Introduction ................................................................................................................. 1
Parallel Execution Concepts ....................................................................................... 2
  Why use Parallel Execution? ................................................................................. 2
  The Theory of Parallel Execution ..................................................................... 2
Parallel Execution Concepts ....................................................................................... 5
  Processing parallel SQL statements ................................................................. 5
  In-Memory Parallel Execution .......................................................................... 16
Controlling Parallel Execution ..................................................................................... 19
  Managing the degree of parallelism ................................................................. 20
  Workload management for Parallel Execution ............................................. 24
Conclusion ................................................................................................................ 30
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, terabytes of data can be processed in minutes, not hours or days. Parallel execution uses multiple processes to accomplish a single task. The more effectively the database can leverage all hardware resources – multiple cores, multiple IO channels, multiple nodes in a cluster - the more efficiently queries and other database operations will be processed.

Large data warehouses should always use parallel execution to achieve good performance. Specific operations in OLTP applications, such as batch operations, can also significantly benefit from parallel execution. The paper covers three main topics:

• **Parallel execution fundamental concepts** – why should you use parallel execution and what are the fundamental principles behind all parallel execution.

• **Oracle’s parallel execution implementation and enhancements** – here you will become familiar with Oracle’s parallel architecture, learn Oracle-specific terminology around parallel execution, and understand the basics of how to control and identify parallel SQL processing. This section also covers enhancements specific to Oracle Database 11g Release 1 and Release 2.

• **Controlling parallel execution** in Oracle Database – this last section shows how to enable and control parallelism within the Oracle environment giving you an overview of what a DBA will be thinking about.
Parallel Execution Concepts

Parallel execution is a commonly used method of speeding up operations by splitting a task in smaller sub tasks. In this section we will discuss the basic reasoning around parallel execution and the basic concepts. Furthermore we will discuss the Oracle Parallels Execution concepts in detail.

Why use Parallel Execution?

Imagine that your task is to count the number of cars in a street. There are two ways to do this, one, you can go through the street by yourself and count the number of cars or you can enlist a friend and then the two of you can start on opposite ends of the street, count cars until you meet each other and add the results of both counts to complete the task.

Assuming your friend counts equally fast as you do, you expect to complete the task of counting all cars in a street in roughly half the time compared to when you perform the job all by yourself. If this is the case then your operations scales linearly; $2x$ the number of resources halves the total processing time.

The database is not very different from the counting cars example. If you allocate twice the number of resources and achieve a processing time that is half of what it was with the original amount of resources, then the operation scales linearly. Scaling linearly is the ultimate goal of parallel processing, both in counting cars as well as in delivering answers from a database query.

The Theory of Parallel Execution

In the counting car example we made some basic assumptions to get to linear scalability. These assumptions reflect some of the theory behind parallel processing.

First of all we chose to use just the two of us to do the counting. Here we decided the parallel degree as it is called in a database. In other words, how many of us would be ideal to solve the problem fastest. The bigger the workload, the more people we could use and of course, if there is a short street with 4 cars, we should avoid any parallelism as it would take longer to decide who starts where than it takes to just count the cars.

So we decided that the overhead of having the two of us count and coordinate is worth the effort. In a database this choice is made by the query optimizer based on the cost of the operation.

Secondly, in the car example we divided the work in 2 equal parts as each of us started on one end of the street and we assumed each counted with the same speed. The same goes for parallel processing in a database. The first step is to divide the data work on in chunks of similar size allowing them to be processed in the same amount of time. Some form of hashing algorithm is often used to divide the data.
This partitioning of data is implemented in two basic ways. The main differentiation is whether or not physical data partitioning is used as a foundation – and therefore a static pre-requisite – for parallelizing the work.

These fundamental approaches are known as **shared everything** architecture and **shared nothing** architecture respectively.

In a shared nothing system, the system is divided into individual parallel processing units. Each processing unit has its own processing power (cores) and its own storage component (disk) and its CPU cores are solely responsible for its individual data set on its own disks. The only way to access a specific piece of data is to use the processing unit that owns this subset of data. Such systems are also commonly known as Massively Parallel Processing (MPP) systems. In order to achieve a good workload distribution shared nothing systems have to use a hash algorithm to partition data evenly across all available processing units. The partitioning strategy has to be decided upon initial creation of the system.

As a result, shared nothing systems introduce mandatory, fixed parallelism in their systems in order to perform operations that involve table scans; the fixed parallelism completely relies on a fixed static data partitioning at database or object creation time. Most non-Oracle data warehouse systems are shared nothing systems.

Oracle Database relies on a shared everything architecture. This architecture does not require any pre-defined data partitioning to enable parallelism; however by using Oracle Partitioning, Oracle Database can deliver the exact same parallel processing capabilities as a shared nothing system. It does so however without the restrictions of the fixed parallel access encompassed in the data.
layout. Consequently Oracle can parallelize almost all operations in various ways and degrees, independent of the underlying data layout.

By using a shared everything architecture Oracle allows flexible parallel execution and high concurrency without overloading the system, using a superset of parallel execution capabilities over shared nothing vendors.
Parallel Execution Concepts

The Oracle Database provides functionality to perform complex tasks in parallel, without manual intervention. Operations that can be executed in parallel include but are not limited to:

- Data loads
- Queries
- RMAN backups
- Index builds
- Gathering statistics
- And more

This paper focuses on SQL parallel execution only, which consists of parallel query, parallel DML (Data Manipulation Language) and parallel DDL (Data Dictionary Language). While the paper focuses on Oracle Database 11g – Release 1 and Release 2 - the information in this paper is generic for Oracle Database 10g and higher, unless explicitly stated.

Processing parallel SQL statements

When you execute a SQL statement in the Oracle Database it is decomposed into individual steps or row-sources, which are identified as separate lines in an execution plan. Below is an example of a simple SQL statement that touches just one table and its execution plan. The statement returns the total number of customers in the CUSTOMERS table:

```
SELECT count(*) FROM customers c;
```

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Cost (XCPU)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td>1</td>
<td>5 (0)</td>
<td>00:00:01</td>
</tr>
<tr>
<td>1</td>
<td>SORT AGGREGATE</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>TABLE ACCESS FULL CUST</td>
<td></td>
<td>630</td>
<td>5 (0)</td>
<td>00:00:01</td>
</tr>
</tbody>
</table>

Figure 2: serial execution plan of a count(*) on the Customers table

A more complex serial execution plan would be one that includes a join between multiple tables. In the example below, information about purchases made by customers is requested. This requires a join between the CUSTOMERS and SALES tables.

```
SELECT c.name, s.purchase_date, s.amount
FROM customers c, sales s
WHERE s.customer_id = c.id;
```
If you execute a statement in parallel, the Oracle Database will parallelize as many of the individual steps as possible and reflects this in the execution plan. If we were to re-execute the two statements above in parallel we could get the following execution plans\(^1\).

These plans look quite a bit different than before, mainly because we are having additional logistical processing steps due to the parallel processing that we did not have before.

---

1 Parallel plans will look different in versions prior to Oracle Database 10g.
SQL parallel execution in the Oracle database is based on a few fundamental concepts. The following section discusses these concepts that help you understand the parallel execution setup in your database and read the basics of parallel SQL execution plans.

**Query Coordinator (QC) and parallel servers**

SQL parallel execution in the Oracle Database is based on the principles of a coordinator (often called the Query Coordinator – QC for short) and parallel execution (PX) server processes. The QC is the session that initiates the parallel SQL statement and the PX servers are the individual sessions that perform work in parallel. The QC distributes the work to the PX servers and may have to perform a minimal – mostly logistical – portion of the work that cannot be executed in parallel. For example a parallel query with a SUM() operation requires a final adding up of all individual sub-totals calculated by each PX server.

The QC is easily identified in the parallel execution plans above as 'PX COORDINATOR' (for example ID 1 in Figure 6 shown above). The process acting as the QC of a parallel SQL operation is the actual user session process itself.

The PX servers are taken from a pool of globally available PX server processes and assigned to a given operation (the setup is discussed in a later section). All the work shown below the QC entry in our sample parallel plans (Figure 4, Figure 5) is done by the PX servers.

![Diagram](image-url)

*Figure 6: Parallel Execution with the Query Coordinator and a set of PX server processes*
PX server processes can be easily identified on the OS level, for example on Linux they are the oracle processes ORA_P***:

<table>
<thead>
<tr>
<th>Oracle</th>
<th>ID</th>
<th>Name</th>
<th>TQ</th>
<th>IN_OUT</th>
<th>PQ Distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>oracle 11131</td>
<td>1 0 Sep25</td>
<td>ora_p024_dbwl</td>
<td>00:10:26</td>
<td></td>
<td></td>
</tr>
<tr>
<td>oracle 11135</td>
<td>1 0 Sep25</td>
<td>ora_p025_dbwl</td>
<td>00:11:04</td>
<td></td>
<td></td>
</tr>
<tr>
<td>oracle 11137</td>
<td>1 0 Sep25</td>
<td>ora_p025_dbwl</td>
<td>00:11:12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>oracle 11139</td>
<td>1 0 Sep25</td>
<td>ora_p027_dbwl</td>
<td>00:10:47</td>
<td></td>
<td></td>
</tr>
<tr>
<td>oracle 11201</td>
<td>1 0 Sep25</td>
<td>ora_p023_dbwl</td>
<td>00:10:56</td>
<td></td>
<td></td>
</tr>
<tr>
<td>oracle 11203</td>
<td>1 0 Sep25</td>
<td>ora_p023_dbwl</td>
<td>00:11:00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>oracle 11205</td>
<td>1 0 Sep25</td>
<td>ora_p030_dbwl</td>
<td>00:11:07</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 7: parallel server processes seen on the OS level using 'ps -ef'

Going back to the example of counting the cars, there would be a third person – the QC - telling you and your friend – the two PX servers - to go ahead and count the cars; this is equivalent to the operation with the ID 2 in Figure 5, illustrated in Figure 8.

You can do exactly the same on the road that is being done internally in the database with the SQL and execution plan shown in Figure 8: You and your friend will go ahead and count the cars on your side; this is equivalent to the operations with the ID 4, ID 5, and ID 6, where ID 5 is the equivalent to tell each one of you to only count the cars on your side of the road (details to follow in the granule section).

![Diagram of SQL and execution plan with Query Coordinator and Parallel Servers]

Figure 8: QC and PX server process show on an execution plan
Finally, each of you tells the third person your individual subtotals (ID 3) and he then adds up the final result (ID 1). This is the hand-over from the PX servers (processes doing the actual work) to the QC for final “assembly” of the result for returning it to the user process.

**Producer/consumer model**

Continuing with our car counting example, imagine the job is to count the total number of cars per car color. If you and your friend cover one side of the road each, each one of you potentially sees the same colors and gets a subtotal for each color, but not the complete result for the street. You could go ahead, memorize all this information and tell it back to the third person (the “person in charge”). But this poor individual then has to sum up all of the results by himself – what if all cars in the street were a different color? The third person would redo exactly the same work as you and your friend just did.

To parallelize the counting, you ask two more friends to help you out: They both walk in the middle of the road, one of them taking count of all dark colors, the other one of all bright colors (assuming this “car color separation” is approximately splitting the information in half).

Whenever you count a new car, you tell the person that is in charge of this color about the new encounter – you **produce** the information, **redistribute** it based on the color information, and the color counter **consumes** the information. At the end, both color counting friends tell their result the person in charge and you're done; we had two sets, each with two friends doing a part of the job, working hand in hand.

That's how the database works: In order to execute a statement in parallel efficiently sets of PX servers work in pairs: one set is producing rows (producer) and one set is consuming the rows (consumer). For example for the parallel join between the SALES and CUSTOMERS tables (Figure 5) rows have to be redistributed based on the join key to make sure that matching join keys from both tables are sent to the same PX server process doing the join. In this example one set of PX servers reads and sends the data from table CUSTOMERS (producer) and another set receives the data (consumer) and joins it with table SALES, as shown in Figure 9.
Operations (row-sources) that are processed by the same set of PX servers can be identified in an execution plan by looking in the TQ column. As shown in Figure 9, the first slave set (Q1,00) is reading table CUSTOMERS in parallel and producing rows that are sent to slave set 2 (Q1,02) that consumes these records and joins then to record coming from the SALES table (Q1,01).

Whenever data is distributed from producers to consumers you will also see an entry of the form :TQxxxxx (Table Queue x) in the NAME column. Please disregard the content of the other columns for now.

This has a very important consequence for the number of PX server processes that are spawned for a given parallel operation: the producer/consumer model expects two sets of PX servers for a parallel operation, so the number of PX server processes is twice the requested degree of parallelism (DOP). For example, if the parallel join in Figure 9 runs with parallel degree of 4, then 8 PX server processes will be used for this statement, 4 producers and 4 consumers.

The only case when PX servers do not work in pairs is if the statement is so basic that only one set of PX servers can complete the entire statement in parallel. For example select count(*) from customers requires only one PX server set (see Figure 4).

Granules

A granule is the smallest unit of work when accessing data. Oracle database uses a shared everything architecture, which from a storage perspective means that any CPU core in a configuration can access any piece of data; this is the most fundamental architectural difference between Oracle and all other database vendors on the market. Unlike all other systems, Oracle can – and will - choose this smallest unit of work solely dependent on a query's requirements.
The basic mechanism the Oracle Database uses to distribute work for parallel execution is block ranges on disk – so-called block-based granules. This methodology is unique to Oracle and is independent of whether the underlying objects have been partitioned. Access to the underlying objects is divided into a large number of granules, which are given out to PX servers to work on (and when a PX server finishes the work for one granule the next one is given out).

Figure 10: Block-based granule in the customer count example.

The number of granules is always much higher than the requested DOP in order to get an even distribution of work among parallel server processes. The operation 'PX BLOCK ITERATOR' shown in Figure 10 literally is the iteration over all generated block range granules.

Although block-based granules are the basis to enable parallel execution for most operation, there are some operations that can benefit from the underlying data structure and leverage individual partitions as granules of work. With partition-based granules only one PX server performs the work for all data in a single partition. The Oracle Optimizer considers partition-based granules if the number of (sub)partitions accessed in the operation is at least equal to the DOP (and ideally much higher if there may be skew in the sizes of the individual (sub)partitions). The most common operations that use partition-based granules are partition-wise joins, which will be discussed later.

Based on the SQL statement and the degree of parallelism, the Oracle Database decides whether block-based or partition-based granules lead to a more optimal execution; you cannot influence this behavior.

In the car counting example, one side of the street – or even a block of a long street - could be considered the equivalent of a block-based granule. The existing data volume – the street – is subdivided into physical pieces on which the parallel servers – you and your friend – are working on independently.

Data redistribution

Parallel operations – except for the most basic ones – typically require data redistribution. Data redistribution is required in order to perform operations such as parallel sorts, aggregations and joins. At the block-granule level there is no knowledge about the actual data contained in an
individual granule. Data has to be redistributed as soon as a subsequent operation relies on the actual content. Remember the last car example? The car color mattered, but you don't know – or even control – what color cars are parked where on the street. You redistributed the information about the amount of cars per color to the additional two friends based on their color responsibility, enabling them to do the total counting for the colors they're in charge of.

Data redistribution takes place between individual PX servers either within a single machine, or, across multiple machines in a Real Application Clusters (RAC). Of course in the latter case interconnect communication is used for the data redistribution.

Data redistribution is not unique to the Oracle Database. In fact, this is one of the most fundamental principles of parallel processing, being used by every product that provides parallel capabilities. The fundamental difference and advantage of Oracle's capabilities, however, is that parallel data access (discussed in the granules section earlier) and therefore the necessary data redistribution are not constrained by any given hardware architecture or database setup.

Shared-nothing database systems also require data redistribution unless operations can take advantage of partition-wise joins (as explained further down in this section). In shared-nothing systems parallel operations that cannot benefit from a partition-wise join – such as a simple three-way table join on two different join keys - always make heavy use of interconnect communication. Because the Oracle Database also enables parallel execution within the context of a node, parallel operations do not always have to use interconnect communication, thus avoiding a potential bottleneck at the interconnect.

The following section will explain Oracle's data redistribution capabilities using the simple example of table joins without any secondary data structures, such as indexes or materialized views.

Serial join

In a serial join a single session reads both tables and performs the join. In this example we assume two large tables CUSTOMERS and SALES are involved in the join. The database uses full table scans to access both tables. For a serial join the single serial session (red arrows) can perform the full join because all matching values from the CUSTOMERS table are read by one process. Figure 11 below depicts the serial join².

---

² Please note that the figures in this section represent logical diagrams to explain data redistribution. In an actual database environment data would typically be striped across multiple physical disks, accessible to any parallel server. This complexity has deliberately been left out from the images.
Processing the same simple join in parallel, a redistribution of rows will become necessary. PX servers scan parts of either table based on block ranges and in order to complete the join, rows have to be distributed between PX servers. Figure 12 depicts the data redistribution for a parallel join at a DOP 2, represented by the green and red arrow respectively. Both tables are read in parallel by both the red and green process (using block-range granules) and then each PX server has to redistribute its result set based on the join key to the subsequent parallel join operator.

SELECT <column> 
FROM Customers c, 
    Sales s 
WHERE s.cust_id=c.cust_id 

Figure 11: Serial Join where red arrow is the serial session

Parallel joins
There are many data redistribution methods. The following 5 are the most common ones:

**HASH**: Hash redistribution is very common in parallel execution in order to achieve an equal distribution of work for individual PX servers based on a hash distribution. Hash (re)distribution is the basic parallel execution enabling mechanism for most data warehouse database systems.

**BROADCAST**: Broadcast redistribution happens when one of the two result sets in a join operation is much smaller than the other result set. Instead of redistributing rows from both result sets the database sends the smaller result set to all PX servers in order to guarantee the individual servers are able to complete their join operation. The small result set may be produced in serial or in parallel.

**RANGE**: Range redistribution is generally used for parallel sort operations. Individual PX servers work on data ranges so that the QC does not have to do any sorting but only to present the individual parallel server results in the correct order.

**KEY**: Key redistribution ensures result sets for individual key values to be clumped together. This is an optimization that is primarily used for partial partition-wise joins (see further down) to ensure only one side in the join has to be redistributed.

**ROUND ROBIN**: Round-robin data redistribution can be the final redistribution operation before sending data to the requesting process. It can also be used in an early stage of a query when no redistribution constraints are required.

As a variation on the data redistribution methods you may see a **LOCAL** suffix in a parallel execution plan on a Real Application Clusters (RAC) database. **LOCAL** redistribution is an
optimization in RAC to minimize interconnect traffic for inter-node parallel queries. For example you may see a **BROADCAST LOCAL** redistribution in an execution plan indicating that the row set is produced on the local node and only sent to the PX servers on that node.

![Data redistribution for a simple parallel join using HASH distribution.](image)

Data redistribution is shown in the SQL execution plan in the **PQ Distrib** column. The execution plan for the simple parallel join illustrated in Figure 13.

**Parallel partition-wise joins**

If at least one of the tables accessed in the join has been partitioned on the join key the database may decide to use a partition-wise join. If both tables are equi-partitioned on the join key the database may use a full partition-wise join. Otherwise a partial partition-wise join may be used in which one of the tables is dynamically partitioned in memory followed by a full partition-wise join.
A partition-wise join does not require any data redistribution because individual PX servers will work on the equivalent partitions of both joined tables.

As shown in Figure 14, the red PX server reads data partition one of the CUSTOMERS table AND data partition one of the SALES table; the equi-partitioning of both tables on the join key guarantees that there will no matching rows for the join outside of these two partitions. The PX server will always be able to complete the full join by reading just these matching partitions. The same is true the green PX server, too, and for any pair of partitions of these two tables. Note that partition-wise joins use partition-based granules rather than block-based granules.

The partition-wise join is the fundamental enabler for shared nothing systems. Shared nothing systems typically scale well as long as they can take advantage of partition-wise joins. As a result, the choice of partitioning (distribution) in a shared nothing system is key as well as the access path to the tables. Operations that do not use partition-wise operations in an MPP system often do not scale well.

In-Memory Parallel Execution

Traditionally, parallel processing by-passed the database buffer cache for most operations, reading data directly from disk (via direct path IO) into the PX server’s private working space. Only objects smaller then PARALLEL_MIN_TABLE_THRESHOLD (default 2% of DB_CACHE_SIZE) would by cached in the database buffer cache of an instance, and most objects accessed in parallel are larger than this limit. This behavior meant that parallel processing rarely took advantage of the available memory other than for its private processing. However, over the last decade hardware systems have evolved quite dramatically; the memory capacity on a typical
database server is now in the double or triple digit Gigabyte range. This, together with Oracle’s compression technologies and the capability of Oracle Database 11g Release 2 to leverage the aggregated database buffer cache of a Real Application Clusters (RAC) environment now enables caching of objects in the Terabyte range.

In-Memory Parallel Execution (In-Memory PX) introduced in Oracle Database 11g Release 2, takes advantage of this large aggregated database buffer cache. By having PX servers access objects via the database buffer cache they can scan data at least 10 X faster than they can on disk.

With In-Memory PX, when a SQL statement is issued in parallel, a check is conducted to determine if the objects accessed by the statement should be cached in the aggregated buffer cache of the system; an object can either be a table, an index, or in the case of partitioned objects one or multiple partitions.

The decision to use the aggregated buffer cache is based on an advanced set of heuristics that include; the size of an object, the frequency at which the object changes and is accessed, and the size of the aggregated buffer cache. If the object meets these criteria it will be fragmented (broken up into pieces) and distributed to all participating nodes: each fragment will be deterministically mapped (affinitized) to a specific RAC node and stored in its buffer cache. If the object is hash partitioned then each partition becomes a fragment, otherwise the mapping is based on the FileNumber and ExtentNumber.

Once a fragment has been mapped all subsequent accesses of that fragment will happen on that node. If a subsequent parallel SQL statement that requires the same data is issued from anywhere in the cluster, the PX servers on the nodes where the data resides will access the data in its buffer cache and return only the result to the node where the statement was issued; no data is moved between nodes via Cache Fusion.

If an object is not considered to be cached it will be accessed via direct path IO to prevent the contents of the buffer cache being unnecessarily displaced.
While not a new concept in Oracle Database 11g Release 2, table functions are an important aspect of parallelism. The goal of a set of table functions is to build a parallel processing pipeline leveraging the parallel processing framework in the database.

A table function encapsulates complex logic in a PL/SQL construct while allowing you to process your data in parallel. A table function also allows you to stream data to the next consumer, building up the pipeline with providers and consumers. This behavior can be best viewed when looking at a small SQL example:

```sql
select *
from table(oracle_map_reduce.reducer(cursor(
    select * from table(oracle_map_reduce.mapper(cursor(
        select * from sls))) map_result));
```

As can be seen, the inner most action is a select from a table, this data then streams into a parallel table function which, after manipulating the data streams it into the next table function. That table function then acts as the data source for the ultimate select.

As the naming above implies, this is very similar to the current popular MapReduce framework used on top of Hadoop. For more information refer to this article.
Controlling Parallel Execution

By default the Oracle Database is enabled for parallel execution and when Automatic Degree of Parallelism (Auto DOP) is active, the database automatically decides if a statement should execute in parallel or not and what DOP it should use. The decision to use parallel execution and the DOP chosen are based on the resource requirements of a statement. If you choose not to activate AUTO DOP then you have three ways to enable a query to execute in parallel:

1) Enable the table(s) for parallel execution:
   ```sql
   alter table sales parallel;
   alter table customers parallel;
   ```
   Use this method if you generally want to execute operations accessing these tables in parallel.

2) Use a parallel hint in the SQL statement:
   ```sql
   SELECT /*+ parallel(c) parallel(s) */
   c.state_province,
   sum(s.amount) revenue
   FROM customers c,
   sales s
   WHERE s.cust_id = c.cust_id
   AND s.purchase_date
   BETWEEN TO_DATE('01-JAN-2007','DD-MON-YYYY')
   AND TO_DATE('31-DEC-2007','DD-MON-YYYY')
   AND c.country = 'United States'
   GROUP BY c.state_province;
   ```
   This method is mainly useful for testing purposes, or if you have a particular statement or few statements that you want to execute in parallel, but most statements run in serial.

3) Use an alter session command:
   ```sql
   alter session force parallel query ;
   ```
   This method is useful if your application always runs in serial except for this particular session that you want to execute in parallel. A batch operation in an OLTP application may fall into this category.
Now that you know how to enable parallel execution, you may wonder how you control the number of PX servers used by each SQL statement. Obviously, you want to use a lot of resources to reduce the response times, but if too many operations take this approach, the system may soon be starved for resources, as you can't use more resources than you have. The Oracle Database also has built-in limits and settings to prevent system overload and ensures that the database remains available to applications. The database initialization parameter PARALLEL_MAX_SERVERS is a good example of one of these limits. All processes in the database require resources, including memory and while active, CPU and I/O resources. The system will not allocate more PX servers on a system than the setting of this initialization parameter.

Understand your target workload

Parallel execution can enable a single operation to utilize all system resources. While this may not be a problem in certain scenarios there are many cases where this would not be desirable. Consider the workload to which you want to apply parallel execution to get optimum use of the system while satisfying your requirements.

Single-user workload

The single-user workload is a workload in which there is only a single operation executing on the database and the objective is for this operation to finish as fast as possible. An example for this type of workload is a large overnight batch load that populates database tables or gathers statistics. Also benchmark situations often measure maximum performance in a single-user workload. In a single-user workload all resources can be allocated to improve performance for the single operation.

Multi-user concurrent workload

Most production environments have a multi-user workload. Users concurrently execute queries – often ad-hoc type queries – and/or concurrent data load operations take place.

In a multi-user environment, workload resources must be divided amongst concurrent operations. End-users will expect a fair amount of resources to be allocated to their operation in order to get predictable response times.

Managing the degree of parallelism

The number of parallel execution servers associated with a single operation is known as the degree of parallelism (DOP). Oracle’s parallel execution framework enables you to either explicitly chose - or even enforce - a specific DOP, or you can rely on Oracle to control it.
DEFAULT parallelism

In the earlier example we simply specify that an object be accessed in parallel. We did not specify the DOP. In such cases Oracle used the so-called DEFAULT parallelism. DEFAULT parallelism uses a formula to determine the DOP based on the system configuration\(^3\), typically \(2 \times \text{CPU\_COUNT}\); in a cluster configuration \(2 \times \text{CPU\_COUNT} \times \text{ACTIVE\_INSTANCE\_COUNT}\). So, on a four node cluster with each node having 8 CPU cores, the default DOP would be \(2 \times 8 \times 4 = 64\).

The DEFAULT algorithm was designed to use maximum resources assuming the operation will finish faster if you use more resources. DEFAULT parallelism targets the single-user workload. In a multi-user environment DEFAULT parallelism will rapidly starve system resources leaving no available resources for other users to execute in parallel.

Fixed Degree of Parallelism (DOP)

Unlike the DEFAULT parallelism, a specific DOP can be requested from the Oracle database. For example, you can set a fixed DOP at a table or index level:

```sql
ALTER table customers parallel 8;
ALTER table sales parallel 16;
```

In this case queries accessing just the customers table use a requested DOP of 8, and queries accessing the sales table will request a DOP of 16. A query accessing both the sales and the customers table will be processed with a DOP of 16 and potentially allocate 32 parallel servers (producer/consumer); whenever different DOPs are specified, Oracle is using the higher DOP\(^4\).

Automatic Degree of Parallelism

When Automatic Degree of Parallelism (Auto DOP) is active, the database automatically decides if a statement should execute in parallel or not and what DOP it should use. The decision to use parallel execution and the DOP chosen are based on the resource requirements of a statement. If the estimated elapse time for the statement is less than `PARALLEL_MIN_TIME_THRESHOLD` (default is `AUTO` which meaning 10 seconds) the statement will run serial.

If the estimated elapse time is greater than `PARALLEL_MIN_TIME_THRESHOLD` the optimizer uses the cost of all scan operations (full table scan, index fast full scan, and so on) in the

---

\(^3\) We are oversimplifying here for the purpose of an easy explanation. The multiplication factor of two is derived by the init.ora parameter `parallel_threads_per_cpu`, an OS specific parameter that is set to two on most platforms

\(^4\) Some statements do not fall under this rule, such as a parallel CREATE TABLE AS SELECT; a discussion of these exceptions is beyond the scope of this paper.
execution plan to determine the so-called ideal DOP for the statement. The actual scan costs are directly dependent on size of the accessed objects, ensuring a deterministic DOP (as long as the size of the accessed objects is not changing dramatically).

However, the optimizer will cap the actual DOP used to ensure parallel server processes do not flood the system. This cap is set by the parameter PARALLEL_DEGREE_LIMIT. The default for value for this parameter is CPU, which means DOP is limited by the number of CPUs on the system. The formula used to derive PARALLEL_DEGREE_LIMIT is

\[ \text{PARALLEL_THREADS_PER_CPU} \times \text{CPU_COUNT} \times \text{ACTIVE_INSTANCE_COUNT} \]

The optimizer will compare its ideal DOP with PARALLEL_DEGREE_LIMIT and take the lowest value.

\[ \text{ACTUAL DOP} = \min(\text{IDEAL DOP, PARALLEL_DEGREE_LIMIT}) \]

It is possible to set the PARALLEL_DEGREE_LIMIT to a specific number, thus allowing you to control the maximum DOP that can get used on your system.

The following diagram shows in detail how the decisions to parallelize a statement and what DOP to use are made.

![Diagram](image)

Figure 17: How Auto DOP works

The final DOP selected is shown and explained in the notes section of an execution plan, visible either using the explain plan command or V$SQL_PLAN. The plan below was generated on a single instance database running on an 8 CPU server. In the note section, you will notice that a
DOP of 16 has been selected. 16 is the maximum DOP allowed by PARALLEL_DEGREE_LIMIT on this system (2 * 8).

Auto DOP in Oracle Database 11g release 2 is controlled by the initialization parameter PARALLEL_DEGREE_POLICY:

MANUAL: The default value for PARALLEL_DEGREE_POLICY is MANUAL, which means Auto DOP is not active.

LIMITED: When set to LIMITED, Auto DOP is only applied to statements that access tables or indexes decorated with the PARALLEL clause but without an explicit DOP. Tables and indexes that have a specific or fixed DOP specified will use that specified DOP.

AUTO: When set to AUTO, Auto DOP will be applied to ALL SQL statements executed on the system, regardless if they access an object decorated with a PARALLEL clause or not. Given this behavior, it is possible for more SQL statement to execute in parallel on Oracle Database 11g Release 2 then did on previous releases.

This initialization parameter can be applied on a system or session level. Furthermore, Auto DOP can be manually invoked for a specific SQL statement using the hint PARALLEL(AUTO):

SELECT /*+ PARALLEL(AUTO) */ SUM(l_quantity * l_extendedprice) total_rev FROM Lineitem;

Adaptive parallelism

Prior to the introduction of Auto DOP, adaptive parallelism looked at the individual SQL statement to determine the ideal DOP for a statement. The old adaptive parallelism capabilities were assessing the parallel resources given to an individual statement based on the actual system workload: Oracle decided at SQL execution time whether a parallel operation should receive the
requested DOP or be throttled down to a lower DOP based on the concurrent workload on the system.

In a system that makes aggressive use of parallel execution by using a high DOP the adaptive algorithm would throttle down the DOP with only few operations running in parallel. While the algorithm will still ensure optimal resource utilization, users may experience inconsistent response times. Using solely the adaptive parallelism capabilities in an environment that requires deterministic response times is not advised.

Adaptive parallelism is controlled through the database initialization parameter PARALLEL_ADAPTIVE_MULTI_USER.

Auto DOP is the preferred method over adaptive parallelism to control the DOP in a multi-user environment.

Workload management for Parallel Execution

Regardless of your expected workload pattern you want to ensure that Oracle’s parallel execution capabilities are used most optimally for your environment. This implies two basic tasks, (a) controlling the usage of parallelism and (b) ensuring that the system does not get overloaded while adhering to the potential different priorities for different user classes in the case of mixed workload environments.

Once a SQL statement starts execution at a certain DOP it will not change the DOP throughout its execution. As a consequence, if you start with a low DOP – either as a result of adaptive parallel execution or because there were simply not enough parallel servers available - it may take longer than anticipated to complete the execution of the SQL statement. If the completion of a statement is time-critical then you may want to either guarantee a minimal DOP and not execute a statement if the appropriate resources are not available or by delaying (queueing) its execution until the necessary resources become available.

Guaranteeing a minimal DOP

You can guarantee a minimal DOP by using the initialization parameter PARALLEL_MIN_PERCENT. This parameter controls the minimal percentage of parallel server processes that must be available to start the operation; it defaults to 0, meaning that Oracle will always execute the statement, irrespective of the number of available parallel server processes.

For example, if you want to ensure to get at least 50% of the requested parallel server processes for a statement:

```sql
SQL> alter session set parallel_min_percent=50 ;

SQL> SELECT /*+ PARALLEL(s,128) */ count(*)
FROM sales s ;

SELECT /*+ PARALLEL(s,128) */ count(*)FROM sales s
``
ERROR at line 1:
ORA-12827: insufficient parallel query slaves available

If there are insufficient parallel query servers available – in this example less than 64 PX servers for a simple SQL statement (or less than 128 slaves for a more complex operation, involving producers and consumers) - you will see ORA-12827 and the statement will not execute. You can capture this error in your code and retry later.

Statement Queuing

With statement Queuing, Oracle will queue SQL statements that require parallel execution if the desired number of PX server processes are not available. Once the necessary resources become available, the SQL statement will be dequeued and allowed to execute. By queuing the statements rather than allowing them to execute with a lower DOP or even serial, Oracle guarantees that the statement will execute with the requested DOP.

The statement queue is a First In - First Out (FIFO) queue based on the time a statement was issued. Statement queuing will kick-in once the number of parallel server processes active on the system is equal to or greater than PARALLEL_SERVERS_TARGET. By default, this parameter is set to

\[ 4 \times CPU\_COUNT \times PARALLEL\_THREADS\_PER\_CPU \times ACTIVE\_INSTANCE\_COUNT \]

PARALLEL_SERVERS_TARGET is not the maximum number of PX server allowed on the system, but the number available to run parallel statements before statement queuing will be used. It is set lower than the maximum number of PX servers allowed on the system, controlled by PARALLEL_MAX_SERVERS, to ensure each parallel statement will get all of the PX server...
resources required and to prevent overloading the system with PX servers. Note all serial (non-parallel) statements will execute immediately even if statement queuing has been activated.

You can identify which SQL statements are being queued using \[GV\|V\]$SQL_MONITOR or the SQL MONITOR screens in Oracle Enterprise Manager (EM).

\[
\text{SELECT sql_id, sql_text}
\text{FROM [GV|V]$SQL_MONITOR}
\text{WHERE status='QUEUED';}
\]

There are also two wait events to help identity if a statement has been queued. A statement waiting on the event ‘PX QUEUING: statement queue’ is the first statement in the statement queue. Once the necessary resources become available for this statement, it will be dequeued and executed. All other statements in the queue will be waiting on ‘ENQ_JX SQL statement queue’. Only when a statement gets to the head of the queue will the wait event switch to ‘PX QUEUING: statement queue’.

It is possible to bypass the statement queue by using a hint NO_STMT_QUEUING.

\[
\text{SELECT /*+ NO_STMT_QUEUING */ SUM(l_quantity * l_extendedprice) total_rev}
\text{FROM Lineitem;}
\]

Statement queuing is only active when the parameter PARALLEL_DEGREE_POLICY is set to AUTO. If you want to ensure a statement will not execute until it can get all of its desired parallel server processes when statement queuing is not active you can use the hint STMT_QUEUING.
Resource Manager

The Oracle Database Resource Manager (DBRM) enables you to prioritize work within an Oracle Database and restrict access to resource for certain groups of users. It is highly recommended to use DBRM if a system is CPU bound, as it will protect high priority users or jobs from being impacted by lower priority work. It provides this protection by allocating CPU time to different jobs based on their priority. In order to use DBRM you will need to create consumer groups, which are groups of users based on a given characteristics, for example username or role. You then create a resource plan that specifies how the resources are to be distributed among various consumer groups. The resources include percentages of CPU time, number of active sessions, and amount of space available in the undo tablespace. You can also restrict parallel execution for users within a consumer group. DBRM is the ultimate final deciding factor in determining the maximum degree of parallelism, and no user in a consumer group (using a specific resource plan) will ever be able to run with a higher DOP than the resource group’s maximum.

For example, if your resource plan has a policy of using a maximum DOP of 4 and you request a DOP of 16, your SQL will run with a DOP of 4. This is true irrespective of how the DOP of 16 was derived, whether explicitly as table attribute or hint or whether it was derived through Oracle’s Auto DOP capabilities.

Furthermore, DBRM can control the maximum number of active sessions for a given resource group. So for the shown resource plan DW_USERS a maximum of 4 sessions are allowed to be active, resulting in a total maximum resource consumption of 4 (sessions) x 4 (DOP) x 2 (slave sets) = 32 PX servers.

Initialization parameters controlling parallel execution

There are 14 different initialization parameters that can be used to control or govern parallel execution behavior in the database. However, you really only need to worry about three:

- **PARALLEL_MAX_SERVERS**: the maximum number of PX servers that can be started by the database instance. In order to execute an operation in parallel, PX servers must be available (i.e. not in use by another parallel operation). By default the value for `PARALLEL_MAX_SERVERS` is derived from other database settings and typically defaults to `10 * CPU_COUNT * THREADS_PER_CPU`. Going back to the example of counting cars and using help from friends: `PARALLEL_MAX_SERVERS` is the maximum number of friends that you can call for help.

- **PARALLEL_MIN_SERVERS**: the minimum number of PX servers that are always started when the database instance is running. `PARALLEL_MIN_SERVERS` enables you to avoid any delay in the execution of a parallel operation by having the PX servers already spawned. By default the value is set to 0. In the counting cars example `PARALLEL_MIN_SERVERS` is the number of friends that are there with you that you don't have to call in order to start the job of counting the cars.
**PARALLEL_DEGREE_POLICY**: controls whether or not In-Memory PX, Auto DOP and statement queuing are enabled. The default value is **MANUAL**, which disables these new Oracle Database 11gR2 feature for backward compatibility.

When set to **LIMITED** only Auto DOP will be enabled. In-Memory PX and statement queuing are disabled. Auto DOP is only applied to statements that access tables or indexes decorated with the PARALLEL clause but without specifying an explicit DOP as table attribute; tables and indexes that have a specific DOP specified will use that specified DOP.

When set to **AUTO** Auto DOP, statement queuing, and In-Memory PX are all enabled. Auto DOP will be applied to all SQL statement regardless to whether or not they access object that have been explicitly decorated with a PARALLEL clause.

The table below gives a brief explanation of the other 11 parallel execution initialization parameters.

<table>
<thead>
<tr>
<th>PARAMETER NAME</th>
<th>DEFAULT VALUE</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>PARALLEL_ADAPTIVE_MULTI_USER</td>
<td>TRUE</td>
<td>Throttles DOP of a statement based on concurrent work load. Can lead to non-deterministic response times</td>
</tr>
<tr>
<td>PARALLEL_AUTOMATIC_TUNING</td>
<td>FALSE</td>
<td>Deprecated, retained for backward compatibility only</td>
</tr>
<tr>
<td>PARALLEL_DEGREE_LIMIT</td>
<td>CPU</td>
<td>Controls MAX DOP used with Auto DOP. CPU means DEFAULT DOP</td>
</tr>
<tr>
<td>PARALLEL_EXECUTION_MESSAGE_SIZE</td>
<td>16KB</td>
<td>Size of the buffers used by the parallel server processes to communicate with each other and the QC</td>
</tr>
<tr>
<td>Parameter</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>-------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>PARALLEL_FORCE_LOCAL</td>
<td>FALSE</td>
<td>In a RAC environment controls if parallel server process will be limited to the node the statement is issued on or not.</td>
</tr>
<tr>
<td>PARALLEL_INSTANCE_GROUP</td>
<td>N/A</td>
<td>Used in conjunction with INSTANCE_GROUPS parameter, it restrict parallel query operations to a limited number of instances.</td>
</tr>
<tr>
<td>PARALLEL_IO_CAP_ENABLED</td>
<td>FALSE</td>
<td>Deprecated, retained for backward compatibility only.</td>
</tr>
<tr>
<td>PARALLEL_MIN_PERCENT</td>
<td>0</td>
<td>Minimum percentage of the requested number of parallel execution processes required for parallel execution.</td>
</tr>
<tr>
<td>PARALLEL_MIN_TIME_THRESHOLD</td>
<td>AUTO</td>
<td>Minimum execution time a statement should have before AUTO DOP kicks in. Default 10 seconds.</td>
</tr>
<tr>
<td>PARALLEL_SERVERS_TARGET</td>
<td>4X DEFAULT DOP</td>
<td>Number of parallel server processes allowed to run before statement queuing kicks in.</td>
</tr>
<tr>
<td>PARALLEL_THREADS_PER_CPU</td>
<td>2</td>
<td>Number of parallel processes a CPU can handle during parallel execution.</td>
</tr>
</tbody>
</table>
Conclusion

The objective of parallel execution is to reduce the total execution time of an operation by using multiple resources concurrently. Resource availability is the most important prerequisite for scalable parallel execution.

The Oracle database provides a powerful SQL parallel execution engine that can run almost any SQL-based operation – DDL, DML and queries – in the Oracle Database in parallel. This paper provides a detailed explanation of how parallel execution is implemented in the Oracle Database and provides a step by step guide to enabling and using SQL parallel execution successful.

Data warehouses should always leverage parallel execution to achieve good performance. Specific operations in OLTP applications, such as batch operations, can also significantly benefit from parallel execution. With the new capabilities of Oracle Database 11g release 2 - In-Memory parallel execution, Auto DOP, and statement queueing - any mixed workload environment will also benefit from Oracle’s parallel execution framework.