Parallel Execution with
Oracle Database 10g Release 2

An Oracle White Paper
June 2005
EXECUTIVE OVERVIEW

Parallel execution is one of the fundamental database technologies that enable organizations to manage and access terabytes of data. Without parallelism, these large databases, commonly used for data warehouses but increasingly found in operational systems as well, would not exist.

Simply defined, parallel execution is the ability to apply multiple CPU and IO resources to the execution of a single database operation. While every major database vendor today provides parallel capabilities, there remain key differences in the architectures provided by the various vendors.

This paper discusses Oracle Database 10g Release 2’s parallel execution architecture and shows its superiority over alternative architectures for real-world applications. In particular, a major advantage of Oracle Database 10g Release 2’s parallel execution architecture is its ability to leverage the underlying hardware infrastructure to its fullest extent – using every single CPU, every byte of memory, and the fully available I/O bandwidth at any point in time. This paper also talks about the seamless integration of Oracle’s parallel execution component with other, business-critical components, such as Oracle Real Application Clusters.

INTRODUCTION

Databases today, irrespective of whether they are data warehouses, operational data stores, or OLTP systems, contain a wealth of information. However, finding and presenting the right information in a timely fashion can be a challenge because of the vast quantity of data involved.

Parallel execution is the capability that addresses this challenge. Using parallelism, a terabyte of data can be processed in minutes, not hours or days. Parallelism achieves this performance by leveraging all of the hardware resources available: multiple CPU’s, multiple IO channels, multiple storage arrays and disk drives, and large volumes of memory. The more effectively the database software can leverage all of these resources, the more efficiently queries and other database operations will be processed.
Moreover, today’s database applications have increasingly complex workloads, not only with large numbers of concurrent users but also with different types of users. Thus, the parallel query architecture should not only ensure that all of the resources of the underlying hardware platform are fully utilized, but should furthermore allocate those resources appropriately among multiple concurrent requests. A request supporting the strategic decision of the CEO is more important than any batch reporting, and the parallel query architecture should account for these business requirements by allocating resources based not only upon request itself but also upon whom made the request as well as the amount of available resources currently available.

Oracle addresses these requirements by providing a parallel execution architecture, which not only provides industry-leading performance but also is uniquely adaptive and dynamic. Oracle Database 10g Release 2’s parallel execution architecture takes full advantage of every hardware investment -- SMP, cluster, or MPP -- guaranteeing an optimal throughput and continuous, optimized system usage at any point in time. The Oracle Database 10g Release 2 database controls and balances all parallel operations, based upon available resources, request priorities and actual system load.

**DESIGN STRATEGIES FOR PARALLELIZATION – STATIC VERSUS DYNAMIC**

Parallelism is the idea of breaking down a single task into multiple smaller, distinct parts. Instead of one process doing all the work, the task can then be parallelized, having multiple processes working concurrently on the smaller units. This leads to tremendous performance improvements and optimal system utilization. The most critical part, however, is to make the decision how to divide the original single task into smaller units of work.

Traditionally, two approaches have been used for the implementation of parallel execution of database systems. The main differentiation is whether or not the physical data layout is used as a base – and static pre-requisite – for dividing, thus parallelizing, the work.

**Shared Everything**
- No Data Partitioning Required

**Shared Nothing**
- Static Data Partitioning is a pre-requirement

*Figure 1: Design strategies for parallelization
Shared Nothing versus Shared Everything architecture*
Static Parallelism through Physical Data Partitioning – Shared Nothing

In pure shared nothing database architectures, database files have to be partitioned on the nodes of a multi-computer system to enable parallel processing. Each node ‘owns’ a subset of the data and all access to this data is performed exclusively by the owning node, using a single process or thread with no provision for intra-partition parallelism (Instead of referring to a ‘node’ you can also find terms like ‘virtual processors’, a mechanism to emulate a shared nothing node on an SMP machine; for simplicity reasons, we will refer to a node when discussing shared nothing architectures). In other words, a pure shared nothing system uses a partitioned or restricted access approach to divide the work among multiple processing nodes. Data ownership by nodes changes relatively infrequently—database reorganization to address changing business needs, adding or removing nodes, and node failure are the typical reasons for change in ownership and always imply manual administration effort.

Conceptually, it is useful to think of a pure shared nothing system as being very similar to a distributed database. A transaction executing on a given node has to send messages to other nodes that own the data being accessed and coordinate the work done on the other nodes, to perform the required read/write activity. Message passing to other nodes, requesting to execute a specific operation (function) on their own data sets is commonly known as function shipping. On the other hand, if simply data is requested from a remote node, the complete data set must be accessed and shipped from the owning node to the requesting nodes (data shipping).

This approach has some basic disadvantages and is not capable to address the scalability and high availability requirements of today’s high-end environments:

- First, the shared nothing approach is not optimal for use on shared everything SMP hardware. The requirement to physically partition data in order to derive the benefits of parallelism is clearly an artificial and outdated requirement in a shared everything SMP system, where every processor has direct, equal access to all the data.

- Second, the rigid partitioning-based parallel execution strategy employed in the shared nothing approach often leads to skewed resource utilization, e.g. when it is not necessary to access all partitions of a table, or when larger non-partitioned tables, owned by a single node, are part of an operation. In such situations, the tight ownership model that prevents intra-partition parallel execution fails to utilize all available processing power, delivering sub-optimal use of available processing power.

- Third, due to the fact of having a physical data partition to node relationship, shared nothing systems are not flexible at all to adapt to changing business requirements. When the business grows, you
cannot easily enlarge your system incrementally to address your growing business needs. You can upgrade all existing nodes, keeping them symmetrical and avoiding data repartitioning. In most cases upgrading all nodes is too expensive; you have to add new nodes and to reorganize – to physically repartition – the existing database. Having no need for reorganization is always better than the most sophisticated reorganization facility.

- Finally, shared nothing systems, due to their use of a rigid restricted access scheme, fail to fully exploit the potential for high fault-tolerance available in clustered systems.

Undoubtedly, massively parallel execution based on a shared nothing architecture with static data distribution can parallelize and scale under laboratory conditions. However, the above-mentioned deficiencies have to be addressed appropriately in every real-life environment to satisfy today’s high-end mission-critical requirements.

A review and a more detailed discussion about the fundamental differences of the various cluster architectures and the disadvantages of shared nothing systems can be found in several Real Application Clusters related white papers, such as *Oracle 9i Real Application Clusters – Cache fusion delivers scalability* of February 2002, available on the Oracle Technology Network, [http://technet.oracle.com/](http://technet.oracle.com/).

**Dynamic Parallelism at Execution Time – Shared Everything**

With Oracle’s dynamic parallel execution framework, all data is shared, and the decision for parallelization and dividing the work into smaller units is not restricted to any predetermined static data distribution done at database setup (creation) time.

Every query has its own characteristics of accessing, joining, and processing different portions of data. Consequently, each SQL statement undergoes an optimization and parallelization process when it is parsed. When the data changes, if a more optimal execution or parallelization plan becomes available, or you simply add a new node to the system, Oracle can automatically adapt to the new situation. This provides the highest degree of flexibility for parallelizing any kind of operation:

- The physical data subsetting for parallel access is dynamically optimized for each query’s requirements before the statement is executed.
- The degree of parallelism is optimized for every query. Unlike in a shared nothing environment, there’s no necessary minimal degree of parallelism to invoke all nodes to access all data – the fundamental requirement to reach all of the data.
Operations can run in parallel, using one, some, or all nodes of a Real Application Clusters, depending on the current workload, the characteristics, and the importance of the query.

As soon as the statement is optimized and parallelized, all subsequent parallel subtasks are known. The original process becomes the query coordinator; parallel execution servers (PX servers) are allocated from the common pool of parallel execution servers on one or more nodes and start working in parallel on the operation.

Like in a shared nothing architecture, each parallel execution server in a shared everything architecture works independently on its personal subset of data. Data or functions are shipped between the parallel processes similar – or even identical – to the above discussed shared nothing architecture. When the parallel plan of a request is determined, every parallel execution server knows its data set and tasks, and the inter-process communication is as minimal as in a shared-nothing environment.

However, unlike the shared nothing architecture, each SQL statement executed in parallel is optimized without the need to take any physical database layout restrictions into account. This enables the most optimal data subsetting for each parallel execution, thus providing equal and in most cases even better scalability and performance than pure shared nothing architectures. Subsequent steps of a parallel operation are combined and processed by one Parallel Execution server whenever beneficial, reducing the necessity of function and/or data shipping even more.

**Why is Shared Everything superior to Shared Nothing?**

Shared nothing architectures have its roots back to the times when massively parallel processing (MPP) systems were seen as the only hardware environment capable of doing scalable high-end parallel computing. Each node in an MPP system has its own system components – CPU, memory, and disks – and works on a distinct subtask, not being able to share any of its resources.

That was yesterday. Today, the most successful and widely used parallel hardware systems are symmetric multi-processor systems (SMP), either standalone or as loosely coupled clusters. SMP systems utilize multiple processors that share common memory and disk resources and hence are also known as ‘shared everything’ systems.

Supporters of the pure shared nothing architecture always claim that a shared everything architecture – and especially cluster environments - would lack the scalability for high-end environments and cause significant overhead that makes this architecture unusable for high-end usage with a high degree of parallelism and/or concurrency. **This is not true.** Today’s available hardware and software technology solved all issues of yesterday, such as high-speed cluster
interconnects or Oracle’s Cache Fusion™ architecture, the foundation of Oracle Real Application Clusters.

Oracle’s dynamic parallel execution framework, built on the same fundamental design for parallel supercomputing as shared nothing software, shares all the benefits, yet enhances its functionality and overcomes the architectural deficiencies of the shared nothing approach. Software based on the shared-nothing philosophy can be seen as the first, yet outdated, generation of parallel execution for databases.

One should notice that newer implementations of the shared nothing approach for parallelization are trying to overcome the aforementioned deficiencies with the introduction of sharing mechanisms - basic components of Oracle’s unique implementation of dynamically parallelized execution. Those are namely:

- Several virtual nodes running on single, large SMP boxes, emulating shared nothing nodes on a shared everything hardware.
- Intra-partition Parallelism – ‘shared everything’ data access on a single node within a shared nothing system
- Node failover mechanisms to overcome the availability drawbacks, requiring physical shared disk access. Unless you have spare nodes, a failover can cause unbalanced workload on the system.

**Parallelism in daily life**

We will now step through a simple example, demonstrating the differences and the superiority of Oracle’s dynamic parallel execution: In some respects, the architectures of static parallelism versus dynamic parallelism echo a basic problem from queuing theory. For example, consider what happens when you go to a post office. The post office has multiple windows, each staffed by a postal worker.

One approach is to have customers like you line up in front of every window, and wait until the clerk for that window can help them. Another approach is to have a single line for all windows, and the customer at the front of the line goes to the first available window.

In many respects, static parallelism is analogous to the first approach, while dynamic parallelism is analogous to the second. As everyone who has waited in lines knows, having a single line generally minimizes the average wait-time. The analogy is that the postal windows are available parallel execution servers, and the customers are the individual portions of database operations that need to be completed.

In a dynamic model, the query coordinator doles out database operations to the first available parallel execution server. Before a query starts, there is no way to tell which operation will be completed by which parallel execution server. In a static model, the work is distributed to all of the parallel execution servers at the
In fact, the static parallelism approach takes this one step further. The static parallelism approach not only requires one customer line for each window, but also specifies that a given customer can only line up in front of a specific window (just as a given set of data can only be accessed by a single node). That is, the post office has one window for all customers with last names starting with the letters A through the letter D, and another window for all customers with last names starting with the letters E through H, and so on. If a customer named Mr. Adams is standing in a long line in the A-D window and notices that the E-H line is empty, Mr. Adams cannot switch lines because the A-D window is the only window that is allowed to assist him. What if even the clerk of the A-D window becomes sick or needs some additional information available at Window J-L? How does the office being run with the static parallelism approach handle peaks like seasonal events?

Which post office would you visit?

ORACLE’S PARALLEL EXECUTION ARCHITECTURE

The ‘Parallel Everything’ philosophy

By building the parallel execution framework following the strategic decision to enable a shared everything access, Oracle can take advantage of doing operations in parallel for everything in the database. Whether data is loaded into a single partition, new index structures are built, multiple users are analyzing your sales data of the last month, or the system is recovering an interrupted transaction, dealing with one partition only – all these operations can be done highly scalable in parallel, using the most optimal degree of parallelism and the most optimal parallel (physical) units of work. Oracle Database 10g Release 2 and its dynamic, scalable parallel everything capabilities enables you to taking full advantage of your hardware investment at any point in time, guaranteeing optimal throughput and continuous, optimized system usage.

How Oracle parallelizes an operation

An Oracle server consists of an Oracle database and an Oracle server instance; in the case of Oracle Real Application Clusters, an instance is running on each participating node in the cluster, commonly accessing the same Oracle database. Every instance provides a pool of parallel executions servers. This pool will be used by any parallel operation of the instance or other instances within the same cluster. Unlike shared nothing systems, no static data row assignment to node takes place. You choose an adequate number of execution servers based on your system’s hardware capabilities, decoupled from any physical database creation. Adding additional hardware resources when your business increases, such as
additional CPUs, is therefore a plug-and-play and does not require any physical repartitioning. Avoiding a physical redistribution of data at all is always better than the cleverest implementation of it.

When executing a parallel operation, the parallel execution coordinator obtains parallel execution servers from the pool and assigns them to the operation. After the statement has been processed completely, the parallel execution servers return to the pool. The number of parallel execution servers used for an operation is the degree of parallelism (DOP) and is determined at query execution time, based on the system’s current workload and other criteria, such as the business priority of a user’s query. We will discuss the details of how Oracle determines the degree of parallelism later in this paper.

Oracle does not have the architectural built-in constraint of static parallelism like shared nothing systems, where the static number of parallel execution servers is determined at object creation time. In shared nothing environments you need at least one process per data partition; without physical data redistribution, the minimal DOP for an object cannot be changed.

The knowledge about parallel everything operations is built into the optimizer; when the execution plan of a statement is determined, the decision about the DOP and the most optimal parallelization method for each operation is made. For example, the parallelization method might be to parallelize a full table scan by block range or parallelize an index range scan by partition. Furthermore, the redistribution requirements of each operation are determined. An operation’s redistribution requirement is the way in which the rows operated on by the operation must be divided or redistributed among the parallel execution servers. Oracle’s parallel-aware optimization uses enhanced redistribution mechanisms like broadcasting of small row sets for join optimization, or a lean redistribution between distinct parallel execution servers. The shared everything approach of Oracle also eliminates all data redistribution necessary due to static node affinity of data, a problem shared nothing architectures often address with duplicating the same data redundantly on several or even all nodes.

After determining the redistribution requirement for each operation in the execution plan, the optimizer determines the order in which the operations must be performed. With this information, the optimizer determines the final parallel plan and the data flow between parallel operations of the statement.

Figure 2 illustrates the data flow for a query that joins an employees and department tables:

```sql
SELECT department_name, MAX(sal), AVG(sal)
FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY department_name;
```
Operations that require the output of other operations are known as parent operations. In Figure 2 the SORT GROUP BY operation is the parent of the HASH JOIN operation because SORT GROUP BY requires the HASH JOIN output.

Parent operations can begin consuming rows as soon as the child operations have produced rows. In the previous example, while the parallel execution servers are producing rows in the FULL SCAN dept operation, another set of parallel execution servers can begin to perform the HASH JOIN operation to consume the rows. Note that this second set of parallel execution servers combines the HASH JOIN operation with the parallel FULL SCAN of emp to minimize data redistribution and communication for the SQL statement.

Each of the two operations performed concurrently is given its own set of parallel execution servers. Therefore, both query operations and the data flow tree itself have parallelism. The parallelism of an individual operation is called intraoperation parallelism and the parallelism between operations in a data flow tree is called interoperation parallelism.

Figure 2: Example parallel data flow

Figure 3 shows the execution plan for the above-discussed parallel operation. The execution plan gives us additional information about the parallel query processing, namely if data distribution has taken place and how the data was distributed between sets of parallel execution servers (column ‘PQ Data Distribution’). The identifier for a set of parallel execution servers is shown in column ‘PX set’. We will now have a closer look into this sample execution plan to highlight some of Oracle’s internal optimizations for parallel execution.
Whenever possible, subsequent operations are combined and being processed by the same set of parallel execution servers; this minimizes the data distribution and communication for a SQL statement. In our sample query, operation Id 4, 3, and 2 are processed by the same set of parallel execution servers (PX SET 1). No data distribution is necessary; these operation steps are combined parallel operations. Furthermore, you can see that the row set derived by operation Id 5 (FULL SCAN dept) is broadcasted to the subsequent operation Id 3 (HASH JOIN), thus optimizing the data redistribution between those operations.

The query plan also shows an additional operation step that is not seen in Figure 1. Before the data is redistributed after the parallel HASH JOIN operation Id 3, each parallel execution server is pre-sorting and pre-aggregating its personal data set (SORT GROUP BY, operation Id 2) before sending it out to the subsequent - and final – sort and aggregation operation (operation Id 1). The data redistribution between these operations is a hash distribution, ensuring an equally balanced workload for all parallel execution servers working on
operation Id 1. After the final operation, the result sets are returned to the user process (operation Id 0) in a random order from all parallel execution servers as soon as those have finished their work (no sort order was specified in the query). Figure 4 shows the parallel data flow with the above-discussed parallel execution optimizations.

**Dynamic Determination of Work Units**

Unlike parallel operations in shared nothing architectures, Oracle’s parallel execution determines the most optimal subset of physical data to be accessed based on the user’s request, the DOP, and the actual system load. Oracle does not rely on any physical data distribution across nodes.

In Oracle, the basic unit of work in parallelism is called a **granule**. Oracle divides the operation being parallelized (for example, a table scan, table update, or index creation) into granules. Parallel execution servers execute the operation one granule at a time. The number of granules and their size correlates with the DOP and the object size. There is no way you can enforce a specific granule strategy as Oracle makes this decision internally.

Depending on the parallel operation, Oracle either chooses the granule based on block ranges or a complete partition as the optimal granule strategy. Looking at the example discussed earlier, Oracle chooses block range granules for the full table scan operations to optimize and balance the work distribution for all affected PX servers. An operation like a parallel partition-wise join on the other hand uses partition granules to minimize the data redistribution between the PX servers. We will discuss the optimization of partition-wise joins later in this paper in more detail.
Providing the flexibility of choosing granules operation-specific enables Oracle to optimize parallel execution far beyond the capabilities of shared nothing architectures.

Figure 5 illustrates how the FULL SCAN of emp will be parallelized in a shared nothing environment and with Oracle’s approach. Assuming the same degree of parallelism and the same amount of data with similar hardware capabilities, you will encounter similar to identical execution times for both architectures. Note that the degree of parallelism is static for the shared nothing architecture.

What does it mean to have static, predefined parallelism?

Figure 6 shows you how the same full table scan will be parallelized when the amount of data has doubled, and no manual administrative action has taken place to physically redistribute any data. Unlike the shared nothing architecture where the degree of parallelism is static, Oracle can automatically choose a higher degree of parallelism (in our sample the DOP also doubles) to accommodate the changed environment. The execution time for the operation – not the degree of parallelism – stays stable and predictable.

The Parallel Execution Server Pool

The total number of available parallel execution (PX) servers is dynamically and automatically adjusted by the database system. If the number of parallel operations processed concurrently by an instance changes significantly, Oracle automatically changes the number of parallel execution servers in the pool.

If the number of parallel operations increases, Oracle creates additional PX servers to handle incoming requests. However, it will never exceed the upper limit of PX servers, which is either manually defined by the database.
Parallel Automatic Tuning manages all relevant database parameters for optimized parallel execution, based on the available system resources.

administered or automatically determined by the database, based on the system's hardware characteristics. This prevents the system from being overloaded with too many concurrent parallel processes; no hardware system can use more resources than available.

If the number of parallel operations decreases, Oracle terminates any parallel execution servers that have been idle for a threshold period of time.

With Parallel Automatic Tuning enabled, Oracle takes care of all relevant database settings for parallel execution. In most cases, Oracle's automated settings are at least as efficient as manually derived parameters and avoid manual tuning errors due to wrong user-load or system-resource calculations or assumptions.

However, you can override any of the default settings done by the database to give you the highest degree of flexibility and to accommodate your application specifics.

Controlling the Degree of Parallelism

The degree of parallelism (DOP) for a given SQL statement is driven by three influencing factors:

1. Available HW resources, determining the parallel capabilities of a system:
   Depending on the available hardware resources, only a specific maximum of parallel processes should run on the system concurrently not to overload the system. Note that this is a hardware constraint and not a limitation of the database.

2. Current workload on a system when the statement is started:
   The current workload on a system is an important factor in

Oracle's Automatic, Adaptive Degree of Parallelism balances all parallel operations for optimal system throughput.
determining the DOP that can be allocated to execute a parallel operation.

3. Enabled parallelism based on object attribute:
   All tables and indexes have a parallel property defined. This attribute should be set to NOPARALLEL or PARALLEL DEFAULT. The latter one enables parallel execution for a specific object in general. Alternatively, if you want to explicitly control parallel execution, you can request a specific DOP for an object by setting its parallel property to a concrete value, e.g. PARALLEL 32.

4. Business priority of a request:
   Every user belongs to a so-called resource consumer group. Using the Database Resource Manager, you restrict the allowed resource consumption that a resource consumer group – representing a business function - can have; for example, the maximum degree of parallelism represents such a database resource. More information about Oracle’s Database Resource Manager can be found on the Oracle Technology Network, [http://technet.oracle.com/](http://technet.oracle.com/).

When a system is overloaded and the input DOP is larger than the default DOP, the algorithm uses the default degree as input. The system then calculates a reduction factor that it applies to the input DOP. For example, using a 16-CPU system, when the first user enters the system and it is idle, it will be granted a DOP of 32. The next user will be given a DOP of eight, the next four, and so on. If the system settles into a steady state of eight users issuing queries, all the users will eventually be given a DOP of 4, thus dividing the system evenly among all the parallel users. This ensures the most optimal throughput of your system at any point in time.

Taking the current workload under account and reducing the DOP before an operation even starts is the better alternative than blindly starting everything with the predictable degree of parallelism; the latter approach is used by the shared nothing systems (they cannot change the DOP) and leads to throttling down some of the database activities later on during the execution to not exhaust the system. The software being able to use more resources than available has to be invented yet.

Although this always guarantees optimal throughput of your database system, this might not be the way you want your application to behave. So you can specify the minimum percentage of requested parallel execution servers that must be available in order for the operation to execute. This strategy ensures that your SQL statement executes predictable within well-defined boundaries. If the minimum percentage of requested parallel execution servers is not available, the SQL statement does not execute and returns an error.
Types of Parallelism

All business critical operations can be parallelized. The following section will discuss the various types of parallel operations in large-scale environments:

Parallel query

The most common parallel operations in large-scale environments are queries. Since Oracle does not request any blocking read locks, high-concurrency environments with hundreds of diversified ad-hoc queries are not uncommon for large-scale installations. Especially those environments benefit from the above discussed automated, adaptive control of parallel operations.

You can parallelize queries and subqueries in SELECT statements. For most queries Oracle chooses block granules, guaranteeing an equal workload distribution over all participating PX servers. An exception worth noting is a parallel HASH JOIN with one or more partitioned table involved, where the optimizer decides to use partition granules. Joining equi-partitioned objects partition pair-wise in parallel is much more efficient and performant than arbitrarily joining in parallel. We will discuss parallel partition-wise hash joins later in more detail.

Parallel DDL

Data warehouse environments often face the task of creating large objects — rebuild indexes, create intermediate staging tables, or issue a partition maintenance operation that move data. All these kinds of operations can be parallelized for partitioned and non-partitioned objects.

Parallel DML

Parallel DML (PARALLEL INSERT, UPDATE, and DELETE) uses parallel execution mechanisms to speed up or scale up large DML operations against large database tables and indexes.

Parallel DML operations are mainly used to speed up large DML operations against large database objects. Parallel DML complements parallel query in providing you with both querying and updating capabilities for your large databases. Although Oracle provides an efficient, scaleable parallel DML framework, experience shows that intermediate objects, created with the CREATE TABLE AS SELECT command, are more common in large-scale warehouse environments than the classical UPDATE or DELETE commands.

Others

In addition to parallel SQL execution, Oracle can use parallelism for every important database operations as well. We’re not going to discuss these functionalities in detail. Those functionalities include

- Parallel recovery
- Parallel propagation (replication)
• Parallel read from external tables
• Parallel load (the SQL*Loader utility)

Parallel Execution and Partitioning
To ensuring an equally balanced workload for a parallel operation, the units of work for all participating processes should be equally sized. Predetermined physical data partitioning is the key enabling technology in shared nothing environments for parallel execution. All data partitions, the solely foundation for determining units of work for shared nothing systems, must be equally sized. That’s the main and sole reason why the fundamental partitioning strategy for shared nothing systems is a based on a HASH-distribution. Distributing data based on hash values guarantees equally sized data partitions, but the location of records becomes totally unrelated to the records’ business content; hash distribution is a purely physical distribution.

Unlike shared nothing systems, Oracle does not rely on having Oracle Partitioning to enable parallel execution, neither on SMP, MPP, or cluster environments. A Partitioning strategy can be chosen based upon the system’s business requirements and is first hand totally unrelated to any parallel operation. Any operation, whether accessing one, some, or all partitions of a table can be parallelized with any unrestricted DOP, depending only on your specific environment and business requirements, such as HW resources, data volume or available time window.

Furthermore, in conjunction with parallel execution, Oracle Partitioning can dramatically improve performance in large-scale environments. This paper does not discuss partitioning in detail (extensive material about Oracle Partitioning can be found on the Oracle Technology Network, http://technet.oracle.com/). It discusses specific areas where partitioning enables enhanced parallel operations or the combination of partitioning and parallel execution shows even more dramatic performance enhancements than parallel execution only.

Partition pruning
Partition pruning is an essential performance feature for large-scale environments. In partition pruning, the cost-based optimizer analyzes FROM and WHERE clauses in SQL statements to eliminate unneeded partitions when building the partition access list. This enables Oracle to perform operations only on those partitions that are relevant to the SQL statement. Partition pruning dramatically reduces the amount of data retrieved from disk and shortens the use of processing time, improving query performance and resource utilization.

Since the most commonly used Partitioning methods with Oracle are logical rather than a purely physical (hash) partitioning strategy, it is most likely that many of your queries will benefit from partition pruning. For example, suppose an application contains an Orders table storing a historical record of orders, and that this table has been partitioned by week. All queries requesting the most
recent orders of the current week would only access a single partition of the Orders table. If the Orders tables had 2 years of historical data, this query would access one partition instead of 104 partitions. This query could potentially execute 100x faster simply because of partition pruning.

Shared nothing systems, relying on hash partitioning across nodes for enabling parallel execution, are not able to provide the same level of sophisticated operations. Nevertheless, let’s assume for a moment, a shared nothing system would use a logical partitioning strategy for the above-mentioned Orders table. Let’s also assume the system would provide the same partition pruning capabilities than Oracle. What would it mean for a parallel operation, if only a single partition must be accessed to satisfy a business query? Accessing only ONE partition would mean that only ONE node could be used to process the request. The foundation for parallelism in shared nothing systems would be undermined, leading to poor and skewed performance of the system.

With Oracle, partition pruning does not affect or interfere with parallel execution. Unlike in shared nothing environments, a query; accessing one, some, or all partitions; can always be parallelized with any DOP; executed on one, some, or all nodes of a cluster.

**Partition-wise joins**

A partition-wise join takes place when the tables to be joined are equi-partitioned on the join key or the operation is executed in parallel and one of the tables is partitioned on the join key. Partition-wise joins improve query response time by minimizing the amount of data exchanged among parallel execution servers when joins execute in parallel. This significantly reduces response time and makes more efficient use of both CPU and memory resources. In Oracle Real Application Clusters environments, partition-wise joins also avoid or at least limit the data traffic over the interconnect, which helps to achieving good scalability for massive join operations. This is exactly the same mechanism that is used by shared nothing architectures to minimize their internode communication as well.

**Figure 7: Example parallel full partition-wise join**
Partition-wise joins can be full or partial. In a full partition-wise join operation, both tables are equi-partitioned on the join key. Unlike full partition-wise joins, partial partition-wise joins only have one table being partitioned on the join key, not both tables. The partitioned table is referred to as the reference table. Partial partition-wise joins are more common than full partition-wise joins. To execute a partial partition-wise join, Oracle dynamically redistributes usually the smaller, possibly nonpartitioned table based on the partitioning key of the larger, partitioned reference table. Once the other table is dynamically repartitioned, the execution is similar to a full partition-wise join.

The performance advantage that partial partition-wise joins have over joins in non-partitioned tables is that the reference table is not moved during the join operation.

Vendors supporting the shared-nothing parallel approach mainly rely on partition-wise joins for getting query performance. This is a natural consequence of having predetermined affinity between data and nodes and the missing possibility to access data remotely. To improve query performance in such environments, both matching partitions should have affinity to the same node. Partition pairs must be spread over all nodes to avoid bottlenecks and to use all CPU resources available on the system. Although this might bring you performance benefits for one class of queries, it limits you in the data model you can choose and does not address the changing business requirements and types of queries. For example, if you optimize your Orders table for the most efficient analysis of customer-related type of queries, you will partition the customers and orders table on their join key customer_id. This enables partition-wise joins for those two tables and guarantees node affinity for the partitions that must be joined. However, any product-related types of business query aren’t optimized and involve data redistribution across nodes, thus less optimal query processing. Reacting on changing business requirements causes substantial additional administrative costs and downtime, limiting the practical applicability but necessary to keeping the system performant:

- Frequent Need for Repartitioning: When data access requirements change over time, as they frequently do in real-life, data has to be repartitioned to achieve acceptable performance levels. Further, repartitioning is also required with any of the following:
  - Significant growth in data volumes
  - Addition of processing nodes
  - Shifts in data distribution patterns resulting in skewed partitions

**ORACLE’S PARALLEL ARCHITECTURE ON EVERY HARDWARE PLATFORM**

Oracle’s parallel execution framework is built into the Oracle Database 10g Release 2 database and has been optimized for clustered and non-clustered
environments. Consequently, it offers the same functionality transparently to any application, whether you’re using Oracle on a large SMP machine, or Oracle Real Application Clusters on an MPP machine, or a loosely coupled cluster.

**Parallel Execution in a single-node SMP Environment**

Running an Oracle database on a single-node SMP box enables you to taking full advantage of the discussed parallel execution capabilities. The hardware provides the resources for parallel execution, the CPU power and I/O bandwidth. Oracle takes care of the rest. Unlike shared nothing systems where you need several artificial ‘virtual’ shared nothing nodes (instances) running a single machine, you have one Oracle instance up and running. Every statement executed against the database can be parallelized without any special parallel-aware setup of or data layout, providing any degree of parallelism up to the system’s limits. The communication between parallel execution servers is within the same instance and done through fast in-memory operations. A single machine is used as such, running a single database instance, and leveraging all available system resources up to the limit. A shared nothing system, on the other hand, needs to run multiple instances on a single machine to emulate a shared nothing environment. Each instance has its own set of processes and memory allocations, acting the same way than in clustered environments. Shared nothing systems are therefore always less efficient on an SMP machine, adding the overhead of running multiple instances and forcing them to communicate like on a shared nothing hardware architecture.

**Parallel Execution in a Real Application Clusters Environment**

Oracle’s first parallel database, Oracle Parallel Server, was introduced with Oracle V6 on VMS operating systems. Oracle Database 10g Release 2’s Real Application Clusters (RAC), the fourth generation of Oracle’s parallel database architecture, provides linear scalability and availability for high volume, mission-critical applications, thus providing unlimited power and growth for your systems.

These benefits are achieved by fully exploiting the power and redundancy of any kind of clustered computer systems. Oracle Real Application Clusters uses modern cluster interconnect facilities to reduce disk I/O and can exploit the emerging high bandwidth low latency interconnects to provide linear scalability. It fully exploits the expanded CPU, memory and disk resources of the clustered system to drive more transactions.

Oracle’s parallel execution framework is built into the Oracle Database 10g Release 2 database and has been optimized for clustered and non-clustered environments. Consequently, it offers the same functionality transparently to any application, whether you’re using Oracle on a large SMP machine or Oracle Real Application Clusters on an MPP machine or a loosely coupled cluster.
Choosing the best approach for any type of cluster

Parallel execution performs efficiently in Real Application Clusters because parallel processing, for example, distributes portions of a large SQL statement across multiple instances. The transaction completes more quickly because it executes on multiple CPUs.

In Real Application Clusters, Oracle determines at runtime whether it will run parallel execution server processes on only one instance, or whether it will run processes on multiple instances. In general, Oracle tries to use only one instance when sufficient resources are available. This reduces the cross-instance message traffic and synchronization. Potential specifics of the cluster environments are also taken into account. For example, on some hardware systems, powerful data locality capabilities were more relevant when shared nothing hardware systems were popular. However, almost all current cluster systems use a shared disk architecture, enabling the same I/O access to all available disks for all nodes.

On **loosely coupled systems**, Oracle’s parallel execution technology uses a function shipping strategy to perform work on remote nodes whenever possible. This delivers efficient parallel execution and eliminates unneeded internode data transfers over the interconnect.

On **shared nothing (MPP) systems**, each node has direct hardware connectivity to a subset of disk devices. On these systems it is more efficient to access local devices from the owning nodes. Real Application Clusters exploits this affinity of devices to nodes and delivers performance that is superior to shared nothing systems using multi-computer configurations and a shared disk architecture.

As with other elements of Cache Fusion, Oracle’s strategy works transparently without data partitioning. Oracle dynamically detects the disk on which the target data resides and makes intelligent use of the data’s location in the following two ways:

- Oracle spawns parallel execution server processes on nodes where the data to be processed is located
- Oracle assigns local data partitions to each sub-process to eliminate or minimize internode data movement

**Dynamic cluster- and parallel-aware query optimization**

Oracle’s cost-based optimizer considers parallel execution when determining the optimal execution plans. An evaluation of the costs of alternative access paths—table scans versus indexed access, for example—takes into account the degree of parallelism available for the operation. This results in Oracle selecting execution plans that are optimized for parallel execution.

An average mid-range cluster systems having four nodes, each cluster node having four CPU’s, is not uncommon. Oracle’s setup on such a system would be one Oracle instance on each node, all accessing the same single database.
number of parallel execution servers on each node would be set to reflect each node’s capabilities for parallel processing; for example 16 PX servers per node, a total of 64. The equivalent setup for a shared nothing system on this hardware would be to have 16 virtual nodes – and data partitions – per cluster node, a total of 64 partitions. In the shared nothing environment, every object has to be partitioned that way if you want to access it in parallel. Furthermore, all of these objects then have to be accessed with a parallel degree of 64, involving all cluster nodes, all the time.

Oracle on the other hand makes intelligent decisions in Real Application Clusters environments with regard to intranode and internode parallelism. In intranode parallelism, for example, if a SQL statement requires a parallel degree of six, the load is balanced over all nodes, and six parallel execution servers are idle on the local node (the node to which the user is connected), the SQL statement is processed using local resources (figure8). This eliminates query coordination overhead across multiple nodes. The shared nothing system has to execute all statements with 64 parallel processes to being able to access all the data, involving all nodes, and communicating between all nodes.

Continuing with this example: if there are only two parallel execution servers idle on the local node, then those two parallel execution servers and four of another node are used to complete the SQL statement; Oracle uses internode parallelism (figure9). The shared nothing system still has to use all 64 processes on all nodes.

Oracle’s parallel execution technology is able to dynamically detect idle CPUs and assign work to these idle CPUs from the execution queue of the CPUs with greater workloads. In this way, Oracle efficiently re-distributes the query
workload across all of the CPUs in the system. Real Application Clusters extends these efficiencies to clusters by re-distributing the work across all the nodes of the cluster.

**Cluster load balancing for multiple concurrent parallel operations**

Load balancing distributes parallel execution server processes to spread CPU and memory use evenly among nodes. It also minimizes communication and remote I/O. Oracle does this by allocating servers to the nodes that are running the fewest number of processes.

The load balancing algorithm maintains an even load across all nodes. For example, if a DOP of eight is requested on an eight-node system with one CPU for each node and two slave sets are necessary for the operation, two PX servers are allocated on each node. If the entire parallel execution server group fits on one node, the load balancing algorithm places all the processes on a single node to avoid communications overhead. So, if the same statement discussed above is issued on a two-node cluster with 16 CPUs for each node, then the algorithm allocates all 16 parallel execution server processes on one node only.

Shared nothing systems claim to have built-in load balancing by using intelligent hash partitioning strategies for their objects. This might be true as long as you are only dealing with objects being partitioned across all nodes. However, this is not true for any object that is either non-partitioned or partitioned only across some nodes. The data affinity to a node determines statically the location where the data has to be processed.

**CONCLUSION**

Whether you’re choosing to use an SMP, MPP or cluster system: Oracle’s scalable parallel execution framework always provides you the most optimal throughput for your hardware architecture. The ability of parallel execution is only limited by the system’s – not the database – capabilities.

Oracle’s parallel execution framework is built into the Oracle Database 10g Release 2 database and has been optimized for clustered and non-clustered environments. Consequently, it offers the same functionality transparently to any application, whether you’re using Oracle on a large SMP machine or Oracle Real Application Clusters on an MPP machine or a loosely coupled cluster.

Built on the same foundation of parallel supercomputing than older shared nothing implementation, Oracle has outgrown its ancestor’s capabilities by far, providing the most scalable, flexible, and complete parallel execution architecture on the market.