Chapter 13

PARALLEL SQL

Parallel SQL enables a SQL statement to be processed by multiple threads or processes simultaneously.

Today's widespread use of dual and quad core processors means that even the humblest of modern computers running an Oracle database will contain more than one CPU. Although desktop and laptop computers might have only a single disk device, database server systems typically have database files spread—striped—across multiple, independent disk devices. Without parallel technology—when a SQL statement is processed in *serial*—a session can make use of only one of these CPUs or disk devices at a time. Consequently, serial execution of a SQL statement cannot make use of all the processing power of the computer. Parallel execution enables a single session and SQL statement to harness the power of multiple CPU and disk devices.

Parallel processing can improve the performance of suitable SQL statements to a degree that is often not possible by any other method. Parallel processing is available in Oracle Enterprise Edition only.

In this chapter we look at how Oracle can parallelize SQL statements and how you can use this facility to improve the performance of individual SQLs or the application as a whole.

UNDERSTANDING PARALLEL SQL

In a serial—nonparallel—execution environment, a single process or thread[1] undertakes the operations required to process your SQL statement, and each action must complete before the succeeding action can commence. The single Oracle process might only leverage the power of a single CPU and read from a single disk at any given instant. Because most modern hardware platforms include more than a single CPU and because Oracle data is often spread across multiple disks, serial SQL execution cannot take advantage of all the available processing power.

For instance, consider the following SQL statement:

```
  SELECT   *
    FROM   sh.customers
ORDER BY   cust_first_name, cust_last_name, cust_year_of_birth
```

If executing without the parallel query option, a single process would be responsible for fetching all the rows in the CUSTOMERS table. The same process would be responsible for sorting the rows to satisfy the ORDER BY clause. Figure 13-1 illustrates the workflow.

We can request that Oracle execute this statement in parallel by using the PARALLEL hint:

```
  SELECT   /*+ parallel(c,2) */ *
    FROM   sh.customers c
ORDER BY   cust_first_name, cust_last_name, cust_year_of_birth
```

If parallel processing is available, the CUSTOMERS table will be scanned by two processes[2] in parallel. A further two processes will be employed to sort the resulting rows. A final process—the session that issued the SQL in the first place—combines the rows and returns the result set. The process that requests and coordinates the parallel processing stream is the *Query coordinator*. Figure 13-2 illustrates this sequence of events.

[1] A process is a unit of execution with its own private memory. A thread is also a unit of execution but shares memory with other threads within a process. On UNIX and Linux Oracle servers, tasks are implemented as processes and on Windows as threads.

[2] Because the PARALLEL hint requested a Degree of Parallelism (DOP) of 2.

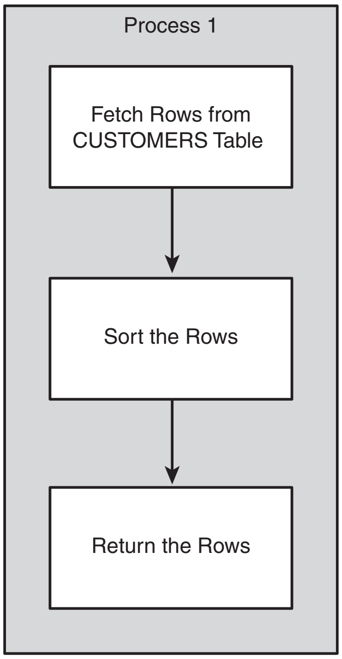


FIGURE 13-1 Serial execution of a SQL statement.

Oracle supports parallel processing for a wide range of operations, including queries, DDL, and DML:

- ❑ Queries that involve table or index range scans
- ❑ Bulk insert, update, or delete operations
- ❑ Table and index creation
- ❑ The collection of object statistics using DBMS_STATS (see Chapter 7, "Optimizing the Optimizer")
- ❑ Backup and recovery operations using Recovery Manager (RMAN)

PARALLEL PROCESSES AND THE DEGREE OF PARALLELISM

The Degree of Parallelism (DOP) defines the number of parallel streams of execution that will be created. In the simplest case, this translates to the number of parallel slave processes enlisted to support your SQL's execution. However, the number of parallel processes is more often twice the DOP. This is because each step in a nontrivial execution plan needs to feed data into the subsequent step, so two sets of processes are required to maintain the parallel stream of processing.

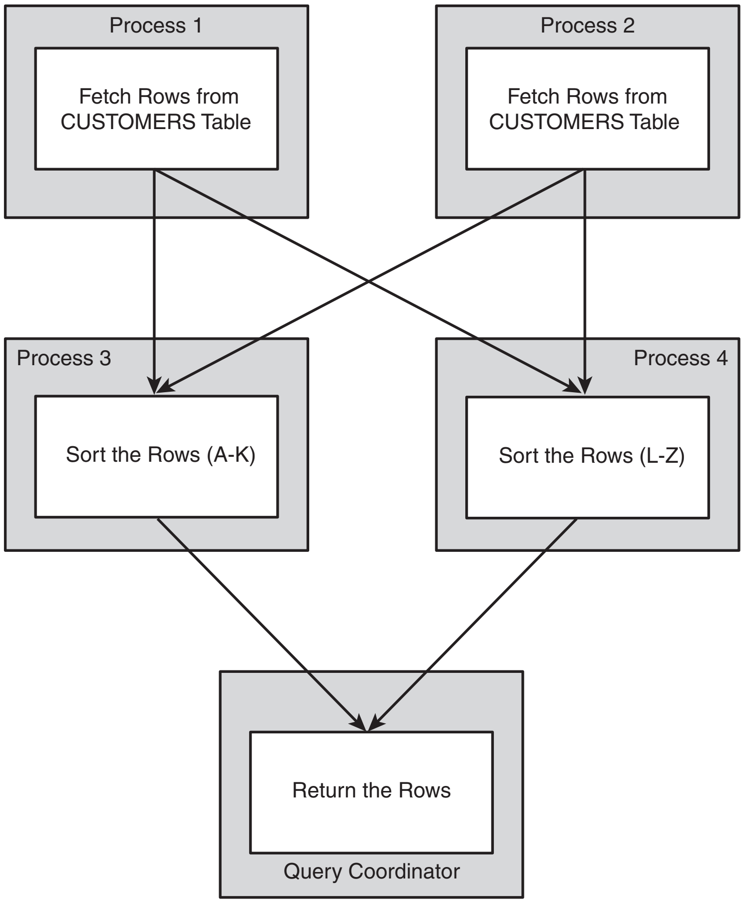


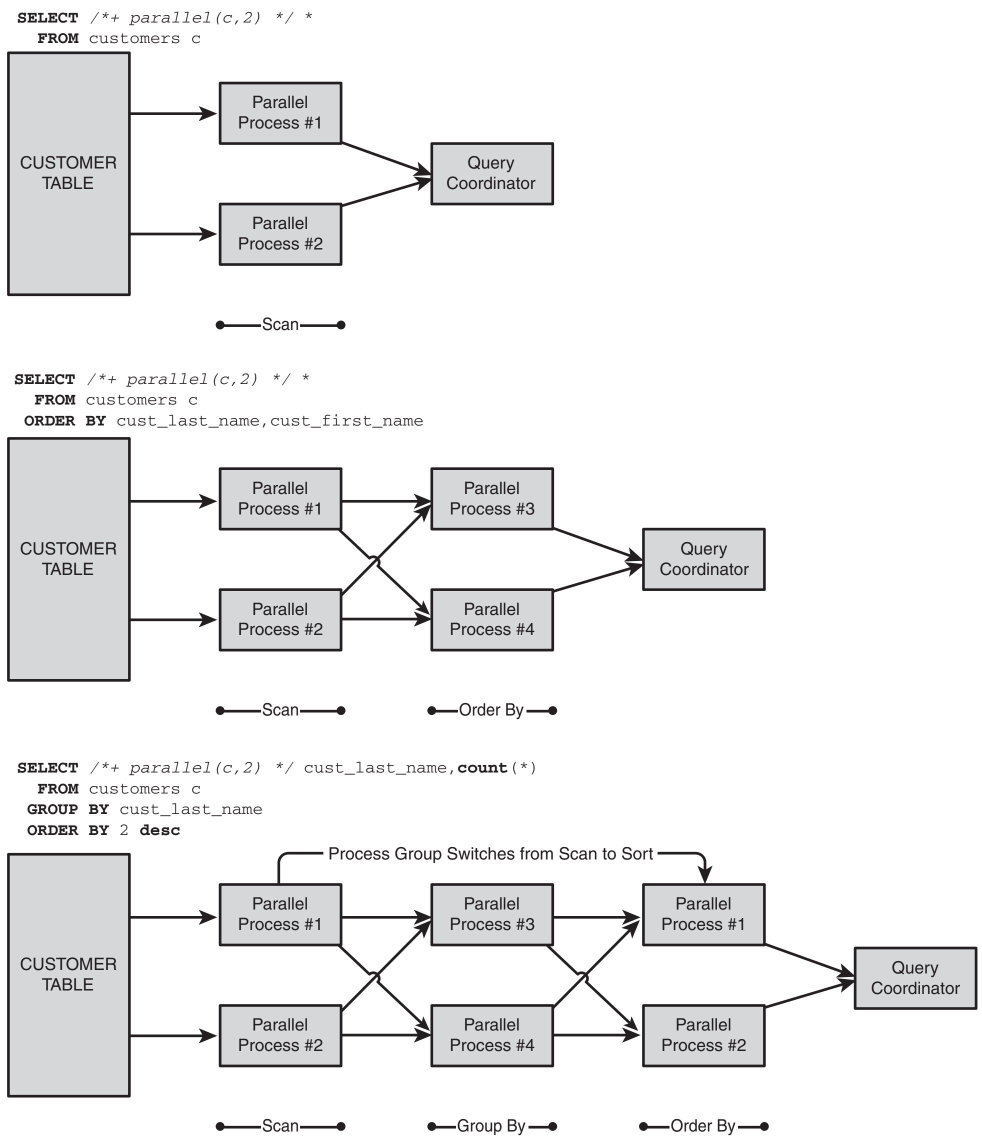
FIGURE 13-2 Parallel Execution.

For instance, if the statement includes a full table scan, an ORDER BY and a GROUP BY, three sets of parallel processes are required: one to scan, one to sort, and one go group. Because Oracle reuses the first set of parallel processes (those that performed the scan) to perform the third operation (the GROUP BY), only two sets of processes are required. As a result of this approach, the number of parallel slaves allocated should never be more than twice the DOP.

Figure 13-3 shows how parallel slaves are allocated for a DOP of 2.

PARALLEL SLAVE POOL

The Oracle server maintains a pool of parallel slave processes available for parallel operations. The database configuration parameters PARALLEL_MIN_ SERVERS



```
SELECT /*+ parallel(c,2) */ *
  FROM customers c
```

```
SELECT /*+ parallel(c,2) */ *
  FROM customers c
 ORDER BY cust_last_name,cust_first_name
```

```
SELECT /*+ parallel(c,2) */ cust_last_name,count(*)
  FROM customers c
 GROUP BY cust_last_name
 ORDER BY 2 desc
```

FIGURE 13-3 Parallel process allocation for a DOP of 2.

and PARALLEL_MAX_SERVERS determine the initial and maximum size of the pool. If insufficient slaves are currently active but the pool has not reached its maximum value, Oracle will create more slaves. After a configurable period of inactivity, slave processes will shut down until the pool is again at its minimum size.

If there are insufficient query processes to satisfy the DOP requested by your statement, one of the following outcomes results:

- ❑ If there are some parallel query slaves available, but less than requested by your SQL statement, your statement might run at a reduced DOP.
- ❑ If there are no parallel query slaves available, your statement might run serially.
- ❑ Under specific circumstances, you might get an error. This will only occur if the database parameter PARALLEL_MIN_PERCENT has been set to a value that is higher than the percentage of required slaves that are available.
- ❑ In Oracle 11g Release 2 and forward, your SQL execution might be delayed until sufficient parallel servers are available.

See the “Parallel Configuration Parameters” section later in this chapter for more information on how to configure these outcomes.

PARALLEL QUERY IO

We discussed in Chapter 2, “Oracle Architecture and Concepts,” and elsewhere, how the Oracle buffer cache helps reduce disk IO by buffering frequently accessed data blocks in shared memory. Oracle has an alternate IO mechanism, *direct path* IO, which it can use if it determines that it would be faster to bypass the buffer cache and perform the IO directly. For instance, Oracle uses direct IO when reading and writing temporary segments for sorting and intermediate result sets. In Oracle 11g onward, Oracle sometimes uses direct path IO in preference to the normal buffered IO for serial table access as well.

When performing Parallel query operations, Oracle normally uses direct path IO. By using direct path IO, Oracle avoids creating contention for the buffer cache and allows IO to be more optimally distributed between slaves. Furthermore, for parallel operations that perform full table scans the chance of finding matching data in the buffer cache is fairly low, so the buffer cache adds little value.

In Oracle 10g and earlier, parallel query always uses direct path IO, and serial query will always use buffered IO.[3] In 11g, Oracle can use buffered IO for parallel query (from 11g release 2 forward), and serial queries might use direct path IO. However, it remains true that parallel queries are less likely to use buffered IO and might, therefore, have a higher IO cost than serial queries. The higher IO cost will, of course, be shared amongst all the parallel processes so the overall performance might still be superior.

[3] Unless the undocumented parameter serial_direct_read has been set to TRUE.

Direct path and buffered IO are discussed in more detail within Chapter 21, "Disk IO Tuning Fundamentals."

PARALLEL PERFORMANCE GAINS

The performance improvements that you can expect to obtain from parallel SQL depend on the suitability of your host computer, Oracle configuration, and the SQL statement. If all the conditions for parallel processing are met, you can expect to get substantial performance improvements in proportion to the DOP employed.

On many systems, the limit of effective parallelism will be determined by segment spread, not by hardware configuration. For instance, if you have 32 CPUs and 64 independent disk devices, you might hope for effective parallelism up to at least a DOP of 32 or maybe even 64. However, if the table you are querying is spread over only 6 disks, you are likely to see performance improvements reduce as you increase the DOP beyond 6 or so.

Figure 13-4 illustrates the improvements gained when increasing the DOP for a SQL statement that performs a table scan and GROUP BY of a single table.

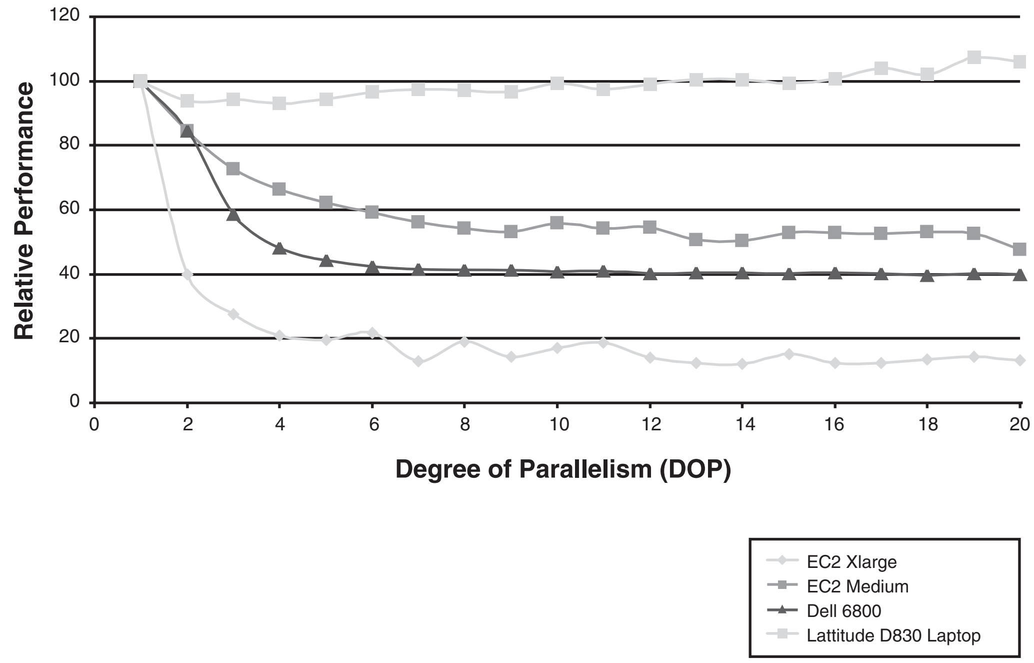


FIGURE 13-4 Improvement gains for various DOPs on various host configurations.

The host configurations shown are

- ❑ An Amazon CPU-intensive Extra Large EC2 image. This is a virtual server running in Amazon's AWS cloud that has the equivalent of 8 × 2.5-GHz CPUs and has storage on a widely striped SAN.
- ❑ An Amazon CPU-intensive Medium EC2 image. This is similar to the extra large image, but has only 2 CPUs.
- ❑ A Dell 6800 4 CPU server with disk storage on a widely striped SAN using ASM.
- ❑ A Dell latitude D830 laptop (my laptop). It is dual core, but all data files are on a single disk.

In each case, the parallel SQL was the only SQL running.

These examples show that for suitably configured systems, performance gains were greater the more CPUs that were available. However, attempting to use parallel on a host that is unsuitable (as in my laptop) is futile at best and counter-productive at worst.

> The performance gains achieved through parallal processing are most dependent on the hardware configuration of the host. To get benefits from parallel processing, the host should possess multiple CPUs and data should be spread across multiple disk devices.

DECIDING WHEN TO USE PARALLEL PROCESSING

A developer once saw me use the parallel hint to get a rapid response to an ad-hoc query. Shortly thereafter, every SQL that developer wrote included the parallel hint, and system performance suffered as the database server became overloaded by excessive parallel processing.

The lesson is obvious: If every concurrent SQL in the system tries to use *all* the resources of the system, parallel makes performance worse, not better. Consequently, we should use parallel only when doing so improves performance without degrading the performance of other concurrent database requests.

The following sections discuss some of the circumstances in which you can effectively use parallel SQL.

YOUR SERVER COMPUTER HAS MULTIPLE CPUS

Parallel processing will usually be most effective if the computer that hosts your Oracle database has multiple CPUs. This is because most operations performed by the Oracle server (accessing the Oracle shared memory, performing sorts, disk accesses) require CPU. If the host computer has only one CPU, the parallel processes might contend for this CPU, and performance might actually decrease.

Almost every modern computer has more than one CPU; dual-core (2 CPUs in a single processor slot) configurations are the minimum found in systems likely to be running an Oracle server including the desktops and laptops running development databases. However, databases running within Virtual machines might be configured with only a single (virtual) CPU.

THE DATA TO BE ACCESSED IS ON MULTIPLE DISK DRIVES

Many SQL statements can be resolved with few or no disk accesses when the necessary data can be found in the Oracle buffer cache. However, full table scans of larger tables—a typical operation to be parallelized—tends to require significant physical disk reads. If the data to be accessed resides on a single disk, the parallel processes line up for this disk, and the advantages of parallel processing might not be realized.

Parallelism will be maximized if the data is spread evenly across the multiple disk devices using some form of striping; we discuss principles of striping in Chapter 21.

THE SQL TO BE PARALLELIZED IS LONG RUNNING OR RESOURCE-INTENSIVE

Parallel SQL suits long running or resource-intensive statements. There is an overhead in activating and coordinating multiple parallel query processes and in co-coordinating the flow of information between these processes. For short-lived SQL statements, this overhead might be greater than the total SQL response time.

Parallel processing is typically used for

❏ Long-running reports
❏ Bulk updates of large tables
❏ Building or rebuilding indexes on large tables
❏ Creating temporary tables for analytical processing
❏ Rebuilding a table to improve performance or to purge unwanted rows

Parallel processing is not usually suitable for transaction processing environments. In these environments, multiple sessions process transactions concurrently. Full use of available CPUs is already achieved because each concurrent transaction can use a different CPU. Implementing parallel processing might actually degrade overall performance by allowing a single user to monopolize multiple CPUs.

Parallel processing is suitable for long-running operations in low-concurrency environments. Parallel processing is less suitable for OLTP style databases.

THE SQL PERFORMS AT LEAST ONE FULL TABLE, INDEX, OR PARTITION SCAN

Parallel processing is generally restricted to operations that include a scan of a table, index, or partition. However, the SQL might include a mix of operations, only some of which involve scans. For instance, a nested loops join that uses an index to join two tables can be fully parallelized providing that the driving table is accessed by a table scan.

Although queries that are driven from an index lookup are not normally parallelizable, if a query against a partitioned table is based on a local partitioned index, each index scan can be performed in parallel against the table partition corresponding to the index partition. We see an example of this later in the chapter.

THERE IS SPARE CAPACITY ON YOUR HOST

You are unlikely to realize the full gains of parallel processing if your server is at full capacity. Parallel processing works well for a single job on an underutilized, multi-CPU machine. If all CPUs on the machine are busy, your parallel processes will bottleneck on the CPU and performance will be degraded.

Remember that when a session uses parallel query, it requests a greater share of machine resources. If many processes simultaneously attempt to run in parallel, the result will usually be that some fail to run at the requested degree of parallelism whereas others acquire more than their fair share of resources.

THE SQL IS WELL TUNED

Parallelizing a poorly tuned SQL might well reduce its execution time. However, you'll also be magnifying the impact of that SQL on the database server and increasing its impact on other sessions. You should make sure that the SQL is efficient before attempting to grant it access to more of the database server's resources. Parallelizing the SQL is *not* an alternative to tuning the SQL.

CONFIGURING PARALLEL PROCESSING

Oracle tries to automate the configuration of the system to maximize the performance of parallel operations. However, there's still a lot of scope for manually tweaking the database and SQL for optimal parallel performance.

DETERMINING THE DEGREE OF PARALLELISM

An optimal DOP is critical for good parallel performance. Oracle determines the DOP as follows:

❑ If parallel execution is indicated or requested, but no DOP is specified, the default DOP is set to twice the number of CPU cores on the system. For a

RAC system, the DOP will be twice the number of cores in the entire cluster. This default is controlled by the configuration parameter PARALLEL_THREADS_PER_CPU.

- ❑ From Oracle 11g release 2 forward, If PARALLEL_DEGREE_POLICY is set to AUTO, Oracle will adjust the DOP depending on the nature of the operations to be performed and the sizes of the objects involved.
- ❑ If PARALLEL_ADAPTIVE_MULTI_USER is set to TRUE, Oracle will adjust the DOP based on the overall load on the system. When the system is more heavily loaded, the DOP will be reduced.
- ❑ If PARALLEL_IO_CAP is set to TRUE in Oracle 11g or higher, Oracle will limit the DOP to that which the IO subsystem can support. These IO subsystem limits can be calculated by using the procedure DBMS_RESOURCE_MANAGER.CALIBRATE_IO.
- ❑ A DOP can be specified at the table or index level by using the PARALLEL clause of CREATE TABLE, CREATE INDEX, ALTER TABLE, or ALTER INDEX.
- ❑ The PARALLEL hint can be used to specify the DOP for a specific table within a query.
- ❑ Regardless of any other setting, the DOP cannot exceed that which can be supported by PARALLEL_MAX_SERVERS. For most SQL statements, the number of servers required will be twice the requested DOP.

As we saw in Figure 13-4, increasing the DOP beyond an optimal point fails to result in further performance increases. However, increasing the DOP beyond optimal *can* have a significant negative effect on overall system performance. Although the SQL being parallelized might not degrade significantly as the DOP increases, load on the system continues to increase and can cause other SQLs running concurrently to suffer reduced response time.

Figure 13-5 shows how increasing the DOP influences CPU utilization. As we hit the optimal DOP—approximately 8 for this system—the reduction in query-elapsed time flattens out. However, the time other sessions spend waiting for CPU to become available continues to increase. Other sessions wanting to access the CPU will need to wait, resulting in degraded response time.

Increasing the DOP beyond the optimal level might overload the host, degrading the performance of other SQLs.

PARALLEL HINTS

The PARALLEL hint can invoke parallel processing. In its simplest form, the hint takes no argument as in the following example:

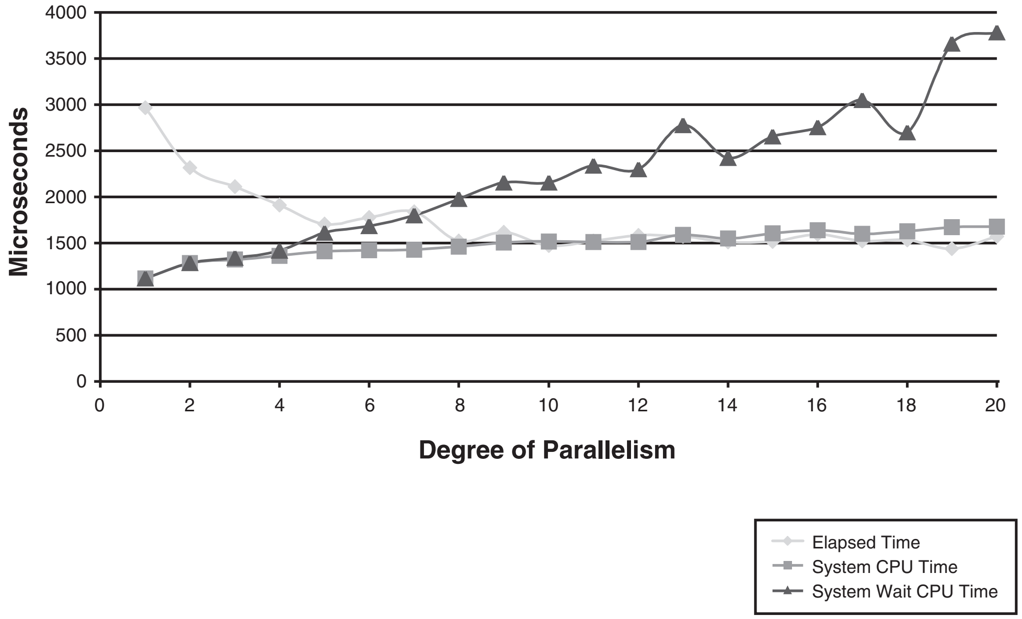


FIGURE 13-5 Increasing the DOP causes increases in system CPU wait times.

```
SELECT /*+ parallel */ *  FROM sh.sales s
```

It’s legal, but not always necessary to specify a table name or alias in the hint:

```
SELECT /*+ parallel(s) */  *  FROM sh.sales s
```

The hint can request a specific DOP:

```
SELECT /*+ parallel(s,8) */ * FROM sh.sales s;
```

The NOPARALLEL hint can be used to suppress parallel processing:

```
SELECT  /*+ noparallel */ COUNT ( * ) FROM sales;
```

In 11g release 2, the AUTO option allows you to request that the AUTO setting for PARALLEL_DEGREE_POLICY be used to calculate the DOP:

```
SELECT  /*+ parallel(auto) */ COUNT ( * ) FROM sales;
```

For ad-hoc query execution, you might want to set an explicit DOP. However, for SQL embedded within an application, this might not be a good idea because the SQL will be less able to adapt to changes in machine configuration (more CPUs for instance), workload (more concurrent sessions), or configuration (changes to the number of parallel slaves or the default DOP). For embedded SQL, it's probably better to omit an explicit DOP or to use the AUTO keyword (in Oracle 11g Release 2 and higher).

PARALLEL CONFIGURATION PARAMETERS

Determining the optimal DOP, especially when taking concurrent system activity into account, is a daunting task. Luckily, Oracle has invested significant effort into automating the process. Each release of Oracle has increased the level of intelligent automation of parallel configuration. In general, you should try Oracle's automation before attempting to manually configure automatic processing.

Nevertheless, significant tweaking is possible; the following lists the significant configuration parameters that you can adjust to optimize parallel SQL:

parallel_adaptive_multi_user	When set to TRUE, Oracle will adjust the DOP to account for the load on the system. On a heavily loaded system, Oracle will reduce the DOP from the requested or default degree.
parallel_degree_limit	In Oracle11g Release 2 and higher, places an absolute limit on the DOP that can be achieved. A value of CPU prevents the DOP from exceeding that specified by parallel_threads_per_cpu. A value of IO sets the maximum to the IO limit determined by running DBMS_RESOURCE_MANAGER.CALIBRATE_IO. AUTO allows Oracle to select a value. An integer value corresponding to a specific DOP might also be specified.
parallel_degree_policy	In 11G release 2 and forward, this parameter controls the means by which the DOP will be calculated. MANUAL equates to the behavior in 11.1 and earlier. If AUTO, the DOP will be calculated based on the types of operations in the SQL statement and the sizes of the tables. AUTO also enables parallel queries to fetch data from the buffer cache rather than using direct path IO and will queue parallel processes if the requested DOP execution is not immediately available.
parallel_execution_message_size	Sets the size of buffers for communication between the processes involved in parallel processing.
parallel_force_local	From Oracle 11g Release 2 forward, this parameter, if set to TRUE, suppresses multi-instance parallelism on RAC clusters.
parallel_io_cap_enabled	This 11g parameter if set to TRUE will limit the DOP to that which Oracle thinks the IO subsystem can support. To use the parameter, you should first use DBMS_RESOURCE_MANAGER.CALIBRATE_IO to determine these IO limits.
parallel_max_servers	The maximum number of parallel servers that can be started. This provides an absolute limit on the amount of concurrent parallel operations that can execute.

parallel_min_percent	If set to nonzero, this parameter determines the minimum acceptable DOP for a query. If the DOP requested or determined cannot be provided due to system load or other parallel processes that are using the parallel server pool, the DOP will be reduced only to the value of PARALLEL_MIN_PERCENT. For instance, if your query requested a DOP of 8 and only 5 were available (5 / 8 = 62%), your query would execute in parallel if PARALLEL_MIN_PERCENT was below 62. If PARALLEL_MIN_PERCENT were above 62, your statement will either terminate with an error or, if PARALLEL_DEGREE_POLICY is set to AUTO, will be queued for later execution.
parallel_min_servers	The minimum number of parallel servers—the number that will be initialized when the database is first started.
parallel_min_time_threshold	Specifies the amount of elapsed time (in seconds) required for a SQL statement to be automatically parallelized. If the estimated elapsed time of a SQL statement exceeds the threshold, Oracle automatically parallelizes the SQL. The default of AUTO results in Oracle automatically calculating a value.
parallel_threads_per_cpu	Sets the number of parallel threads that can be applied per CPU. Oracle generally restricts the DOP so that this limit is not exceeded.

MONITORING PARALLEL SQL

Because multiple processes are involved in parallel execution, explaining, tracing, and monitoring parallel execution can be more complex than for serial SQL.

PARALLEL EXPLAIN PLANS

EXPLAIN PLAN reflects additional steps for a parallelized SQL statement that reflect the additional parallel operations involved in the parallel execution.

For instance, consider this simple SQL statement and explain plan:

```
SQL> EXPLAIN PLAN  FOR
  2     SELECT   * FROM   customers
  3   ORDER BY   cust_last_name;

----------------------------------------
| Id  | Operation          | Name      |
----------------------------------------
|   0 | SELECT STATEMENT   |           |
|   1 |  SORT ORDER BY     |           |
|   2 |   TABLE ACCESS FULL| CUSTOMERS |
----------------------------------------
```

The CUSTOMERS table is scanned, and the rows scanned are sorted.

When the statement is parallelized, additional operations are added to the execution plan:

```
SQL> EXPLAIN PLAN  FOR
  2     SELECT /*+ parallel */ *
  3       FROM   customers
  4   ORDER BY   cust_last_name;

SQL> SELECT * FROM table (DBMS_XPLAN.display
  2                   (null,null,'BASIC +PARALLEL'));

----------------------------------------------------------------------
|Id  | Operation                 | Name      |  TQ  |IN-OUT| PQ Distrib |
----------------------------------------------------------------------
|  0 | SELECT STATEMENT          |           |      |      |            |
|  1 |  PX COORDINATOR           |           |      |      |            |
|  2 |   PX SEND QC (ORDER)      | :TQ10001  | Q1,01| P->S | QC (ORDER) |
|  3 |    SORT ORDER BY          |           | Q1,01| PCWP |            |
|  4 |     PX RECEIVE            |           | Q1,01| PCWP |            |
|  5 |      PX SEND RANGE        | :TQ10000  | Q1,00| P->P | RANGE      |
|  6 |       PX BLOCK ITERATOR   |           | Q1,00| PCWC |            |
|  7 |        TABLE ACCESS FULL| CUSTOMERS| Q1,00| PCWP |            |
----------------------------------------------------------------------
```

The new plan contains a variety of PX steps that describe the parallel operations involved. Let's look at each of these steps:

PX BLOCK ITERATOR	This operation is typically the first step in a parallel pipeline. The BLOCK ITERATOR breaks up the table into chunks that are processed by each of the parallel servers involved.
PX SEND	PX SEND operations simply indicate that data is being sent from one parallel process to another.
PX RECEIVE	PX RECEIVE operations indicate the data being received by one parallel process from another.
PX SEND QC	This is a send operation to the parallel query co-coordinator process.
PX COORDINATOR	This step simply indicates that the parallel query co-coordinator is receiving the data from the parallel streams and returning it to the SQL statement.

Figure 13-6 illustrates how these steps relate to parallel processing with a DOP of 2.

PX SEND and PX RECEIVE operations are associated with distribution options—shown in the "PQ Distrib" column of DBMS_XPLAN—which describe how data is sent from one slave to another. In sort operations it's typical to see

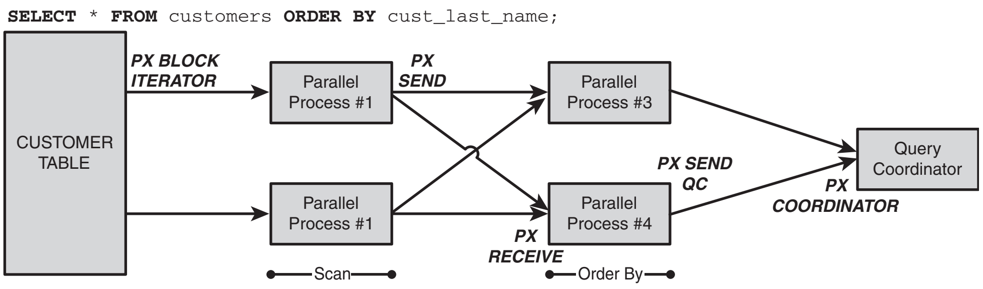


FIGURE 13-6 EXPLAIN PLAN parallel execution steps.

the RANGE option because rows to be sorted are distributed based on the value of the sort columns. For instance when sorting by CUST_FIRST_NAME as in the preceding query, Oracle might send names from A–K to one slave and names from L–Z to the other. Here are the commonly encountered distribution options:

RANGE	Rows are distributed based on ranges of values. This is typical when sort operations are parallelized.
HASH	Rows are distributed to parallel query slaves based on a hash of the value concerned. This is suitable for joins and HASH GROUP BY operations and generally ensures a more even distribution of rows than for RANGE operations.
RANDOM	Rows are randomly assigned to parallel query slaves.
ROUND ROBIN	Rows are distributed one at a time in a circular fashion, just as you would deal cards in a game of poker.

The IN-OUT column of the DBMS_XPLAN output describes how data flows between and within the parallel processes. The column corresponds to the OTHER_TAG column in the PLAN_TABLE table. These columns can contain one of the values shown in Table 13-1.

The presence of a PARALLEL_FROM_SERIAL or S->P tag in the PLAN_TABLE or DBMS_XPLAN output might represent a serial bottleneck in an otherwise parallel execution stream.

TRACING PARALLEL EXECUTION

Using SQL trace to tune our queries becomes somewhat more difficult when the SQL is parallelized. This is because each process involved in the parallel execution has its own trace file. Furthermore, because these processes are shared among all parallelized SQLs and sessions, the trace files contain trace data for other SQLs and sessions in addition to the ones we are interested in.

Table 13-1 Parallel Data Flow Tags

IN-OUT VALUE	OTHER_TAG VALUE	DESCRIPTION
P->P	PARALLEL_TO_PARALLEL	This tag denotes parallel processing that passes results to a second set of parallel processes. For instance, a parallel table scan might have passed results to a parallel sort.
P->S	PARALLEL_TO_SERIAL	This is usually the top level of a parallel query. The results are fed in parallel to the query coordinator.
PCWP PCWC	PARALLEL_COMBINED_WITH_PARENT PARALLEL_COMBINED_WITH_CHILD	The step was executed in parallel. Either the parent step or the child step was also executed in parallel by the same process. For instance, in a parallel nested loops join, the parallel query process scanned the driving table and also issued index lookups on the joined table.
S->P	PARALLEL_FROM_SERIAL	A serial operation that passed results to a set of parallel processes. The presence of this tag can indicate a serial bottleneck within a parallel statement because it suggests that parallel processing might wait on serial processing.

However, it is possible, through a somewhat convoluted process, to trace parallel execution. Here are the steps:

1. Set a unique client identifier in your session using DBMS_SESSION.SET_IDENTIFIER.
2. Enable tracing for that client identifier using DBMS_MONITOR.CLIENT_ID_TRACE_ENABLE.
3. Run your parallel SQL.
4. Use the *trcsess* utility to create a new trace file that contains only trace entries for your client identifier.
5. Analyze the new trace file as usual.

Here we invoke steps 1, 2, and 3:

```
BEGIN
   DBMS_SESSION.set_identifier ('gh pqo test 27');
   DBMS_MONITOR.client_id_trace_enable
     (client_id => 'gh pqo test 27',
      waits => TRUE);
END;
/
```

```
SELECT /*+ parallel */ prod_name, SUM (amount_sold)
  FROM   products JOIN sales
  USING (prod_id)
GROUP BY   prod_name
ORDER BY   2 DESC;
```

Here we perform steps 4 and 5:

```
$ trcsess clientid='gh pqo test 27' output=pqo_test_27.trc *
$ tkprof pqo_test_27.trc pqo_test_27.prf sort='(prsela,fchela,exeela)'

TKPROF: Release 11.1.0.6.0 - Production on Mon Dec 29 19:40:38 2008
Copyright (c) 1982, 2007, Oracle.  All rights reserved.
```

The merged trace file now accurately reflects not only the activity from our invoking session, but also from all the parallel server processes that were involved in executing the query.

To trace a parallel execution, set a Client Identifier and use the trcsess utility to extract trace records for that client identifier into a single file.

Advanced tracing of parallel server activity can also be achieved by using the "_px_trace" facility.[4] For instance

```
ALTER SESSION SET "_px_trace"="compilation","execution","messaging";
```

The 10391 event can also be used to dump information about parallel server allocation:

```
ALTER SESSION SET EVENTS '10391 trace name context forever, level 128';
```

Both of these events generate rather cryptic and sometimes voluminous output and should probably be used only if all other techniques fail to shed light on parallel execution.

THE V$PQ_TQSTAT VIEW

Even with EXPLAIN PLAN and SQL trace output, it's hard to work out exactly how a parallel query executed. For instance, what was the actual DOP? How much work did each parallel server process do?

[4] See Oracle support note 444164.1

The V$PQ_TQSTAT view contains information about the data transferred between each set of parallel query servers, including the number of rows transmitted and received. Unfortunately, the view is visible only from within the session that issued the parallel query and only for the most recent query executed. This limits its usefulness in a production environment, but it is still invaluable when tuning parallel queries.

For instance, consider this parallel query:

```
SQL>   SELECT /*+ parallel */
  2          prod_id, SUM (amount_sold)
  3      FROM   sales
  4  GROUP BY   prod_id
  5  ORDER BY   2 DESC;

----------------------------------------------------------------
| Id  | Operation                    | Name       |    TQ  |IN-OUT|
-----------------------------------------------------------------
|   0 | SELECT STATEMENT             |            |        |      |
|   1 |  PX COORDINATOR              |            |        |      |
|   2 |   PX SEND QC (ORDER)         | :TQ10002   | Q1,02  | P->S |
|   3 |    SORT ORDER BY             |            | Q1,02  | PCWP |
|   4 |     PX RECEIVE               |            | Q1,02  | PCWP |
|   5 |      PX SEND RANGE           | :TQ10001   | Q1,01  | P->P |
|   6 |       HASH GROUP BY          |            | Q1,01  | PCWP |
|   7 |        PX RECEIVE            |            | Q1,01  | PCWP |
|   8 |         PX SEND HASH         | :TQ10000   | Q1,00  | P->P |
|   9 |          HASH GROUP BY       |            | Q1,00  | PCWP |
|  10 |           PX BLOCK ITERATOR  |            | Q1,00  | PCWC |
|  11 |            TABLE ACCESS FULL| SALES      | Q1,00  | PCWP |
----------------------------------------------------------------
```

If we query V$PQ_TQSTAT directly after the query executes[5] we can see the number of rows passed between each of the parallel server sets. Each of the unique TQ_IDs corresponds to one of the interactions between server sets denoted in the execution plan by 'P->P' or 'P->S' values for the IN-OUT column. You can correlate the value of TQ_ID with the TQ column in the EXPLAIN PLAN output.

```
SQL>   SELECT   dfo_number, tq_id, server_Type, MIN (num_rows),
                MAX (num_rows),count(*) dop
  2      FROM   v$pq_tqstat
```

[5] You might need to wait a few seconds to allow parallel server processes to flush their statistics.

```
  3  GROUP BY   dfo_number, tq_id, server_Type
  4  ORDER BY   dfo_number, tq_id, server_type DESC;

DFO_NUMBER      TQ_ID SERVER_TYP MIN(NUM_ROWS) MAX(NUM_ROWS)        DOP
---------- ---------- ---------- ------------- ------------- ----------
         1          0 Producer              72            72          2
         1          0 Consumer              62            82          2
         1          1 Ranger                72            72          1
         1          1 Producer              31            41          2
         1          1 Consumer              35            37          2
         1          2 Producer              35            37          2
         1          2 Consumer              72            72          1
```

For complex parallel SQLs, there might be multiple parallel pipelines that are indicated by different values for the DFO_NUMBER column.

Use the V$PQ_TQSTAT view to measure the actual DOP and amount of data transferred between parallel servers.

OTHER STATISTICS

We can get a view of parallel execution occurring on the system in real time by examining the V$PX_SESSION view, which shows which parallel slave processes are currently executing SQL. Joining V$PX_SESSION to V$SESSION and V$SQL enables us to identify the sessions and SQLs currently employing parallel processing to see the desired and actual DOP:

```
SQL> WITH px_session AS (SELECT qcsid, qcserial#, MAX (degree) degree,
                                MAX (req_degree) req_degree,
                                COUNT ( * ) no_of_processes
                           FROM v$px_session p
                       GROUP BY   qcsid, qcserial#)
     SELECT   s.sid, s.username, degree, req_degree, no_of_processes,
              sql_text
       FROM      v$session s  JOIN px_session p
                ON (s.sid = p.qcsid AND s.serial# = p.qcserial#)
              JOIN v$sql sql
                ON (sql.sql_id = s.sql_id
                    AND sql.child_number = s.sql_child_number)
     /
```

```
       SID USERNAME     DEGREE REQ_DEGREE NO_OF_PROCESSES
---------- -------- ---------- ---------- ---------------
SQL_TEXT
-----------------------------------------------------------
       144 OPSG             18         18              36
select   /*+ parallel(sa,18) */ prod_id,sum(quantity_sold)
   , sum(amount_sold)  from sales_archive sa group by prod
_id  order by 3 desc
```

V$SYSSTAT contains some statistics relating to parallel query downgrades that can help us determine how often parallel queries are being downgraded from the requested DOP:

```
SQL> SELECT  name,value, round(value*100/sum(value) over(),2) pct
  2    FROM  v$sysstat
  3   WHERE  name LIKE 'Parallel operations%downgraded%';

NAME                                          VALUE        PCT
------------------------------------------ ---------- ----------
Parallel operations not downgraded                109      93.97
Parallel operations downgraded to serial            0          0
Parallel operations downgraded 75 to 99 pct         0          0
Parallel operations downgraded 50 to 75 pct         3       2.59
Parallel operations downgraded 25 to 50 pct         2       1.72
Parallel operations downgraded 1 to 25 pct          2       1.72
```

OPTIMIZING PARALLEL PERFORMANCE

Now that we have a solid grounding in the theory of parallel execution, and understand how to influence and measure parallel execution, we are in a good position to formulate some guidelines for optimizing parallel execution. Here are the guidelines for getting the most out of parallel execution:

- ❑ Start with a SQL that is optimized for *serial* execution.
- ❑ Ensure that the SQL is a suitable SQL for *parallel* execution.
- ❑ Ensure that the database server host is suitably configured for parallel execution.
- ❑ Make sure that all parts of the execution plan are parallelized.
- ❑ Ensure that the requested DOP is realistic.
- ❑ Monitor the actual versus requested DOP.
- ❑ Check for skew in data and skew in workload between processes.

Let's now look at each of these guidelines in detail.

START WITH A SQL THAT IS OPTIMIZED FOR SERIAL EXECUTION

An optimal parallel plan might be different from an optimized serial plan. For instance, parallel processing usually starts with a table or index scan, whereas the optimal serial plan might be based on an index lookup. However, you should ensure that your query is optimized for serial execution before parallelizing for these reasons:

- ❑ The structures and methods of serial tuning—indexing, statistics collections, and such—are often essential for good parallel tuning as well.
- ❑ If the resources required for parallel execution are not available, your query might be serialized (depending on the settings of PARALLEL_DEGREE_POLICY and PARALLEL_MIN_PERCENT). In that case, you want to ensure than your parallel query's serial plan is as good as possible.
- ❑ A SQL that is poorly tuned might become an even worse SQL—at least in terms of its impact on other users—when it is permitted to consume more of the database server's CPU and IO resources.

When optimizing a SQL statement for *parallel* execution, start by optimizing the SQL for *serial* execution.

ENSURE THAT THE SQL IS A SUITABLE SQL FOR PARALLEL EXECUTION

Not every SQL can benefit from parallel execution. Here are a few examples of SQLs that probably should *not* be parallelized:

- ❑ SQL statements that have a short execution time when executed serially.
- ❑ SQL statements likely to be run at high rates of concurrency in multiple sessions.
- ❑ SQL statements based on index lookups. Nonpartitioned index lookups or range scans cannot be parallelized. Index full scans *can* be parallelized, however. Index lookups on partitioned indexes can also be parallelized.

Make sure that the SQL to be parallelized is suitable for parallel execution; OLTP type queries are generally not suitable for parallelization.

ENSURE THAT THE SYSTEM IS SUITABLY CONFIGURED FOR PARALLEL EXECUTION

Not all SQLs are suitable for parallel execution, and not all database server hosts are suitable either. In today's world, most physical server hosts will meet the minimum requirements: multiple CPUs and data striped across multiple physical drives. However, some virtual machine hosts might fail to meet those minimum requirements and desktop machines, which typically have only a single disk device, are usually not optimized for parallel execution.

Don't try to use parallel execution on systems that do not meet the minimum requirements (multiple CPUs and data striped across multiple drives).

MAKE SURE THAT ALL PARTS OF THE EXECUTION PLAN ARE PARALLELIZED

In a complex parallel SQL statement, it's important to ensure that all significant steps in the query execution are implemented in parallel. If one of the steps in a complex query is performed in serial, the other parallel steps might have to wait for the serial step to complete, and the advantages of parallelism will be lost. The OTHER_TAG column of the PLAN_TABLE indicates such a step with the PARALLEL_FROM_SERIAL tag and DBMS_XPLAN record S->P in the IN-OUT column.

For instance, in the following example the CUSTOMERS table is parallelized, but the SALES table is not. The join and GROUP BY of the two tables includes many parallelized operations, but the full table scan of SALES is not parallelized and the tell-tale S->P tag shows that SALES rows are fed in serial into subsequent parallel operations:

```
SQL> ALTER TABLE customers PARALLEL(DEGREE 4);

SQL> ALTER TABLE sales NOPARALLEL ;

SQL> EXPLAIN PLAN FOR
  2   SELECT /*+ ordered use_hash(c) */
  3          cust_last_name, SUM (amount_sold)
  4     FROM sales s JOIN customers c
  5          USING (cust_id)
  6    GROUP BY cust_last_name;

SQL> SELECT * FROM table (DBMS_XPLAN.display
(NULL, NULL, 'BASIC +PARALLEL'));


----------------------------------------------------------------------
|Id| Operation                         |Name     |  TQ  |IN-OUT|PQ Distrib|
----------------------------------------------------------------------
| 0| SELECT STATEMENT                  |         |      |      |          |
| 1|  PX COORDINATOR                   |         |      |      |          |
| 2|   PX SEND QC (RANDOM)          |:TQ10002 | Q1,02| P->S |QC (RAND) |
| 3|    HASH GROUP BY               |         | Q1,02| PCWP |          |
| 4|     PX RECEIVE                 |         | Q1,02| PCWP |          |
| 5|      PX SEND HASH              |:TQ10001 | Q1,01| P->P |HASH      |
| 6|       HASH GROUP BY            |         | Q1,01| PCWP |          |
| 7|        HASH JOIN               |         | Q1,01| PCWP |          |
| 8|         BUFFER SORT            |         | Q1,01| PCWC |          |
| 9|          PX RECEIVE            |         | Q1,01| PCWP |          |
|10|           PX SEND BROADCAST    |:TQ10000 |      | S->P |BROADCAST |
|11|            VIEW                |VW_GBC_5 |      |      |          |
|12|             HASH GROUP BY      |         |      |      |          |
|13|              TABLE ACCESS FULL| SALES    |      |      |          |
|14|         PX BLOCK ITERATOR      |         | Q1,01| PCWC |          |
|15|          TABLE ACCESS FULL     |CUSTOMERS| Q1,01| PCWP |          |
----------------------------------------------------------------------
```

A partially parallelized execution plan, such as the preceding one, can deliver the worst of both worlds: Elapsed time is not improved because the serial operation forms a bottleneck on overall execution. Nevertheless, the SQL ties up parallel server processes and might impact the performance of other concurrently executing SQL.

If we set a default degree of parallelism for the SALES table, the serial bottleneck disappears. The full scan of SALES is now performed in parallel, and the S->P bottleneck is replaced by the fully parallelized P->P operation:

```
----------------------------------------------------------------------
| Id  | Operation                      | Name      |    TQ  |IN-OUT|
----------------------------------------------------------------------
|   0 | SELECT STATEMENT               |           |        |      |
|   1 |  PX COORDINATOR                |           |        |      |
|   2 |   PX SEND QC (RANDOM)          | :TQ10003  |  Q1,03 | P->S |
|   3 |    HASH GROUP BY               |           |  Q1,03 | PCWP |
|   4 |     PX RECEIVE                 |           |  Q1,03 | PCWP |
|   5 |      PX SEND HASH              | :TQ10002  |  Q1,02 | P->P |
|   6 |       HASH GROUP BY            |           |  Q1,02 | PCWP |
|   7 |        HASH JOIN               |           |  Q1,02 | PCWP |
|   8 |         PX RECEIVE             |           |  Q1,02 | PCWP |
|   9 |          PX SEND BROADCAST     | :TQ10001  |  Q1,01 | P->P |
|  10 |           VIEW                 | VW_GBC_5  |  Q1,01 | PCWP |
|  11 |            HASH GROUP BY       |           |  Q1,01 | PCWP |
|  12 |             PX RECEIVE         |           |  Q1,01 | PCWP |
|  13 |              PX SEND HASH      | :TQ10000  |  Q1,00 | P->P |
|  14 |               HASH GROUP BY    |           |  Q1,00 | PCWP |
|  15 |                PX BLOCK ITERATOR |         |  Q1,00 | PCWC |
|  16 |                  TABLE ACCESS FULL| SALES      |  Q1,00 | PCWP |
|  17 |           PX BLOCK ITERATOR       |            |  Q1,02 | PCWC |
|  18 |            TABLE ACCESS FULL      | CUSTOMERS |  Q1,02 | PCWP |
-----------------------------------------------------------------------
```

When optimizing a parallelized execution plan, ensure that all relevant steps are executed in parallel: The S->P tag in DBMS_XPLAN or PARALLEL_FROM_SERIAL in the PLAN_TABLE often indicates a serial bottleneck in an otherwise parallel plan.

ENSURE THAT THE REQUESTED DOP IS REALISTIC

We saw previously (in Figure 13-5, for instance), how increasing the DOP beyond the optimal level can place excessive load on the system without improving performance. In worst case scenarios, increasing the DOP beyond optimal can result in a reduction in query elapsed time as well. Therefore, setting an appropriate DOP is important both for the health of the database as a whole, and for the optimal performance of the query being parallelized.

Ensure that your requested or expected DOP is realistic; an overly-high DOP can result in excessive load on the database server without improving the SQL's performance.

MONITOR THE ACTUAL DOP

Your requested DOP might be optimal but not always achievable. When multiple parallelized queries contend for finite parallel execution resources, the DOP might be reduced, or the SQL statement might be run in serial mode.

We previously discussed how Oracle decides on the actual DOP; most importantly the parameters PARALLEL_MIN_PERCENT, PARALLEL_DEGREE_POLICY, and PARALLEL_ADAPTIVE_MULTI_USER control how Oracle changes the DOP and whether a statement runs at reduced parallelism, terminates with error, or is deferred for later processing when insufficient resources exist to run the statement at the requested DOP.

Reductions in the DOP can result in disappointing performance for your parallel SQL. You should monitor query execution to see if such reductions in the DOP are actually occurring. We previously saw how we can use V$PQ_TQSTAT to measure the actual DOP and how we can use statistics in V$SYSTAT to measure parallel downgrades overall.

If you determine that downgraded parallelism is leading to disappointing performance, you might want to revisit your system resources (memory, IO bandwidth), scheduling of parallel SQLs, or revisit your server configuration. Possible options include

- ❑ Rescheduling parallel SQLs so that they do not attempt to run concurrently. Oracle 11g Release 2 can automatically reschedule SQLs if the PARALLEL_DEGREE_POLICY is set to AUTO.
- ❑ Adjusting parallel configuration parameters to allow greater concurrent parallelism. You can do this by increasing PARALLEL_THREADS_PER_CPU or PARALLEL_MAX_SERVERS. The risk here is that the amount of parallel execution will be greater than your system can support, leading to degraded SQL performance.
- ❑ Increasing the power of your database server. You can increase the number of CPUs, the number of instances in a RAC cluster, and the number of disks in your disk array.
- ❑ Adjust PARALLEL_MIN_PERCENT to enable SQLs to run at reduced parallelism rather than signalling an error.

Disappointing parallel performance might be the result of Oracle downgrading the requested DOP due to concurrent load or limits on parallel execution resources.

CHECK FOR SKEW IN DATA AND SKEW IN WORKLOAD BETWEEN PROCESSES

Parallel processing works best when every parallel process in a step has the same amount of work to do. If one slave process has more work than a peer process, the "lazy" slave will wait for the "busy" slave, and we won't get performance improvements in line with the number of processes working on the SQL.

Most of the algorithms that Oracle employs are designed to achieve an even distribution of data; these algorithms include the HASH, ROUND ROBIN, and RANDOM distribution mechanisms. However, when a sort operation is performed, Oracle cannot use these random or pseudo-random mechanisms. Instead, Oracle must distribute data to the slaves based on the sort key columns. We saw an example of this in Figure 13-2 where a parallel process fed rows from A–K to one slave for sorting and rows from L–Z to the other.

If the distribution of data in the sort column is very skewed, this allocation might be uneven. For instance, consider this simple query:

```
SQL> EXPLAIN PLAN
  2     FOR
  3          SELECT /*+ parallel */
  4               cust_last_name, cust_first_name, cust_year_of_birth
  5            FROM   customers
  6        ORDER BY   CUST_LAST_NAME;
```

```
---------------------------------------------------------------------
|Id | Operation                | Name       |   TQ  |IN-OUT| PQ Distrib |
---------------------------------------------------------------------
| 0 | SELECT STATEMENT         |            |       |      |            |
| 1 |  PX COORDINATOR          |            |       |      |            |
| 2 |   PX SEND QC (ORDER)     | :TQ10001   | Q1,01 | P->S | QC (ORDER) |
| 3 |    SORT ORDER BY         |            | Q1,01 | PCWP |            |
| 4 |     PX RECEIVE           |            | Q1,01 | PCWP |            |
| 5 |      PX SEND RANGE       | :TQ10000   | Q1,00 | P->P | RANGE      |
| 6 |       PX BLOCK ITERATOR  |            | Q1,00 | PCWC |            |
| 7 |        TABLE ACCESS FULL| CUSTOMERS  | Q1,00 | PCWP |            |
---------------------------------------------------------------------
```

In the preceding step 5, Oracle distributes data from one set of slaves to another based on the range of values contained in the sort column. If the data is well distributed, all should be well. However, should the data be heavily skewed (perhaps we have an extra large number of Smiths and Zhangs), the distribution of data to slaves might become uneven. For example, the following V$PQ_TQSTAT output shows such an uneven distribution with twice as many rows directed to one slave than the other (I deliberately skewed customer surnames to achieve this):

```
SQL>   SELECT   dfo_number, tq_id, server_Type, MIN (num_rows),
  2             MAX (num_rows), COUNT ( * ) dop
  3      FROM   v$pq_tqstat
  4  GROUP BY   dfo_number, tq_id, server_Type
  5  ORDER BY   dfo_number, tq_id, server_type DESC;

DFO_NUM      TQ_ID SERVER_TYP MIN(NUM_ROWS) MAX(NUM_ROWS)        DOP
------- ---------- ---------- ------------- ------------- ----------
      1          0 Ranger               182           182          1
      1          0 Producer          158968        174512          2
      1          0 Consumer          103262        230218          2
      1          1 Producer          103262        230218          2
      1          1 Consumer          333480        333480          1
```

Unfortunately, there might be little that can be done about such a data skew. Oracle does not appear to take histogram data into account when distributing rows between parallel slaves. If the distribution of rows seems particularly uneven, you can consider changing the DOP or reviewing whether the SQL is truly suitable for parallelizing.

Effective parallelism depends on the even distribution of processing across the parallel slave processes. V$PQ_TQSTAT enables you to evaluate the efficiency of the load balancing across the parallel slaves.

OTHER PARALLEL TOPICS

Most of what we covered so far applies to all parallel execution but focused mainly on single-instance parallel queries involving table scans. Now let's turn our attention to other parallel scenarios.

PARALLEL EXECUTION IN RAC

In a Real Application Clusters (RAC) database, SQL can be parallelized across the multiple instances that make up the cluster. Indeed, Oracle transparently parallelizes across the entire cluster unless you take specific steps to prevent it.

Using all the instances in the cluster enables Oracle to take advantage of all the CPUs of the host computers that support the cluster database and, therefore, will usually lead to better performance than could be achieved by running the SQL on a single instance. Oracle multiples the default DOP by the number of instances in the cluster to take full advantage of the processing power of the cluster.

To see exactly how the query distributes across the instances within the cluster, we can observe the INSTANCE column in V$PQ_TQSTAT. The following gives a good summary of overall parallelism:

```
SQL> SELECT   dfo_number, tq_id, server_Type, MIN (num_rows) min_rows,
  2           MAX (num_rows) max_rows, COUNT ( * ) dop,
  3           COUNT (DISTINCT instance) no_of_instances
  4    FROM   v$pq_tqstat
  5  GROUP BY  dfo_number, tq_id, server_Type
  6  ORDER BY  dfo_number, tq_id, server_type DESC;

DFO_NUMBER      TQ_ID SERVER_TYP   MIN_ROWS   MAX_ROWS   DOP INSTANCES
---------- ---------- ---------- ---------- ---------- ----- ---------
         1          0 Producer          842       1617    48         3
         1          0 Consumer         1056       1239    48         3
         1          1 Producer         8779      38187    48         3
         1          1 Consumer        15331      24572    48         3
         1          2 Producer          107        159    48         3
         1          2 Consumer           64        244    48         3
         1          3 Ranger            479        479     1         1
         1          3 Producer            9         10    48         3
         1          3 Consumer            9         55    48         3
         1          4 Producer            9         10    48         3
         1          4 Consumer            9          9     1         1
```

The above output was generated on a 3 instance RAC cluster in which each instance had 8 CPUs available. Oracle applied the default formula of 2 threads per CPU to achieve a DOP of 48 for the 24 CPUs available across the 3 hosts.

Although parallelism in RAC scales well with additional instances, there is an additional overhead in communication when the parallel slave processes reside on different hosts. The RAC cluster's high-speed interconnect might become taxed if the amount of data transferred is high, and the performance of a RAC-parallelized query might not be quite as good as for a locally parallelized query with an equivalent DOP.

From Oracle 11g Release 2 forward, the parameter PARALLEL_FORCE_LOCAL can be set to restrict parallel processing to the current instance only.

We discuss some further aspects of RAC optimization in Chapter 23, "Optimizing RAC."

PARALLEL INDEX LOOKUPS

Index-based queries are not usually parallelizable; however, if the index involved is a locally partitioned index on a partitioned table, a lookup using that index can be parallelized. Each partition lookup can be performed by a separate process, and a DOP as high as the number of partitions can be achieved.

For example, if the SALES table had a local partitioned index on the CUST_ID column like this:

```
CREATE INDEX sales_i1 ON sales(cust_id) LOCAL;
```

We could use the PARALLEL_INDEX hint to parallelize lookups on specific CUST_ID values:

```
SELECT /*+ parallel_index(s) */ *
FROM sales s
WHERE cust_id = 247;
------------------------------------------------------------------------
| Id  | Operation                                 | Name     |    TQ  |IN-OUT|
------------------------------------------------------------------------
|   0 | SELECT STATEMENT                          |          |        |      |
|   1 |  PX COORDINATOR                           |          |        |      |
|   2 |   PX SEND QC (RANDOM)                     | :TQ10000 |  Q1,00 | P->S |
|   3 |    PX PARTITION HASH ALL                  |          |  Q1,00 | PCWC |
|   4 |     TABLE ACCESS BY LOCAL INDEX ROWID| SALES    |  Q1,00 | PCWP |
|   5 |      INDEX RANGE SCAN                     | SALES_I1 |  Q1,00 | PCWP |
------------------------------------------------------------------------
```

PARALLEL DML

Any DML statement that performs a scan operation can be parallelized, at least for that part of the statement that performs the table reads.

For instance, parts of the following UPDATE statement executes in parallel:

```
SQL> EXPLAIN PLAN
  2     FOR
  3        UPDATE /*+ parallel(s)  */
  4              sales s
  5           SET   unit_price = amount_sold / quantity_sold;


--------------------------------------------------------------------
|Id  | Operation              | Name      |    TQ  |IN-OUT| PQ Distrib |
--------------------------------------------------------------------
|  0 | UPDATE STATEMENT       |           |        |      |            |
|  1 |  UPDATE                | SALES     |        |      |            |
|  2 |   PX COORDINATOR       |           |        |      |            |
|  3 |    PX SEND QC (RANDOM)| :TQ10000  |  Q1,00 | P->S | QC (RAND)  |
|  4 |     PX BLOCK ITERATOR |           |  Q1,00 | PCWC |            |
|  5 |      TABLE ACCESS FULL| SALES     |  Q1,00 | PCWP |            |
--------------------------------------------------------------------
```

The full scan of SALES is parallelized, but note that the UPDATE statement (step 1) is executed outside the parallel processing stream; although the rows to be updated are identified by the parallel processes, the actual updates are performed in serial by the query coordinator.

To perform true parallel DML, you should first enable parallel DML with the following statement:

```
ALTER SESSION ENABLE PARALLEL DML.
```

After we do this, we get a fully parallelized execution plan:

```
SQL> EXPLAIN PLAN
  2     FOR
  3        UPDATE /*+ parallel(s)  */
  4              sales_p s
  5           SET   unit_price = amount_sold / quantity_sold;


--------------------------------------------------------------------
|Id  | Operation              | Name      |    TQ  |IN-OUT| PQ Distrib |
---------------------------------------------------------------------
|  0 | UPDATE STATEMENT       |           |        |      |            |
|  1 |  PX COORDINATOR        |           |        |      |            |
|  2 |   PX SEND QC (RANDOM)  | :TQ10000  |  Q1,00 | P->S | QC (RAND)  |
|  3 |    UPDATE              | SALES_P   |  Q1,00 | PCWP |            |
|  4 |     PX BLOCK ITERATOR |           |  Q1,00 | PCWC |            |
|  5 |      TABLE ACCESS FULL| SALES_P   |  Q1,00 | PCWP |            |
--------------------------------------------------------------------
```

The UPDATE step is now executed by the same parallel server processes that perform the scan of the SALES table. The UPDATE is now fully parallelized.

To fully parallelize a DML statement, issue an ALTER SESSION ENABLE PARALLEL DML statement; otherwise the statement will be only partially parallelized (at best).

Parallel INSERT Inserting rows in parallel is a particularly good use of parallel DML, especially for bulk operations in which the input data is in another table (such as a staging or transaction table). In this scenario, it's important to parallelize both the SELECT and INSERT operations. For instance, here we parallelize the INSERT but not the SELECT that performs the table scan on the SALES_UPDATE table:

```
SQL> EXPLAIN PLAN FOR
  2   INSERT /*+ parallel(s) */
  3     INTO sales s
  4   SELECT * FROM sales_updates;

--------------------------------------------------------------------
|Id| Operation                    | Name         |  TQ  |IN-OUT|PQ Distrib|
-------------------------------------------------------------------
| 0| INSERT STATEMENT             |              |      |      |          |
| 1|  PX COORDINATOR              |              |      |      |          |
| 2|   PX SEND QC (RANDOM)        |:TQ10001      |Q1,01 | P->S |QC (RAND) |
| 3|    LOAD AS SELECT            |SALES         |Q1,01 | PCWP |          |
| 4|     BUFFER SORT              |              |Q1,01 | PCWC |          |
| 5|      PX RECEIVE              |              |Q1,01 | PCWP |          |
| 6|       PX SEND ROUND-ROBIN|:TQ10000      |      | S->P |RND-ROBIN |
| 7|        TABLE ACCESS FULL |SALES_UPDATES|      |      |          |
---------------------------------------------------------------------
```

The full table scan of SALES_UPDATE is processed serially, and the S->P tag should raise a red flag, indicating parallel processing waiting on serial processing.

This plan is more perfectly parallelized:

```
SQL> EXPLAIN PLAN FOR
  2   INSERT /*+ parallel(s) */
  3     INTO    sales s
  4   SELECT /*+ parallel(u) */ *
  5      FROM   sales_updates u;
```

```
---------------------------------------------------------------------
|Id| Operation                | Name          |  TQ   |IN-OUT| PQ Distrib |
---------------------------------------------------------------------
| 0| INSERT STATEMENT         |               |       |      |            |
| 1|  PX COORDINATOR          |               |       |      |            |
| 2|   PX SEND QC (RANDOM)    |:TQ10000       |Q1,00  | P->S | QC (RAND)  |
| 3|    LOAD AS SELECT        |SALES          |Q1,00  | PCWP |            |
| 4|     PX BLOCK ITERATOR    |               |Q1,00  | PCWC |            |
| 5|      TABLE ACCESS FULL|SALES_UPDATES|Q1,00  | PCWP |            |
---------------------------------------------------------------------
```

> When parallelizing an INSERT from a SELECT, remember to parallelize both the INSERT and SELECT steps, using two hints if necessary.

By default, parallel insert uses the direct load APPEND method, creating new data blocks and appending them directly to the segment, bypassing the buffer cache. We talk about the pros and cons of direct load inserts in Chapter 14, “DML Tuning.” However, for now it’s enough to note that direct path insert is usually the best choice for parallel insert because otherwise the parallel slaves might contend for latches, free lists, and data buffers. However, if you want to use the conventional insert method—inserting rows into existing data blocks where appropriate and utilizing the buffer cache—you can use the NOAPPEND hint:

```
SQL> EXPLAIN PLAN FOR
  2    INSERT /*+ parallel(s) noappend */
  3      INTO    sales s
  4    SELECT /*+ parallel(u) */ *
  5      FROM   sales_updates u;
```

```
------------------------------------------------------------------
|Id| Operation                  | Name       |  TQ   |IN-OUT|PQ Distrib|
------------------------------------------------------------------
| 0| INSERT STATEMENT           |            |       |      |          |
| 1|  PX COORDINATOR            |            |       |      |          |
| 2|   PX SEND QC (RANDOM)      |:TQ10000    |Q1,00  | P->S |QC (RAND) |
| 3|    LOAD TABLE CONVENTIONAL |SALES       |Q1,00  | PCWP |          |
| 4|     PX BLOCK ITERATOR      |            |Q1,00  | PCWC |          |
| 5|      TABLE ACCESS FULL     |SALES_UPDAT|Q1,00  | PCWP |          |
------------------------------------------------------------------
```

Parallel MERGE The MERGE statement combines the functionality of INSERT and UPDATE into a single statement. A MERGE execution plan usually involves an outer join between the target table and the source tables. To optimize the merge, we most truly need to optimize that outer join.

We look more at MERGE optimization in Chapter 14.

Merge statements can be fully parallelized, although you normally want to ensure that both target and source tables are parallelized. For instance, in the following example we supply PARALLEL hints for both the source and target tables:

```
SQL> EXPLAIN PLAN FOR
     MERGE /*+ parallel(s) parallel(u) */ INTO sales s
        USING sales_updates u
        ON (s.prod_id=u.prod_id AND s.cust_id=u.cust_id
            AND s.time_id=u.time_id
            AND s.channel_id=u.channel_id
            AND s.promo_id = u.promo_id)
       WHEN MATCHED THEN
     UPDATE SET  s.amount_sold  =u.amount_sold,
                 s.quantity_sold=u.quantity_sold
     WHEN NOT MATCHED THEN
     INSERT VALUES ( u.prod_id, u.cust_id, u.time_id  ,
                     u.channel_id, u.promo_id,
                     u.quantity_sold, u.amount_sold);
```

```
--------------------------------------------------------------------
|Id| Operation                        | Name           |    TQ  |IN-OUT|
--------------------------------------------------------------------
| 0| MERGE STATEMENT                  |                |        |      |
| 1|  PX COORDINATOR                  |                |        |      |
| 2|   PX SEND QC (RANDOM)            | :TQ10003       |  Q1,03 | P->S |
| 3|    MERGE                         | SALES          |  Q1,03 | PCWP |
| 4|     PX RECEIVE                   |                |  Q1,03 | PCWP |
| 5|      PX SEND HYBRID (ROWID PKEY)| :TQ10002       |  Q1,02 | P->P |
| 6|       VIEW                       |                |  Q1,02 | PCWP |
| 7|        HASH JOIN OUTER BUFFERED |                |  Q1,02 | PCWP |
| 8|         PX RECEIVE               |                |  Q1,02 | PCWP |
| 9|          PX SEND HASH            | :TQ10000       |  Q1,00 | P->P |
|10|           PX BLOCK ITERATOR      |                |  Q1,00 | PCWC |
|11|            TABLE ACCESS FULL     | SALES_UPDATES |  Q1,00 | PCWP |
|12|         PX RECEIVE               |                |  Q1,02 | PCWP |
|13|          PX SEND HASH            | :TQ10001       |  Q1,01 | P->P |
|14|           PX BLOCK ITERATOR      |                |  Q1,01 | PCWC |
|15|            TABLE ACCESS FULL     | SALES          |  Q1,01 | PCWP |
--------------------------------------------------------------------
```

DBMS_PARALLEL_EXECUTE Parallel DML is an incredibly powerful way to speed up bulk DML. However, it has the disadvantage of applying all changes in a single transaction. This results in the generation of long-standing locks, requires large undo segments, and runs the risk of expensive rollback operations should the operation fail.

The DBMS_PARALLEL_EXECUTE package, introduced in Oracle 11g Release 2, helps to resolve this dilemma by enabling you to execute parallel DML in smaller "chunks," each of which is committed individually. The package enables you to restart the job should any of the individual chunked operations fail.

The following code shows an example of DBMS_PARALLEL_EXECUTE in action:

```
DECLARE
   v_dml_sql     VARCHAR2(1000);
   v_task_name   VARCHAR2(1000)
                 := 'dbms_parallel_execute demo';
   v_status      NUMBER;
BEGIN
   DBMS_PARALLEL_EXECUTE.CREATE_TASK(
        task_name => v_task_name);

   DBMS_PARALLEL_EXECUTE.CREATE_CHUNKS_BY_ROWID(
        TASK_NAME => v_task_name,
        TABLE_OWNER => USER, TABLE_NAME => 'SALES',
        BY_ROW => TRUE, CHUNK_SIZE => 1000);

   v_dml_sql :=
         'UPDATE sales SET   unit_price = '
      || '        amount_sold / quantity_sold '
      || ' WHERE rowid BETWEEN :start_id AND :end_id ';

   DBMS_PARALLEL_EXECUTE.RUN_TASK(TASK_NAME => v_task_name,
        SQL_STMT => v_dml_sql, LANGUAGE_FLAG => DBMS_SQL.NATIVE,
        PARALLEL_LEVEL => 2);

   v_status := DBMS_PARALLEL_EXECUTE.TASK_STATUS(
              task_name  => v_task_name);

   IF v_status = DBMS_PARALLEL_EXECUTE.FINISHED THEN
      DBMS_PARALLEL_EXECUTE.DROP_TASK(task_name => v_task_name);
   ELSE
   -- could use dbms_parallel_execute.resume_task here to retry
   -- if required
      raise_application_error(-2001,
33            'Task ' || v_task_name || ' abnormal termination: status='
34            || v_status);
35         END IF;
36      END;
```

DBMS_PARALLEL_EXECUTE works in two phases. The first phase, shown on line 10 above, uses one of the CREATE_CHUNK procedures to define the table chunks that are to be processed. There are a number of ways of chunking, including defining chunks using custom SQL. In this example, we use the CREATE_CHUNKS_BY_ROWID procedure that simply creates chunks that have a sequential set of ROWIDS. These rows tend to be in contiguous blocks, and this method will result in a fairly even distribution of rows. The approximate size of each chunk is defined by the CHUNK_SIZE argument (line 13).

The second phase executes a SQL statement to work on the chunks. The SQL statement, shown on lines 15-18 in our example, must define bind variables :START_ID and :END_ID that are used to feed in the ROWID ranges or—if you use a different chunking strategy—column values that define the chunks. The SQL statement is fed into the RUN_TASK procedure that also specifies the DOP to be used in the operation (line 22).

DBMS_PARALLEL_EXECUTE runs the SQL statement against each chunk using the DOP specified. A COMMIT will be executed after each chunk has been processed. This means that if there is an error, only some of the rows in the table will have been processed. If this occurs, you can use the RESUME_TASK procedure to restart the operation on the chunks that have not been processed.

Consider the 11g Release 2 DBMS_PARALLEL_EXECUTE package when you want to issue parallel DML without the drawbacks of performing all the DML in a single transaction.

PARALLEL DDL

The DDL statements CREATE INDEX and CREATE TABLE AS SELECT statements can both be parallelized. The CREATE TABLE AS SELECT statement parallelizes in much the same way as a parallel INSERT. Parallel CREATE INDEX parallelizes the table or index scan necessary to create the index blocks, sorts the rows in parallel, and builds the index leaf and branch blocks in parallel.

In both cases, the DOP is controlled by the PARALLEL clause of the CREATE INDEX or CREATE TABLE statement. That DOP is then set for subsequent query operations that use the index or table.

Here is an example of CREATE INDEX:

```
SQL> EXPLAIN PLAN FOR
  2  CREATE INDEX sales_i ON sales(prod_id,time_id)
           PARALLEL(DEGREE DEFAULT);

-------------------------------------------------------------------
|Id| Operation                   | Name      |    TQ   |IN-OUT|PQ Distrib |
-------------------------------------------------------------------
| 0| CREATE INDEX STATEMENT      |           |         |      |           |
| 1|  PX COORDINATOR             |           |         |      |           |
| 2|   PX SEND QC (ORDER)        | :TQ10001  |  Q1,01  | P->S |QC (ORDER) |
| 3|    INDEX BUILD NON UNIQUE| SALES_I   |  Q1,01  | PCWP |           |
| 4|     SORT CREATE INDEX       |           |  Q1,01  | PCWP |           |
| 5|      PX RECEIVE             |           |  Q1,01  | PCWP |           |
| 6|       PX SEND RANGE         | :TQ10000  |  Q1,00  | P->P |RANGE      |
| 7|        PX BLOCK ITERATOR |           |  Q1,00  | PCWC |           |
| 8|         TABLE ACCESS FULL| SALES     |  Q1,00  | PCWP |           |
-------------------------------------------------------------------
```

Here is a parallel CREATE TABLE AS SELECT:

```
SQL> EXPLAIN PLAN FOR
  2  CREATE TABLE sales_copy  PARALLEL(DEGREE DEFAULT)
        AS SELECT * FROM sales;
------------------------------------------------------------------
|Id| Operation               | Name        |    TQ   |IN-OUT|PQ Distrib|
------------------------------------------------------------------
| 0| CREATE TABLE STATEMENT |             |         |      |          |
| 1|  PX COORDINATOR        |             |         |      |          |
| 2|   PX SEND QC (RANDOM)  | :TQ10000    |  Q1,00  | P->S |QC (RAND) |
| 3|    LOAD AS SELECT      | SALES_COPY |  Q1,00  | PCWP |          |
| 4|     PX BLOCK ITERATOR  |             |  Q1,00  | PCWC |          |
| 5|      TABLE ACCESS FULL | SALES       |  Q1,00  | PCWP |          |
------------------------------------------------------------------
```

SUMMARY

In this chapter we looked at the parallel execution facilities provided by the Oracle RDBMS, how to use these to improve SQL performance, and how to optimize the performance of SQL running in parallel.

Parallel processing uses multiple processes or threads to execute a single SQL statement. Providing that the system is suitably configured, parallel processing can result in big improvements in SQL throughput, though at the cost of an increased load on the system.

The Degree of Parallelism (DOP) defines the amount of parallelism that is applied to your SQLs. For simple SQLs the DOP equates to the number of parallel processes, but in most nontrivial statements, twice as many processes will be required to achieve a pipeline of parallel processing.

Parallel processing might be indicated if

- ❑ The database server has multiple CPUs.
- ❑ The data is distributed across multiple disk devices.
- ❑ The SQL is long running or resource-intensive.
- ❑ Free resources are available on the system to support the additional overhead associated with parallel processing.
- ❑ The SQL involves a full table or index scan, or locally partitioned index lookups.

You can use EXPLAIN PLAN and DBMS_XPLAN to determine the parallel execution plan and V$PQ_TQSTAT to determine the actual DOP achieved.

The key principles for optimizing parallel SQL are

- ❑ Starting with an SQL that is optimized for serial execution
- ❑ Ensuring that the SQL is suitable for parallel execution
- ❑ Determining that that the database server host is suitably configured for parallel execution
- ❑ Ensuring that all the steps in the SQL are parallelized
- ❑ Configuring a realistic DOP
- ❑ Monitoring the actual DOP and determining the approach when that DOP cannot be achieved (downgrade, defer, or fail)
- ❑ Checking for skew in data and in workload between processes

You can apply parallel processing across instances in a RAC cluster, making full use of all the resources of the entire cluster database. You can also apply parallel processing to DML or DDL statements.