BATTLE OF THE NODES: RAC PERFORMANCE MYTHS

Introduction

This paper is to explore various misconceptions about Real Applications Cluster (RAC) and performance issues encountered in RAC databases. Scripts are provided, inline, wherever possible for reproducing test cases.

This paper is NOT designed as a step by step approach, rather designed as a guideline. Every effort has been taken to reproduce the test results. It is possible for the test results to differ slightly due to version/platform differences.

Myth 1: Performance of a node doesn't affect performance of other nodes

In a classic RAC architectural design pattern, node wise workload management is performed. Few nodes are dedicated to support critical business functionality and one or two nodes are dedicated to support adhoc and reporting business functionality. This workload management stems from the fact that by insulating adhoc and reporting functionality to one node, performance of the critical functionality in other nodes will not be affected. While there is some merit to this argument (and it does provide hardware level insulation), this methodology does not provide RAC global fusion insulation. Further, all forms of adhoc and costly SQL statements are executed from these nodes in the hope that the performance of other nodes (which are critical to business functions) will not be affected. This is an incorrect assumption. In RAC, all nodes contribute to Cache Fusion performance and performance issues in one node will be visible in other nodes.

Further, size of reporting node is kept smaller with fewer resources. This results in over use of reporting nodes, typically, with a resource usage exceeding that node capacity. In Figure 1.1, online users are using all instances except instance 4. Instance 4 is allocated for batch, reporting and adhoc query tool user loads.

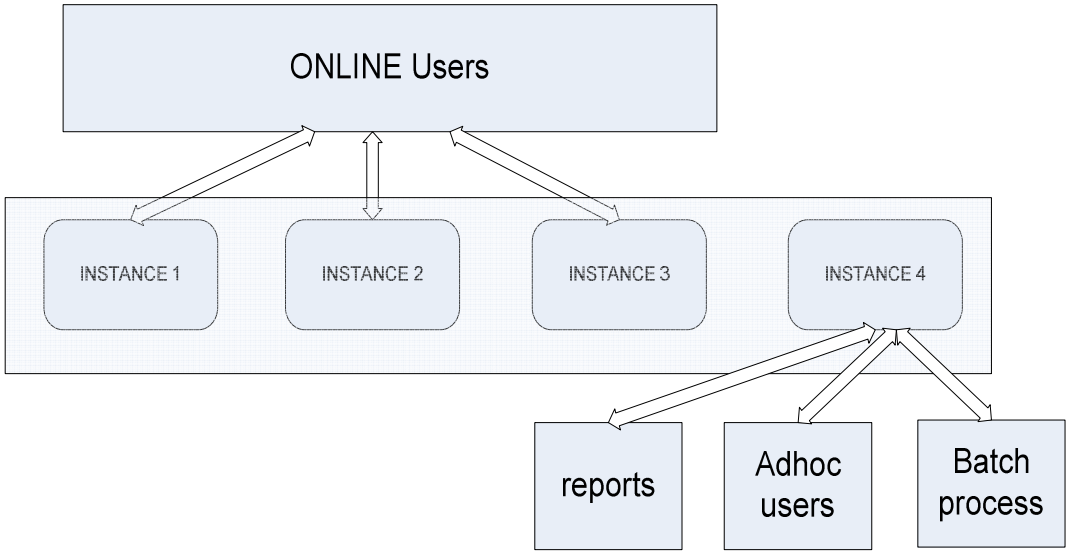

Figure 1.1 Typical RAC setup

From version 9i, thanks to cache fusion, blocks are transferred from remote cache if suitable block is found in remote cache avoiding disk reads. Block transfer is performed by LMS processes. If LMS processes are running in normal priority (as in version 9i) and If these LMS processes are suffering from CPU starvation due to excessive resource usage in a node, then performance of the cache fusion traffic will be affected in other nodes. Figure 1.2 shows that LMS processes running in all four nodes are communicating also termed as Cache Fusion.

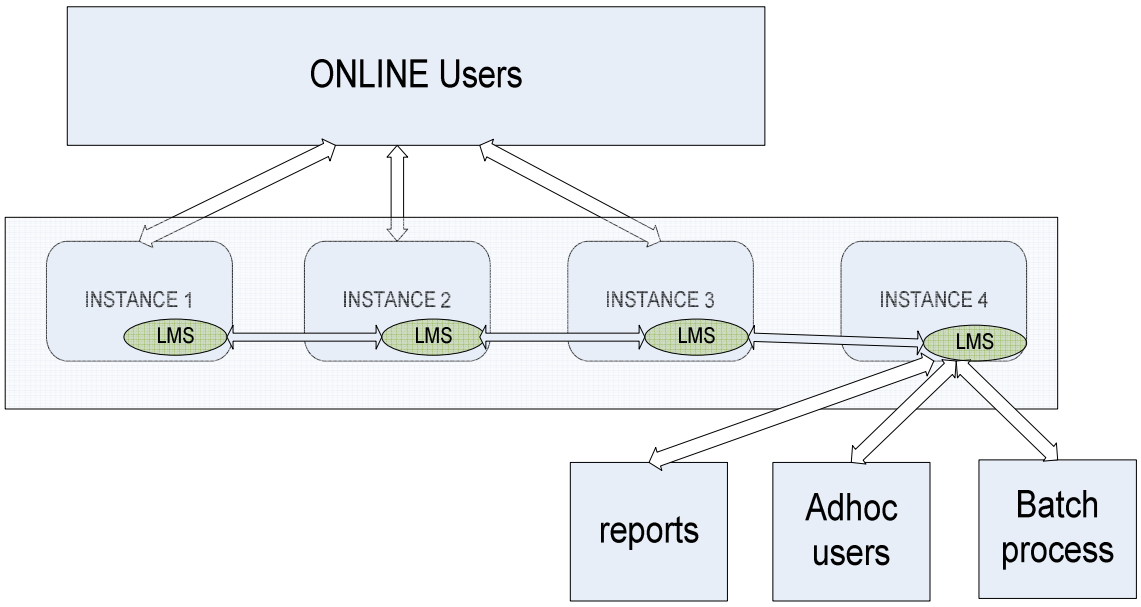

Figure 1.2 LMS process and Cache fusion

There are two major issues with off-loading all batch work to one node:

1. Instance 4 can receive much more Global Cache traffic. If the batch processes or adhoc SQL statements are not tuned, then this will increase buffer gets. If the buffers are already in the remote instances supporting online users, then those instances' LMS processes must work harder to send the buffers to reporting node. To complicate things further, Oracle Database version 10g introduces Dynamic ReMastering of resource and if an instance accesses an object much more than other instances then those objects might be remastered. So, an object needed for OLTP performance might be accidentally remastered to instance 4.
2. Instance 4 can transmit more Global Cache traffic. If the buffers are transferred to instance 4 excessively or if the objects are remastered to instance 4, then Global communication between nodes might increase too.

Before a buffer can be sent to the remote cache, LGWR must perform a log flush for Consistent Read (CR) buffers if the blocks were constructed by applying undo records. LMS process will wait for an event known as 'gcs log flush' before transmitting the buffer to a server process in the remote node. If CPU or I/O usage of a node is saturated, then performance of that LMS process will suffer. Typically, Global cache CR waits such as 'gc cr grant 2 way' (10g), 'global cache cr request' latency increases due to global communication or cache fusion latencies. This global cache performance is blamed on interconnect network hardware or path. But, interconnect is probably performing fine and global cache latency is caused by LMS process latencies.

In real life, this myth is already in play and it is quite hard to change its dynamics. Of course, there are following ways to work around this issue:

1. More LMS processes? Typical response to this issue, if LMS processes have been identified as a culprit, to increase number of LMS processes. This change has negative effect on performance in many cases. If a RAC node is suffering from resource starvation, then adding more processes typically results in further degradation in performance.

   Modern CPUs have cache affinity designed in them and so, processes tend to be executed in the same CPU sets to improve efficiencies. By adding more LMS processes, TLB misses and cache thrashing increases. This can be evidenced by carefully monitoring xcalls/migrates/TLB misses in mpstat and trapstat outputs. In summary, few busy LMS processes are better then many quasi-busy LMS processes.

   Twice the number of interconnect hardware is a good starting point. For example, if you have 4 network cards supporting cluster interconnect traffic, then start with 8 LMS processes.

2. LMS process priority? LMS processes are also running in normal priority in Oracle Database version 9i. In versions 9i, increasing LMS process priority to RT or FX (with larger CPU quanta scheduling schemes) helps. Oracle Corporation has already identified this potential issue and in release 10g, this issue is resolved: LMS processes are running with RT priority.

   Two parameters control this behaviour[1]:
   a. _high_priority_processes: This parameter controls processes that will have higher priority and v$parameter describtes this parameter as "High priority process name mask" with default value of LMS* or LMS*|VKTM* depending upon few other parameters.
   b. _os_sched_high_priority: This controls whether OS scheduling high priority is enabled or not and defaults to 1. By default it is enabled and LMS processes runs in RT mode.

By enabling higher priority for LMS processes, we will be able to reduce the effect of report node resource usage issues.

Myth 2: All global cache performance issues are due to interconnect

Much of the global cache performance is blamed on interconnect or interconnect hardware. That is not necessarily the case. Let's consider few lines printed from statspack output in listing 1.1 from an Oracle 9i version database. This listing shows few lines from a statspack captured during a customer performance issue.

```
Global Cache and Enqueue Services - Workload Characteristics
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                  Avg global enqueue get time (ms):      8.9

       Avg global cache cr block receive time (ms):     63.3
   Avg global cache current block receive time (ms):     42.1
```

[1] In some cases, I have noticed that running these processes in RT can have negative effect too. Especially, if the node size is very small and overloaded, then kernel and other critical processes doesn't get enough CPU. This can result in node evictions due to timeout failures.

```
          Avg global cache cr block build time (ms):      0.3
           Avg global cache cr block send time (ms):      0.1
Global cache log flushes for cr blocks served %:      4.5
          Avg global cache cr block flush time (ms):     51.5

       Avg global cache current block pin time (ms):      0.0
      Avg global cache current block send time (ms):      4.8
Global cache log flushes for current blocks served %:      0.1
     Avg global cache current block flush time (ms):     30.0
```

Listing 1.1 Global Cache CR Performance

From the listing 1.1 we can see that average global cache CR block receive time is 63.3ms and it is very high. It is easy (and most probably incorrect) to conclude that interconnect hardware is causing this performance issue. There are many other factors that can cause Global Cache performance issues and those factors need to be considered. We need to look at more performance metrics before arriving to a conclusion. Usually, with decent hardware, interconnect performance is within an acceptable range.

In the scenario above performance of Global cache for CR mode of transmission is affected. It might be easy to conclude, since this wait is for CR buffers, as an interconnect issue. But, Global cache receive time is affected by few other factors including Log writer performance issues.

So, global cache latency can be roughly broken down to three major components:

> Interconnect message latency from and to the LMS processes +
> LMS processing latency +
> LGWR processing latency to complete Log flush (if required).

It is important to review performance metrics in all nodes. Review of workload characteristics in node 3 statspack throws light in to this problem. Listing 1.2 is showing few lines from the statspack report from node 3. I think, the problem is visible from listing 1.2. Both interconnect hardware and LMS processes are working fine. But, LGWR writes are suffering from extreme slow I/O response[2]. After resolving LGWR performance, Global cache response times were back to normal and acceptable range.

```
Global Cache and Enqueue Services - Workload Characteristics
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                    Avg global enqueue get time (ms):      0.3

         Avg global cache cr block receive time (ms):     10.4
    Avg global cache current block receive time (ms):      3.2

          Avg global cache cr block build time (ms):      0.1
           Avg global cache cr block send time (ms):      0.0
     Global cache log flushes for cr blocks served %:      5.0
          Avg global cache cr block flush time (ms):   4380.0

       Avg global cache current block pin time (ms):      0.0
      Avg global cache current block send time (ms):      0.1
Global cache log flushes for current blocks served %:      0.1
     Avg global cache current block flush time (ms):      0.0
```

Listing 1.2 Global Cache CR and LGWR performance

[2] Which happened to be a faulty hardware and a firmware bug.

That leads us to our next diversion. In addition to the discussions in myth 1 for LMS processes, it is also prudent to consider following:

1. LGWR processes might also need to run with higher priority, in addition to LMS processes (only applicable to 9i and above). But, this needs to be carefully tested. In few cases, LGWR might be stealing off the CPU from the processes generating redo and cause performance degradation.
2. Better write throughput for redo log files is quite essential. Higher interconnect traffic eventually lead higher or hyperactive LGWR. Consider using non-buffered I/O and asynchronous I/O for redo log files. Asynchronous I/O is very important if there are multiple log group members. If asynchronous I/O is not in use, then LGWR must write to each online log file member sequentially, before releasing commit markers. This can lead to performance degradation.
3. In solaris platform, priocntl can be used to increase priority of LGWR and LMS processes. This is needed only in Oracle Database version 9i. In Oracle version 10g, parameter _high_priority_processes can be set to "LMS*|LGWR*|VKTM" to increase LGWR priority to RT.
   priocntl -e -c class –m userlimit -p priority
   priocntl -e -c RT -p 59 `pgrep -f ora_lgwr_${ORACLE_SID}`
   priocntl -e -c FX -m 60 -p 60 `pgrep -f ora_lms[0-9]*_${ORACLE_SID}`
4. Binding: Another option is bind LMS and LGWR processes to specific processors or processor sets. This should reduce TLB thrashing and improve efficiency of LMS/LGWR processes.
5. It is worthwhile to fence interrupts to one or two processors. Refer psradm for further reading in Solaris environments. But, of course, processor binding and fencing is only applicable to servers with high number of CPUs. Engaging interrupt fencing in the servers with few CPUs will have negative consequences.

Myth 3: Inter-node parallel execution is excellent and use it aggressively

In Real Application Cluster databases, a query can be executed in parallel allocating slaves from multiple nodes. For example, in a 6 node RAC cluster, a parallel query can allocate 4 slaves from each node with a combined parallelism of 24. Allocating slaves from multiple nodes to execute SQL statements *can* cause performance issues. Let me be very clear here, I am not suggesting that inter-node parallelism never should be used, only that all ramifications carefully considered.

Intra instance parallelism is controlled by parameters such as parallel_min_servers, parallel_max_servers and other parallel* parameters. Inter-node parallel execution is controlled by RAC specific parameters. Specifically, instance_group and parallel_instance_group parameters determine the allocation of parallel servers in the local and remote nodes (until version 10g). From Oracle Database version 11g onwards, you need to use database services to implement inter-node parallel execution.

Parameter instance_group dictates the group membership of an instance. For example, we could specify that instance group OLTP is made up of instances 1,2 and 5. Parameter parallel_instance_group dictates the parallel slave instance group. It is easier to explain these two parameters with few examples:

Say, there are three nodes in this RAC cluster inst1, inst2 and inst3.

Scenario #1:

In this first scenario, an instance_group 'all' encompassing all three instances is setup and the parameter parallel_instance_group is set to 'all'. So, parallel slaves for a parallel SQL statements initiated from any instance *can* be allocated from all three nodes. In the figure 1.3 show how parallel slaves can be executed from all three nodes with global co-ordinator process is executing in node 1. Further, these slaves are not randomly allocated and there is much intelligence and usually least loaded nodes are selected, execution plan of the SQL statements, type of SQL statement etc.

Parameters:

```
inst1.instance_groups='inst1','all'
inst2.instance_groups='inst2','all'
inst3.instance_groups='inst3','all'
inst1.parallel_instance_group='all'
```

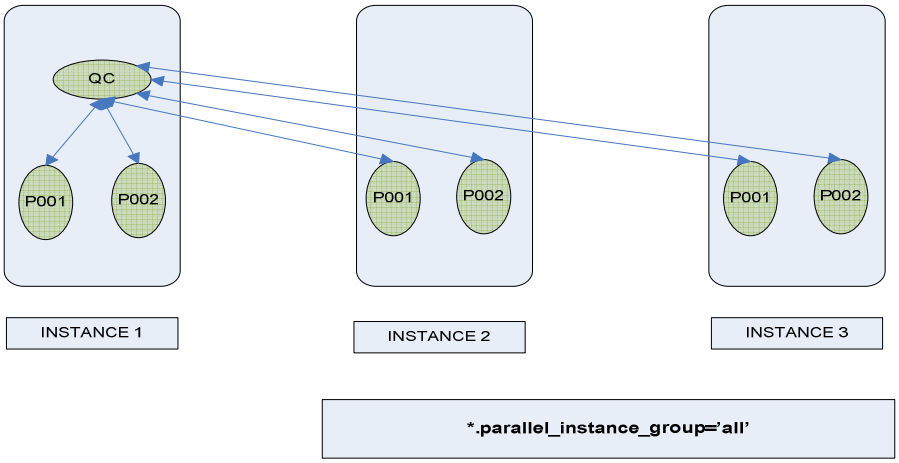

Figure 1.3 Parallel query slaves allocated in all three nodes

Scenario #2:

In this scenario, both inst1 and inst2 are members of instance_group inst12. But, inst3 is not a member of this instance_group. Figure 1.3 depicts a scenario where only two nodes are participating to execute this parallel SQL statement.

```
inst1.instance_groups='inst12','all'
inst2.instance_groups='inst12','all'
inst3.instance_groups='inst3','all'
inst1.parallel_instance_group='inst12'
```

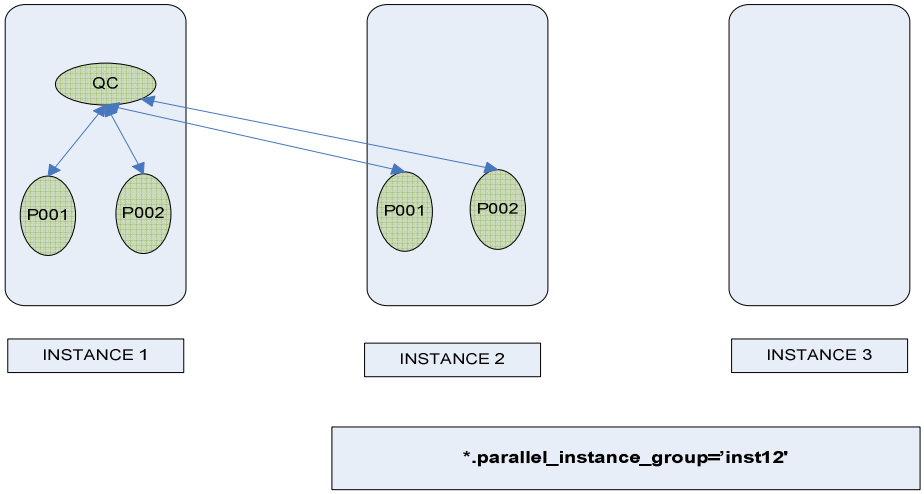

Figure 1.3 Parallel query slaves allocated in two nodes

Parallel Query and RAC optimization

We also need to discuss recent optimization changes, especially, in 10g and 11g with PQ slaves allocation, communication and RAC. We will consider two SQL statements to establish these optimization features:

SQL 1

Consider the SQL statement in Listing 1.3 performing a simple aggregation of a big table with 8 parallel slaves. Parallel_instance_group was set to inst12 so as to allocate parallel slaves from instance 1 and instance 2. Due to aggregation step, there can be twice as many as slaves allocated and so 16 slaves are allocated from two instances for this SQL statement.

```
select /*+ parallel ( t1, 8) */
      min (t1.CUSTOMER_TRX_LINE_ID +t1.CUSTOMER_TRX_LINE_ID ) ,
      max ( t1.SET_OF_BOOKS_ID+t1.set_of_books_id ),
      avg(t1.SET_OF_BOOKS_ID +t1.set_of_books_id),
      avg( t1.QUANTITY_ORDERED + t1.QUANTITY_ORDERED ),
      max(t1.ATTRIBUTE_CATEGORY ), max(t1.attribute1),
      max(t1.attribute2)
from
BIG_TABLE t1;
```

Listing 1.3 Parallel SQL with simple aggregation

Listing 1.4 shows the allocation of slaves between two instances. Eight slaves are allocated in instance 1 and 8 slaves are allocated in instance 2, Query coordinator (QC) process is running from instance 1. Notice that amount of communication between query co-ordinator process and the slaves running in instance 2 under the size column. Size column[3] is printing the amount of interconnect traffic between QC process running in instance 1 and slaves running in instance 2. It looks like parallel slave processes are aggregating data and sending only aggregated results to the Query Co-ordinator process. Query co-ordinator must perform second level aggregation to derive final results. In a

[3] Size column is calculated from a Solaris tool. Until 11g, PQ traffic is not accounted in cluster traffic in the database. These measurements were taken in Oracle Database version 10g. To measure UDP traffic over the interconnect, a Solaris tool udpsnoop.d was used. This tool is part of dtrace tool kit.

nutshell, for these types of SQL statements with simple aggregation, interconnect traffic is kept to a minimum[4].

```
   INST Username      QC/   Slave SID    QC      Req. Act. Size
                      Slave   Set                DOP  DOP
------- ------------ ----- ----- ----- -------- ---- ---- -----
      1 SYS          QC            10933  10933                3314
      1  - p000      (Slave)     1 10958  10933   16    16
      1  - p001      (Slave)     1 10948  10933   16    16
      1  - p002      (Slave)     1 10953  10933   16    16
      1  - p003      (Slave)     1 10925  10933   16    16
      1  - p004      (Slave)     1 10916  10933   16    16
      1  - p005      (Slave)     1 10938  10933   16    16
      1  - p006      (Slave)     1 10951  10933   16    16
      1  - p007      (Slave)     1 10946  10933   16    16

      2  - p000      (Slave)     1 10949  10933   16    16  2152
      2  - p001      (Slave)     1 10937  10933   16    16  2152
      2  - p002      (Slave)     1 10946  10933   16    16  2152
      2  - p003      (Slave)     1 10956  10933   16    16  2152
      2  - p004      (Slave)     1 10902  10933   16    16  2152
      2  - p005      (Slave)     1 10981  10933   16    16  2152
      2  - p006      (Slave)     1 10899  10933   16    16  2152
      2  - p007      (Slave)     1 10927  10933   16    16  2152
```

Listing 1.4 Parallel slave allocation and message size

SQL 2

Consider the SQL statement in Listing 1.5. This SQL statement is performing slightly complex aggregation. Two tables are joined in this example using HASH JOIN technique and then aggregated to derive final result set.

```
select /*+ parallel ( t1, 16)   parallel (t2, 16)  */
    t1.CUSTOMER_TRX_LINE_ID , t1.SET_OF_BOOKS_ID,
max(t1.ATTRIBUTE_CATEGORY ), s
um( t2.QUANTITY_ORDERED), sum(t1.QUANTITY_ORDERED)
from
   backup.big_table1_part t1,
   backup.big_table1_part t2
where t1.CUSTOMER_TRX_LINE_ID = t2.CUSTOMER_TRX_LINE_ID
group by
    t1.CUSTOMER_TRX_LINE_ID , t1.SET_OF_BOOKS_ID
;
```

Listing 1.5 Parallel SQL for a slightly complex execution

Listing 1.6 shows the slave allocation for the SQL statement in listing 1.5. You can see that 16 slaves were allocated in instance 1 and 16 slaves were allocated in instance 2. Also notice the amount of interconnect traffic between the PQ slaves in instance 2 and the QC process. Approximately, total interconnect traffic for this SQL statement in one direction was 2.4GB.

```
1 SYS QC  9868   9868
1  - p007      (Slave)    1 9890   9868     1616
1  - p006      (Slave)    1 9949   9868     1616
1  - p005      (Slave)    1 9871   9868     1616
1  - p004      (Slave)    1 9878   9868     1616
1  - p003      (Slave)    1 9898   9868     1616
```

[4] Size column is empty for local slaves since udpsnoop.d is measuring traffic over interconnect and local slaves are not using the interconnect for communication with QC process.

```
1  - p002      (Slave)    1 9969   9868     1616
1  - p001      (Slave)    1 9897   9868     1616
1  - p000      (Slave)    1 9924   9868     1616

1  - p015      (Slave)    2 9867   9868     1616  128020044
1  - p014      (Slave)    2 9908   9868     1616  132011920
1  - p013      (Slave)    2 9917   9868     1616  127770024
1  - p012      (Slave)    2 9938   9868     1616  128154240
1  - p011      (Slave)    2 9877   9868     1616  183490248
1  - p010      (Slave)    2 9895   9868     1616  181900075
1  - p009      (Slave)    2 9942   9868     1616  181828128
1  - p008      (Slave)    2 9912   9868     1616  124428800

2  - p023      (Slave)    1 9904   9868     1616
2  - p022      (Slave)    1 9941   9868     1616
2  - p021      (Slave)    1 9928   9868     1616
2  - p020      (Slave)    1 9870   9868     1616
2  - p019      (Slave)    1 9880   9868     1616
2  - p018      (Slave)    1 9934   9868     1616
2  - p017      (Slave)    1 9910   9868     1616
2  - p016      (Slave)    1 9913   9868     1616

2  - p031      (Slave)    2 9902   9868     1616  127068484
2  - p030      (Slave)    2 9883   9868     1616  126904080
2  - p029      (Slave)    2 9882   9868     1616  127767353
2  - p028      (Slave)    2 9920   9868     1616  128154145
2  - p027      (Slave)    2 9916   9868     1616  128096875
2  - p026      (Slave)    2 9903   9868     1616  182490908
2  - p025      (Slave)    2 9893   9868     1616  181899025
2  - p024      (Slave)    2 9897   9868     1616  181611641
```

Listing 1.6 Parallel SQL for a slightly complex execution

This convincingly prove that amount of interconnect communication increases if parallel slaves are allocated from multiple instances to execute a parallel query. Essentially, if the application needs to use parallelism across many instances, then interconnect hardware must be properly designed and configured.
A white paper supplied by Oracle Corporation also has recommendations along the same line. Listing 1.7 shows few lines from the white paper available free from the URL:
http://www.oracle.com/technology/products/bi/db/11g/pdf/twp_bidw_parallel_execution_11gr1.pdf

> "If you use a relatively weak interconnect, relative to the I/O bandwidth from the server to the storage configuration, then you may be better of restricting parallel execution to a single node or to a limited number of nodes; inter-node parallel execution will not scale with an undersized interconnect. As a general rule of thumb, your interconnect must provide the total I/O throughput of all nodes in a cluster (since all nodes can distribute data at the same point in time with the speed data is read from disk); so, if you have a four node cluster, each node being able to read 1GB/sec from the I/O subsystem, the interconnect must be able to support 4 x 1GB/sec = 4GB/sec to scale linearly for operations involving inter-node parallel execution. It is not recommended to use inter-node parallel execution unless your interconnect satisfies this requirement (or comes very close)."

Listing 1.7 Few lines from Oracle White paper.

In summary, if you are planning to allow parallelism to spawn multiple nodes, verify that interconnect hardware can support it. Especially, if the SQL statements are executed concurrently by many processes then interconnect hardware can quickly become a bottleneck. In fact, this is a customer performance issue we resolved, in which, parallelism on many queries were turned on

causing enormous performance issues. Since the Global cache statistics for Parallel queries are not captured in the statspack they were not easily visible.

Myth 4: Run same batch process concurrently from both nodes.

Let me tell you a story about customer performance issue. A batch process was designed to accept organization_id as an argument and spawns multiple processes to complete the business functionality in an E-Business Suite 11i environment. Client decided to schedule these batch programs to execute from 2 nodes, in order to use CPUs from both nodes effectively. Programs will process one set of organization_id in one node and other set of organization_ids in the second node. While I can understand the need to use all CPUs effectively to improve performance, it arises from fallacy that logical isolation of the batch process is good enough. Since each disjoint set of organization_ids will be processed by a node, there is sufficient logical isolation, but no physical isolation. In a nutshell, even though these processes are operating on two disjoint set of logical rows, physically these rows are within the same physical block, i.e. no effective use of partitioning.

Few batch processes were started from node 1 with a set of organization_ids and few of them started from node 2 with another set of organization_ids. Each of these batch processes in turn started with at least 5 processes in each node. Performance of the whole application and the batch process was unacceptable.

These batch processes were concurrently accessing same set of blocks from both nodes. As we discussed already, transfer of CR blocks needs log flush and this increases LGWR activity. Further, only one instance can hold a buffer in CURRENT mode, and of course, CURRENT mode transfers also needs log flush. Essentially, both CR and CUR mode global cache transfer increased drastically and LGWR activity also increased.

Typical global cache waits are 1-5ms for a decent hardware setup and a typical single block or multi-block reads are in the order of 2-10ms. But, a typical access to SGA buffer in local instance in nano-seconds in the range of 10-100ns. In a non-RAC setup, these batch processes will access buffers in nano-seconds and were working fine in development environment. With RAC and with this design of logical-isolation-but-no-physical-isolation, waits increased to 1-3ms from 10-100ns, which is 1000 times worse. In addition, these transfers were introducing hot spots for LGWR processing and interconnect transfer leading to a performance impact for the application.

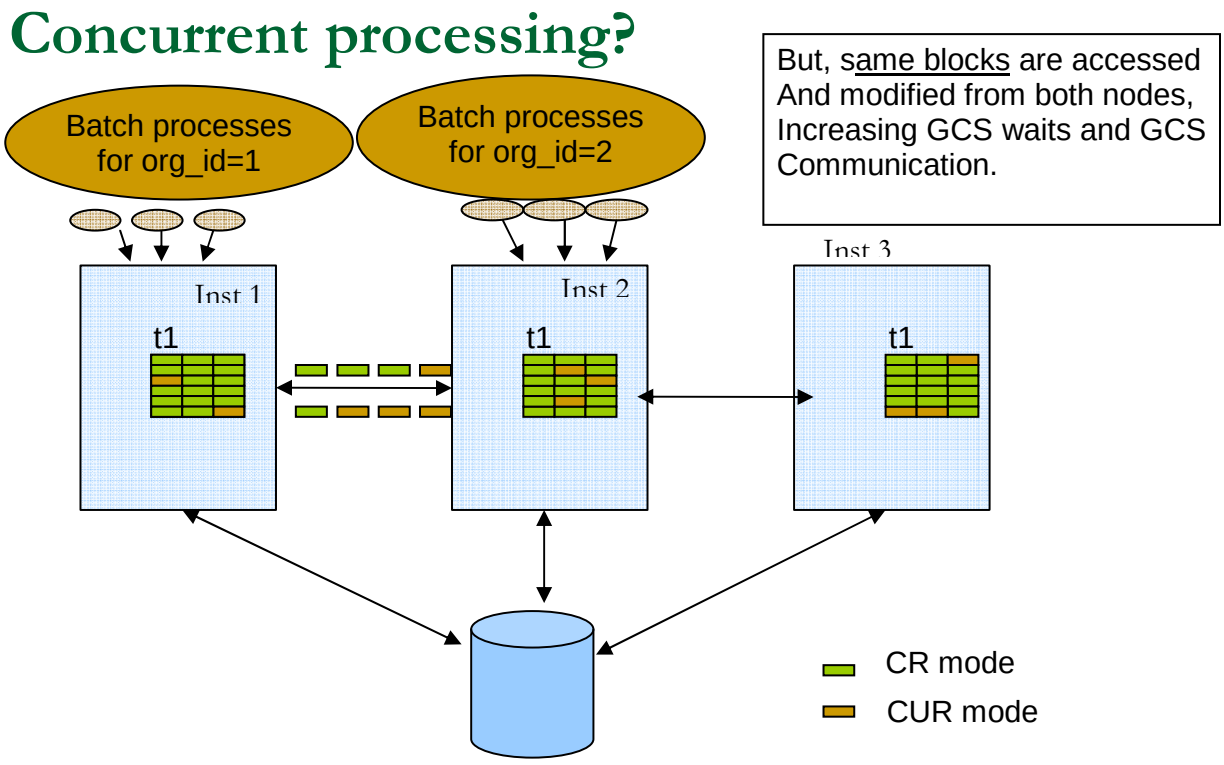

Figure 1.4 Concurrent processing

Listing 1.8 shows few lines from the Statspack report during this issue. (5 minute statspack from one node, database version 9.2).

```
Top 5 Timed Events
~~~~~~~~~~~~~~~~~~
                                                                      % Total
Event                                               Waits    Time (s) Ela Time
--------------------------------------------- ------------ ----------- --------
CPU time                                                         3,901    46.17
PL/SQL lock timer                                       80       2,316    27.42
global cache cr request                            123,725         670     7.93
global cache null to x                              13,551         446     5.28
buffer busy global CR                               12,383         215     2.54
```

Listing 1.8 Few lines from statspack output

Global cache wait events add up to 17%. Since 'PL/SQL lock timer' event is form another process and can be considered idle waits. If we omit 'PL/SQL lock timer' then total global cache wait events are adding up to of 35% of elapsed time (approximately).

Trace file for that batch process also shows similar problem.

```
Elapsed times include waiting on following events:
  Event waited on                          Times   Max. Wait  Total Waited
  ----------------------------------------  Waited  ----------  ------------
  log file sync                               13        0.08          0.28
  latch free                                2185        0.17         32.76
  global cache cr request                  21719        0.31         71.18
  db file sequential read                     50        0.04          0.33
```

```
global cache s to x                          791        0.06          5.31
row cache lock                               113        0.02          0.41
global cache null to x                      4881        0.30         97.41
buffer busy global CR                       3306        0.23         15.26
global cache open x                          903        0.06          5.83
buffer busy waits                           1462        0.98         17.90
global cache busy                            644        0.98         10.36
buffer deadlock                              224        0.02          0.07
global cache null to s                        89        0.15          0.77
buffer busy global cache                    1066        0.31         27.62
enqueue                                      302        0.17          2.61
KJC: Wait for msg sends to complete          102        0.00          0.00
global cache open s                           62        0.01          0.13
cr request retry                               4        0.00          0.00
```

Listing 1.8 Few lines from the trace file

In summary, we need to consider isolation at physical layer. If two batch processes are accessing same table blocks aggressively, then they need to run from same node. Proper partitioning will help provided that special attention given to global indices.

Myth 5: Set Sequence to nocache in RAC environments to avoid gaps

Sequence values are cached in instance memory and by default 20 values are cached. As instance cache is transient, loss of an instance can result in loss of cached sequence values. Permanent record of highest possible value from any instance is kept track in SEQ$ table. SQL statements accessing a sequence within an instance will access instance SGA. As each instance caches its own sequence values it is highly likely that SQL statements accessing this sequence from different instance will create gaps in sequence values.

To avoid so called gaps, designers usually create sequences with attributes nocache and order. It is not exactly a good practice as we will discuss in this topic.

Sequence of operation accessing this sequence with cache value set to 20 is:

1. Assume that starting value of that sequence cust_id_sq in SEQ$ [5] table high water mark value is 9. Also assume that cache value for that sequence is set to 20.
2. First access to that sequence from instance 1 will cache values from 10-29.
3. High water mark for that sequence in SEQ$ table updated to 29.
4. First access to that sequence from instance 2 will cache values from 30-49.
5. High water mark for that sequence in SEQ$ table updated to 49.
6. From instance 1, subsequent access to that sequence will return values from 10 until 29.
7. After the value of 29 sequence cache for that sequence is essentially depleted and next access to that sequence cust_id_sq in instance1 will return values from 50-69
8. High water mark of that sequence in SEQ$ high water mark updated to 69.

[5] DBA_SEQUENCES.LAST_NUMBER is an alias for SEQ$.highwater column.

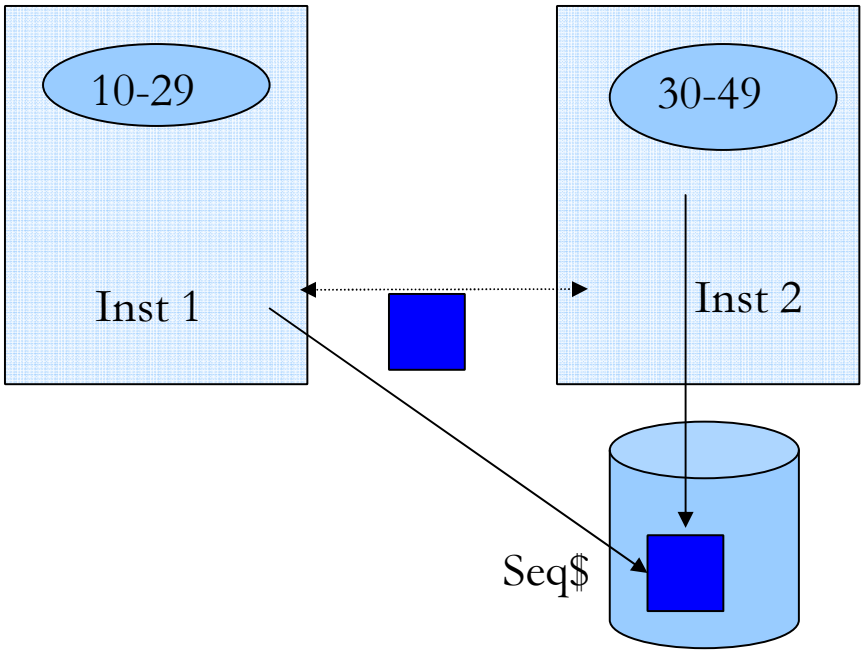

Figure 1.5 Sequence processing with cache value of 20

In summary,

1. 60 accesses to the sequence cust_id_sq resulted in 3 updates to the SEQ$ table block for that row.
2. These three changes may or may not result in physical write of that block. (due to SGA caching and lazy DBWR write schemes).
3. Gaps in sequence possible. Sequence values can be lost if instance crashes or shared pool is flushed.
4. Changes to SEQ$ table block, results in cache fusion transfer.

Let's consider the scenario, if sequence cache value is set to nocache or ORDER. Following listing explains the operational details for this sequence operations:

1. Assume that starting value of that sequence cust_id_sq in SEQ$ table high water mark value is 9. Also assume that cache value for that sequence is set to 20.
2. First access to sequence from instance 1 will return value of 10.
3. SEQ$ high water mark for that sequence is updated to 10 for that sequence.
4. First access to that sequence cust_id_sq from instance 2 will return value of 11.
5. SEQ$ high water mark is updated to 11 for that sequence.
6. From instance 1, subsequent access to that sequence will return value as 12.
7. SEQ$ high water mark updated to 12.

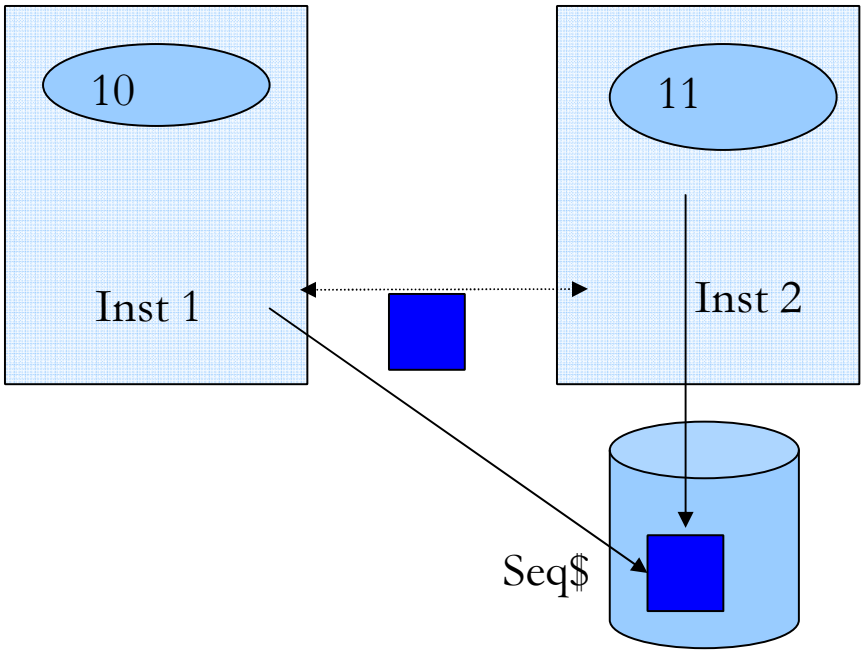

Figure 1.6 Sequence processing with cache value of 1

In summary,

1. Three accesses to this sequence results in three block changes.
2. No gaps in sequence possible.
3. But, SEQ$ table blocks transferred back and forth.
4. This will increase the stress on updates to row cache and 'row cache lock' waits will be visible.
5. Increases global cache communication too.

Test case:

Listing 1.8 shows a simple test case. 10,000 rows are inserted in to a table using a sequence generated values with a PL/SQL loop. Rows are committed after every 1000 row inserts.SQLTRACE is turned on at level 8 using event 10046 syntax.

```
set timing on
alter session set events '10046 trace name context forever, level 8';
declare
  l_v1 varchar2(512);
  l_n1 number :=0;
begin
 for loop_cnt in 1 .. 10000
  loop
      -- Random access
      -- Also making undo blocks to be pinged..
      insert into t1
      select t1_seq.nextval, lpad( loop_cnt, 500, 'x') from dual;
      if mod(loop_cnt, 1000) =0 then
        commit;
```

```
      end if;
    end loop;
  end;
  /
```

Listing 1.8 Few lines from the trace file

Tkprof output of above code executions in single node printed in the Listing 1.9. Since the cache of this sequence is 1, table seq$ is updated 10000 times. But, there are not any other performance issues that we notice.

```
INSERT INTO T1 SELECT T1_SEQ.NEXTVAL, LPAD( :B1 , 500, 'x') FROM DUAL
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute  10000      5.28       7.66          1        794      25670       10000
Fetch        0      0.00       0.00          0          0          0           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total    10001      5.29       7.66          1        794      25670       10000


update seq$ set increment$=:2,minvalue=:3,maxvalue=:4,cycle#=:5,order$=:6,
  cache=:7,highwater=:8,audit$=:9,flags=:10 where  obj#=:1

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse    10000      0.32       0.30          0          0          0           0
Execute  10000      2.74       3.04          0      10000      20287       10000
Fetch        0      0.00       0.00          0          0          0           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total    20000      3.06       3.34          0      10000      20287       10000
```

Listing 1.9 TKPROF output for PL/SQL loop execution from one node.

Listing 1.10 prints tkprof output when the same test case is executed from both nodes. Notice the increase in waits for 'row cache lock' events. Total elapsed time also increased to 81 seconds and 62 seconds were spent waiting for 'row cache lock' event.

```
INSERT INTO T1 SELECT T1_SEQ.NEXTVAL, LPAD( :B1 , 500, 'x') FROM DUAL
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute  10000      8.02      81.23          0       1584      27191       10000
Fetch        0      0.00       0.00          0          0          0           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total    10001      8.02      81.23          0       1584      27191       10000


Elapsed times include waiting on following events:
  Event waited on                             Times   Max. Wait  Total Waited
  ----------------------------------------   Waited  ----------  ------------
  row cache lock                               5413        2.93         62.86
  gc current block 2-way                         63        0.16          0.41
  gc cr block 2-way                              46        0.00          0.06
```

Listing 1.10 Few lines from the trace file

These test results shows that sequences that are accessed heavily, in RAC environments, suffers from horrible performance issues if the sequence cache value is set to nocache.

If you need to have a sequence with no gaps, then from Oracle version 10g onwards, you can create the sequence with cache order. With sequence cache and order, RAC uses DFS messages to communicate latest sequence numbers without incurring additional overhead.

Myth 6: Small tables should not be indexed in RAC:

I think, this probably stems from the misunderstanding of RAC environments. Smaller tables tend to be cached in memory and with bigger SGA & cache fusion, much of the small table blocks will stay in cache. This somehow resulted in a myth that "small tables should not be indexed in RAC". Surprisingly, I heard this in a conference call with a senior DBA.

But, that isn't true. Small tables should be indexed too.

Following test case proves this. First, we crate a small table and insert 10K rows in to that table.

```
set timing on
drop table t_small2;
create table t_small2 (n1 number, v1 varchar2(10) )
 tablespace users
 ;
insert into t_small2 select n1, lpad(n1,10,'x')
from  (select level n1 from dual connect by level <=10001 );
commit;


select segment_name, sum(bytes)/1024 from dba_segments where
segment_name='T_SMALL2'
and owner='CBQT' group by segment_name
SQL> /
SEGMENT_NAME                                SUM(BYTES)/1024
------------------------------------------- ---------------
T_SMALL2                                                256
```

Listing 1.11 Test case for myth 6

T_SMALL2 is a smaller table with a size of 256KB. In the test case in listing 1.12, code accesses t_small2 table in a loop one hundred thousand times randomly. Trace is also enabled by setting event 10046 with level 8.

```
alter session set events '10046 trace name context forever , level 8';
set serveroutput on size 100000
declare
  v_n1 number;
  v_v1 varchar2(512);
  b_n1 number;
begin
 for i in 1 .. 100000 loop
   b_n1 := trunc(dbms_random.value (1,10000));
   select n1, v1 into v_n1, v_v1 from t_small2 where n1 =b_n1;
 end loop;
exception
  when no_data_found then
   dbms_output.put_line (b_n1);
end;
/
```

Listing 1.12 Test case for myth 6

Executing above test case, in two instances of RAC cluster, shows that it took 66 seconds to complete this PL/SQL code block.

```
SELECT N1, V1
FROM
 T_SMALL2 WHERE N1 =:B1
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute 100000      2.81       3.08          0          1          0           0
Fetch   100000     62.72      63.71          0    3100000          0      100000
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total   200001     65.54      66.79          0    3100001          0      100000

Rows     Row Source Operation
-------  ---------------------------------------------------
 100000  TABLE ACCESS FULL T_SMALL2 (cr=3100000 pr=0 pw=0 time=63391728 us)
```

Listing 1.13 TKPROF output of test case with out index

Adding an index to column n1 and repeating the test case in Listing 1.12 shows that script completed in 3.4 seconds.

```
REM adding an index and repeating test
create index t_small2_n1 on t_small2(n1);

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute 100000      1.64       1.61          0          2          0           0
Fetch   100000      1.79       1.78         23     300209          0      100000
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total   200001      3.43       3.40         23     300211          0      100000

Rows     Row Source Operation
-------  ---------------------------------------------------
 100000  TABLE ACCESS BY INDEX ROWID T_SMALL2 (cr=300209 pr=23 pw=0 time=1896719
us)
 100000   INDEX RANGE SCAN T_SMALL2_N1 (cr=200209 pr=23 pw=0 time=1109464
us)(object id 53783)
```

Listing 1.14 TKPROF output of test case with index

In summary, Consider indexing small tables, even if the SGA is quite big. Accessing through many rows is CPU intensive and costly.

Myth 6: Bitmap index performance is worse compared to single instance

Bitmap indices are suitable for mostly read only tables and works optimally for low cardinality data. Bitmap indices are not suitable for columns with excessive DML changes. But, there is a notion that bitmap indices and RAC does not mix and that is a myth.

SQL performance does not decrease just because bitmap index and RAC are mixed. In the Listing 1.14 we cselect from a table 10000 times using a random number generator.

```
create bitmap index t_large2_n4 on t_large2(n4);

alter session set events '10046 trace name context forever , level 8';
set serveroutput on size 100000
declare
```

```
  v_n1 number;
  v_v1 varchar2(512);
  b_n1 number;
begin
 for i in 1 .. 100000 loop
   b_n1 := trunc(dbms_random.value (1,10000));
   select count(*) into v_n1 from t_large2 where n4 =b_n1;
 end loop;
exception
  when no_data_found then
   dbms_output.put_line (b_n1);
end;
/
```

Listing 1.14 TKPROF output of test case with index

Test results for a single threaded execution in a RAC instance is depicted in the Listing 1.15.

```
SELECT COUNT(*) FROM T_LARGE2 WHERE N4 =:B1

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute 100000      2.87       2.93          2          2          0           0
Fetch   100000      1.86       2.03         78     200746          0      100000
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total   200001      4.73       4.97         80     200748          0      100000

Rows     Row Source Operation
-------  ---------------------------------------------------
 100000  SORT AGGREGATE (cr=200746 pr=78 pw=0 time=2854389 us)
 100000   BITMAP CONVERSION COUNT (cr=200746 pr=78 pw=0 time=1766444 us)
```

Listing 1.15 TKPROF output of test case with index

Now, lets' add what happens if we execute this from multiple nodes: Listing 1.16 shows the tkprof output from a node. This shows that there is nothing wrong in having a bitmap index in RAC environments.

```
SELECT COUNT(*)
FROM
 T_LARGE2 WHERE N4 =:B1

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.01          0          0          0           0
Execute 100000      2.82       2.95          0          2          0           0
Fetch   100000      1.90       1.94          3     200753          0      100000
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total   200001      4.73       4.90          3     200755          0      100000

Misses in library cache during parse: 1
```

Conclusion

We disproved various myths centered RAC and performance.

About the author

Riyaj Shamsudeen has 15+ years of experience in Oracle and 14+ years as an Oracle DBA/Oracle Applications DBA. He is the principal DBA behind ora!nternals

(http://www.orainternals.com - performance/recovery/RAC/EBS consulting company). He specializes in RAC, performance tuning and database internals and frequently blogs about them in http://orainternals.wordpress.com . He has authored many articles such as internals of locks, internals of hot backups, redo internals etc. He also teaches in community colleges in Dallas such as North lake college and El Centro College. He is a proud member of OakTable network. He co-authored the book "Expert Oracle Practices: Oracle database administration from Oak Table".

References

1. Oracle support site. Metalink.oracle.com. Various documents

2. Internal's guru Steve Adam's website
   www.ixora.com.au

3. Jonathan Lewis' website
   www.jlcomp.daemon.co.uk

4. Julian Dyke's website
   www.julian-dyke.com

5. 'Oracle8i Internal Services for Waits, Latches, Locks, and Memory'
   by Steve Adams

6. Tom Kyte's website
   Asktom.oracle.com

Appendix #1: Environment details

Linux/Solaris 10

Oracle version 10gR2 and 9iR2

RAC

Locally managed tablespaces

No ASM

No ASSM