table is normally very large compared with all the dimension tables combined.

To illustrate data scale up, we use the following example: Suppose the fact table of a data warehouse occupies around 90% of the space in the database. Assume

the job is to produce a report that groups data in the fact table according to some criteria specified by its dimensions.

For example, the processing of this operation on a single processor takes one hour. If the size of the fact table is then doubled up, it is sensible to double up the number of processors. If the same process now takes one hour, a linear scale up has been achieved.

If the process now takes longer than one hour, say for example 75 minutes, then the scale up is equal to 0.8, which is less than 1. Therefore, a sublinear scale up is obtained.

1.3.3 Parallel Obstacles

A number of factors work against efficient parallel operation and can diminish both speed up and scale up, particularly: (i) start up and consolidation costs, (ii) interference and communication, and (iii) skew.

Start Up and Consolidation Costs

Start up cost is associated with initiating multiple processes. In a parallel operation consisting of multiple processes, the start up time may overshadow the actual processing time, adversely affecting speed up, especially if thousands of processes must be started. Even when there is a small number of parallel processes to be started, if the actual processing time is very short, the start up cost may dominate the overall processing time.

Consolidation cost refers to the cost associated with collecting results obtained from each processor by a host processor. This cost can also be a factor that prevents linear speed up.

Parallel processing normally starts with breaking up the main task into multiple subtasks in which each subtask is carried out by a different processing element. After these subtasks have been completed, it is necessary to consolidate the results produced by each subtask to be presented to the user. Since the consolidation process is usually carried out by a single processing element, normally by the host processor, no parallelism is applied, and consequently this affects the speed up of the overall process.

Both start up and consolidation refer to sequential parts of the process and cannot be parallelized. This is a manifestation of the *Amdahl law*, which states that the compute time can be divided into the parallel part and the serial part, and no matter how high the degree of parallelism in the former, the speed up will be asymptotically limited by the latter, which must be performed on a single processing element.

For example, a database operation consists of a sequence of 10 steps, 8 of which can be done in parallel, but 2 of which must be done in sequence (such as start up and consolidation operations). Compared with a single processing element, an 8-processing element machine would attain a speed up of not 8 but somewhere around 3, even though the processing element cost is 8 times higher.

To understand this example, we need to use some sample figures. Assume that 1 step takes 1 minute to complete. Using a single processor, it will take 10 minutes, as there are 10 steps in the operation. Using an 8-processor machine, assume each step is allocated into a separate processor and it takes only 1 minute to complete the parallel part. However, the two sequential steps need to be processed by a single processor, and it takes 2 minutes. In total, it takes 3 minutes to finish the whole job using an 8-processor machine. Therefore, the speed up is 3.33, which is far below the linear speed up (speed up = 8). This example illustrates how the sequential part of the operations can jeopardize the performance benefit offered by parallelism.

To make matters worse, suppose there are 100 steps in the operation, 20 of which are sequential parts. Using an 80-processor machine, the speed up is somewhat under 5, far below the linear speed up of 80. This can be proven in a similar manner.

Using a single-processor machine, the 100-step job is completed in 100 minutes. Using an 80-processor machine, the elapsed time is 21 minutes (20 minutes for the sequential part and 1 minute for the parallel part). As a result, the speed up is equal to 4.8 (speed up = $100/21 = 4.76$). Figure 1.3 illustrates serial and parallel parts in a processing system.

Interference and Communication

Since processes executing in a parallel system often access shared resources, a slowdown may result from the *interference* of each new process as it competes with existing processes for commonly held resources. Both speed up and scale up are affected by this phenomenon.

Very often, one process may have to communicate with other processes. In a synchronized environment, the process wanting to *communicate* with others may be forced to wait for other processes to be ready for communication. This waiting time may affect the whole process, as some tasks are idle waiting for other tasks.

Figure 1.4 gives a graphical illustration of the waiting period incurred during the communication and interference among parallel processes. This illustration uses the example in Figure 1.3. Assume there are four parallel processes. In Figure 1.4, all parallel processes start at the same time after the first serial part has been

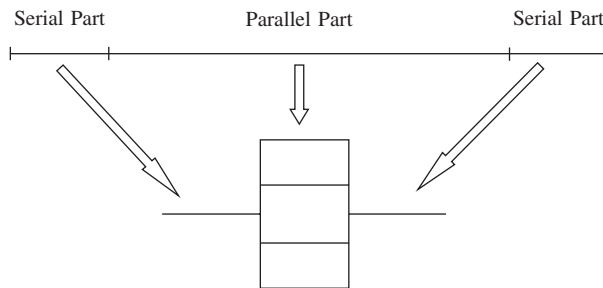


Figure 1.3 Serial part vs. parallel part

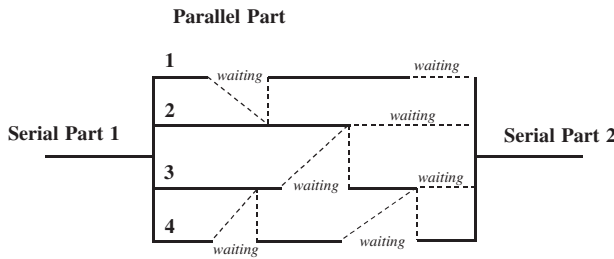


Figure 1.4 Waiting period

completed. After parallel part 1 has been going for a while, it needs to wait until parallel part 2 reaches a certain point in the future, after which parallel process 1 can continue. The same thing happens to parallel part 4, which has to wait for parallel part 3 to reach a certain point. The latter part of parallel part 4 also has to wait for parallel part 3 to completely finish. This also happens to parallel part 3, which has to wait for parallel part 2 to be completed. Since parallel part 4 finishes last, all other parallel parts have to wait until the final serial part finishes off the whole operation. All the waiting periods and their parallel part dependencies are shown in Figure 1.4 by dashed lines.

Skew

Skew in parallel database processing refers to the unevenness of workload partitioning. In parallel systems, equal workload (load balance) among all processing elements is one of the critical factors to achieve linear speed up. When the load of one processing element is heavier than that of others, the total elapsed time for a particular task will be determined by this processing element, and those finishing early would have to wait. This situation is certainly undesirable.

Skew in parallel database processing is normally caused by uneven data distribution. This is sometimes unavoidable because of the nature of data that is not uniformly distributed. To illustrate a skew problem, consider the example in Figure 1.5. Suppose there are four processing elements. In a uniformly distributed workload (Fig. 1.5(a)), each processing element will have the same elapsed time, which also becomes the elapsed time of the overall process. In this case, the elapsed time is t_1 . In a skewed workload distribution (Fig. 1.5(b)), one or more processes finish later than the others, and hence, the elapsed time of the overall process is determined by the one that finishes last. In this illustration, processor 2 finishes at t_2 , where $t_2 > t_1$, and hence the overall process time is t_2 .

1.4 FORMS OF PARALLELISM

There are many different forms of parallelism for database processing, including (i) interquery parallelism, (ii) intraquery parallelism, (iii) interoperation parallelism, and (iv) intraoperation parallelism.

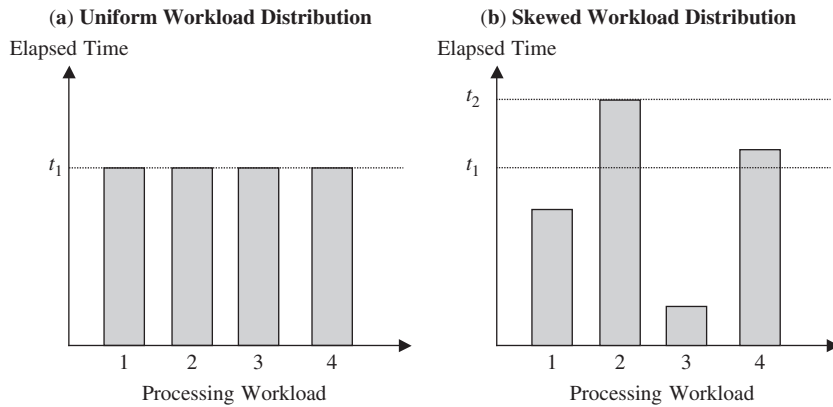


Figure 1.5 Balanced workload vs. unbalanced workload (skewed)

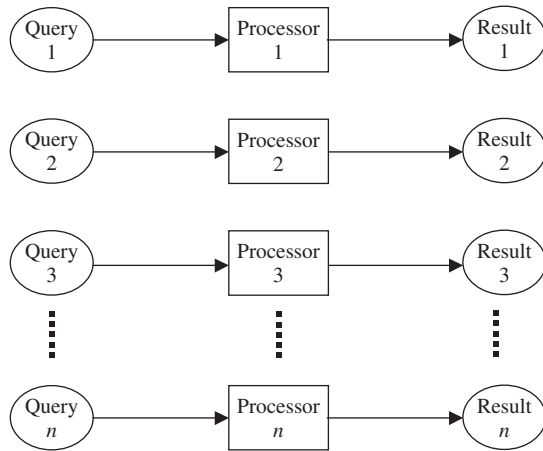


Figure 1.6 Interquery parallelism

1.4.1 Interquery Parallelism

Interquery parallelism is “parallelism among queries”—that is, different queries or transactions are executed in parallel with one another. The primary use of interquery parallelism is to scale up transaction processing systems (i.e., transaction scale up) in supporting a large number of transactions per second.

Figure 1.6 gives a graphical illustration of interquery parallelism. Each processor processes a query/transaction independently of other processors. The data that each query/transaction uses may be from the same database or from different databases.

In comparison with single-processor database systems, these queries/transactions will form a queue, since only one query/transaction can be processed at any given time, resulting in longer completion time of each query/transaction, even

though the actual processing time might be very short. With interquery parallelism, the waiting time of each query/transaction in the queue is reduced, and subsequently the overall completion time is improved.

It is clear that transaction throughput can be increased by this form of parallelism, by employing a high degree of parallelism through additional processing elements, so that more queries/transactions can be processed simultaneously. However, the response time of individual transactions is not necessarily faster than it would be if the transactions were run in isolation.

1.4.2 Intraquery Parallelism

A query to a database, such as sort, select, project, join, etc, is normally divided into multiple operations. *Intraquery parallelism* is an execution of a single query in parallel on multiple processors and disks. In this case, the multiple operations within a query are executed in parallel. Therefore, intraquery parallelism is “parallelism within a query.”

Use of intraquery parallelism is important for speeding up long-running queries. Interquery parallelism does not help in this task, since each query is run sequentially.

Figure 1.7 gives an illustration of an intraquery parallelism. A user invokes a query, and in processing this, the query is divided into n subqueries. Each subquery is processed on a different processor and produces subquery results. The results obtained with each processor need to be consolidated in order to generate final query results to be presented to the user. In other words, the final query results are the amalgamation of all subquery results.

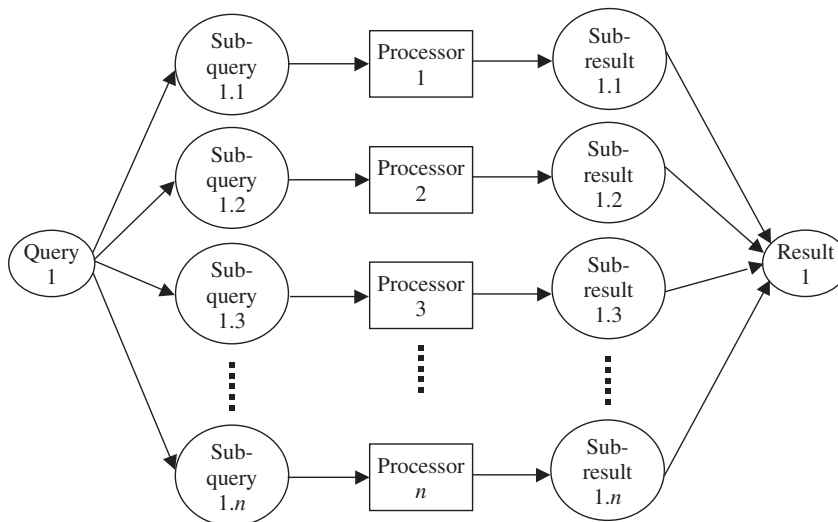


Figure 1.7 Intraquery parallelism

Execution of a single query can be parallelized in two ways:

- *Intraoperation parallelism.* We can speed up the processing of a query by parallelizing the execution of each individual operation, such as parallel sort, parallel search, etc.
- *Interoperation parallelism.* We can speed up the processing of a query by executing in parallel the different operations in a query expression, such as simultaneously sorting and searching.

1.4.3 Intraoperation Parallelism

Since database operations work on tables containing large data sets of records, we can parallelize the operations by executing them in parallel on different subsets of the table. Hence, intra-operation parallelism is often called *partitioned parallelism*—that is, parallelism due to the data being partitioned.

Since the number of records in a table can be large, the degree of parallelism is potentially enormous. Consequently, intra-operation parallelism is natural in database systems.

Figure 1.8 gives an illustration of intraoperation parallelism. This is a continuation of the previous illustration of intraquery parallelism. In intraoperation parallelism, an operation, which is a subset of a subquery, works on different data fragments to create parallelism. This kind of parallelism is also known as “Single Instruction Multiple Data” (SIMD), where the same instruction operation works on different parts of the data.

The main issues of intraoperation parallelism are (i) how the operation can be arranged so that it can perform on different data sets, and (ii) how the data is partitioned in order for an operation to work on it. Therefore, in database processing, intraoperation parallelism raises the need for formulating parallel versions of basic sequential database operations, including: (i) parallel search, (ii) parallel sort, (iii) parallel group-by/aggregate, and (iv) parallel join. Each of these parallel algorithms will be discussed in the next few chapters.

1.4.4 Interoperation Parallelism

Interoperation parallelism is where parallelism is created by concurrently executing different operations within the same query/transaction. There are two forms of interoperation parallelism: (i) *pipelined parallelism* and (ii) *independent parallelism*.

Pipeline Parallelism

In pipelining, the output records of one operation *A* are consumed by a second operation *B*, even before the first operation has produced the entire set of records

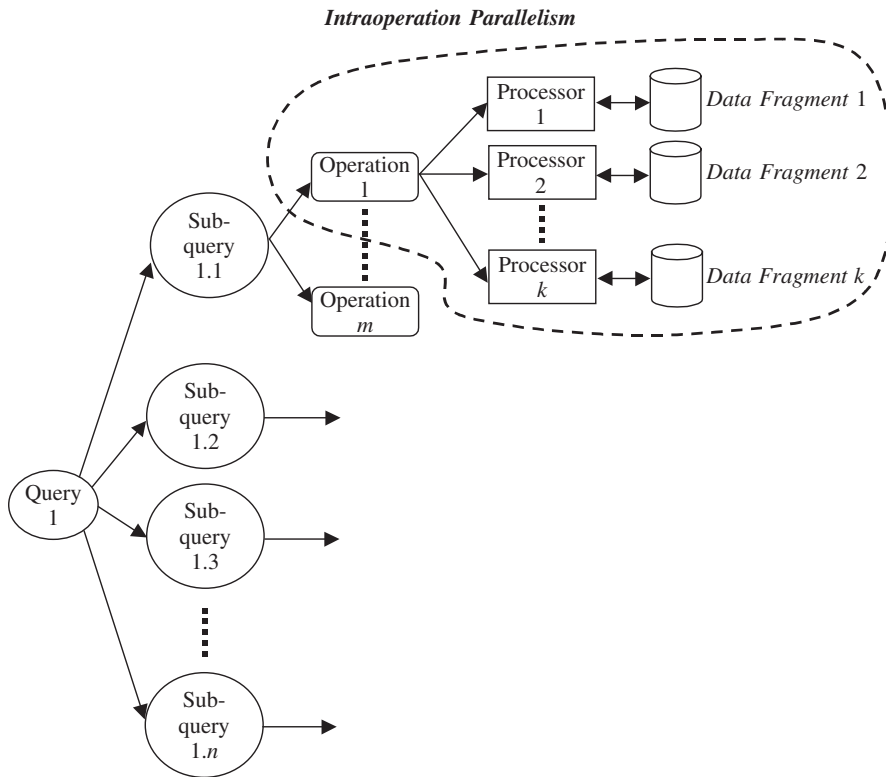


Figure 1.8 Intraoperation parallelism

in its output. It is possible to run A and B simultaneously on different processors, such that B consumes records in parallel with A producing them.

Pipeline parallelism is influenced by the practice of using an assembly line in the manufacturing process. In parallel database processing, multiple operations form some sort of assembly line to manufacture the query results.

The major advantage of pipelined execution is that we can carry out a sequence of such operations without writing any of the intermediate results to disk.

Figure 1.9 illustrates pipeline parallelism, where a subquery involving k operations forms in a pipe. The results from each operation are passed through the next operation, and the final operation will produce the final query results.

Bear in mind that pipeline parallelism is not sequential processing, even though the diagram seems to suggest this. Each operation works with a volume of data. The operation takes one piece of data at a time, processes it, and passes it to the next operation. Each operation does not have to wait to finish processing all data allocated to it before passing them to the next operation. The latter is actually a sequential processing. To emphasize the difference between sequential and

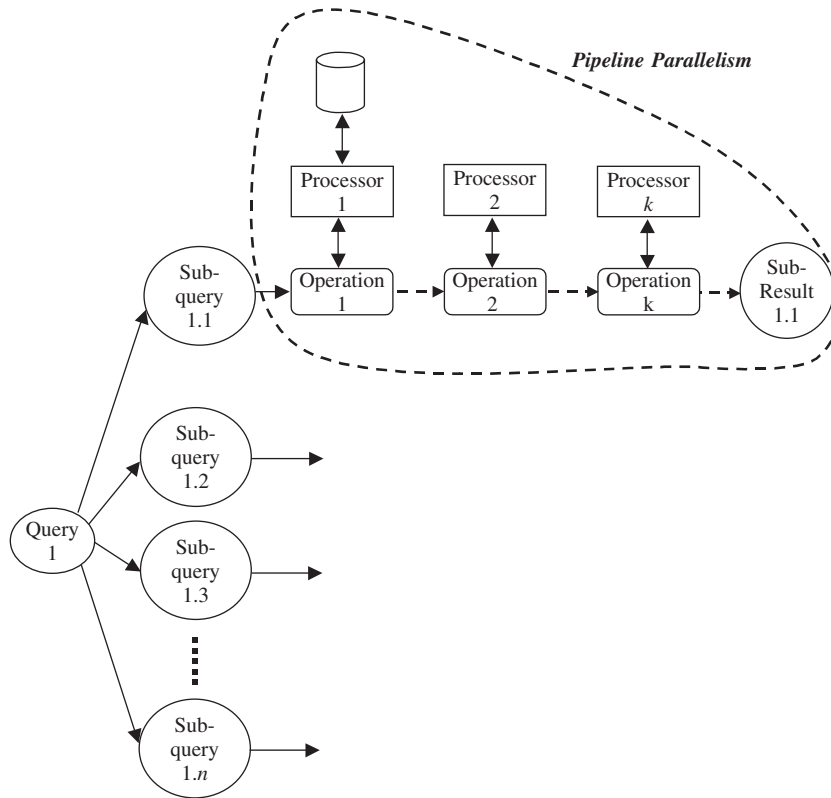


Figure 1.9 Pipeline parallelism

pipeline parallelism, we use a dotted arrow to illustrate pipeline parallelism, showing that each piece of data is passed through the pipe as soon as it has been processed.

Pipelined parallelism is useful with a small number of processors but does not scale up well for various reasons:

- Pipeline chains generally do not attain sufficient length to provide a high degree of parallelism. The degree of parallelism in pipeline parallelism depends on the number of operations in the pipeline chain. For example, a subquery with 8 operations forms an assembly line with 8 operators, and the maximum degree of parallelism is therefore equal to 8. The degree of parallelism is then severely limited by the number of operations involved.
- It is not possible to pipeline those operators that do not produce output until all inputs have been accessed, such as the set-difference operation. Some operations simply cannot pass temporary results to the next operation without having fully completed the operation. In short, not all operations are suitable for pipeline parallelism.

- Only marginal speed up is obtained for the frequent cases in which one operator's execution cost is much higher than that of the others. This is particularly true when the speed of each operation is not uniform. One operation that takes longer than the next operation will regularly require the subsequent operation to wait, resulting in a lower speed up. In short, pipeline parallelism is suitable only if all operations have uniform data unit processing time.

Because of the above limitations, when the degree of parallelism is high, the importance of pipelining as a source of parallelism is secondary to that of partitioned parallelism.

Independent Parallelism

Independent parallelism is where operations in a query that do not depend on one another can be executed in parallel, for example, Table 1 *join* Table 2 *join* Table 3 *join* Table 4. In this case, we can process Table 1 *join* Table 2 in parallel with Table 3 *join* Table 4.

Figure 1.10 illustrates independent parallelism. Multiple operations are independently processed in different processors accessing different data fragments.

Like pipelined parallelism, independent parallelism does not provide a high degree of parallelism, because of the possibility of a limited number of independent operations within a query, and is less useful in a highly parallel system, although it is useful with a lower degree of parallelism.

1.4.5 Mixed Parallelism – A More Practical Solution

In practice, a mixture of all available parallelism forms is used. For example, a query joins 4 tables, namely, Table 1, Table 2, Table 3, and Table 4. Assume that the order of the join is Table 1 joins with Table 2 and joins with Table 3 and finally joins with Table 4. For simplicity, we also assume that the join attribute exists in the two consecutive tables. For example, the first join attribute exists in Table 1 and Table 2, and the second join attribute exists in Table 2 and Table 3, and the last join attribute exists in Table 3 and Table 4. Therefore, these join operations may form a bushy tree, as well as a left-deep or a right-deep tree.

A possible scenario for parallel processing of such a query is as follows.

- *Independent parallelism:*

The first join operation between Table 1 and Table 2 is carried out in parallel with the second join operation between Table 3 and Table 4.

Result1 = Table 1 join Table 2, in parallel with

Result2 = Table 3 join Table 4.

- *Pipelined parallelism:*

Pipeline Result1 and Result2 into the computation of the third join. This means that as soon as a record is formed by the first two join operations (e.g., Result1 and Result2), it is passed to the third join, and the third join can start

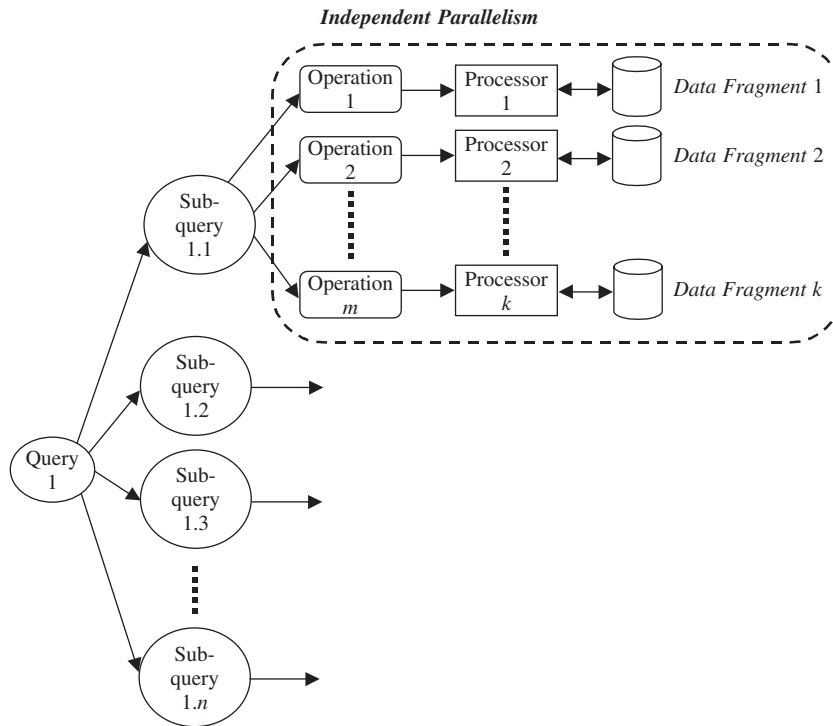


Figure 1.10 Independent parallelism

the operation. In other words, the third join operation does not wait until the first two joins produce their results.

- *Intraoperation parallelism:*

Each of the three join operations above is executed with a partitioned parallelism (i.e., parallel join). This means that each of the join operations is by itself performed in parallel with multiple processors.

Figure 1.11 gives a graphical illustration of a mixed parallelism of the above example.

1.5 PARALLEL DATABASE ARCHITECTURES

The motivation for the use of parallel technology in database processing is influenced not only by the need for performance improvement, but also by the fact that parallel computers are no longer a monopoly of supercomputers but are now in fact available in many forms, such as systems consisting of a small number but powerful processors (e.g., SMP machines), clusters of workstations (e.g., loosely

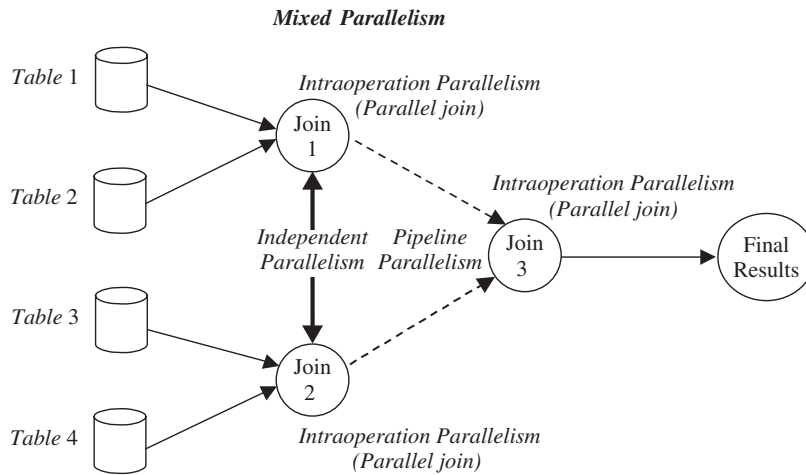


Figure 1.11 Mixed parallelism

coupled shared-nothing architectures), massively parallel processors (MPP), and clusters of SMP machines (i.e., hybrid architectures).

It is common for parallel architectures especially used for data-intensive applications to be classified according to several categories: (i) *shared-memory*, (ii) *shared-disk*, (iii) *shared-nothing*, and (iv) *shared-something*.

1.5.1 Shared-Memory and Shared-Disk Architectures

Shared-memory architecture is an architecture in which all processors share a common main memory and secondary memory. When a job (e.g., query/transaction) comes in, the job is divided into multiple slave processes. The number of slave processes does not have to be the same as the number of processors available in the system. However, normally there is a correlation between the maximum number of slave processes and the number of processors. For example, in Oracle 8 parallel query execution, the maximum number of slave processes is $10 \times$ number of CPUs.

Since the memory is shared by all processors, processor load balancing is relatively easy to achieve, because data is located in one place. Once slave processes have been created, each of them can then request the data it needs from the central main memory. The drawback of this architecture is that it suffers from memory and bus contention, since many processors may compete for access to the shared data. Shared-memory architectures normally use a bus interconnection network. Since there is a limit to the capacity that a bus connection can handle, data/message transfer along the bus can be limited, and consequently it can serve only a limited number of processors in the system. Therefore, it is quite common for a

shared-memory machine to be equipped with no more than 64 processors in a computer system box.

In *shared-disk* architecture, all processors, each of which has its own local main memory, share the disks. Shared-disk architecture is very similar to shared-memory architecture in the sense that the secondary memory is shared, not the main memory.

Because of the local main memory in each processor that keeps active data, data sharing problems can be minimized and load balancing can largely be maintained. On the other hand, this architecture suffers from congestion in the interconnection network when many processors are trying to access the disks at the same time.

The way a job is processed is also very similar. Once slave processes have been created, each process requests the data to be loaded from the shared disk. Once the data is loaded, it is kept in its processor's main memory. Therefore, the main difference between shared-disk and shared-memory architecture is the memory hierarchy that is being shared. From a memory hierarchy point of view, shared-memory and shared-disk architecture share the same principle. In shared-memory architecture, although "everything" (e.g., main memory and secondary memory) seems to be shared, each processor may have its own cache, which nowadays can be quite large (e.g., up to 4 megabytes). If we then assume that this cache acts similarly to main memory in a shared-disk architecture, then the difference between these two architectures is narrowed.

In the context of the computing platform, *shared-memory* and *shared-disk* architectures are normally found in *Symmetric Multi Processor (SMP)* machines. A typical SMP machine consists of several CPUs, ranging from 2 to 16 CPUs. A larger number of CPUs is not too common because of the scaling up limitation. Each CPU maintains its own cache, and the main-memory is shared among all the CPUs. The sizes of main-memory and caches may vary from machine to machine. Multiple disks may be attached to an SMP machine, and all CPUs have the same access to them. The operating system normally allocates tasks according to the schedule. Once a processor is idle, a task in the queue will be immediately allocated to it. In this way, balancing is relatively easy to achieve. Figure 1.12 gives an illustration of an SMP architecture.

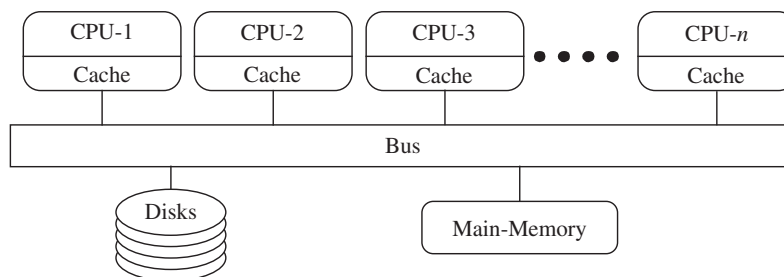


Figure 1.12 An SMP architecture

1.5.2 Shared-Nothing Architecture

A *shared-nothing* architecture provides each processor with a local main memory and disks. The problem of competing for access to the shared data will not occur in this system, but load balancing is difficult to achieve even for simple queries, since data is placed locally in each processor and each processor may have an unequal load. Because each processor is independent of others, it is often claimed that scaling up the number of processors without adversely affecting performance is achievable.

The way a shared-nothing architecture is used in parallel database processing is that when a job comes in, it comes into a processor in the system. Depending on the type of machine, this processor might be a host processor. In other cases, there is the notion of a host processor, meaning that any processor can receive a job. The processor that receives the job then splits the job into multiple processes to be allocated to each processor. Once each processor has its share of a piece of the job, it loads the data from its own local disk and starts (e.g., computation or data distribution when required) processing it.

The load imbalance problem is not only due to the size of local data in each processor that might be different from that of other processors, but also due to the need for data redistribution during processing of the job. Data redistribution is normally based on some distribution function, which is influenced by the value of the data, and this may further create workload imbalance. The skewness problem has been a major challenge in database processing using a shared-nothing architecture.

Shared-nothing architecture ranges from the workstation farm to *Massively Parallel Processing (MPP)* machines. The range is basically divided by the speed of the network, which connects the processing units (i.e., CPUs containing primary and secondary memory). For a workstation farm, the network is a slower Ethernet; whereas for MPP, the interconnection is done via a fast network or system bus. Whether it be a slow or fast network, the processing units communicate among each other via the network, as they do not share common data storage (i.e., main memory or secondary memory). Because the data storage is not shared but localized, shared-nothing architecture is often called *distributed-memory* architecture. Figure 1.13 shows a typical shared-nothing architecture.

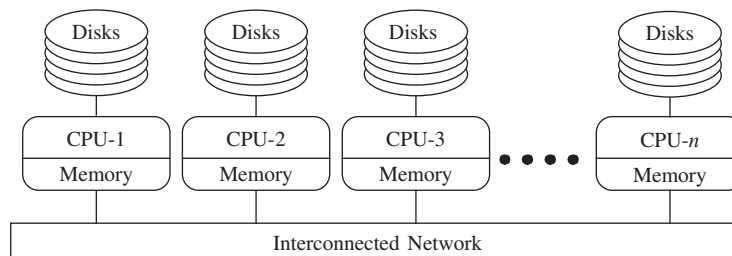


Figure 1.13 A shared-nothing architecture

1.5.3 Shared-Something Architecture

A *shared-something* architecture compromises the extensibility limitation of shared-memory architecture and the load balancing problem of shared-nothing architecture. This architecture is a mixture of shared-memory and shared-nothing architectures. There are a number of variations to this architecture, but basically each *node* is shared-memory architecture connected to an interconnection network a la shared-nothing architecture. As each shared-memory (i.e., SMP machine) maintains a group of processing elements, a collection of these groups is often called a “*cluster*,” which in this case means *clusters of SMP* architecture. Figure 1.14 shows the architecture of clusters of SMP.

Obvious features of a shared-something architecture include flexibility in the configuration (i.e., number of nodes, size of nodes) and lower network communication traffic as the number of nodes is reduced. Intraquery parallelization can be isolated to a single multiprocessor shared-memory node, as it is far easier to parallelize a query in a shared-memory than in a distributed system and moreover, the degree of parallelism on a single shared-memory node may be sufficient for most applications. On the other hand, interquery parallelization is consequently achieved through parallel execution among nodes.

The popularity of cluster architectures is also influenced by the fact that processor technology is moving rapidly. This also means that a powerful computer today will be out of date within a few years. Consequently, computer prices are falling not only because of competitiveness but also because of the above facts. Therefore, it becomes sensible to be able to plug in new processing elements to the current system and to take out the old ones. To some degree, this can be done to an SMP machine, considering its scaling limitations and that only identical processors can be added into it. With MPP machines, although theoretically not imposing scaling limitations, their configurations are difficult to alter, and hence they cannot keep up with up-to-date technology, despite the high price of MPP machines. On the other hand, SMP machines are becoming popular because of

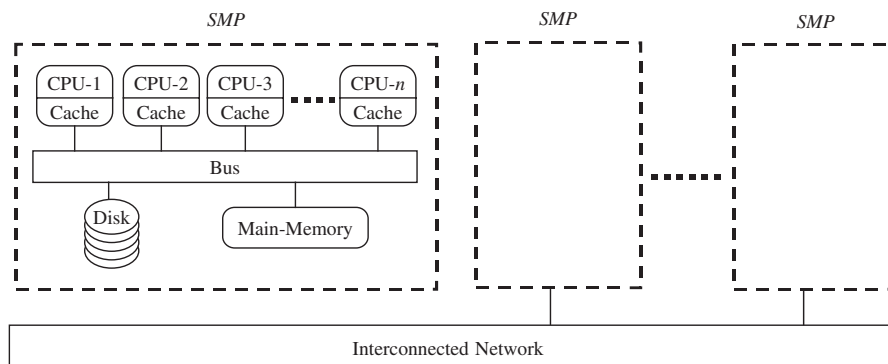


Figure 1.14 Cluster of SMP architectures

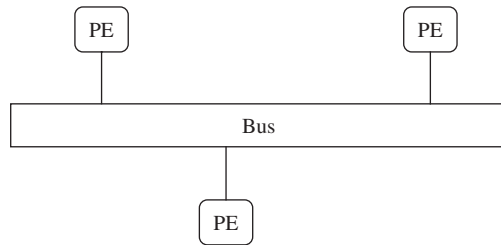


Figure 1.15 Bus interconnection network

their competitiveness in relative pricing and power, so it becomes easier and more feasible to add SMP machines to an interconnection network. Therefore, the cluster of SMP becomes demanding.

1.5.4 Interconnection Networks

Regardless of the kind of architectures, when several processors are connected, they are connected in an interconnection network. There are several types of interconnection networks for parallel computing systems. In this section, we cover the three most popular interconnection networks, namely, (i) bus, (ii) mesh, and (iii) hypercube.

Bus

A bus can be illustrated as a single communication line, which connects a number of processing elements (*PEs*). We use the notion of processing element here, as it can be a single processor or a processor with its own local memory. Figure 1.15 gives an illustration of a bus interconnection network.

With a bus interconnection network, all the system components can send data on and receive data from a single communication bus. Bus architectures work well for small numbers of processors. However, they do not scale well with increasing parallelism since the bus can handle communication from only one component at a time. A bus interconnection network is normally used in the SMP architectures.

Mesh

The processing elements are arranged as nodes in a Grid, and each processing element is connected to all its adjacent components in the Grid. Figure 1.16 gives an illustration of a mesh structure. In this example, the mesh is a 3×4 structure.

Processing elements that are not directly connected can communicate with one another by routing messages via a sequence of intermediate nodes that are directly connected to one another. For example, the processing element at the top left-hand corner wanting to send a message to the processing element at the bottom right hand corner in Figure 1.16 must pass through several intermediate processing elements.

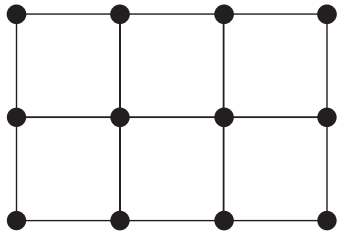


Figure 1.16 Mesh interconnection network

The number of communication links grows as the number of components grows, and the communication capacity of a mesh therefore scales better with increasing parallelism.

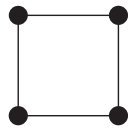
Hypercube

The processing elements are numbered in binary, and a processing element is connected to another if the binary representations of their numbers differ in exactly one. Thus each of the n components is connected to $\log(n)$ other components.

Figure 1.17 gives two examples of a hypercube interconnection network. In a 2-dimensional hypercube, it is exactly the same as a 2×2 mesh structure. In a 3-dimensional hypercube, the processing elements are connected as in a cube. Theoretically, the degree of dimension in a hypercube can be arbitrary. To visualize a higher degree of dimension, a 4-dimensional hypercube, it looks like a cube inside a bigger cube (possibly like a room), in which the corner of the inner cube is connected with the corner of the outer cube. In a 5-dimensional hypercube, the 4-dimensional hypercube is inside another bigger cube, and so on.

In a hypercube connection, a message from a processing element can reach any other processing element by going via at most $\log(n)$ links, where n is the number of processing elements connected in a hypercube interconnection network. For example, in a 2-dimensional hypercube, there are 4 processing elements, and

2-dimensional



3-dimensional

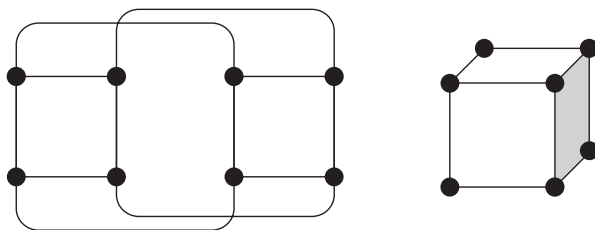


Figure 1.17 Hypercube interconnection network

therefore the longest route from one processing element to another is equal to $\log(4) = 2$. In a 3-dimensional hypercube connecting 8 processing elements, the longest route takes only 3 links. It can be concluded that the degree of the dimension also determines the longest route from one processing element to the other.

1.6 GRID DATABASE ARCHITECTURE

Although the Grid technology has attracted much research attention during the last decade, it is only recently that data management in Grids including data Grid and Grid databases have attracted research attention. This was because the underlying middleware architecture, such as Globus, Legion, etc., had not been established.

Figure 1.18 shows the general architecture of databases working in a Grid environment. The Grid database architecture typically works in a wide area, spanning multiple institutions, and has autonomous and heterogeneous environment. Global data-intensive collaborative applications such as earth observation and simulation, weather forecasting, functional genomics, etc. will need to access geographically distributed data without being aware of the exact location of interesting data.

Various Grid services, for example, metadata repository services, lookup services, replica management services, etc., shown in Figure 1.18 will help to provide seamless access to data. Accounting services will be helpful in calculating the cost of using the resources held by other organizations. Authorization and security services will set the level of authorization and security required to access other organizations' resources. This depends on the extent to which the organizations are trusted. Various trust models have been developed for accessing resources. Apart from basic services, other application-specific services can be developed and plugged into the middleware. All these services together constitute the Grid middleware.

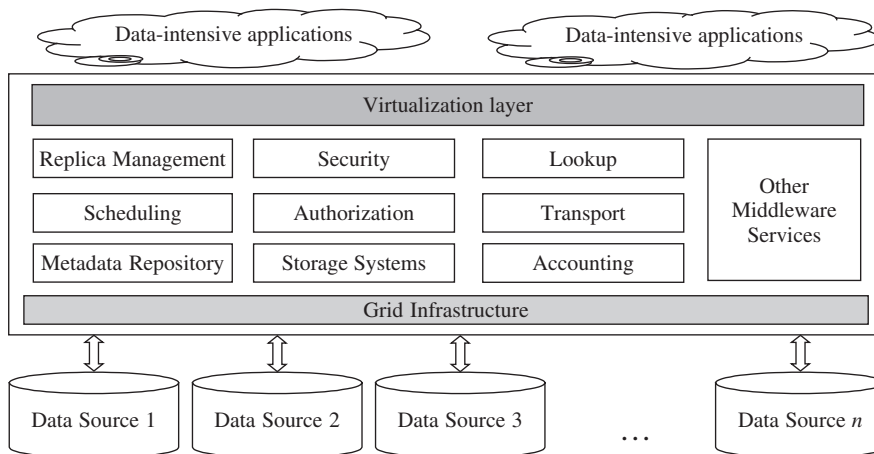


Figure 1.18 Data-intensive applications working in Grid database architecture

Grid middleware provides a complete suite of software for delivering various services, such as resource discovery, metadata management, job and queue management, security, encryption, authentication, file transfer, remote process management, storage access, Quality of Service, replica and coherency control, etc. Middleware toolkits are being developed to provide high-level services. The Globus toolkit is the most accepted Grid middleware, and is still evolving rapidly.

The middleware infrastructure is still evolving, but there is an understanding of the basic functionalities of existing middleware in the form of Globus, Condor, Legion, etc. Recent progress in data management in Grids is evident from the efforts of the Data Access and Integration Working-Group (DAIS-WG) and the Transaction Management Research-Group (TM-RG) of the GGF, a standardization body for Grids.

Apart from core Grid services mentioned above, user-level Grid middleware consists of programming tools and application development environment for managing global resources with the help of resource brokers. The transaction management protocols are lower-level correctness protocols, and they use some of the higher-level services from the middleware.

Most of the efforts in data management in Grid architecture have been focused on providing high-level services for locating the data, efficient access methods, transporting the data, making the replication decision, data caching, etc. This book introduces lower-level traditional transactional requirements and demonstrates how the Grid infrastructure affects the working of individual sites. It focuses on maintaining data consistency at individual data sites when these sites join the Grid infrastructure to make resources available to a wide and collaborative environment, while maintaining autonomy. It explores the lower-level transactional requirements of Grid databases rather than providing a higher-level service to the user.

All resources, for example, computational resources, storage resources, network, programs, etc., are represented as service in Open Grid Service Architecture (OGSA). A service-oriented approach is also used for the abstraction of underlying data stored in different types of data resources. *Data virtualization* is used to denote such abstraction of data resources. The Grid Data Distribution (GDD) model supports dynamic and efficient data distribution for providing data services. Figure 1.19 shows the data virtualization approach. The interfaces shown are OGSA's data service interface.

Data virtualization will help in accessing data stored on different physical media, maintained in different syntaxes, managed by different software systems, and made available by different protocols. Attempts to standardize the virtualization approach are being made by various Research- and Working-Groups of the GGF.

Referring to Figure 1.19, from the data management perspective, an important point is that Grids do not have Direct Attached Storage (DAS). Earlier computing infrastructures, such as monolithic, open, and distributed, all have DAS. A research survey carried out on storage systems shows that about 53% of businesses use DAS as their storage medium; 58% of organizations consider interoperability to be a

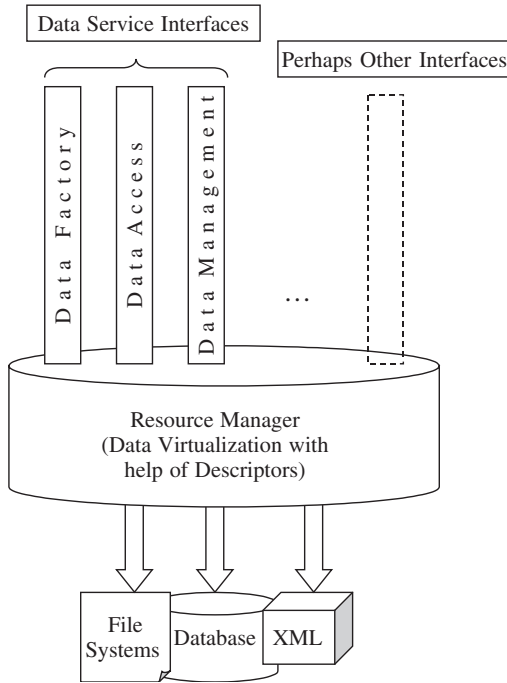


Figure 1.19 Data virtualization approach in Grid environment

critical issue. While interoperability and DAS have contradictory requirements, the good news is that more than 60% of organizations are considering moving toward network-based storage systems. This will make the integration of data sources convenient with the Grid.

Most of the work done in Data Grid infrastructure assumes the existence of file systems like Network File System (NFS) for data storage. Considering the global vision of Grids, it is believed that Grids must also integrate database systems into the infrastructure to support a wide range of applications. Hence, databases offer a much richer set of operations such as queries and transactions.

Oracle has launched its Grid-enabled database systems, denoted with a suffix *g* in its versions. Oracle's Real Application Cluster (RAC) can run the application workload on a cluster of servers. Two main distinct features are (i) integrated clusterware and (ii) automatic workload management. Integrated clusterware provides cluster connectivity, messaging and locking, recovery, etc. Automatic workload management provides the dynamic allocation of workloads to the servers. Allocation rules can be dynamically and automatically defined to allocate processing resources. Thus the main focus is on providing dynamic load balancing and allocation of workloads. It is greatly different from the autonomous, heterogeneous, and cross-institution working environment, or DAIS-WG or TM-RG. The earlier versions were not designed for heterogeneous and global Grids, but for intra-institutional dynamic workload management.

1.7 STRUCTURE OF THIS BOOK

This book covers two main elements, namely:

1. *Parallel Query Processing* and
2. *Grid Transaction Management*

The first element on parallel query processing mainly deals with read-only queries, whereas the second element on Grid databases deals with read as well as write transactions.

The book is structured in four parts: **Part I** gives an introduction to the topic, including this chapter. Since an analytical model is an important part of performance evaluation of any system, an introduction of analytical models will also be included in this part.

Part II and **Part III** concentrate on *Parallel Query Processing*. These parts feature parallel algorithms and approaches for all important database processing operations. This ranges from the very basic database operations, such as data searching and sorting, to the most complex database operations involving complex database computation, like universal quantifier, complex join, etc. Understanding these algorithms is critical in order to fully comprehend how parallel database processing works and enhances the performance of modern database applications. Basically, Part II describes the basic of query parallelism including parallel search, parallel sort, and parallel join, whereas Part III focuses on more complex query parallelism, such as groupby-join, indexing, universal quantification, and scheduling.

An understanding of the pseudocode of parallel algorithms is not enough to fully comprehend the behavior of each parallel algorithm and its contribution to performance improvement. We need to be able to describe their cost models. Analytical models for each parallel algorithm can be used to understand the internal components of each algorithm, to predict the performance of each algorithm, and to compare with other algorithms. Extensive cost models are also included in each chapter to describe the behavior of each parallel query processing method.

Part IV focuses on the second element of this book, namely, *Grid Transaction Management*. This covers the ACID properties of transactions as well as replication in a Grid environment. Transaction issues such as consistency, atomicity and recovery, which are more relevant in a database environment, are brought into focus. Data consistency issues are addressed in the presence of write transactions.

The final part, **Part V**, presents other data intensive applications in which parallelism might be applied in order to achieve high performance. These include parallel Online Analytical Processing (parallel OLAP) and parallel data mining techniques. The use of parallelism in these data-intensive applications is unavoidable because of the large volume of data to be processed.

1.8 SUMMARY

This chapter focuses on three fundamental questions in parallel query processing, namely, *why*, *what*, and *how*, plus one additional question based on the technological support. The more complete questions and their answers are summarized as follows.

- *Why is parallelism necessary in database processing?*
Because there is a large volume of data to be processed and reasonable (improved) elapsed time for processing this data is required.
- *What can be achieved by parallelism in database processing?*
The objectives of parallel database processing are (i) linear speed up and (ii) linear scale up. Superlinear speed up and superlinear scale up may happen occasionally, but they are more of a side effect, rather than the main target.
- *How is parallelism performed in database processing?*
There are four different forms of parallelism available for database processing: (i) interquery parallelism, (ii) intraquery parallelism, (iii) intraoperation parallelism, and (iv) interoperation parallelism. These may be combined in parallel processing of a database job in order to achieve a better performance result.
- *What facilities of parallel computing can be used?*
There are four different parallel database architectures: (i) shared-memory, (ii) shared-disk, (iii) shared-nothing, and (iv) shared-something architectures.

Distributed computing infrastructure is fast evolving. The architecture was monolithic in 1970s, and since then, during the last three decades, developments have been exponential. The architecture has evolved from monolithic, to open, to distributed, and lately virtualization techniques are being investigated in the form of Grid computing. The idea of Grid computing is to make computing a commodity. Computer users should be able to access the resources situated around the globe without knowing the location of the resource. And a pay-as-you-go strategy can be applied in computing, similar to the state-of-the-art gas and electricity distribution strategies. Data storages have reached petabyte size because of the increase in collaborative computing and the amount of data being gathered by advanced applications. The working environment of collaborative computing is hence heterogeneous and autonomous.

1.9 BIBLIOGRAPHICAL NOTES

The work in parallel databases began in around the late 1970s and the early 1980s. The term “Database Machine” was used, which focused on building special parallel machines for high-performance database processing. Two of the first papers in database machines were written by Su (*SIGMOD* 1978), entitled “Database Machines,” and by Hsiao (*IEEE Computer* 1979), entitled “Database Machines are

Coming, Database Machine are Coming.” A similar introduction was also given by Langdon (*IEEE TC* 1979) and by Hawthorn (*VLDB* 1980). A more complete survey on database machine was given by Song (*IEEE Database Engineering Bulletin* 1981). The work on the database machine was compiled and published as a book by Ozkarahan (1986). Although the rise of database machines was welcomed by many researchers, a critique was presented by Boral and DeWitt (1983). A few database machines were produced in the early 1980s. The two notable database machines were Gamma, led by DeWitt et al. (*VLDB* 1986 and *IEEE TKDE* 1990), and Bubba (Haran et al., *IEEE TKDE* 1990).

In the 1990s, the work on database machines was then translated into “Parallel Databases”. One of the most prominent papers was written by DeWitt and Gray (*CACM* 1992). This was followed by a number of important papers in parallel databases, including Hawthorn (*PDIS* 1993) and Hameurlain and Morvan (*DEXA* 1996). A good overview on research problems and issues was given by Valduriez (*DAPD* 1993), and a tutorial on parallel databases was given by Weikum (*ICDT* 1995).

Ongoing work on parallel databases is supported by the availability of parallel machines and architectures. An excellent overview on parallel database architecture was given by Bergsten, Couprie, and Valduriez (*The Computer Journal* 1993). A thorough discussion on the shared-everything and shared-something architectures was presented by Hua and Lee (*PDIS* 1991) and Valduriez (*ICDE* 1993). More general parallel computing architectures, including SIMD and MIMD architectures, can be found in widely known books by Almasi and Gottlieb (1994) and by Patterson and Hennessy (1994).

A new wave of *Grid databases* started in the early 2000s. A direction on this area is given by Atkinson (*BNCOD* 2003), Jeffery (*EDBT* 2004), Liu et al. (*SIGMOD* 2003), and Malaika et al. (*SIGMOD* 2003). One of the most prominent works in Grid databases is the DartGrid project by Chen, Wu et al., who have reported their project in *Concurrency and Computation* (2006), at the *GCC* conference (2004), at the Computational Sciences conference (2004), and at the *APWeb* conference (2005).

Realizing the importance of parallelism in database processing, many commercial DBMS vendors have included some parallel processing capabilities in their products, including Oracle (Cruanes et al. *SIGMOD* 2004) and Informix (Weininger *SIGMOD* 2000). Oracle has also implemented some grid facilities (Poess and Othayoth *VLDB* 2005). The work on parallel databases continues with recent work on shared cache (Chandrasekaran and Bamford *ICDE* 2003).

1.10 EXERCISES

- 1.1. Assume that a query is decomposed into a serial part and a parallel part. The serial part occupies 20% of the entire elapsed time, whereas the rest can be done in parallel. Given that the one-processor elapsed time is 1 hour, what is the *speed up* if 10 processors are used? (For simplicity, you may assume that during the parallel processing of the parallel part the task is equally divided among all participating processors).

- 1.2. Under what conditions may *superlinear speed up* be attained?
- 1.3. Highlight the differences between *speed up* and *scale up*.
- 1.4. Outline the main differences between *transaction scale up* and *data scale up*.
- 1.5. Describe the relationship between the following:
 - Interquery parallelism
 - Intraquery parallelism
- 1.6. Describe the relationship between the following:
 - Scale up
 - Speed up
- 1.7. *Skewed workload distribution* is generally undesirable. Under what conditions that parallelism (i.e. the workload is divided among all processors) is not desirable.
- 1.8. Discuss the strengths and weaknesses of the following parallel database architectures:
 - Shared-everything
 - Shared-nothing
 - Shared-something
- 1.9. Describe the relationship between parallel databases and Grid databases.
- 1.10. Investigate your favourite Database Management Systems (DBMS) and outline what kind of *parallelism* features have been included in their query processing.
- 1.11. For the database in the previous exercise, investigate whether the DBMS supports the *Grid* features.