TUNING PARALLEL EXECUTION

Introduction

The first time I heard of Parallel Query was in 1993, when my boss returned from the IOUG conference. He was an experienced database guy and I still remember his comment that 'this could be as big a step forward as B-tree indexing'.

I heard nothing about it for another 2 or 3 years, working at various sites as a contractor and the facility wasn't released until Oracle 7.16. Then I worked at a site where one of the DBAs decided we should 'give it a try' to improve the performance of our overnight batch schedule. The results were disastrous so it was switched off again because no one had the time to investigate why. Much of the time, that's the day-to-day reality of life as a working DBA. Periodically I would hear about someone trying this wonderful feature with negative results so it was never used as much as Oracle must have hoped.

As Oracle extended the server's capabilities over time to include parallel DML and the like, the name of this functionality has been changed to Parallel Execution, but you'll hear people using both names.

Parallel Execution Architecture

I think that one of the reasons why Parallel Execution is not used at many sites is that Oracle's basic Instance Architecture works so well. One of the foundations of the architecture is multiple processes running concurrently that use multi-CPU servers effectively. In a typical Production OLTP system there will be a large number of concurrent sessions. In most configurations, each will have their own dedicated server process to communicate with the instance. When all of these server connection processes are considered, the reality is likely to be that you are using your available CPU resource quite effectively, particularly when you consider the various background processes as well!

However, what about those situations when you have a resource-intensive job to run, whether it be a long running report query, large data load or perhaps an index creation? That's where parallel execution can prove extremely useful. It allows the server to take a single large task, break into separate streams of work and pass those streams to parallel execution (PX) slave processes for completion. Because the PX slaves are separate processes (or threads in a Windows environment), the operating system is able to schedule them and provide timely CPU resource in the same way that it would schedule individual user sessions. In fact, each PX slave *is* just like a normal dedicated server connection process in many ways so it's like setting four or eight normal user sessions to work on one problem. Of course, those 'users' need to behave in a co-ordinated manner (probably unlike most real users!).

As Oracle's parallel capabilities have been developed, most tasks can be executed in parallel now. According to the 9.2.0.5 documentation, Oracle supports parallel execution of the following operations: -

Access methods

> For example, table scans, index fast full scans, and partitioned index range scans.

Join methods

> For example, nested loop, sort merge, hash, and star transformation.

DDL statements

> CREATE TABLE AS SELECT, CREATE INDEX, REBUILD INDEX, REBUILD INDEX PARTITION, and MOVE SPLIT COALESCE PARTITION

DML statements

> For example, INSERT AS SELECT, updates, deletes, and MERGE operations.

Miscellaneous SQL operations

> For example, GROUP BY, NOT IN, SELECT DISTINCT, UNION, UNION ALL, CUBE, and ROLLUP, as well as aggregate and table functions.

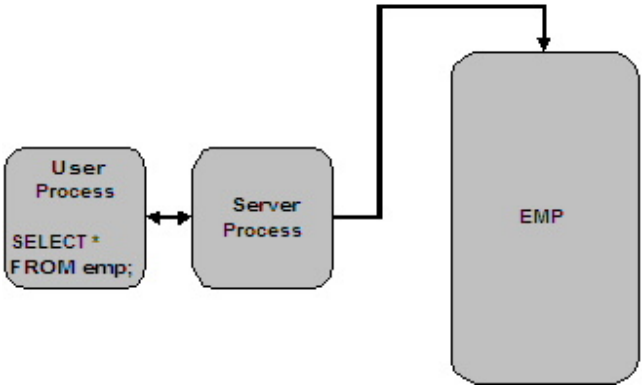
However, unless a task is likely to run for a fair amount of time (minutes or hours, rather than seconds) there's not much

point in splitting it into parallel streams. The process initialisation, synchronisation and messaging effort might take longer than the original single-threaded operation! In practice, you can simplify the possibilities for the SELECT statements that I'm going to focus on to those that include one of the following: -

- Full Table Scans
- Partition Scans (Table or Indexes)

Or, to put it another way, those operations that are likely to process significant volumes of data. There wouldn't be any significant benefits to having multiple processes retrieving a handful of rows via a selective index. First, let's look at the default Single-threaded architecture.

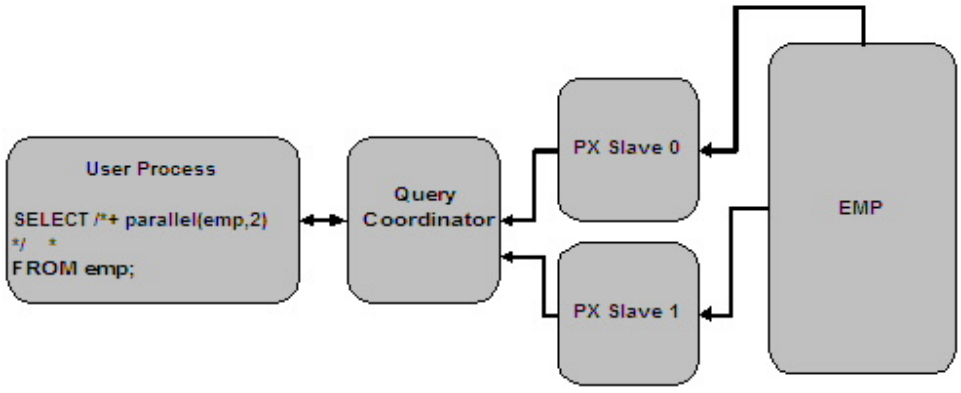
Figure 1 - Standard Single-threaded Architecture using Dedicated Server Process



This should be very familiar. The User Process (on the client or server) submits a SELECT statement that requires a full table scan of the EMP table and the Dedicated Server Process is responsible for retrieving the results and returning them to the User Process.

Let's look at how things change when we enable Parallel Execution.

Figure 2 – Parallel Full Table Scan – Degree 2



This time, the server is going to process the query in parallel as a result of the optimizer hint. When the server sees that the requested Degree of Parallelism (DOP) for the emp table is two the dedicated server process becomes the Query Coordinator. It makes a request for two PX slaves and, if it's able to acquire them, it will divide all of the blocks that it would have had to scan in the emp table into two equal ranges. Then it will send a SQL statement similar to the following

to each of the slave processes.

```
SELECT /*+ Q1000 NO_EXPAND ROWID(A1) */ A1."EMPNO",A1."ENAME",A1."JOB",
  A1."MGR",A1."HIREDATE",A1."SAL",A1."COMM",A1."DEPTNO"
FROM
 "EMP" PX_GRANULE(0, BLOCK_RANGE, DYNAMIC)  A1
```

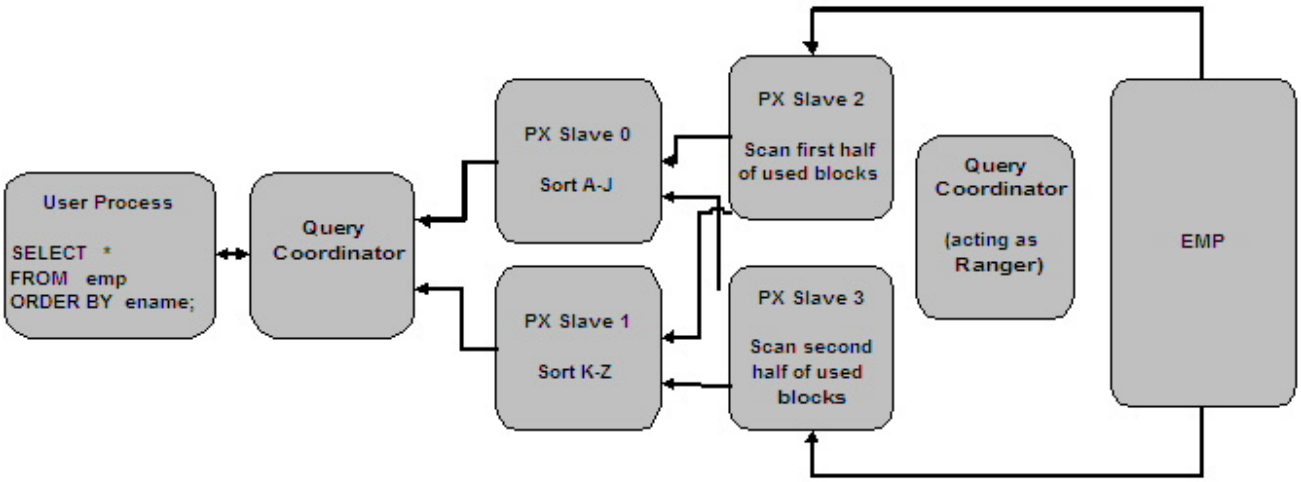
(Note that Oracle 10g changes this approach slightly, so that the SQL statement associated with the slave processes will be the same as the Query Co-ordinator, i.e.

```
SELECT /*+ parallel (emp,2) */
*
FROM EMP
```

Although this makes things a little easier to follow, it's more difficult to get at the detail of what various px slaves are doing because you can't use the SQL statement to differentiate them or find the cost for a given slave's SQL statement.)

As the data is retrieved from the emp table, it will be returned to the query co-ordinator which will, in turn, return the data to the user process. The way that all of the data is moved between the processes is using areas of memory called parallel execution message buffers or table queues. These can be stored in either the Shared Pool or Large Pool. Now let's look at another parallel query.

Figure 3 –Parallel Full Table Scan with Sort – Degree 2



The first thing to note is that there is no PARALLEL hint in the query and yet Oracle chooses to use Parallel Execution to process it. The reason is that the emp table itself also has a parallel DEGREE setting which allows us to specify whether Oracle should attempt to parallelise operations against that table and in this case, it's been set to 2[1] :-

```
ALTER TABLE emp PARALLEL (DEGREE 2);

SELECT table_name, degree
```

[1] Remember that when you use Parallel Execution, you will automatically be using the Cost-based optimizer so make sure that you have appropriate statistics on the schema objects if you are going to set their degree of parallelism!

```
FROM user_tables
WHERE table_name = 'EMP';

TABLE_NAME                     DEGREE
------------------------------ ----------
EMP                                 2
```

Hold on a minute! We requested a DOP of two and yet there are four PX slaves being used to process our request. This is because Oracle will often use two sets of PX slaves for a specific action. The first set produces rows (and are known as producers) and the second set (called consumers) consumes the rows produced by the producers. So, in this case, Oracle can see that we are going to have to perform a sort because the NAME column isn't indexed, so it requests 4 PX slaves – two sets of two. The first set are responsible for scanning the EMP table and the second set for sorting the data as it's delivered by the producers. As you can see, though, the type of operation defines the way the workload is distributed between the slaves in each set. For the full table scan, it's based on block ranges. For the sort, the QC process acts as a Ranger and divides up the sort activity by calculating the correct range of values for each slave to sort so that they'll process a reasonably equal number of rows. The sort slaves will receive statements similar to the following.

```
SELECT A1.C0 C0,A1.C1 C1,A1.C2 C2,A1.C3 C3,A1.C4 C4,A1.C5 C5,A1.C6 C6,A1.C7
  C7
FROM
 :Q1000 A1 ORDER BY A1.C0
```

However, it's actually more complicated than that. Oracle can also split an individual SQL statement into multiple actions that can be performed in parallel, known as Data Flow Operations or DFOs. For example, in this statement, there are three actions that can be parallelised – the two full table scans and the join operation.[2]

```
select c.unit_price
from costs c, products p
where c.prod_id = p.prod_id;
```

There a couple of important points to note :-

- Each PX slave in a given set must be able to communicate with *all* of the slaves in the other set so as the DOP increases, the number of connections will increase rapidly. As Jonathan Lewis points out in his article 'The Parallel Query Option in Real Life' (http://www.jlcomp.demon.co.uk/pqo_2_i.html), the number of inter-slave communication paths is DOP-squared. Even with a DOP of two, you can see this means four inter-slave connections, a DOP of four would need 16 connections and so on.

- The maximum number of processes required for a simple query is 2 x *DOP* plus the Query Co-ordinator. However, if there are multiple DFOs in the plan then additional PX slaves may be acquired[3] and slave sets may be re-used.

Instance Configuration

Switching Parallel Execution on at the Instance Level is surprisingly easy and only requires a few parameters to be set. However, you need to think carefully about the overall effects on the server first. It's worth reading this document and as many of the references listed at the end of this paper as you can before deciding on the values that you think are best for your particular application.

---

[2] This is from Metalink Note 280939 which discusses this subject in some detail, although the clarity leaves a bit to be desired.

[3] Jonathan Lewis sent me an example statement that will fire up 34 slaves on Oracle 10g and 22 slaves on 9i, so things are more complicated than they might appear!

Important – I've tried to be helpful in suggesting some initial values for these parameters, but your database and application is unique, so you should use these suggestions as starting points. There are no easy answers.

parallel_automatic_tuning

Default Value FALSE
Recommended Value TRUE

This parameter was first introduced in Oracle 8i and its very presence is instructive! I remember in version 6 being able to tune the various areas of the dictionary (or row) cache using the dc_ parameters. When 7 came along, the facility was taken away and that particular cache became self-tuning. Oracle has attempted much the same at various stages in the development history of the server, with varying degrees of success! Other examples include automatic PGA management and System Managed Undo. To me, this parameter is a sign that users have experienced difficulty in configuring PX themselves so Oracle is trying to make the job easier. In this case, I think they probably have a point. When parallel_automatic_tuning=true, it has several effects

.

- The message buffers are stored in the Large Pool rather than the Shared Pool to improve efficiency. However, you need to be aware of this and set large_pool_size accordingly. The calculation for this is in the documentation (see the list of references at the end)
- It sets various parameters to more sensible values than their defaults (e.g. parallel_execution_message_size is increased from 2Kb to 4KB)
- Importantly, it enables the parallel adaptive multi-user algorithm (see next section)

According to Oracle, this parameter is deprecated in 10G as the default values for parallel configuration parameters are optimal. However, when I've tried changing the value, apart from generating a warning, it seems to me that the behaviour is the same as previous versions! Perhaps this is one of those situations where Oracle will be making some far-reaching changes in approach in future versions and this is an early sign.

parallel_adaptive_multi_user

Default Value FALSE
Automatic Tuning Default TRUE
Recommended Value TRUE

Parallel_adapative_multi_user is one of the most significant PX configuration parameters and you need to be fully aware of its effects. Imagine a situation where perhaps we have an Oracle Discoverer report running against a Data Warehouse that takes 8 minutes to run. So we modify the DOP setting on the relevant tables to see if using PX will improve performance. We find that a DOP of 4 gives us a run-time of 90 seconds and the users are extremely happy. However, to achieve this, the report is using a total of nine server processes. We decide to stress test the change to make sure that it's going to work, don't we? Or maybe we just decide to release this to production because it's such a fantastic improvement! The only difference is going to be whether we have a disastrous test (bearable) or dozens of unhappy users (unbearable).

The problem is that we've just multiplied the user population by nine and, whilst this worked fantastically well with just one report running, it probably won't scale to large user populations unless you have some *extremely* powerful hardware. The likelihood is that it won't take long before Oracle manages to suck every last millisecond of CPU away and the effect on the overall server performance will be very noticeable!

To give you an example, one day we were testing our overnight batch run that used multiple job streams running in parallel (using the scheduling tool) each of which was parallelised using PX. In effect, we had over a hundred server processes running on a server with a handful of CPUs. It ran quickly enough for our purposes but, while it was running, it was difficult to do anything else on the server at all. It was so slow that it appeared to be dead.

Clearly, the last thing we want is for our server to grind to a halt. The users wouldn't be getting their reports back at all, never mind in 8 minutes! So Oracle introduced the Adaptive Multi-User facility to address this problem. Effectively, Oracle will judge how busy the server is when it's doling out the PX slaves and, if it decides the machine is too busy, it will give you a smaller degree of parallelism than requested. So the feature acts as a limiter[4] on PX activity to prevent the server from becoming overloaded (and everyone losing out), but allows PX to be used when the number of concurrent PX users drops and it will be more beneficial.

However, let's question and be clear on the impact of this. To help me do this, I'm going to use a quote:-

> 'This provides the best of both worlds and what users expect from a system. They know that when it is busy, it will run slower.'
> Effective Oracle by Design. Thomas Kyte

Tom Kyte is a man that I admire very much for all the work he's done for the Oracle community and for his considerable technical and communication skills. It's difficult for me to find anything he has to say about PX that I'd disagree with. In fact, I've revised this section based on a short but thought-provoking dialogue between us. However, what I'm interested here is in different opinions and perspectives (including the user perspective!) and I'm not questioning any of the technical detail of Tom's argument.

Ask yourself this question – 'Do my users expect the same report to run 4 or 8 times more slowly depending on what else is going on on the server?'. I'm not talking about 90 seconds versus 100 seconds, more like 90 seconds against 8 minutes, at unpredictable moments (from the point of view of the user). How about two users invoking the same report, one a minute or two after the other, but experiencing very different response times?

In my opinion, the statement doesn't reflect what a lot of users are like at all. The one thing they *don't* want is unpredictable performance. In fact, I worked at a site recently where the managers were very particular about the fact that they wanted a reasonable but, more important, *consistent* level of performance. Then again, you might argue that users would also be extremely unhappy if the server becomes overloaded and everyone's reports stop returning any data until someone steps in and kills some of them!

It's a complex balance that Oracle is trying to maintain so I think their solution is very sensible and I recommend it. Just remember the implications and be able to articulate them to your users

parallel_max_servers and parallel_min_servers

Default Value	Derived from the values of `CPU_COUNT, PARALLEL_AUTOMATIC_TUNING` and `PARALLEL_ADAPTIVE_MULTI_USER`
Recommended Value	Completely dependant on number of CPUs – use your initiative?

As Oracle uses PX for user requests, it needs to allocate PX slaves and it does this from a pool of slaves. These two parameters allow you to control the size of the pool and are very straightforward in use. The most difficult thing is to decide on the maximum number of slaves that you think is sensible for your server. I've seen people running dozens or hundreds of slaves on a 6 CPU server. Clearly that means that each CPU could be trying to cope with 10-20 or more processes and this probably isn't a good idea. Don't forget, though, that this is precisely what your home PC is doing all of the time! However if your disk subsystem is extremely slow, it may be that a number of slaves per CPU is beneficial because most of your processes are spending most of their time waiting on disk i/o rather than actually doing anything! However, that needs to be balanced against the extra work that the operating system is going to have to do managing the run queue.

[4] For those of you interested in audio recording, a soft-knee compressor is probably a closer relation

A sensible range of values is perhaps 2-to-4 x the number of CPUs. The most important thing is to perform some initial stress testing and to monitor CPU and disk usage and the server's run queue carefully!

parallel_threads_per_cpu

Default Value	OS Dependent, but usually 2
Automatic Tuning Default	2
Recommended Value	Increase if i/o-bound, decrease if CPU-bound

This is closely related to the previous parameter. Although it may be worth increasing this from the default of 2, you should be careful that you don't overload the CPUs as a result.

parallel_execution_message_size

Default Value	2Kb
Automatic Tuning Default	4Kb
Recommended Value	4-8Kb

This parameter controls the size of the buffers used to pass messages between the various slaves and the query coordinator. If a message is larger than this size, then it will be passed in multiple pieces, which may have a slight impact on performance. Tellingly, parallel_automatic_tuning increases the size from the default of 2Kb to 4Kb so this is probably a useful starting point, but it may be worth increasing to 8Kb or even larger. Bear in mind, though, that increasing this value will also increase the amount of memory in the Large or Shared Pool, so you should check the sizing calculations in the documentation and increase the relevant parameter appropriately

Other significant parameters

In addition to the parallel_ parameters, you should also think about the effect that all of the additional PX slaves will have on your server. For example, each is going to require a process and a session and each is going to be using a sub-task SQL statement that will need to exist in the Shared SQL area. Then we need to think about all of the additional work areas used for sorting for example. The documentation is very good in this area, though, so I'll refer you to that.

Data Dictionary Views

The easiest approach to high-level real-time performance monitoring is to use data dictionary views. There is some information in the standard views, such as V$SYSSTAT, which I'll come to later. First, though, let's take a look at the PX specific views. These begin with either V$PQ or V$PX, reflecting the change in Oracle's terminology over time. Typically, the V$PX views are the more recent and Oracle change the views that are available reasonably frequently so it's always worth using the query below to find out what views are available on the version that you're using.

```
SELECT table_name
FROM dict
WHERE table_name LIKE 'V%PQ%'
OR table_name like 'V%PX%';


TABLE_NAME
------------------------------
V$PQ_SESSTAT
V$PQ_SYSSTAT
V$PQ_SLAVE
V$PQ_TQSTAT
V$PX_BUFFER_ADVICE
```

```
V$PX_SESSION
V$PX_SESSTAT
V$PX_PROCESS
V$PX_PROCESS_SYSSTAT
```

V$PQ_SESSTAT

V$PQ_SESSTAT shows you PX statistics for your *current session*.

```
SELECT * FROM v$pq_sesstat;

STATISTIC                      LAST_QUERY SESSION_TOTAL
------------------------------ ---------- -------------
Queries Parallelized                    1             2
DML Parallelized                        0             0
DDL Parallelized                        0             0
DFO Trees                               1             2
Server Threads                          7             0
Allocation Height                       7             0
Allocation Width                        1             0
Local Msgs Sent                       491           983
Distr Msgs Sent                         0             0
Local Msgs Recv'd                     491           983
Distr Msgs Recv'd                       0             0
```

It's a nice easy way to confirm that your queries are being parallelised and also gives you a taste of the amount of messaging activity that's required even for a fairly straightforward task.

V$PQ_SYSSTAT

This view is useful for getting an instance-wide overview of how PX slaves are being used and is particularly helpful in determining possible changes to parallel_max_servers and parallel_min_servers. For example if 'Servers Started' and 'Servers Shutdown' were constantly changing, maybe it would be worth increasing parallel_min_servers to reduce this activity.

V$PX_PROCESS_SYSSTAT contains similar information.

```
SELECT * FROM v$pq_sysstat WHERE statistic LIKE 'Servers%';

STATISTIC                           VALUE
------------------------------ ----------
Servers Busy                            0
Servers Idle                            0
Servers Highwater                       3
Server Sessions                         3
Servers Started                         3
Servers Shutdown                        3
Servers Cleaned Up                      0
```

V$PQ_SLAVE and V$PX_PROCESS

These two views allow us to track whether individual slaves are in use or not and track down their associated session details.

```
SELECT * FROM v$px_process;
```

```
SERV STATUS            PID SPID                SID    SERIAL#
---- --------- ---------- ------------ ---------- ----------
P001 IN USE             18 7680                144         17
P004 IN USE             20 7972                146         11
P005 IN USE             21 8040                148         25
P000 IN USE             16 7628                150         16
P006 IN USE             24 8100                151         66
P003 IN USE             19 7896                152         30
P007 AVAILABLE          25 5804
P002 AVAILABLE          12 6772
```

V$PQ_TQSTAT

V$PQ_TQSTAT shows you table queue statistics for the current session and you must have used parallel execution in the current session for this view to be accessible. I like the way that it shows the relationships between slaves and the query coordinator very effectively[5]. For example, after running this query against the 25,481 row attendance table: -

```
SELECT /*+ PARALLEL (attendance, 4) */ *
FROM attendance;
```

The contents of V$PQ_SYSSTAT look like this: -

```
break on dfo_number on tq_id[6]

SELECT dfo_number, tq_id, server_type, process, num_rows, bytes
FROM v$pq_tqstat
ORDER BY dfo_number DESC, tq_id, server_type DESC[7], process;

DFO_NUMBER      TQ_ID SERVER_TYP PROCESS      NUM_ROWS      BYTES
---------- ---------- ---------- ---------- ---------- ----------
         1          0 Producer   P000             6605     114616
                      Producer   P001             6102     105653
                      Producer   P002             6251     110311
                      Producer   P003             6523     113032
                      Consumer   QC              25481     443612
```

We can see here that four slave processes have been used acting as row Producers, each processing approximately 25% of the rows, which are all consumed by the QC to return the results to the user. Whereas for the following query: -

```
SELECT /*+ PARALLEL (attendance, 4) */ *
FROM attendance
ORDER BY amount_paid;
```

We'll see something more like this.

```
break on dfo_number on tq_id

SELECT dfo_number, tq_id, server_type, process, num_rows, bytes
```

[5] Actually, the contents of this view are so useful and interesting that I'm planning another paper which will be full of example contents

[6] Jonathan Lewis suggested that additional breaks on the report makes each producer/consumer step stand out nicely.

[7] Jonathan also pointed out that it would be a good idea to use descending order on the server_type column because this lists "ranger, producer, consumer in that order - which is the real activity order". I agree that this makes it much easier to follow what's going on.

```
FROM v$pq_tqstat
ORDER BY dfo_number DESC, tq_id, server_type DESC, process;

DFO_NUMBER      TQ_ID SERVER_TYP PROCESS      NUM_ROWS      BYTES
---------- ---------- ---------- ---------- ---------- ----------
         1          0 Ranger     QC                372      13322
                      Producer   P004             5744     100069
                      Producer   P005             6304     110167
                      Producer   P006             6303     109696
                      Producer   P007             7130     124060
                      Consumer   P000            15351     261380
                      Consumer   P001            10129     182281
                      Consumer   P002                0        103
                      Consumer   P003                1        120
                    1 Producer   P000            15351     261317
                      Producer   P001            10129     182238
                      Producer   P002                0         20
                      Producer   P003                1         37
                      Consumer   QC              25481     443612
```

There are a few new things going on here

- The QC acts as a Ranger, which works out the range of values that each PX slave should be responsible for sorting. By implication, it also lets the Producer slaves that will be reading the table know which sorting Consumer slave is the correct 'destination' for a row.
- P004, P005, P006 and P007 are scanning 25% of the blocks each.
- P0001, P002, P003 and P004 act as Consumers of the rows being produced by P004-P007 and perform the sorting activity
- They also act as Producers of the final sorted results for the QC to consume (now that it's finished it's other activities).

What is a little worrying from a performance point of view is that P000 and P001 seem to be doing a lot more work than P002 and P003 which means that they will run for much longer and we're not getting the full benefit of a degree 4 parallel sort. It's a good idea to look at the range of values contained in the sort column.

```
SELECT amount_paid, COUNT(*)
FROM attendance
GROUP BY amount_paid
ORDER BY amount_paid
/

AMOUNT_PAID   COUNT(*)
----------- ----------
        200          1
        850          1
        900          1
       1000          7
       1150          1
       1200      15340
       1995      10129
       4000          1
```

This indicates where the problem lies. We have extremely skewed data because the vast majority of rows have one of only two values, so it's very difficult to parallelise a sort on this column!

V$PX_SESSTAT

This view is a bit like V$SESSTAT but also includes information about which QC and which Slave Set each session belongs to, which allows us to see a given statistic (e.g. Physical Reads) for all steps of an operation.

```
SELECT stat.qcsid, stat.server_set, stat.server#, nam.name, stat.value
FROM v$px_sesstat stat, v$statname nam
WHERE stat.statistic# = nam.statistic#
AND nam.name LIKE 'physical reads%'
ORDER BY 1,2,3
```

```
     QCSID SERVER_SET    SERVER# NAME                      VALUE
---------- ---------- ---------- -------------------- ----------
       145          1          1 physical reads                0
       145          1          2 physical reads                0
       145          1          3 physical reads                0
       145          2          1 physical reads               63
       145          2          2 physical reads               56
       145          2          3 physical reads               61
```

Monitoring the Parallel Adaptive Multi-user Algorithm

If you are using the Parallel Adaptive Multi-User algorithm, it's vital that you are able to check whether any particular operations have been severely downgraded because the server is too busy. There are additional statistics in V$SYSSTAT that show this information.

```
SELECT name, value FROM v$sysstat WHERE name LIKE 'Parallel%'
```

```
NAME                                                             VALUE
---------------------------------------------------------------- ----------
Parallel operations not downgraded                               546353
Parallel operations downgraded to serial                            432
Parallel operations downgraded 75 to 99 pct                         790
Parallel operations downgraded 50 to 75 pct                        1454
Parallel operations downgraded 25 to 50 pct                        7654
Parallel operations downgraded 1 to 25 pct                        11873
```

Clearly, you should be most concerned about any operations that have been downgraded to serial as these may be running many times more slowly than you'd expect. This information is also available in a Statspack report so it's easy to get a view over a period of time. Unfortunately the name column is truncated in the report, which makes it a little difficult to read, but you soon get used to this.

Monitoring the SQL being executed by slaves

As with most dictionary views, we can write queries that combine them to show us interesting or useful information. To offer just one small example, this query will show us the SQL statements that are being executed by active PX slaves. (N.B. The slave must be active, otherwise the SID and SERIAL# it was previously associated with is not contained in the v$px_process view)

```
set pages 0
column sql_test format a60

select p.server_name,
      sql.sql_text
from v$px_process p, v$sql sql, v$session s
```

```
WHERE p.sid = s.sid
and p.serial# = s.serial#
and s.sql_address = sql.address
and s.sql_hash_value = sql.hash_value
/
```

As I mentioned earlier, you'll see completely different results if you run this query on Oracle 10g than on previous versions. First some example results from Oracle 9.2: -

```
P001 SELECT A1.C0 C0,A1.C1 C1,A1.C2 C2,A1.C3 C3,A1.C4 C4,A1.C5 C5,
     A1.C6 C6,A1.C7 C7 FROM :Q3000 A1 ORDER BY A1.C0
```

Whereas on 10g the results look like this: -

```
P000 SELECT /*+ PARALLEL (attendance, 2) */ * FROM attendance ORD
     ER BY amount_paid

P003 SELECT /*+ PARALLEL (attendance, 2) */ * FROM attendance ORD
     ER BY amount_paid

P002 SELECT /*+ PARALLEL (attendance, 2) */ * FROM attendance ORD
     ER BY amount_paid

P001 SELECT /*+ PARALLEL (attendance, 2) */ * FROM attendance ORD
     ER BY amount_paid
```

This is an example of a more general change in 10g. When tracing or monitoring the PX slaves, the originating SQL statement is returned, rather than a block range query as shown earlier in this document. I think this makes it much easier to see at a glance what a particular long-running slave is really doing, rather than having to tie it back to the QC as on previous versions. However, it makes things much trickier when trying to access the execution plan for different slaves[8].

Session Tracing and Wait Events

Tracing an application that uses parallel execution is a little more complicated than tracing non-parallel statements in a few ways.

- A trace file will be generated for each slave process as well as for the query co-ordinator.
- As I've just mentioned, it's time consuming prior to 10g to identify precisely what application operation a PX slave is involved in processing.
- The trace files for the slave processes may be created in background_dump_dest, rather than the standard user_dump_dest. This is version dependant and the trace file for the query coordinator will be in user_dump_dest in any case
- As a result of all the synchronisation and message passing that occurs between the different processes, there are a number of additional wait events.

You won't find too much specific information about tracing Parallel Execution because it's based on exactly the same principles as standard tracing, with the few differences mentioned above. The biggest problem tends to be the large number of trace files that you have to analyse!

---

[8] There's an excellent recent discussion of 10g px tracing techniques in Mark Rittman's weblog at http://www.rittman.net/archives/001240.html

Parallel-specific Wait Events

The first thing to get used to when monitoring PX, whether it be using Statspack at the high level or event tracing at the low level, is that you are going to see a lot more wait events, including types that you won't have seen before. Here are some of the Parallel-specific wait events.

Events indicating Consumers are waiting for data from Producers

- PX Deq: Execute Reply
- PX Deq: Table Q Normal

Oracle's documentation states that these are idle events because they indicate the normal behaviour of a process waiting for another process to do its work, so it's easy to ignore them. However, if you have excessive wait times on these events it could indicate a problem in the slaves. To give you a real-world example, here is the top timed events section of a Statspack report from a production system I worked on.

Event	Waits	Timeouts	Time (s)	(ms)	/txn
direct Path read	2,249,666	0	115,813	51	25.5
PX Deq: Execute Reply	553,797	22,006	75,910	137	6.3
PX qref latch	77,461	39,676	42,257	546	0.9
library cache pin	27,877	10,404	31,422	1127	0.3
db file scattered read	1,048,135	0	25,144	24	11.9

The absolute times aren't important here, just the events. First, it's worth knowing that PX slaves use direct path reads for full table scans and index fast full scans, rather than db file scattered reads. You may already be used to direct path reads because they're used with temporary segments for example. On this system, which was a European-wide Data Warehouse, we were performing long-running SELECT statements as part of the overnight batch run, so a high level of disk I/O was inevitable. (Whether an average wait time of 51 ms is acceptable when you've spent a small fortune on a Hitachi SAN is another matter!)

The next event is PX Deq: Execute Reply, which Oracle considers to be an idle event as I've mentioned. So we ignore that and move down to the next event. The PX qref latch event can often mean that the Producers are producing data quicker than the Consumers can consume it. On this particular system, very high degrees of parallelism were being used during an overnight batch run so a great deal of messaging was going on. Maybe we could increase parallel_execution_message_size to try to eliminate some of these waits or we might decrease the DOP.

But the real problem that we were able to solve was the next event – library cache pin. This event represents Oracle trying to load code into the Library Cache so you wouldn't normally expect to see a significant percentage of wait time for this event unless the Shared Pool is really struggling (which it was on this system).

So next we drill down and start to try session tracing to establish the source of these events. Initially I was unsuccessful in tracking them down until I realised that the PX Deq: Execute Reply was a useful hint. The fact is that many of these wait events were happening in the PX slaves and many of the PX Deq: Execute Reply events were caused by the QC waiting for the PX slaves, which were waiting for the library cache pin latch! So sometimes idle events are important.

Eventually it turned out to be a pretty bad bug in earlier versions of 9.2 (fixed in 9.2.0.5) that caused some of our 2-minute SQL statements to occasionally take 2 hours to run. (Yes, that really does say 2 hours.) Anyway, back to more wait events.

Events indicating producers are quicker than consumers (or QC)

- PX qref latch

I've found that PX qref latch is one of the events that a system can spend a lot of time waiting on when using Parallel Execution extensively (as you can see from the earlier Statspack example). Oracle suggest that you could try to increase `parallel_execution_message_size` as this might reduce the communications overhead, but this could make things worse if the consumer is just taking time to process the incoming data.

Synchronisation Message Events

- PX Deq Credit: need buffer
- PX Deq: Signal Ack
- PX Deq: Join Ack

Although you will see a lot of waits on these synchronisation events – the slaves and QC need to communicate with each other - the time spent should not be a problem. If it is, perhaps you have an extremely busy server that is struggling to cope and reducing the Degree of Parallelism and parallel_max_servers would be the best approach.

Query Coordinator waiting for the slaves to parse their SQL statements

- PX Deq: Parse Reply

Long waits on this event would tend to indicate problems with the Shared Pool as the slaves are being delayed while trying to parse their individual SQL statements. (Indeed, this was the event I would have expected to see as a result of the bug I was talking about earlier but the library cache pin waits were appearing in the Execute phase of the PX slave's work.) Again, the best approach is to examine the trace files of the PX slaves and track down the problem there.

Partial Message Event

- PX Deq: Msg Fragment

This event indicates that parallel_execution_message_size may be too small. Maybe the rows that are being passed between the processes are particularly long and the messages are being broken up into multiple fragments. It's worth experimenting with message size increases to reduce or eliminate the impact of this.

Some Common Sense

One of my favourite descriptions of performance tuning, although I can't remember where I first heard it, is that it is based on 'informed common sense'. That really captures my own experiences of performance tuning. Yes, you need to use proper analysis techniques and often a great deal of technical knowledge, but that's all devalued if you're *completely missing the point*. So let's take a step away from the technical and consider the big picture.

- Don't even think about implementing Parallel Execution unless you are prepared to invest some time in initial testing, followed by ongoing performance monitoring. If you don't, you might one day hit performance problems either server-wide or on an individual user session that you'd never believe (until it happens to you).

- Parallel Execution is designed to utilise hardware as heavily as possible. If you are running on a single-CPU server with two hard disk drives and 512Mb RAM, don't expect significant performance improvements just because you switch PX on. The more CPUs, disk drives, controllers and RAM you have installed on your server, the better the results are going to be.

- Although you may be able to use Parallel Execution to make an inefficient SQL statement run many times faster, that would be incredibly stupid. It's essential that you *tune the SQL first*. In the end, doing more work than you should be, but more quickly, is still doing more work than you should be! To put it another way, don't use PX as a

dressing for a poorly designed application. Reduce the workload to the minimum needed to achieve the task and *then* start using the server facilities to make it run as quickly as possible. Seems obvious, doesn't it?

- If you try to use PX to benefit a large number of users performing online queries you may eventually bring the server to its knees. Well, maybe not if you use the Adaptive Multi-User algorithm, but then it's *essential* that both you and, more important, your users understand that response time is going to be *very* variable when the machine gets busy.

- Using PX for a query that runs in a few seconds is pointless. You're just going to use more resources on the server for very little improvement in the run time of the query. It might well run more slowly!

- Sometimes when faced with a slow i/o subsystem you might find that higher degrees of parallelism are useful because the CPUs are spending more time waiting for i/o to complete. Therefore they are more likely to be available for another PX slave (that isn't waiting on i/o) to use. This was certainly my experience at one site. However, it's also true that using PX will usually *lead* to a busier i/o subsystem because the server is likely to favour full scans over indexed retrieval. There are no easy answers here - you really need to carry out some analysis of overall system resource usage to identify where the bottlenecks are and adjust the configuration accordingly.

- Consider whether PX is the correct parallel solution for overnight batch operations. It may be that you can achieve better performance using multiple streams of jobs, each single-threaded, or maybe you would be better with one stream of jobs which uses PX. It depends on your application so the only sure way to find out is to *try the different approaches*.

Conclusion

Oracle's Parallel Execution capability can improve the performance of long-running tasks significantly by breaking the tasks into smaller sub-tasks that can execute in parallel. The intention is to use as much hardware resource as possible to deliver results more quickly. However, it works best

- On a server which has spare CPU, RAM and i/o throughput capacity
- For tasks which run for more than a few seconds
- For a limited number of concurrent users

If you can meet all of these requirements then the performance improvements can be dramatic but you should consider the potential downsides carefully

- Tracing sessions becomes more difficult, although things are supposed to become easier with 10g
- Unless you are using the Adaptive Multi-user facility you may find your server grinding to a halt one day.
- If you are using the Adaptive Multi-user facility you may find one or more user sessions slowing down dramatically under heavy server workloads.

As with many aspects of Oracle, it's important to plan an effective implementation and test it as thoroughly as possible before inflicting it on your users but when used appropriately, parallel execution is hard to beat.

Bibliography and Resources

The best source of information on Parallel Execution is the Oracle documentation. It's amazing how often I find the (free) manuals far superior to (paid-for) 3rd party books! Specifically, the Data Warehousing Guide contains a couple of relevant chapters :-

Using Parallel Execution

http://download-west.oracle.com/docs/cd/B10501_01/server.920/a96520/tuningpe.htm#19664

Parallelism and Partitioning in Data Warehouses

http://download-west.oracle.com/docs/cd/B10501_01/server.920/a96520/parpart.htm#745

There are also a number of useful resources on Metalink, including a dedicated section containing the most useful notes that you can access by selecting 'Top Tech Docs', 'Database', 'Performance and Scalability' and then 'Parallel Execution' from your Metalink home page. The following documents are particularly relevant to this paper.

184417 – Where to track down the information on Parallel Execution in the Oracle documentation!

203238 – Summary of how Parallel Execution works.

280939 - Checklist for Performance Problems with Parallel Execution (but also contains a useful explanation of DFO effects on number of PX slaves used)

119103 – Parallel Execution Wait Events (contains links to event-specific information)

201799 – init.ora parameters for Parallel Execution

240762 - pqstat PL/SQL procedure to help monitor all of the PX slaves running on the instance or for one user session

202219 – Script to map PX slaves to Query Co-ordinator (An alternative to using the procedure in note 240762.1)

275240 – Discusses Bug no. 2533038 in the Parallel Adaptive Multi-User algorithm (fixed in 9.2.0.3) which makes it sensitive to idle sessions, leading to an under-utilised system.

238680 – Investigating ORA-4031 errors caused by lack of memory for queues.

242374 – Discusses some of the issues around PX session tracing (but not in any great depth)

237328 – Direct Path Reads (brief discussion)

The following books contain some useful information about Parallel Execution.

Harrison, Guy. *Oracle SQL High Performance Tuning*. Prentice Hall.

Kyte, Thomas. *Effective Oracle by Design*. Oracle Press.

Lewis, Jonathan. *Practical Oracle 8i – Building Efficient Databases*. Addison Wesley.

Mahapatra,Tushar and Mishra Sanjay. *Oracle Parallel Processing*. O'Reilly & Associates, Inc.

There's also an excellent Parallel Execution conference paper by Jeff Maresh. As well as covering server configuration it contains more developer-orientated information about using PX within your application than I've covered in this document.

Maresh, Jeff. *Parallel Execution Facility Configuration and Use*. http://www.evdbt.com/PX_2003.doc

And, as usual Jonathan Lewis has some previous articles on this subject! Although these are geared to Oracle 7.3, much of the content makes sense across versions.

Lewis, Jonathan. http://www.jlcomp.demon.co.uk/ind_pqo.html

Finally, here's a nice PX slave monitoring query from the "Co-operative Oracle Users' FAQ"

http://www.jlcomp.demon.co.uk/faq/pq_proc.html

Acknowledgements

I'd like to thank the following individuals for first class feedback, without which this article would have been fundamentally flawed. Although I haven't incorporated all of their suggestions, I've tried to correct any technical errors that they identified. In the end, any errors are mine and there were a lot more of them before these guys helped out.

Glyn Betts, Sun Microsystems
Andrew Campbell, Sun Microsystems
Carl Dudley, University of Wolverhampton
Jari Kuhanen, Sun Microsystems
Tom Kyte, Oracle Corp.
Jonathan Lewis, JL Computing

About the author

Doug Burns is an independent consultant who has 14 years experience working with Oracle in a range of industries and applications and has worked as a course instructor and technical editor for both Oracle UK and Learning Tree International. He can be contacted at dougburns@yahoo.com and this document and other articles are available on his website at http://doug.burns.tripod.com.