How Many Slaves?

Parallel Execution and the 'Magic of 2'

Abstract and Scope

Several sources have suggested to me that the Magic of Two, as described by Cary Millsap, comes into play when setting the best degree of parallelism. This paper describes tests of various degrees of parallelism against different hardware configurations to investigate what factors should be considered when choosing an appropriate Degree of Parallelism (DOP). It does not provide a comprehensive introduction to the workings of parallel execution. That knowledge is assumed and is available in Oracle's online documentation; my previous paper on this subject; and other sources (see References).

Introduction

Oracle's Parallel Execution functionality (sometimes known as Parallel Query or simply parallelism), offers the possibility of improving the performance of long-running tasks by dividing the workload between multiple slave processes that can run concurrently and use multiple CPU servers effectively. This facility is increasingly attractive in modern business applications where data volumes are growing, but Parallel Execution (PX) can only work well for suitable workloads on suitable system configurations.

If you look at the various online Oracle forums you will find questions similar to the following.

- *"What degree of parallelism should I use for query y on equipment x?"*
- *"What is a sensible setting for parallel_max_servers?"*

The usual and most sensible answer is 'it depends'.

However, another answer that you'll often find is 'about two times the number of CPUs in the server'. This advice is often based, either directly or indirectly, on Cary Millsap's paper "Batch Queue Management and the Magic of '2' " (Millsap, 2000).

A couple of years ago I was working on a European-wide Data Warehouse project that depended on PX to deliver online user reports within agreed times and also made very intensive use of PX throughout the overnight batch load. That system used a parallel DEGREE setting of two on all objects for online use although the projected number of concurrent users was several hundred, with perhaps a maximum of 50 reports running concurrently. In practice, this could mean a few hundred concurrently active PX slave processes and parallel_max_slaves was set to 256. Applying a little common sense, I couldn't imagine how the eight CPUs in the server were ever going to be able to support 256 very active processes, so I spent a couple of months arguing the case for serial operation. Apparently the load tests had already been run, though, and had 'proved' that a DOP of two resulted in the best performance. (The fact that report creation times and overall system performance could be wildly unpredictable was a significant inconvenience, though)

The experience prompted a previous paper – 'Suck It Dry – Tuning Parallel Execution' (Burns, 2004). When I asked a few Oracle experts to help me out by reviewing that paper, a couple of comments kept bothering me

1. I suggested a range of 2-10 * no. of CPUs for parallel_max_servers. One reviewer questioned the wisdom of this. The problem was that I knew of at least one site that claimed to have successfully used 5 * no. of CPUs. The claim had come from someone I would usually trust and I hear that they're using 8 * no. of CPUs today because they've found that results in the best response times on their system. The DW that I'd worked on was also using substantially higher degrees of parallelism, albeit less successfully.

2. I ventured in that paper (with no proof) that if the I/O subsystem was particularly slow, then higher degrees of parallelism might be appropriate based on the assumption that while the CPUs were waiting on I/O, other work could be performed. Two reviewers noted this suggestion and commented along the lines of 'I know what you mean, but that could make things much worse'. In fact, introducing parallelism is likely to lead to an increase in I/O activity because Oracle could favour parallel full table scans.

The different points of view convinced me that I wanted to understand the problem better. Meanwhile, I presented the paper at the UKOUG Unix SIG and when the subject of a sensible number of slaves per CPU came up, someone in the front row mentioned something about Cary Millsap and 'Two' (or was it two Cary Millsaps? That seems unlikely, at least in the Oracle community). It was a quick in-and-out trip that day so I didn't have time to follow up on that comment, but it stuck with me. I noticed Tom Kyte mention the Power of Two in an asktom.oracle.com question and answer thread (see References), and I started to dig around. Eventually I came across Cary's paper – "Batch Queue Management and the Magic of '2'" and read it.

The Magic of Two

Although I have a long appreciation of Cary's work, I hadn't read this paper. In it, Cary explains in some detail why there is a limit to how many batch processes a single CPU in an SMP server can execute efficiently. Although the paper applies most specifically to Oracle Applications systems and to batch processes (which I'll come back to shortly), Cary describes the appropriate number of batch processes to run on a server as being between 1 and 2 times the number of CPUs. That goes some way to explaining this section of the Oracle 9.2 Reference Manual (I've emphasised the key lines in bold italics)

PARALLEL_THREADS_PER_CPU

Property	Description
Parameter type	Integer
Default value	***Operating system-dependent, usually 2***
Modifiable	ALTER SYSTEM
Range of values	Any nonzero number

PARALLEL_THREADS_PER_CPU specifies the default degree of parallelism for the instance and determines the parallel adaptive and load balancing algorithms. The parameter describes the number of parallel execution processes or threads that a CPU can handle during parallel execution.

The default is platform-dependent and is adequate in most cases

So Oracle will use (CPU * 2) by default. However, there's another interesting suggestion in the next section of the documentation.

You should decrease the value of this parameter if the machine appears to be overloaded when a representative parallel query is executed. You should increase the value if the system is I/O bound.

The argument is expanded in the Oracle 10.2 documentation.

PARALLEL_THREADS_PER_CPU enables you to adjust for hardware configurations with I/O subsystems that are slow relative to the CPU speed and for application workloads that perform few computations relative to the amount of data involved. If the system is neither CPU-bound nor I/O-bound, then the PARALLEL_THREADS_PER_CPU value should be increased. This increases the default DOP and allow better utilization of hardware resources. The default for

PARALLEL_THREADS_PER_CPU on most platforms is two. However, the default for machines with relatively slow I/O subsystems can be as high as eight.

Oracle's documentation implies that if a system is I/O bound, there might be benefits from increasing the number of PX slaves beyond CPU * 2, which was my original suggestion in the previous paper.

In fact if you read Cary's paper carefully, the CPU * 2 guideline is not an absolute, for several reasons

- The paper suggests a range of values from around 1 to 2 * CPUs. Looking at some work that Cary did recently for the Miracle Master Class in early 2006, his advice looks more like this :-
  - If jobs are CPU-intensive, the best number of batch jobs per CPU <2 (nearer to 1)
  - If jobs are I/O-intensive, the best number of batch jobs per CPU >2
  - If CPU and I/O request durations are exactly equal (rare), the best number is CPU * 2
- The paper discusses traditional batch jobs with a similar amount of CPU and I/O activity, so that when one job is performing I/O, the other is able to use the CPU for processing work. I suspect that in the simple case of a parallel full table scan, PX slaves are likely to be more I/O-intensive than CPU-intensive so that they'll spend most of their time waiting on I/O. However, if large Hash Joins, Group Bys and other CPU-intensive tasks are being performed by a second set of slaves, the balance will change.
- The paper doesn't say that things will always be slower when you run more than two jobs per CPU. Instead, it suggests that if you have success in doing so, it could be due to bad server configuration or an I/O-intensive application that could be tuned. Maybe PX slaves have the same characteristics and PX would benefit from having more than 2 * CPU slaves running?

Clearly there are different perspectives on the problem, so I wanted to test different degrees of parallelism on different hardware configurations to find out more about what the sensible limits are and what factors might govern them.

Test Server Configuration

I chose three different servers to run the tests on that are representative of a range of hardware configurations. Although none of the equipment is state-of-the-art, all of the servers are probably still in common use today with the exception of the PC platform (which I hope is not a common Oracle database platform!).

Test Server 1 – Intel Single-CPU PC

Tulip PC

- White Box Enterprise Linux 4 – Kernel 2.6.9
- 1 x 550MHz Pentium 3
- 768Mb RAM
- Single 20Gb IDE HDD

This is an old vanilla PC that I had kicking around and I wasn't expecting it to run parallel statements very well. Using Cary's paper as a guideline, the maximum degree of parallelism I should expect from this server is 2. In fact I expected that there might be no benefit at all when allowing for the Linux, Oracle and monitoring overhead and the single hard disk drive.

Test Server 2 – Intel SMP Server

Intel ISP4400 (SRKA4)

- White Box Enterprise Linux 4 – Kernel 2.6.9
- 4 x 700Mhz Pentium 3 Xeon

- 3.5Gb RAM
- 4 x Seagate Cheetah SCSI Ultra 160 in software RAID-0 (256 Kb stripe) configuration and a separate SCSI-U160 containing the O/S and Oracle software.

This was a top-of-the-line Intel server from around 2000 or 2001 that I bought on eBay for £200. Although the processors are slow and out of date, the overall server infrastructure is very impressive and the SCSI disk drives are only one generation old. Having tested various configurations of striped and non-striped I/O configurations, I settled on Software RAID-0 configured using mdadm.

Test Server 3 – Enterprise SMP Server

Sun E10000

- 12 x 400Mhz CPU
- 12 Gb RAM
- EMC Symmetrix 8730 via a Brocade SAN
- 5 x Hard Disk slices in RAID 1+0 (960 Kb stripe) configuration
- Solaris 8

This was definitely the top of the range of Sun servers when I remember using it for the first time around 1999, but is still in very common use despite being superceded by the E20K and E25K servers. It's an impressive server and the availability of the EMC storage (largely unused by other hosts) ensures this is very similar to the servers that many large business database systems run on.

Operating Systems

The choice of Operating System for the tests was easy. It needed to

- Be available on all platforms
- Have good scripting and monitoring tools
- Be free (as in beer)

Unix or Linux it was, then! Solaris 8 was installed on the E10K and was free by virtue of it being one of the servers at work that's in the middle of being redeployed as a development environment. I was lucky to have access to it at all and o/s upgrades were never going to be on the agenda. At least both Linux and Solaris have the same shell scripting languages and monitoring tools available. For the Intel servers, I chose White Box Linux but Red Hat Advanced Server, or any clone would do, as would SUSE Linux Enterprise Server.

In practice, I think you would find very similar results to me regardless of your operating system choice, as will become clear as we look at the results.

CPU Resource

On the multi-CPU servers I was keen to control the CPU resources available in each server environment for different tests. That way I could prove what benefits an individual CPU would give. If only that option were available in the workplace more often there would be fewer poor purchasing decisions!

Mike (the sysadmin) gave me sudo access to the psradm utility on the Sun E10K so that I could enable and disable CPUs whilst the system was online as follows

```
testhost:[oracle]:psrinfo
12      on-line   since 02/14/06 09:04:33
13      off-line  since 02/15/06 08:02:05
14      off-line  since 02/15/06 08:02:13
15      off-line  since 02/15/06 08:02:16
```

```
16      off-line  since 02/14/06 12:28:31
17      off-line  since 02/14/06 12:28:25
18      off-line  since 02/14/06 12:28:20
19      off-line  since 02/14/06 12:28:15
20      off-line  since 02/14/06 12:28:08
21      off-line  since 02/14/06 12:28:06
22      off-line  since 02/14/06 12:28:03
23      off-line  since 02/14/06 12:27:55
testhost:[oracle]:sudo psradm -n 13
testhost:[oracle]:psrinfo
12      on-line   since 02/14/06 09:04:33
13      on-line   since 02/17/06 11:27:51
14      off-line  since 02/15/06 08:02:13
15      off-line  since 02/15/06 08:02:16
16      off-line  since 02/14/06 12:28:31
17      off-line  since 02/14/06 12:28:25
18      off-line  since 02/14/06 12:28:20
19      off-line  since 02/14/06 12:28:15
20      off-line  since 02/14/06 12:28:08
21      off-line  since 02/14/06 12:28:06
22      off-line  since 02/14/06 12:28:03
23      off-line  since 02/14/06 12:27:55
```

On the ISP4400, I controlled the number of CPUs by editing the /etc/grub.conf file, changing the value of maxcpus and rebooting the server. To make sure that the correct number of CPUs were enabled, I checked /proc/cpuinfo

```
title White Box Enterprise Linux (2.6.9-5.0.5.ELsmp)
        root (hd0,0)
        kernel /vmlinuz-2.6.9-5.0.5.ELsmp ro root=LABEL=/ rhgb maxcpus=3
        initrd /initrd-2.6.9-5.0.5.ELsmp.img

[oracle@ISP4400 pfile]$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 10
model name      : Pentium III (Cascades)
stepping        : 1
cpu MHz         : 698.058
cache size      : 1024 KB
fdiv_bug        : no
hlt_bug         : no
f00f_bug        : no
coma_bug        : no
fpu             : yes
fpu_exception   : yes
cpuid level     : 2
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic mtrr pge mca cmov pat
pse36 mmx fxsr sse
bogomips        : 1376.25

processor       : 1
vendor_id       : GenuineIntel
cpu family      : 6
model           : 10
model name      : Pentium III (Cascades)
stepping        : 1
cpu MHz         : 698.058
cache size      : 1024 KB
fdiv_bug        : no
hlt_bug         : no
f00f_bug        : no
```

```
coma_bug         : no
fpu              : yes
fpu_exception    : yes
cpuid level      : 2
wp               : yes
flags            : fpu vme de pse tsc msr pae mce cx8 apic mtrr pge mca cmov pat
pse36 mmx fxsr sse
bogomips         : 1392.64

processor        : 2
vendor_id        : GenuineIntel
cpu family       : 6
model            : 10
model name       : Pentium III (Cascades)
stepping         : 1
cpu MHz          : 698.058
cache size       : 1024 KB
fdiv_bug         : no
hlt_bug          : no
f00f_bug         : no
coma_bug         : no
fpu              : yes
fpu_exception    : yes
cpuid level      : 2
wp               : yes
flags            : fpu vme de pse tsc msr pae mce cx8 apic mtrr pge mca cmov pat
pse36 mmx fxsr sse
bogomips         : 1392.64
```

I/O Resource

With Mike's assistance on the EMC and under my own steam at home, I tried several disk layouts based on four available Hard Disks in the ISP4400 and five available Hard Disk Slices in the EMC Symmetrix. The two main configurations were

- One discrete filesystem per disk, with file placement of online redo log and several data files on separate devices to spread the I/O load manually.
- One filesystem using all devices in a RAID-1+0 configuration (EMC) or RAID-0 configuration (ISP4400). (It's worth noting that the ISP4400 configuration would be useless for a real world application because it offers no redundancy.) The stripe width on the EMC is fixed at 960Kb and I used a stripe width of 256Kb on the ISP4400. Although I would have preferred to use the Intel RAID controller installed in the ISP4400, I had difficulties getting it to work so I settled for the software solution. In practice, the graphs show that this performed pretty well when compared to the much more expensive EMC equipment on this small number of devices.

The single large Striped (and in the case of the EMC, Mirrored) solution (SAME) gave the best performance and was convenient. With post-testing hindsight, it would be an interesting exercise to run the tests with a variety of different disk configurations as the I/O subsystem is often the bottleneck when using PX.

Oracle Configuration

The choice of Oracle version and configuration was trickier. I was tempted to stick with an older version of Oracle such as 9.2 which I have more experience with so that I would be less likely to run into unexpected problems. Parallel Execution is an area that Oracle is developing constantly with some significant changes from release to release. I'm also keen to write papers that reflect the average business environment, where I would suggest 9.2 far outstrips the number of Oracle 10g installations at the time of writing.

At the same time, I was tempted by Oracle 10.2's significant performance monitoring improvements and spent some time being impressed by the OEM Tuning pack (or is it Grid Control Tuning Pack?). It would also increase the lifetime of the paper if I picked the most up-to-date version and would increase my knowledge of 10G behaviour.

In the end, I decided to go with the most current version of the RDBMS server, 10.2.0.1, but limit myself to using features I needed and that others were likely to be able to use in their own environments.

Again, although the version of Oracle is likely to play a part in some of the results (particularly the multi-user tests), I expect that the results for most of the tests would be very similar on any of the current versions of Oracle.

Key Initialisation Parameters

As for the instance configuration, I played around with various ideas and refined it based on previous test results. Here is the init.ora file I planned to use on all of the servers.

```
aq_tm_processes=0
background_dump_dest='/home/oracle/product/10.2.0/db_1/admin/TEST1020/bdump'
compatible='10.2.0'
control_files='/u01/oradata/TEST1020/control02.ctl'
core_dump_dest='/home/oracle/product/10.2.0/db_1/admin/TEST1020/cdump'
db_block_size=8192
db_cache_size=1024000000
db_domain=''
db_file_multiblock_read_count=128
db_name='TEST1020'
fast_start_mttr_target=300
java_pool_size=0
job_queue_processes=0
large_pool_size=8388608
log_buffer=1048576
log_checkpoint_timeout=999999
open_cursors=300
parallel_max_servers=512
parallel_adaptive_multi_user=false
pga_aggregate_target=838860800
processes=800
query_rewrite_enabled='TRUE'
remote_login_passwordfile='NONE'
service_names='TEST1020'
shared_pool_reserved_size=10485760
shared_pool_size=512000000
sort_area_retained_size=4194304
sort_area_size=4194304
star_transformation_enabled='FALSE'
timed_statistics=TRUE
undo_management='AUTO'
undo_retention=600
undo_tablespace='UNDOTBS1'
user_dump_dest='/home/oracle/product/10.2.0/db_1/admin/TEST1020/udump'
```

In practice, the PC configuration had insufficient memory to support this configuration, so I changed the following parameters there. (As I'll explain later, I had to scale down the data volumes for the PC tests too.)

`db_cache_size=102400000`	**(10% of other servers)**
`pga_aggregate_target=200000000`	**(25% of other servers)**
`shared_pool_size=102400000`	**(20% of other servers)**

This is a modified version of an init.ora file produced by the Database Configuration Assistant. I decided that I would leave as many parameters at their default settings as possible, with a couple of notable exceptions.

- Increase parallel_max_servers beyond the default so that all of the test instances would be able to start up to 512 parallel slaves if requested. This was an important decision because it was entirely dependent on what I was trying to test. If I left parallel_max_slaves at the configuration-specific default, I would be able to show how Oracle limits the number of slave processes available to **`cpu_count * 20`**. However, what I'm trying to show here is

what would happen at the limits if you ignore Oracle's automatic tuning efforts!

- Set parallel_adaptive_multi_user to false, for the same reasons. i.e. To allow me to exceed Oracle's sensible restrictions!
- Increase db_file_multiblock_read_count to 128 and make sure db_block_size was 8K on all platforms to attempt to ensure consistency of read request size.

Here, then, are the PX-specific parameter values.

```
SQL> show parameters parallel

NAME                                 TYPE        VALUE
------------------------------------ ----------- ------------------------------
fast_start_parallel_rollback         string      LOW
parallel_adaptive_multi_user         boolean     FALSE
parallel_automatic_tuning            boolean     FALSE
parallel_execution_message_size      integer     2148
parallel_instance_group              string
parallel_max_servers                 integer     512
parallel_min_percent                 integer     0
parallel_min_servers                 integer     0
parallel_server                      boolean     FALSE
parallel_server_instances            integer     1
parallel_threads_per_cpu             integer     2
recovery_parallelism                 integer     0
```

There are a couple of points to note here

- The `parallel_automatic_tuning` parameter that many sites used on previous versions of Oracle has been deprecated. The reason is that Oracle has implemented much of what `parallel_automatic_tuning` was designed to achieve.
- I left the `parallel_threads_per_cpu` parameter set to 2. This was another decision governed by the philosophy of the tests. I could have increased the degree of parallelism by setting DEGREE on the test tables to DEFAULT and gradually increasing this value and restarting the instance. Instead, I decided to control the DOP more directly using optimizer hints and `parallel_max_servers`. That probably isn't the most sensible approach to take on anything other than an artificial test of this nature.

Test Loads and Scripts

Coming up with a suitable test of PX performance that would suit multiple environments threw up some interesting challenges.

- How to make the tests repeatable and reliable across a number of hardware platforms and possibly operating systems. I would hope that the tests could be reusable by other people on other systems and to be able to re-use them myself with different types of SQL statement.
- How to make the test workloads large enough to be valid but small enough to execute successfully on all platforms and in a reasonable amount of time. For example, my early attempts utilised a single 2 million row table that, by setting PCTFREE to 90, was stretched to use 2.8Gb of disk space. Although this test ran in several minutes on the two Intel platforms, it took a mere 7 seconds to execute on the E10K with EMC storage. When the performance was identical for a five-disk RAID-0 set and a single disk layout, I started to realise that all of the data was likely to be cached. Jonathan Lewis suggested the same when I asked him about the results. After discussion with the SAN admin, I discovered that there was a total of 24Gb of Cache RAM on the EMC so, even if shared with other servers, it was likely to be caching my 2.8Gb fully.
- What performance analysis data should I capture? I decided that I would need operating system statistics as well as Oracle event information to identify some of the underlying causes of performance problems. I would also include a number of diagnostic statements such as queries against v$pq_tqstat and would show the results of the queries so that I could confirm that the tests were functioning correctly.

Test Scripts

For convenience, commented versions of the test scripts are contained in Appendix A. Here is a summary of the scripts

setup.sql	Run once using a DBA account to create the test tablespace, user account and privileges
setup2.sql	Run using the TESTUSER account to create two large test tables – TEST_TAB1 and TEST_TAB2 of 8,192,000 rows each, resulting in two data segments of 10,716 Mb. These tables were used for the single user / large volume tests. **N.B. For the PC tests, I had to reduce the size of these tables to 1/8 of the size on the other servers, purely due to lack of disk space and unreasonably long run times.**
setup3.sql	Run using the TESTUSER account to create eight smaller tables – TEST_TAB[3-10] of 128,000 rows, resulting in ten data segments of 176 Mb. These tables were used for the multi-user / small volume tests **N.B. This script is very clunky and basic! It's a cut-and-paste job and would be better written using PL/SQL and Native Dynamic SQL to be more flexible.**
setup4.sql	This script is identical to setup3.sql but creates tables of 1,024,000 rows, resulting in ten data segments of 147 Mb by using PCTFREE 10 instead of PCTFREE 90. This means that fewer blocks are read in the early stages of query processing, but more data is passed between slaves, joined and sorted.
rolling.sh	Shell script to run two tests for a range of DOPs and CPU counts. • Full Table Scan of TEST_TAB1. • Hash Join of TEST_TAB1 and TEST_TAB2 with GROUP BY. **N.B. Although the choice of tests is specific for this paper, the scripts could be used for any type of SQL statement that generates PX activity by simply replacing the two SQL statements with your own choice.**
session.sh	Shell script to run the same type of Hash Join with GROUP BY as rolling.sh, but session.sh joins the larger test_tab1 to one of the eight smaller tables (i.e. a large table to small table join). However, for the second set of multi-user tests, I modified this script so it used test_tab3 in place

	of test_tab1. (i.e. a small table to small table join) The second joined table is one of the range of smaller tables, based on a 'session' parameter value passed into the script. The idea here is to be able to run multiple concurrent test sessions that don't just access the same data and produce correctly named output files for subsequent analysis.
Multi.sh	Shell script to accept *Number of CPUs* / *Number of Sessions* / *DOP* and execute session.sh *Number of Sessions* times to simulate multiple concurrent user activity.

Monitoring and Tracing

A key function of the shell scripts is that they should collect enough data to be able to analyse the underlying causes of different run times. The output files they generate include

CPU_(SESSION)_DOP.log	Contains the following • Query results and AUTOTRACE plan to confirm that the test is doing the right thing! • Query against V$PQ_TQSTAT as further confirmation • Timings (using set timing on)
CPU_(SESSION)_DOP.trc	Consolidated 10046 trace file assembled using the trcsess utility (see below)
CPU_(SESSION)_DOP.out	tkprof formatted version of the trace file
CPU_(SESSION)_DOP.sysstats	At Jonathan Lewis' suggestion, I gathered System Statistics during each test to check the values of MAXTHR and SLAVETHR. Although I decided against using them in the paper, the rolling.sh script gathers them.

Of course, this means that the scripts generate ***a lot*** of output (one of the reasons I haven't used the System Statistics) so I captured the overall timings into Excel by grep-ing through the *.log files, for example

```
[oracle@ISP4400 PX]$ grep Elapsed 2_*.log
2_10.log:Elapsed: 00:05:14.21
2_10.log:Elapsed: 00:10:39.54
2_11.log:Elapsed: 00:05:15.33
2_11.log:Elapsed: 00:10:37.48
2_128.log:Elapsed: 00:05:42.86
2_128.log:Elapsed: 00:12:20.07
2_12.log:Elapsed: 00:05:14.97
2_12.log:Elapsed: 00:10:41.68
2_16.log:Elapsed: 00:05:18.22
2_16.log:Elapsed: 00:10:42.55
...
```

Then I could identify unusual looking timings and delve deeper using the *.trc or *.out files. I would also regularly select trace files at random whilst waiting for other tests to complete and look through them to get a feel for how jobs were behaving during the different tests. When I identified an event of specific importance, I could look at the trend by grep-ing through the *.out files

```
[oracle@ISP4400 PX]$ grep "PX Deq Credit: send blkd" 2_*.out
2_10.out:  PX Deq Credit: send blkd                          18        0.01    0.03
2_10.out:  PX Deq Credit: send blkd                          18        0.01    0.03
2_11.out:  PX Deq Credit: send blkd                         107        0.83    6.49
2_11.out:  PX Deq Credit: send blkd                         107        0.83    6.49
2_128.out:  PX Deq Credit: send blkd                        1338        0.84 19.66
2_128.out:  PX Deq Credit: send blkd                          39        0.04    0.13
```

```
2_12.out:  PX Deq Credit: send blkd                          31         0.01   0.07
2_12.out:  PX Deq Credit: send blkd                          31         0.01   0.07
2_16.out:  PX Deq Credit: send blkd                          41         0.02   0.29
2_16.out:  PX Deq Credit: send blkd                           1         0.00   0.00
...
```

I'm sure this process would have been much easier, however, had I used something like the Hotsos Profiler or orasrp.

ORCA

Early in the tests, I realised that the server's available hardware resources were the principle factor governing run times. Although this wasn't surprising, I wanted to be able to show this. Initially I ran sar or vmstat in parallel with the test scripts and tried to marry the results in Excel. This was a time-consuming task and interpreting the resulting data was likely to prove overwhelming when presenting it to others.

Fortunately Mike informed me that he had configured Orca on the Sun E10K server, which is a site standard where I work. It offered a couple of things I'd been looking for

- Easy correlation between the individual test phases and the overall server performance
- Pretty graphs for a presentation ;-) without too much additional work.

Of course I had to be careful that the 'big picture' that the graphs represented didn't hide small-scale detail but dipping into the large number of output files produced by the scripts allowed me to sanity-check what the graphs were showing.

Next, the hunt was on for the Linux equivalent. Orca is a tool that reads formatted text files and produces .PNG graphics and HTML files that can be served via HTTP. The tool that actually reads the system data and collates it for Orca's use is called orcallator on Solaris. The Linux equivalent that reads the /proc filesystem is procallator. The version of Orca that I ended up using on the two Intel boxes is 0.28, which includes procallator in the distribution, and I have to say I was pretty impressed by it although, as you'll see, the Solaris and Linux versions have a slightly different look to each other. See References for further information.

Oracle Event 10046 Tracing

I gathered event 10046 trace information on every job. In my previous paper, I highlighted the difficulty of tracing parallel execution jobs because Oracle generates a trace file for the Query Co-ordinator and each of the slaves. If I was running jobs at a DOP of 128, or 10 sessions at a DOP of 4, that was going to be a lot of trace files to wade through!

Fortunately this has become somewhat easier since Oracle 10g introduced the trcsess utility that can generate a consolidated trace file. The general approach I used was

Enable tracing and set tracefile and session identifiers so that Oracle can identify the relevant trace information

```
exec dbms_session.set_identifier('${CPU_COUNT}_${SESSION}_${DOP}');
alter session set tracefile_identifier='${CPU_COUNT}_${SESSION}_${DOP}';
exec dbms_monitor.client_id_trace_enable(client_id => '${CPU_COUNT}_${SESSION}_$
{DOP}');
```

Do work in here, then generate the consolidated trace file for the given client ID, looking at all of the trace files in user_dump_dest and background_dump_dest. Note that I could have been more specific about the trace file names, but I cleared them out between executions of this procedure.

```
trcsess output="${CPU_COUNT}_${SESSION}_$DOP.trc" clientid="${CPU_COUNT}_${SESSI
ON}_$DOP" /home/oracle/product/10.2.0/db_1/admin/TEST1020/udump/test*.trc /home/
oracle/product/10.2.0/db_1/admin/TEST1020/bdump/test*.trc
```

Generate tkprof version of consolidated trace file with the key statements (from my point of view) at the start of the output.

```
tkprof ${CPU_COUNT}_${SESSION}_$DOP.trc ${CPU_COUNT}_${SESSION}_$DOP.out sort=pr
sela,fchela,exeela
```

Note that I used the tkprof output throughout this paper rather than the raw trace files. I could have examined the raw trace files if necessary but decided against this, given the volume of data that I had to analyse.

Test Results – Volume Tests (setup2.sql and rolling.sh scripts)

The Volume Tests consisted of a single session performing a Full Table Scan of an 11Gb table, followed by a Hash Join and Group By of two 11Gb tables. The test steps are in rolling.sh. After a lot of experimentation, I settled on testing the following range of DOPs – 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 16, 24, 32, 48, 64, 96 and 128. This provided a balance between enough detail in the data to identify smaller differences at lower DOPs (likely to be the most useful in practice), whilst having some high DOPs thrown into the mix to see the effect of large-scale parallel operations. Note that you can change the range of DOPs and test every DOP between 1 and 200, but you'll be waiting a long time for the results!

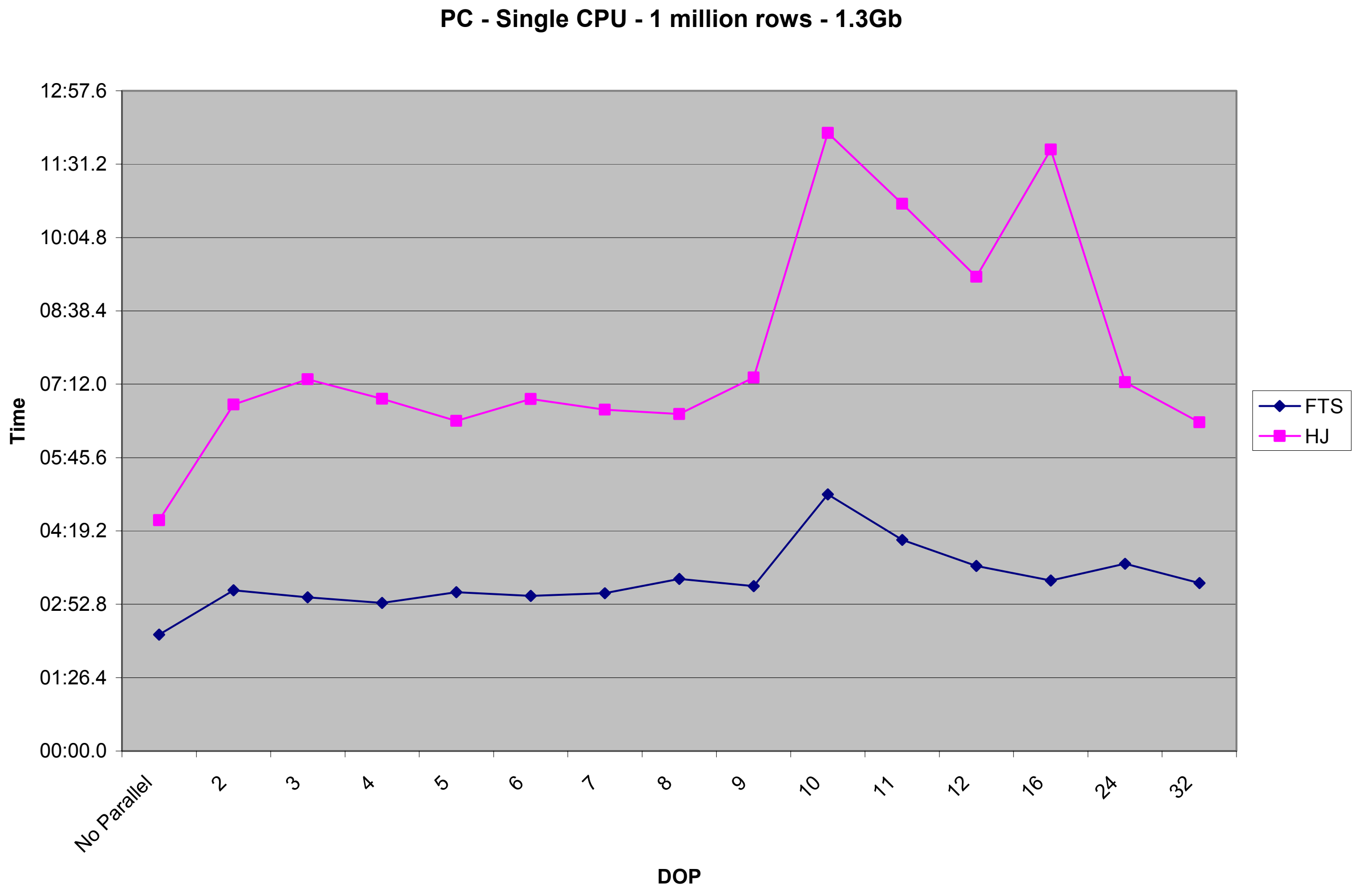
Conclusion – The Timings

Let's start with the conclusion. Before looking at some of the reasons behind the different run times, here they are. Note that the tests were run many times while I was fine-tuning the scripts and interpreting performance data and the run times barely changed, so I'm confident that there are no significant one-off anomalies in the results.

PC

My first conclusion is that running large-scale parallel execution tests on an old single PIII 550 with one IDE disk available is like beating yourself with a stick – fundamentally pointless and slightly painful.

Graph 1 – Single CPU PC / Reduced SGA and Data Sizes / Full Table Scan and Hash Join



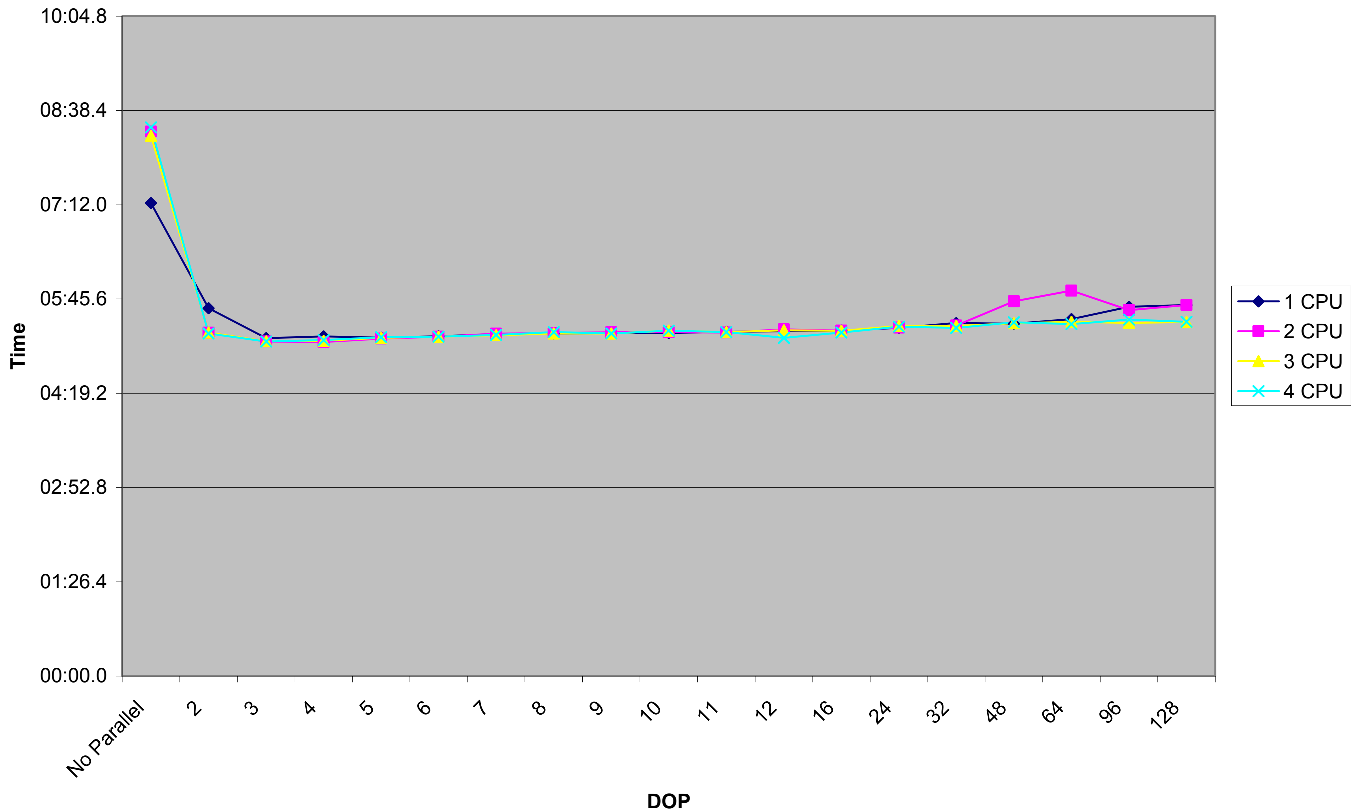
I think the main points to note are :-

- For both the Hash Join (which, remember, will use two slave sets and so twice as many slaves) and the Full Table Scan, the best results are achieved without using PX at all (i.e. the DOP is 1).
- Considering that the data volumes are 1/8 of the volumes used on the other servers, the run times are much longer. This isn't surprising with an older, slower CPU, less available memory and a single IDE disk drive.
- Why are there spikes around a DOPs of 10 and 11? Shocking though it might seem, probably because I happened to be doing a few directory listings to monitor progress. To give the human perspective, the server becomes unusable soon after the parallelism begins.
- The Hash Join example started to fail at a DOP of 64 due to lack of memory, so I gave it up as a lost cause

ISP4400

Graph 2 – ISP4400 / 1-4 CPUs /Full Table Scan

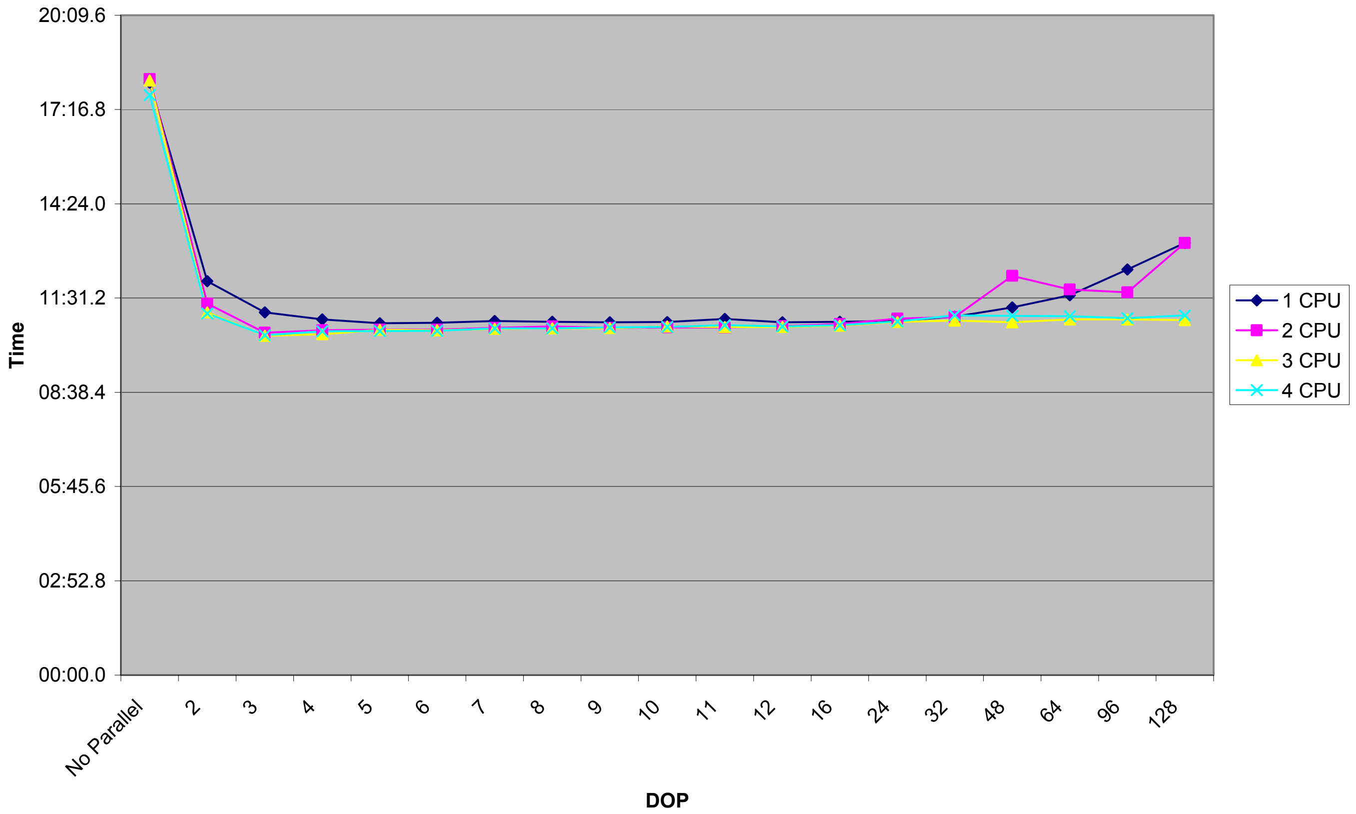
For the SMP graphs, I was initially tempted to change the Y axis, to emphasise the difference in test completion times based on the number of CPUs and the DOP. Then I decided that, rather than trying to show something 'significant' it would be better to show just how *little* benefit PX provided.



I think the main points here are :-

- There is a very significant benefit to be had from moving from non-parallel execution to a DOP of two, even when using a single CPU.
- There isn't much benefit to having more CPUs available. For example, if I had a single CPU server with this disk configuration, how much benefit would I get from paying for and installing additional CPUs to increase performance?
- When hitting DOPs of 128, I was expecting a more serious performance degradation on four CPUs, never mind a single CPU. i.e. I expected the timings to curve back up at higher DOPs. Jonathan Lewis gave me a nudge in the right direction at this point, and I'll discuss this more shortly.

Graph 3 – ISP4400 / 1-4 CPUs / Hash Join with Group By

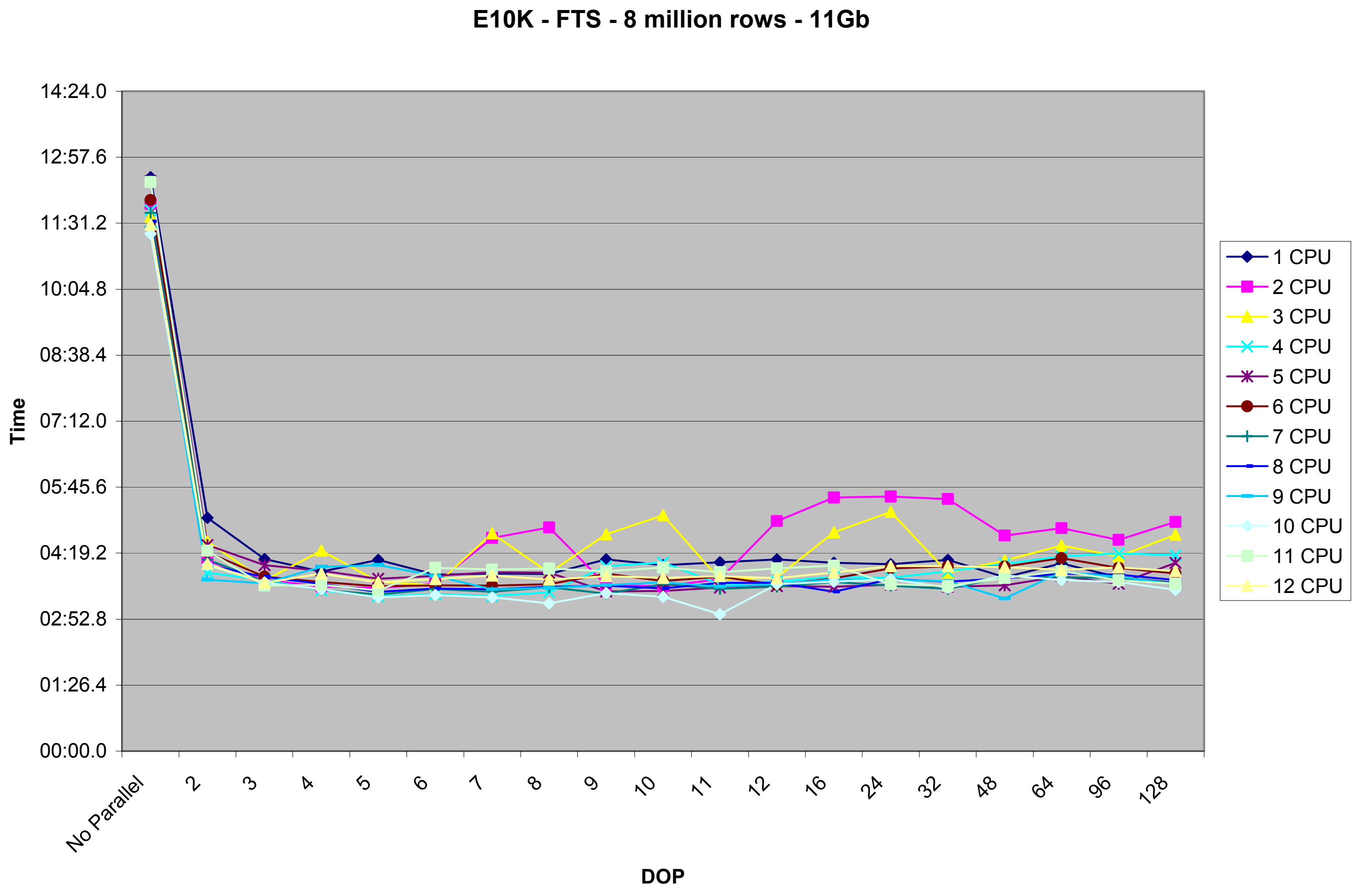


I think the main difference here is :-

- The first evidence of a performance break-down in the 1 CPU and 2 CPU timings, above a DOP of 32. I'd put this down to a couple of possible reasons – a) Double the number of slaves from the Full Table Scan example; and b) This operation is a little more CPU-intensive (that's the nudge Jonathan gave me)

E10K

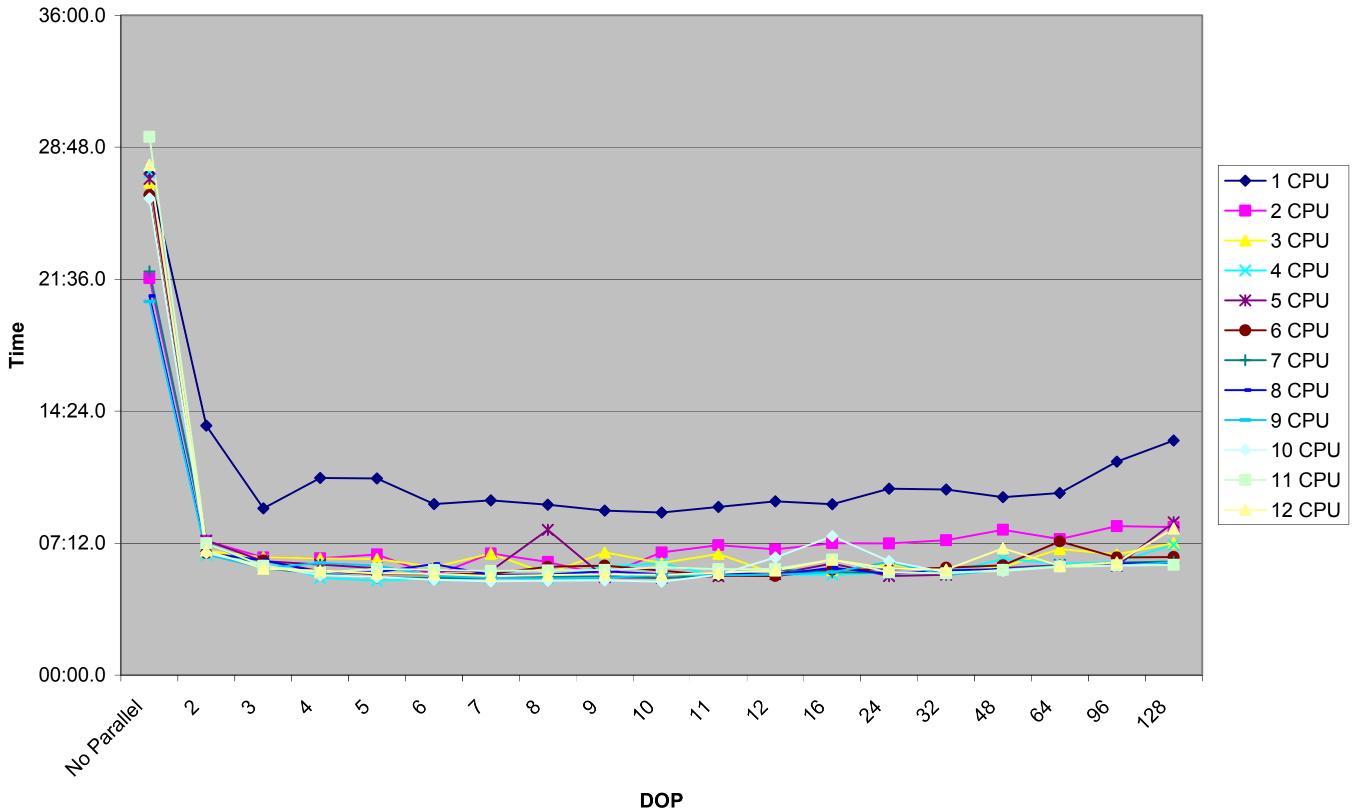
Graph 4 – E10K / 1-12 CPUs / Full Table Scan



A few points here :-

- The performance improvement moving from non-parallel to a DOP of two is more marked than on the ISP4400, perhaps indicating that the EMC storage is capable of and looking to be pushed harder?
- The stand-out point to me remains that the benefits of moving from a DOP of two to a DOP of 128 are very minor.
- There are some benefits of moving beyond a DOP of two – the best results were achieved with DOPs of about 4-12, but there's very little benefit for the extra resources that are consumed.

Graph 5 – E10K / 1-12 CPUs Hash Join with Group By



I see the main differences as being :-

- With double the number of slaves (two slave sets) and a more CPU-intensive plan, the additional CPUs become more beneficial. 1, 2, and even 3 CPUs are noticeably slower. Whether there is any great benefit from any of CPUs 4 to 12 is debatable.
- Once again, the more CPU-intensive and complex plan shows a degradation that begins above a DOP of about 10 and becomes progressively worse, particularly with a single or two CPUs.
- It's worth pointing out an immediately noticeable exception to the two-per-CPU guideline. The shortest execution time with a single CPU is at a DOP of three. As there are two slave sets in use, that means (2 * 3) + 1 = 7 processes.

In summary, with these particular test scripts on these particular server configurations, there's a noticeable benefit using PX at a degree of 2, but very little additional benefit from increasing it further. Then again, the performance of the jobs doesn't get much worse as the DOP increases, so why worry about using higher DOPs? The operating system statistics highlight some of the problems.

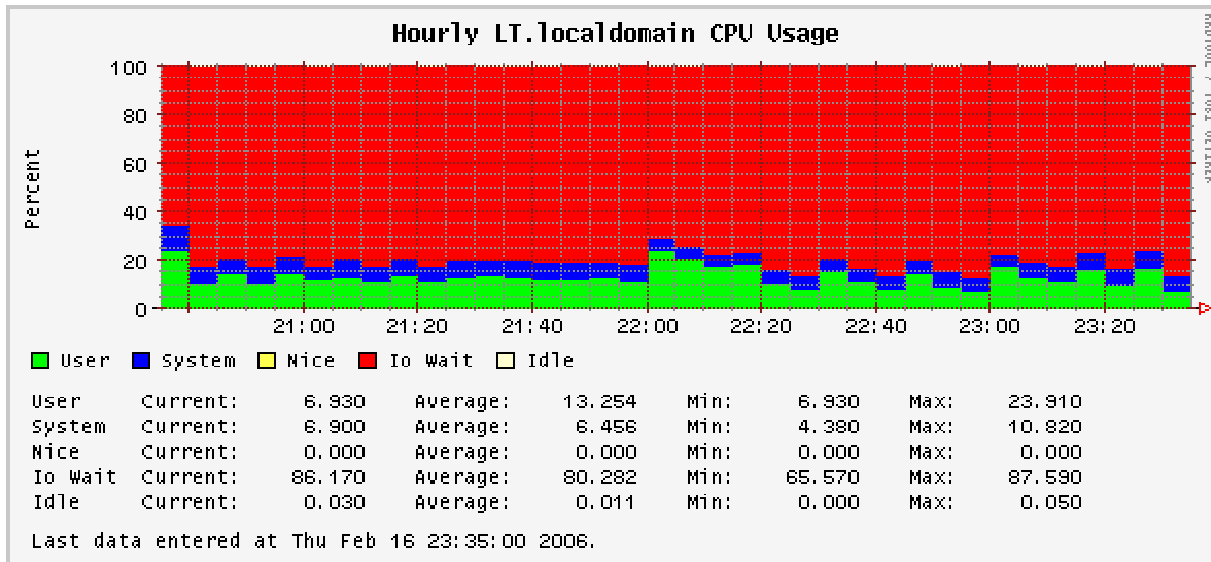
Operating System Statistics

PC

Note that, for the PC tests only, I used the Hourly option on ORCA to see the detail I needed as I was only running a

single CPU test and even that wasn't successful for long. The interval is a short 5 minutes per grid line here and the number of CPUs for the entire period was 1. Contrast this with the later graphs for the SMP servers.

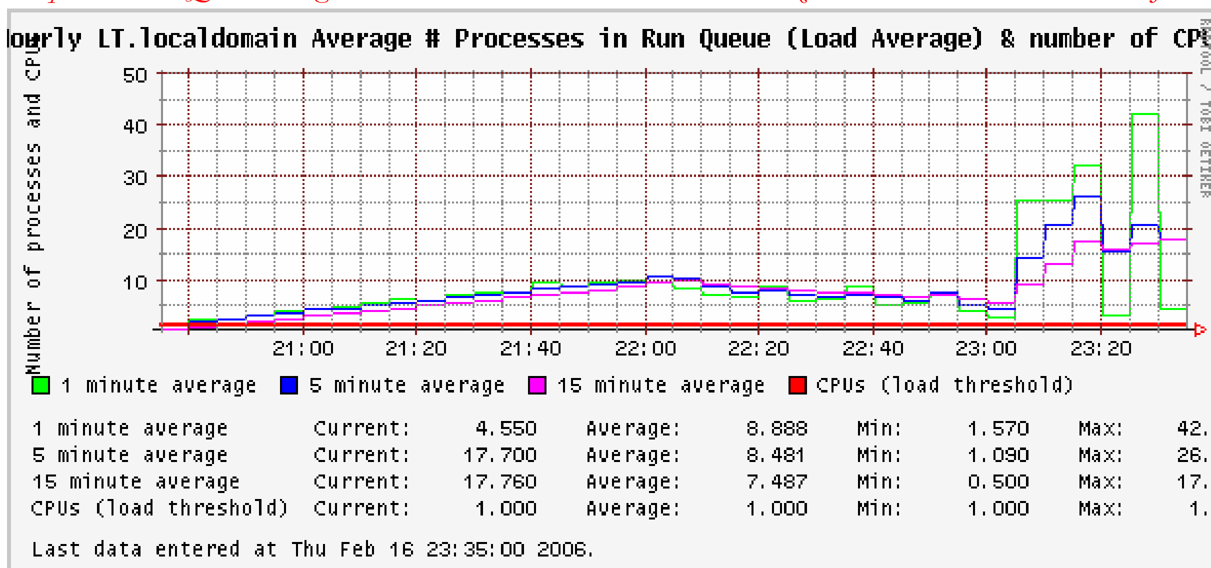
Graph 6 – CPU Usage - Single CPU PC / Reduced SGA and Data Sizes / Full Table Scan and Hash Join



This graph covers a range of DOPs from 1 – 32, after which the statistics didn't look any better. So it covers the same workload as Graph 1, which shows the timings. I think the main points here are :-

- 100% CPU usage throughout the test, albeit most of the time spent waiting on I/O

Graph 7 – Run Queue - Single CPU PC / Reduced SGA and Data Sizes / Full Table Scan and Hash Join

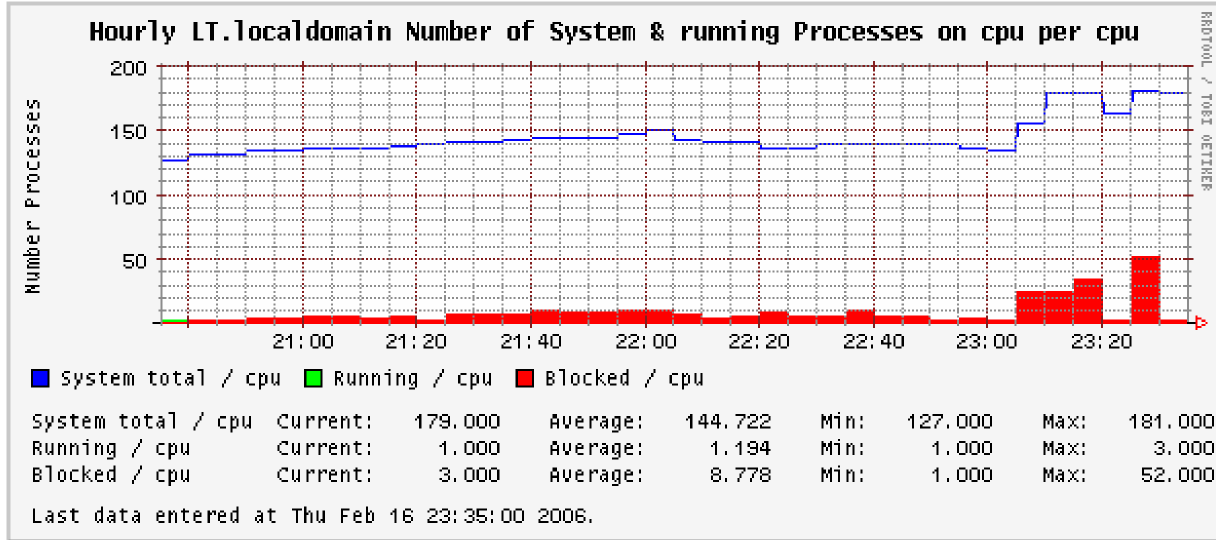


This graph is far more interesting to me because it's a better reflection of the problems you are likely to encounter running PX on a single slow CPU. Whilst Graph 6 shows the CPU usage at 100% throughout the test period, what it fails to show is the negative impact on other users as the DOP increases!

Note how the run queue becomes much worse around the 11pm mark. At this stage, the DOP is increasing from 12 to 16,

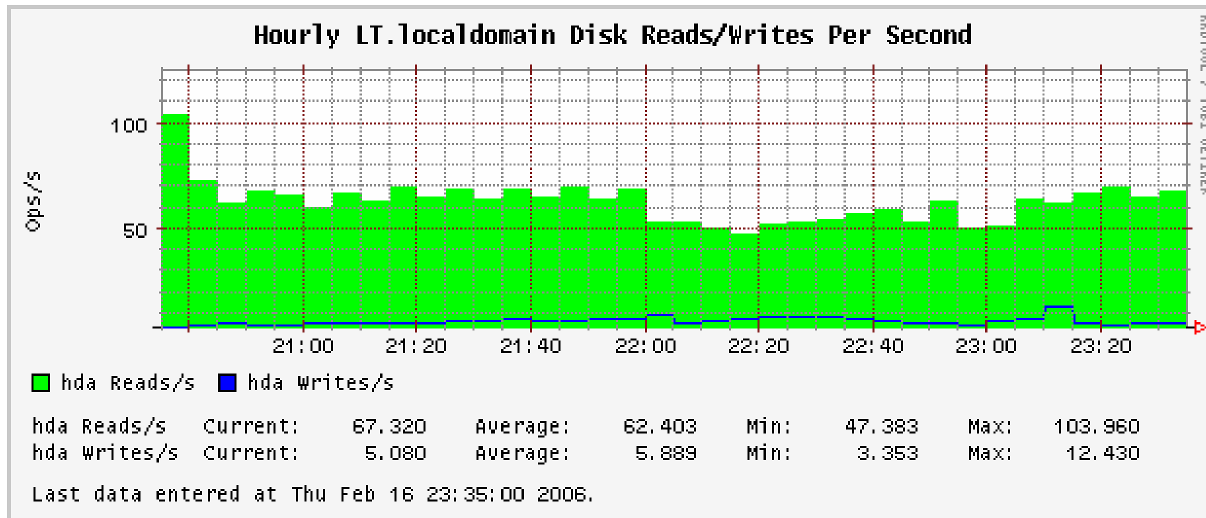
to 24 and 32. The user experience of this is that the server becomes unusable, even when trying to get a simple shell prompt. This is the experience I wanted to highlight in my previous paper – the 'Suck It Dry' moment when you've brought a server to it's knees. In fact, the server started to become unusable at a much earlier stage – around about the parallel 3 or 4 point.

Graph 8 – Total, Blocked and Running Processes - Single CPU PC / Reduced SGA and Data Sizes / Full Table Scan and Hash Join

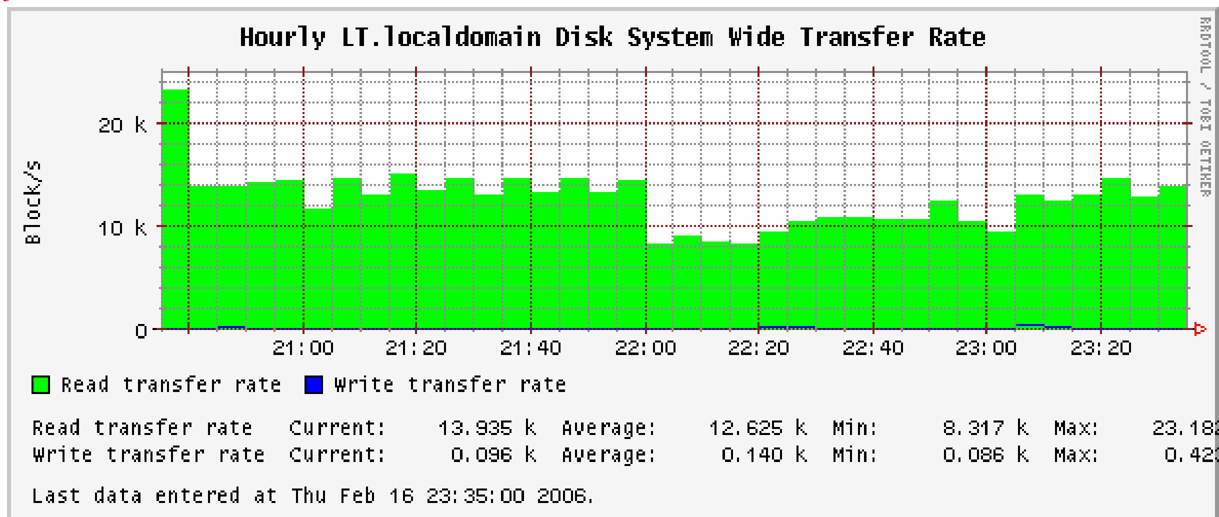


Like Graph 7, this graph shows more of the underlying detail of what's happening while CPU usage is at 100%. All that happens as the DOP increases is that more processes become blocked on each CPU and aren't really serving any useful purpose.

Graph 9 – No. of disk ops per second- Single CPU PC / Reduced SGA and Data Sizes / Full Table Scan and Hash Join



Graph 10 –Disk Transfer Rate (4k blocks) - Single CPU PC / Reduced SGA and Data Sizes / Full Table Scan and Hash Join



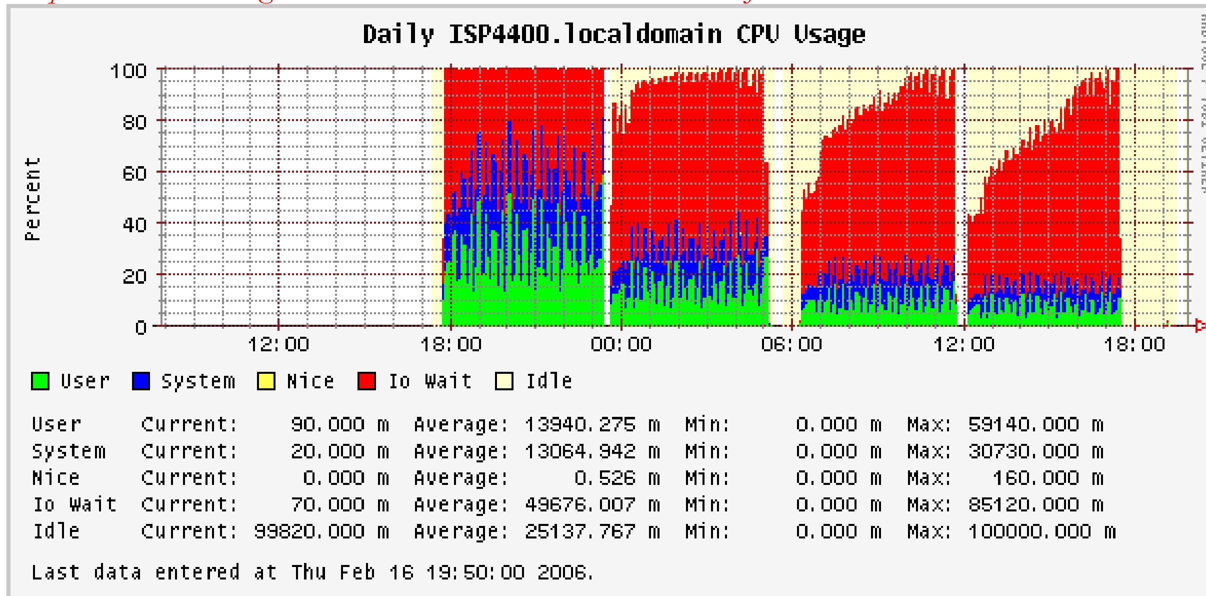
If I had to identify a single limiting factor on all of the volume tests I ran, on all of the different servers, it would be disk I/O. Even when the SQL statements are executing without using PX, they are all reading substantial volumes of data and if the I/O infrastructure isn't capable of reading data quickly enough, adding CPUs or increasing the DOP will achieve nothing beyond making the system unusable. In fact, as this and some of the other graphs show, higher degrees of parallelism can actually decrease I/O throughput a little, presumably due to contention.

ISP4400

For all of the SMP server tests, the graphs use the Orca Daily reports option and show the tests across the entire range of CPUs so that the results can be compared. Note also that these tests are using the full data volumes, rather than the reduced volumes used in the PC tests.

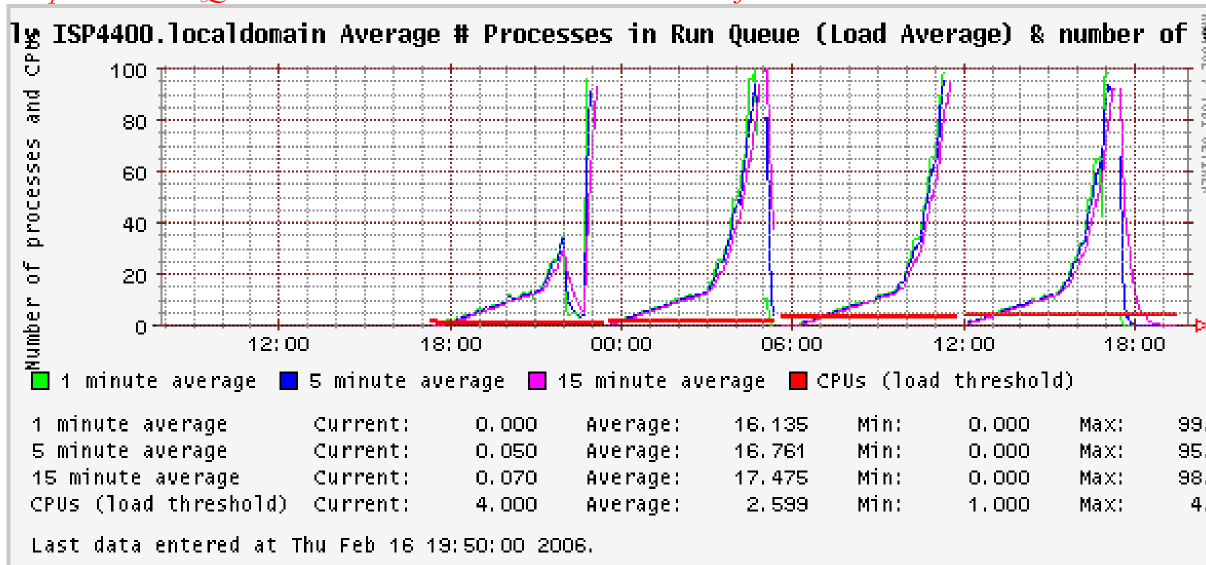
For the ISP4400 graphs, the order of the tests is from a single CPU to all four CPUs.

Graph 11 – CPU Usage – ISP4400 / Full Table Scan and Hash Join



Although this graph looks much as we might expect, it's worth highlighting that even the small amount of CPU idle time with 2, 3 and 4 CPUs is likely to prove extremely important in making the server usable by other processes.

Graph 12 – Run Queue – ISP4400 / Full Table Scan and Hash Join

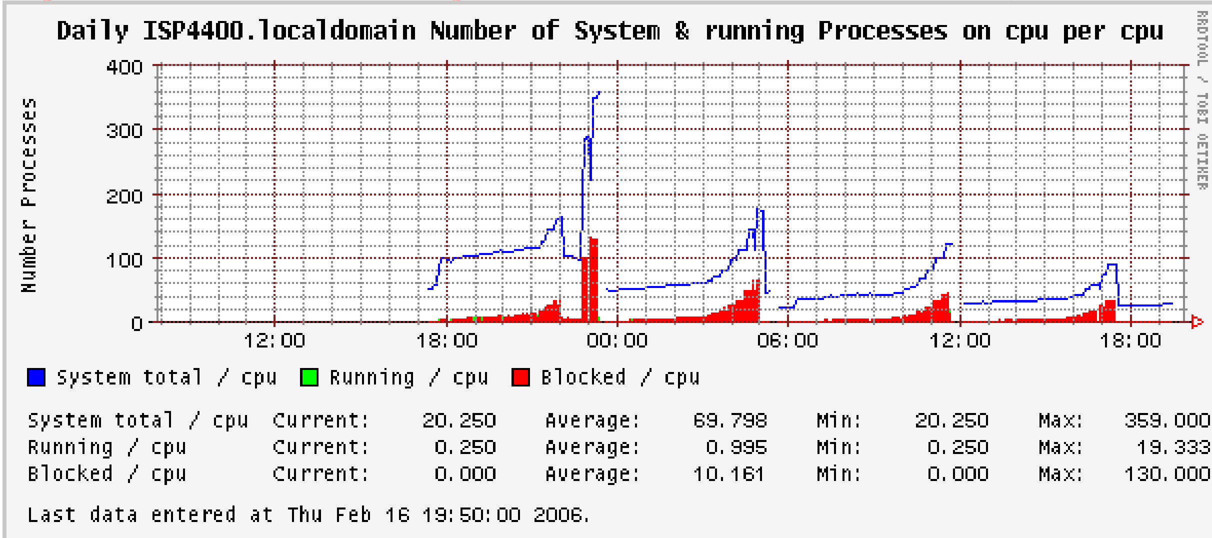


That red line at the bottom of the graphs is the number of CPUs available. The other lines are the processes in the run queue. To give you a flavour of what this means in practice, once the run queue starts to exceed a couple of times the number of CPUs available, the simplest of operations such as navigating through files becomes noticeably slower.

N.B. In many of the graphs for the ISP4400, there's a strange period between 22:00 and 23:00 when the server appears to do nothing! The server can't have rebooted because the tests wouldn't have restarted automatically from where they had left off and when I examined the trace files there was no obvious period of inactivity. My best guess at what the problem was is a statistics collection problem with procallator, particularly bearing in mind that I stopped and restarted it around that time. However, this period shouldn't be confused with the

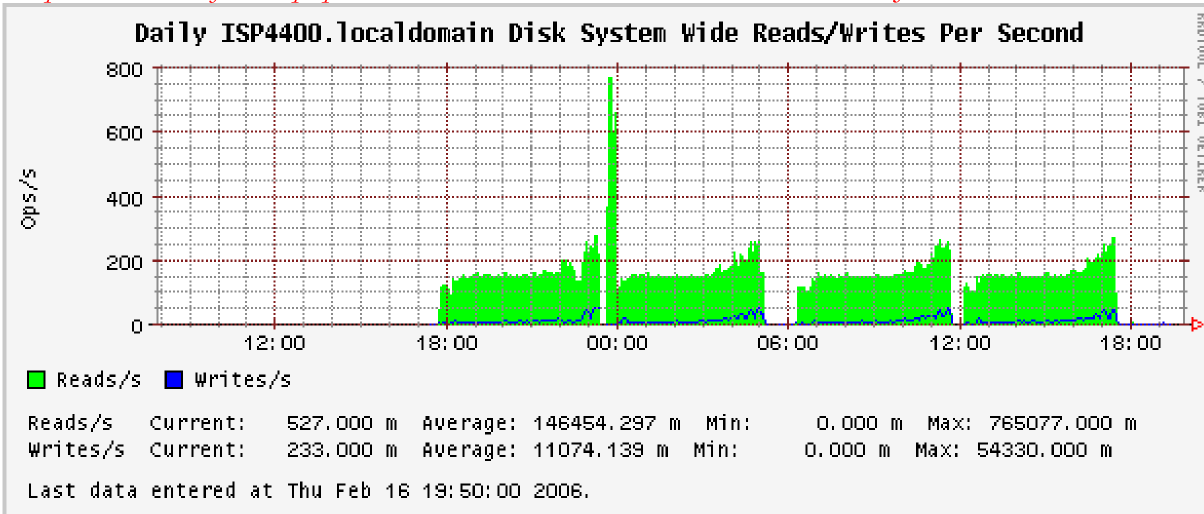
period soon afterwards when the server *was* rebooted to start the 2-CPU test.

Graph 13 – Total, Blocked and Running processes – ISP4400 / Full Table Scan and Hash Join



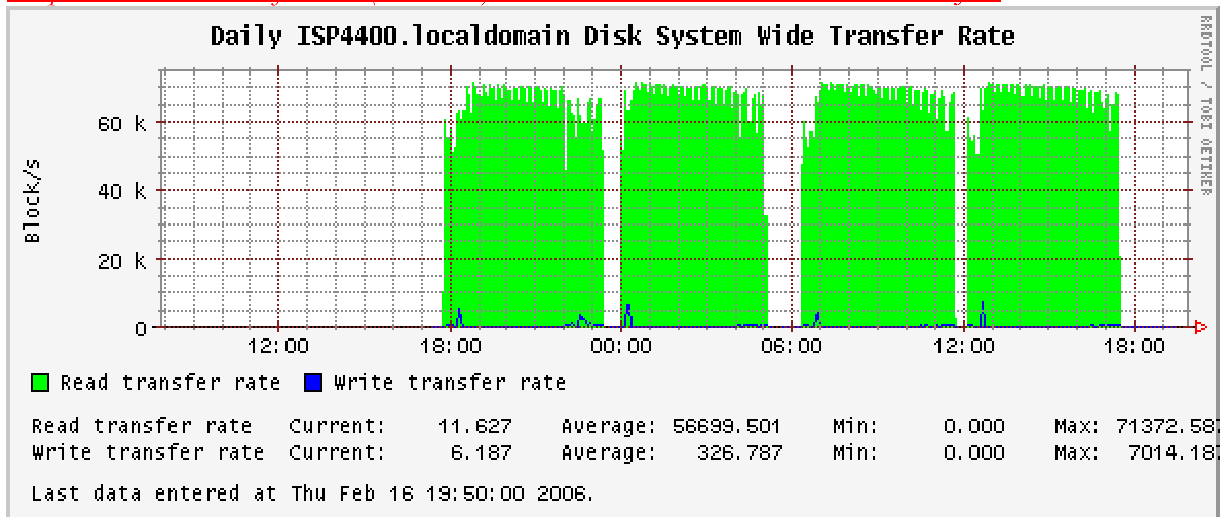
Note the number of blocked processes shown in red, which shows the scale of the bottleneck introduced when trying to run too many processes per CPU. The big improvement comes with the addition of the second CPU.

Graph 14 – No. of disk ops per second – ISP4400 / Full Table Scan and Hash Join



(Note that the spike in this graph is the activity on the system disk., unrelated to the tests.)

Graph 15 – Disk Transfer Rate (4k blocks) – ISP4400 / Full Table Scan and Hash Join

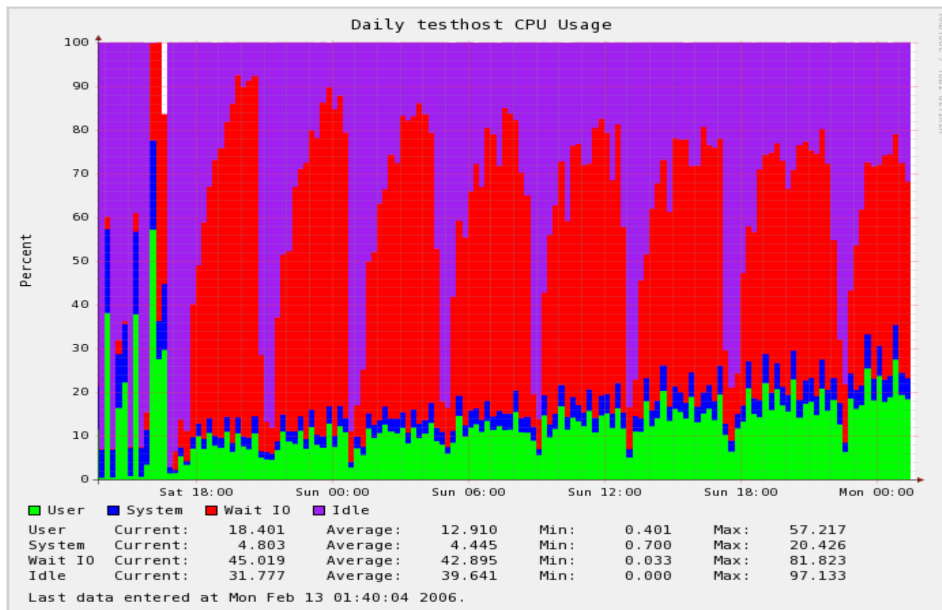


Hopefully it's clear from the previous two graphs that, above the lower DOPs of 1, 2 and maybe 3, the disks simply can't supply data quickly enough to meet the PX slave's demands. If that's the case, then they are wasting CPU and memory for no real benefit.

Sun E10K

With 12 CPUs available, the graphs for the E10K spread over two images. (I could have tested on 1, 2, 4, 6 etc. CPUs but, as the server was sitting there and the tests run happily unattended; I thought I'd test all 12 possibilities.) What is most important to note is that the E10K graphs work show the tests running from 12 CPUs down to 1. i.e. The further to the right on the graph, the fewer the CPUs. (This is the opposite of the ISP4400 graphs.)

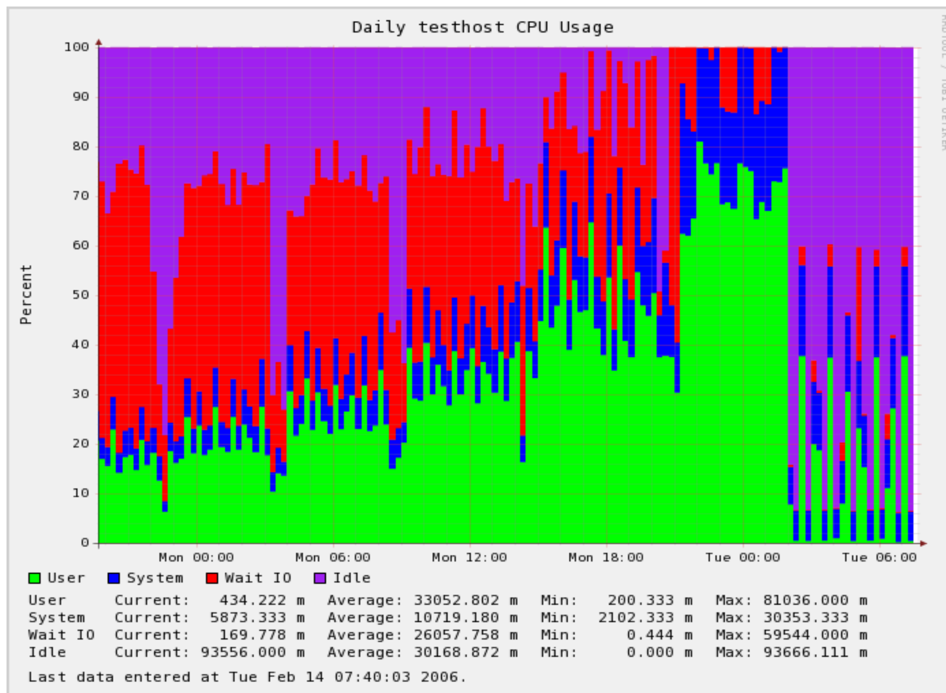
Graph 16 – CPU Usage – Sun E10K – CPUs 12-6 / Full Table Scan and Hash Join



Each spike in this graph represents the execution of the test against a given number of CPUs for all of the DOPs in rolling.sh. Ignoring the far left of the graph (which shows the activity as I was creating the test data), it shows 12-6 CPUs and most of 5.

As you can see, the vast majority of time is spent waiting on I/O, despite the EMC hardware being used. A combination of 12 CPUs running PX jobs is more than the I/O subsystem can cope with, with the result that there's always some idle time, and very little User or System CPU time, although those increase proportionally as the number of CPUs decreases and there's less of a mismatch between the CPU and I/O resources. A noticeable result of the spare CPU resource on this server was that I never really experienced the server slowing down dramatically.

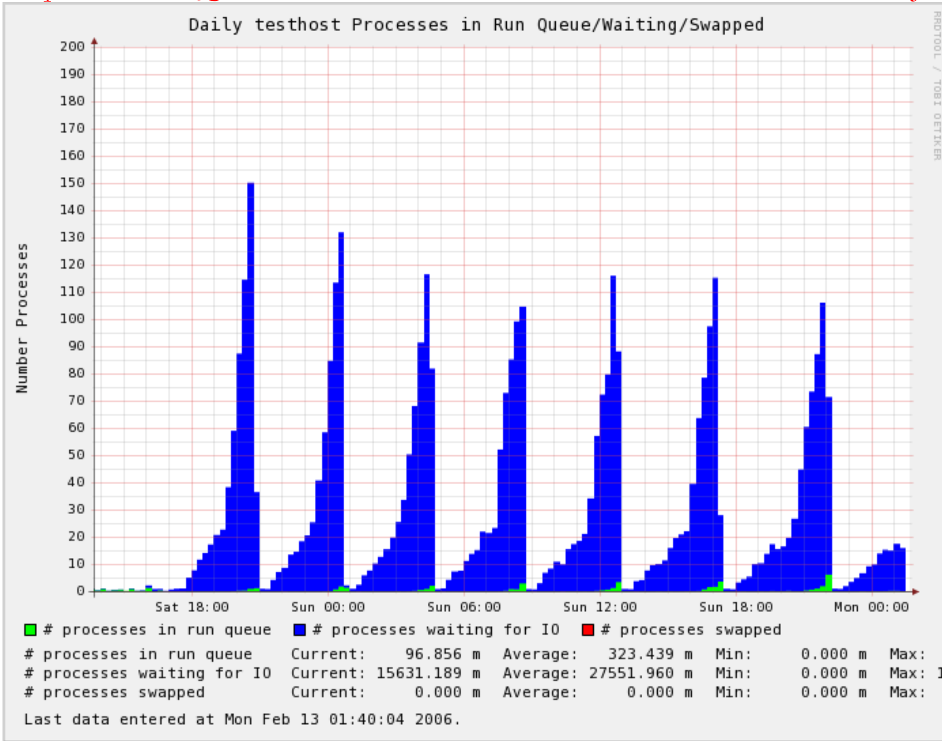
Graph 17 – CPU Usage – Sun E10K – CPUs 5-1 / Full Table Scan and Hash Join



The second part of the E10K CPU usage is much more interesting as the proportion of time spent waiting on I/O decreases and the amount of user and system CPU increases. Particularly interesting is how things start to look when only one CPU is available and System CPU usage increases quite dramatically and there's very little idle time.

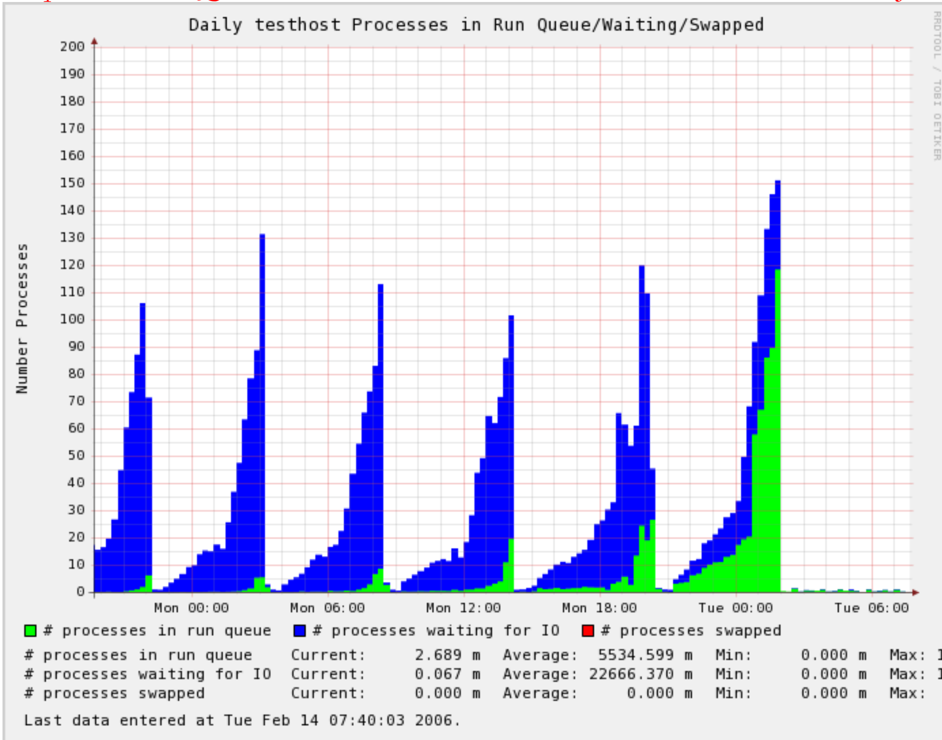
Note also that the spike for each individual test run becomes wider as the jobs take longer to run.

Graph 18 – Run Queue – Sun E10K – CPUs 12-6 / Full Table Scan and Hash Join



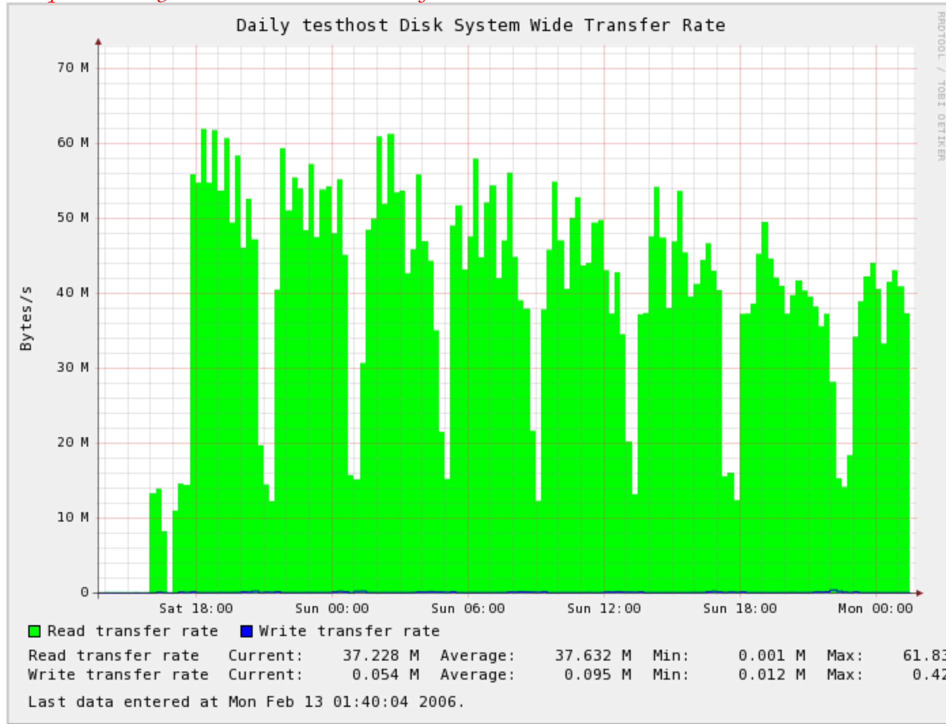
The Run Queue data reflects the CPU usage. The biggest component is the large number of processes blocked waiting on I/O, which decreases as the number of CPUs decreases. However, the processes start to become blocked in the run queue waiting for a CPU, as can be seen from the small but growing green 'bump' at higher DOPs.

Graph 19 – Run Queue – Sun E10K – CPUs 5-1 / Full Table Scan and Hash Join

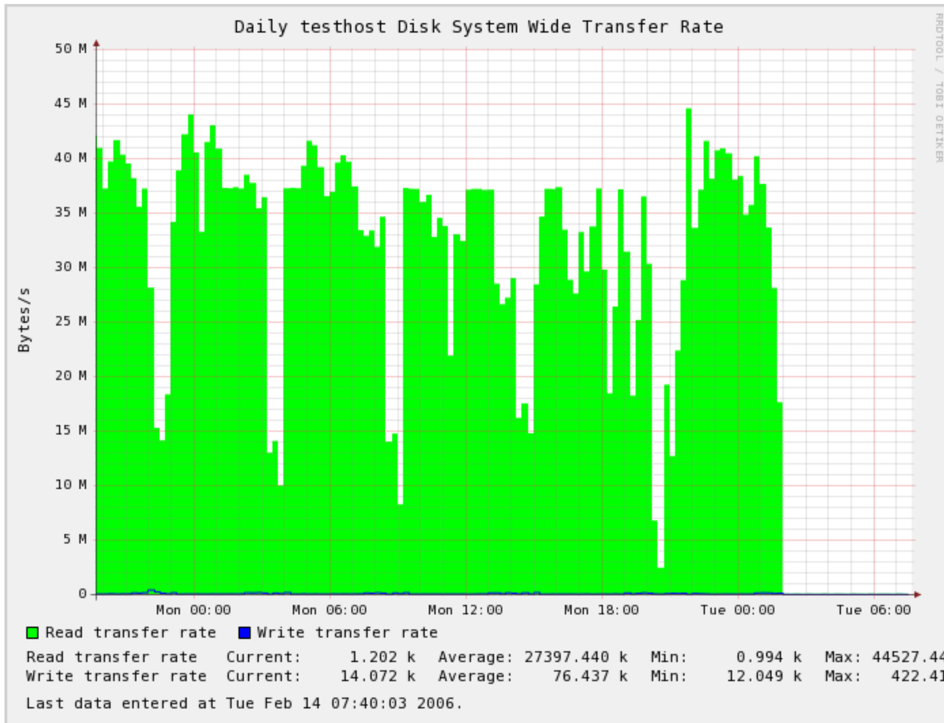


As the number of CPUs decrease, the number of processes blocked on I/O also drops whilst the number of processes in the run queue continue to increase because the system is now increasingly CPU-bound, particularly when a very small number of processors are available.

Graph 20 – System Wide Disk Transfer Rate – Sun E10K – CPUs 12-6 / Full Table Scan and Hash Join



Graph 21 – System Wide Disk Transfer Rate – Sun E10K – CPUs 5-1 / Full Table Scan and Hash Join



Unlike the graphs on the other servers, higher DOPs running on more CPUs seem to be able to achieve higher I/O throughput on the E10K/Symmetrix combination. I'd venture this could be because of increased caching opportunities or prioritisation of this host's requests as the demands increase, but have no evidence of either.[1]

Oracle Timed Events

Although the biggest factors limiting the performance of the tests I ran appeared to be related to the hardware configuration, it's also true that Oracle's management overhead increases with the DOP and I expected the time spent waiting on various timed events to increase. As described in Metalink Note 191103.1, Oracle categorises many PX-related events as being idle events (note, that this is aimed at 9.2) :-

```
Some of these can be considered as "events which you can do nothing about so
don't bother looking at them".
We also say to this events idle wait events.

At the moment we consider the following PX wait events as idle:
- "PX Idle Wait"
- "PX Deq: Execution Msg"
- "PX Deq: Table Q Normal"
- "PX Deq Credit: send blkd"
- "PX Deq: Execute Reply"
- "PX Deq Credit: need buffer"
- "PX Deq: Signal ACK"
- "PX Deque wait"
```

Although you might not be able to do anything about these events they are still worth looking at because they can give an

[1] Mike Scott has another suggestion on this – "This is also perhaps explainable because of higher bandwidth available between host and array that the lower number of cpu runs are incapable of exercising due to CPU contention (remember the CPUs are required also to service the I/O interrupts). Bear in mind also that there are multiple paths to the storage (two, in fact) in a balanced Active/Active config."

indication of underlying problems that could be solved. In any case, I would argue that you *can* do something about the time spent on some of these events - decrease the Degree of Parallelism!

For a general overview of how waits increase with the DOP, here's the tkprof output (including the timed events summary) for the Hash Join/Group By statement from the rolling.sh script, running on a 10-CPU E10K configuration, first without using PX :-

tkprof output – E10K – 10 CPUs – No Parallel

```
SELECT /*+ parallel(tt1, 1) parallel(tt2, 1) */ MOD(tt1.pk_id + tt2.pk_id, 113),
COUNT(*)
FROM test_tab1 tt1, test_tab2 tt2
WHERE tt1.pk_id = tt2.pk_id
GROUP BY MOD(tt1.pk_id + tt2.pk_id ,113)
ORDER BY MOD(tt1.pk_id + tt2.pk_id ,113)

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        9    597.41    1523.35    2730439    2735411          0         113
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total       11    597.41    1523.35    2730439    2735411          0         113

Misses in library cache during parse: 0
Optimizer mode: ALL_ROWS
Parsing user id: 39

Rows     Row Source Operation
-------  ---------------------------------------------------
    113  SORT GROUP BY (cr=2735411 pr=2730439 pw=21204 time=1282752775 us)
8189019   HASH JOIN  (cr=2735411 pr=2730439 pw=21204 time=1238538971 us)
8192000    TABLE ACCESS FULL TEST_TAB1 (cr=1367375 pr=1346953 pw=0
time=1138805959 us)
8192000    TABLE ACCESS FULL TEST_TAB2 (cr=1368036 pr=1362282 pw=0
time=1048630407 us)


Elapsed times include waiting on following events:
  Event waited on                             Times   Max. Wait  Total Waited
  ----------------------------------------   Waited  ----------  ------------
  SQL*Net message to client                       9        0.00          0.00
  db file scattered read                      24447        2.96       1092.54
  db file sequential read                       490        0.00          0.06
  direct path write temp                        684        0.18          1.34
  latch: cache buffers chains                     1        0.00          0.00
  direct path read temp                         684        0.00          0.13
  SQL*Net message from client                     9        0.00          0.02
********************************************************************************
```

There are very few wait events, other than the disk I/O waits that I'd expect when reading Gigabytes of data.

Next is the output for a DOP of 2. Remember that two slave sets will be used for this query, which implies (2 * 2) + 1 = 5 processes.

tkprof output – E10K – 10 CPUs – DOP 2

```
SELECT /*+ parallel(tt1, 2) parallel(tt2, 2) */ MOD(tt1.pk_id + tt2.pk_id, 113),
COUNT(*)
FROM test_tab1 tt1, test_tab2 tt2
WHERE tt1.pk_id = tt2.pk_id
GROUP BY MOD(tt1.pk_id + tt2.pk_id ,113)
ORDER BY MOD(tt1.pk_id + tt2.pk_id ,113)

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        5      0.01       0.00          0          0          0           0
Execute      5    469.92    1558.55    2748070    2785291          0           0
Fetch        9      0.09     389.92          0          0          0         113
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total       19    470.02    1948.48    2748070    2785291          0         113

Misses in library cache during parse: 0
Optimizer mode: ALL_ROWS
Parsing user id: 39

Elapsed times include waiting on following events:
  Event waited on                             Times   Max. Wait  Total Waited
  ----------------------------------------   Waited  ----------  ------------
  PX Deq: Join ACK                                3        0.01          0.01
  os thread startup                               2        0.24          0.48
  PX Deq: Parse Reply                             4        0.02          0.03
  SQL*Net message to client                       9        0.00          0.00
  PX Deq: Execute Reply                         366        1.96        389.24
  PX Deq: Execution Msg                         348        1.96         30.80
  direct path read                            22331        1.03        544.04
  PX Deq Credit: send blkd                      322        1.96         18.74
  PX Deq: Table Q Normal                      51041        0.95        529.26
  PX Deq Credit: need buffer                   6014        0.15         14.06
  direct path write temp                        481        0.11          8.15
  direct path read temp                         481        0.01          0.09
  PX qref latch                                  53        0.00          0.00
  PX Deq: Table Q Get Keys                        2        0.00          0.00
  PX Deq: Table Q Sample                          2        0.00          0.00
  PX Deq: Table Q qref                            2        0.00          0.00
  SQL*Net message from client                     9        0.00          0.02
  PX Deq: Signal ACK                             11        0.10          0.55
  enq: PS - contention                            1        0.00          0.00
********************************************************************************
```

The first PX-related events start to appear.

- The table reads are now being performed using Direct Path Reads as is normal when PX slaves are used.
- There are a couple of waits for **os thread startup** that, although they only take half a second, can soon add up when using very high DOPs. However, it's unlikely to be significant for the types of long-running tasks that you should be using PX for. If you were concerned about this you could increase `parallel_min_servers` so that slaves are left running after use, but I've seen a recommendation from Oracle that it's better to avoid this if possible.
- There are a number of **PX Deq: Execute Reply** events. This is time spent waiting for PX slaves to complete their work and whilst some waits are unavoidable, long wait times are a sign that the slaves are suffering from poor performance.
- The **PX Deq: Execution Message** events are another event that Oracle considers an idle event and there's no easy way to eliminate them**.** This is probably the most common event when using PX and is used when the PX slave is waiting to be told what to do. It is an indicator that there are more PX slaves passing more messages, though, and if the wait times are increasing it may be that the servers CPUs are overloaded. By decreasing the volume of PX message traffic, you'll decrease the waits on this event

- The **PX Deq: Table Q Normal** event (*Metalink Note 270921.)* is yet another idle event (can you see a pattern here?) that indicated that consumer slaves are waiting for producer slaves to send data on for them to process. For example, a slave is waiting for a producer to give it some rows to sort. Whilst some time spent waiting on this is inevitable, it also indicates that the consumers are working faster than the producers can keep up with, so it would be beneficial to improve the performance of the producers if possible or reduce the number of producers by decreasing the DOP.
- The **PX Deq Credit: send blkd** event (*Metalink Note 271767.1)* was one that Jonathan Lewis suggested I keep an eye out for as the DOP increased. This and the related **PX Deq Credit: need buffer** indicate that a producer wants to send data to a consumer, but the consumer is still busy with previous requests so isn't ready to receive it. i.e. it's falling behind. Reducing the DOP would reduce the number of times this happens and how long for.

Finally, here is the output with a DOP of 64 :-

tkprof output – E10K – 10 CPUs – DOP 64

```
********************************************************************************
SELECT /*+ parallel(tt1, 64) parallel(tt2, 64) */ MOD(tt1.pk_id + tt2.pk_id,
113), COUNT(*)
FROM test_tab1 tt1, test_tab2 tt2
WHERE tt1.pk_id = tt2.pk_id
GROUP BY MOD(tt1.pk_id + tt2.pk_id ,113)
ORDER BY MOD(tt1.pk_id + tt2.pk_id ,113)

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse      129      0.13       0.17          0          0          0           0
Execute    129    504.16   42254.40    2737959    3151821          0           0
Fetch        9      2.37     329.70          0          0          0         113
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total      267    506.66   42584.29    2737959    3151821          0         113

Misses in library cache during parse: 0
Optimizer mode: ALL_ROWS
Parsing user id: 39

Elapsed times include waiting on following events:
  Event waited on                             Times   Max. Wait  Total Waited
  ----------------------------------------   Waited  ----------  ------------
  PX Deq: Join ACK                               74        0.00          0.07
  os thread startup                              64        0.25         15.12
  latch: parallel query alloc buffer             12        0.00          0.00
  PX Deq: Parse Reply                            71        0.01          0.20
  PX Deq Credit: send blkd                     4060        1.96        431.91
  PX Deq Credit: need buffer                     17        0.03          0.09
  PX qref latch                                  23        0.00          0.00
  SQL*Net message to client                       9        0.00          0.00
  PX Deq: Execute Reply                        2983        1.95        327.10
  PX Deq: Execution Msg                        3740        1.99        533.44
  PX Deq: Msg Fragment                          488        0.00          0.08
  direct path read                            23774        5.92      20036.51
  PX Deq: Table Q Normal                      57984        1.96      18873.83
  direct path write temp                        320       25.27       1391.28
  direct path read temp                         320        1.05         29.99
  PX Deq: Table Q Sample                         70        0.25         11.96
  PX Deq: Table Q Get Keys                       65        0.22          8.04
  latch free                                      2        0.00          0.00
  SQL*Net message from client                     9        0.00          0.02
  PX Deq: Signal ACK                             16        0.10          0.17
********************************************************************************
```

There are several new things to note

- More and more waits on all of the PX-related events, which isn't surprising as there are two sets of 64 slaves communicating with each other and the QC. For example, not only has the number of waits on **PX Deq: Table Q Normal** increased, the average wait time has increased too, so much more time is being spent waiting for data to arrive from the producer slaves.
- That minor **os thread startup** event is contributing 15 seconds of wait time now. It's still pretty insignificant, but it's interesting how quickly it builds up.

- We're starting to see various latch waits that weren't apparent before. Although the time spent is insignificant, I think they're a sign that the server is becoming busier and that might cause problems for other users connected to the same oracle instance.

Each particular application of Parallel Execution is likely to produce it's on workload profile but hopefully I've highlighted here that, as well as using more system resources to reach the same results, higher DOPs will cause Oracle to wait more often on a wider variety of timed events. That should be completely obvious, but is easy to forget in the rush to throw resources at a slow-running query.

Test Results – Multi-user Tests (setup3.sql and session.sh scripts)

Prompted by Jonathan Lewis, I decided to try some multi-session variations on the tests. The worst performance problems I've experienced have been at sites where PX is used for semi-online operations. i.e. The user's requests take minutes to run, but there are varying numbers of online users waiting for the results. As Jonathan said when I showed him some early results - "The trouble is that you are using PX the way it is meant to be used - massive resources, very few users."

At first I just ran multiple sessions of the rolling.sh test but the run times were unworkable so I had to rethink my approach. I threw together[2] setup3.sql that creates 8 additional smaller tables containing 128,000 rows, each using 170Mb space because of the PCTFREE 90 setting.

Two shell scripts controlled the multi-user tests. I ran multiple sessions of the session.sh script that hash-joins test_tab1 to one of the new smaller tables, based on the session number passed in by the multi.sh script. The latter allowed me to control the number of concurrent sessions to execute. These tests were only executed on the ISP4400 with all 4 CPUs enabled; a range of DOPs of 1 to 4; and from 1 to 12 concurrent sessions.

When I first ran the tests, I found that certain sessions would take 2 or 3 times longer to complete than others. That was because I'd left `parallel_adaptive_multi_user` set to the default value of TRUE so Oracle was scaling back the DOP for some sessions as more sessions requested more PX slaves. This is designed to stop the server from grinding to a halt due to over-use of PX although interestingly, it still allowed sufficient work to take place that I found the server unusable! Using the same goal as the rest of the tests (to push PX beyond what it's designed to achieve) I disabled `parallel_adaptive_multi_user` and re-ran the tests.

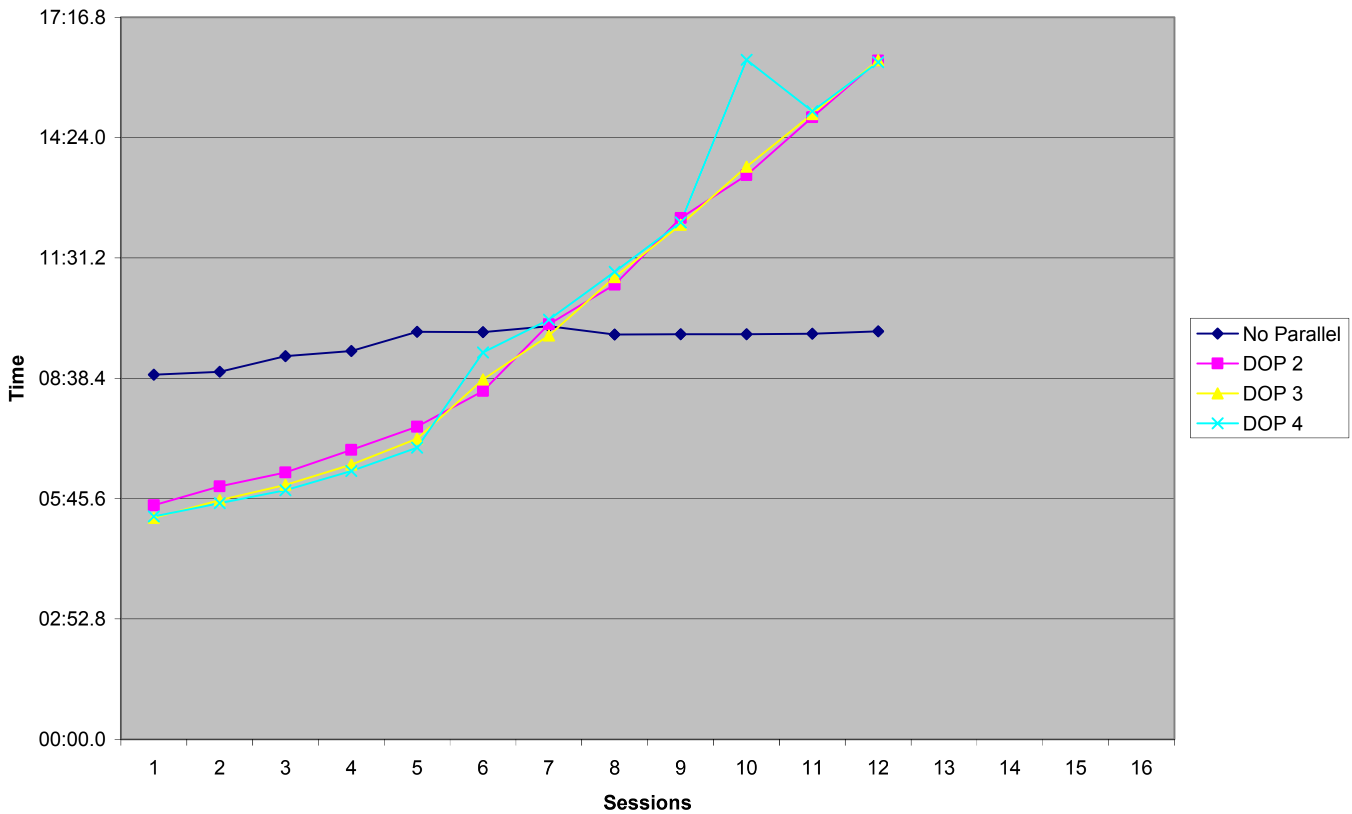
Conclusion – The Timings

Let's start with the timings again. Again, the tests were run many times while I was fine-tuning the scripts and interpreting performance data and the run times barely changed, so I'm confident that there are no significant one-off anomalies in the results. As there were response times for many concurrent sessions, I had to make a decision on which result to graph. I decided to use the maximum response time based on my natural pessimism and experience of user perceptions! In practice, the longest time was well within 10% of the shortest time with the exception of jobs that ran for a very short period of time.

---

[2] You'll see why I use these words when you look at the script!

Graph 22 –ISP4400 4 CPUs – Hash Join 11Gb table to 170Mb tables – 1-12 concurrent sessions / DOP 1-4



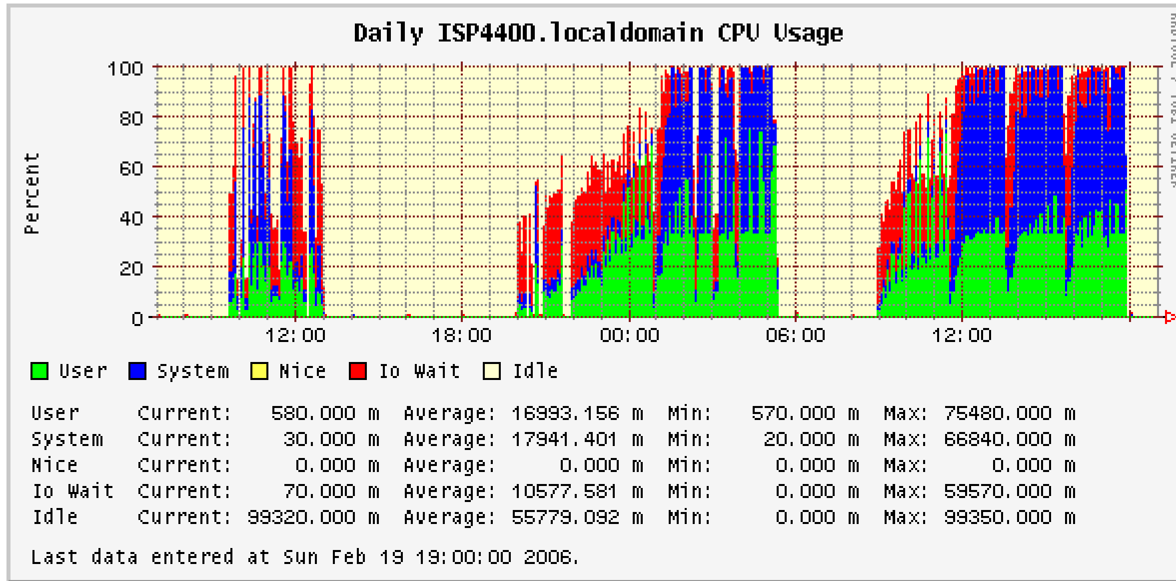
The multi-user tests highlight how inappropriate it is to use PX for typical online reporting, even when users want reasonably quick results from reports run against very large databases. I think the highlights are :-

- Again, when moving from single-threaded execution to parallel execution with a DOP of 2, there is a significant improvement in run time – from eight and a half minutes, down to just over 5 minutes. The difference is not dramatic, but is the type of improvement users would probably notice and welcome.
- For the first 5 or 6 concurrent users running this report job, the job will still complete in less time than if it had been single-threaded, which '*proves*' that PX is appropriate for this report. That is until the 9th and 10th users start using parallel and things start to take longer and longer. At least the 'poor' single-threaded users experience a reasonably consistent response time!
- Worse than that – the effect on the rest of the users of the server (who aren't using PX) becomes noticeable much earlier. When I was running these tests, I found it unbearably slow to get a simple directory listing or vi a file once there were more than two or three concurrent sessions running. So although the reports might be running more quickly than the single-threaded version, everyone else is suffering. This is a key message of Cary Millsap's 'Magic of 2' work. This will become more obvious when we look at the server statistics in a moment.
- I think the most dramatic piece of evidence that PX and multiple sessions don't mix is that these effects are all noticeable at the lowest possible DOP of 2 (admittedly with a query involving two slaves sets) and with less than 10 concurrent users. Of course, because everything looked so much better for the first few users, we might not expect things to get so bad.

This all seemed very familiar to me because I'd seen it on systems I'd worked on in the past, but had never had the

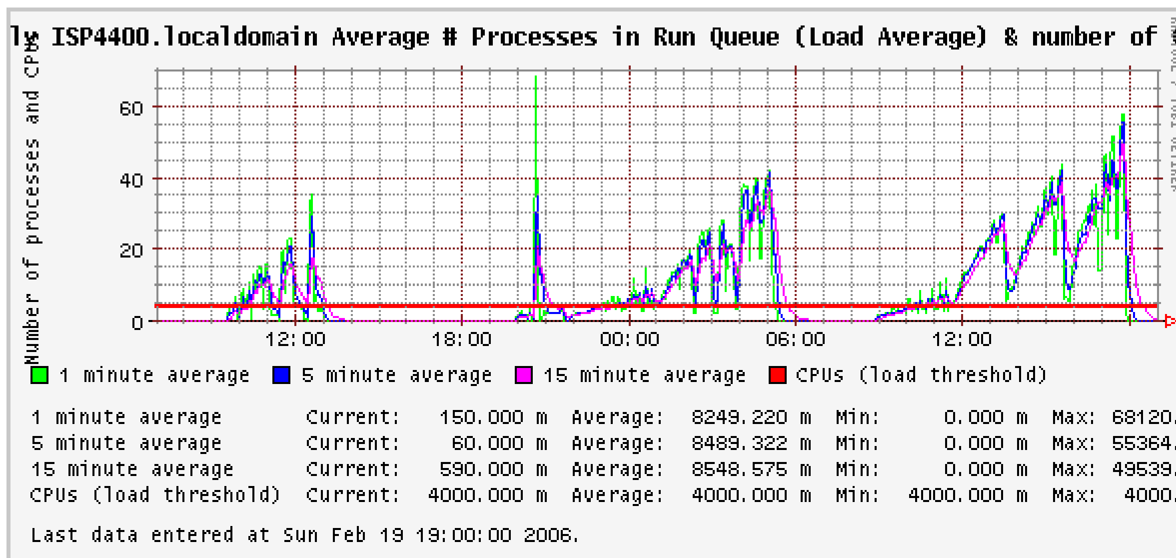
opportunity to run tests like this to prove it.

Graph 23 –CPU Usage – ISP4400 4 CPUs – HJ 11Gb table to 170Mb tables – 1-12 concurrent sessions / DOP 1-4



The CPU usage graph shows the results of the multi-user tests. The only relevant section is at the far right of the graph, where the tests were run between 9am and 6pm. The single-threaded test (first spike) uses much less CPU than all of the parallel tests, which run at 100% CPU utilisation for almost the entire test and system CPU is much higher[3]. Remember each of these spikes represents the same number of concurrent user sessions reading the same volumes of data, so the only reason for the additional workload is the use of Parallel Execution. The first spike is a little wider, so the work was spread out over a longer period of time, but the overall server health looks much better.

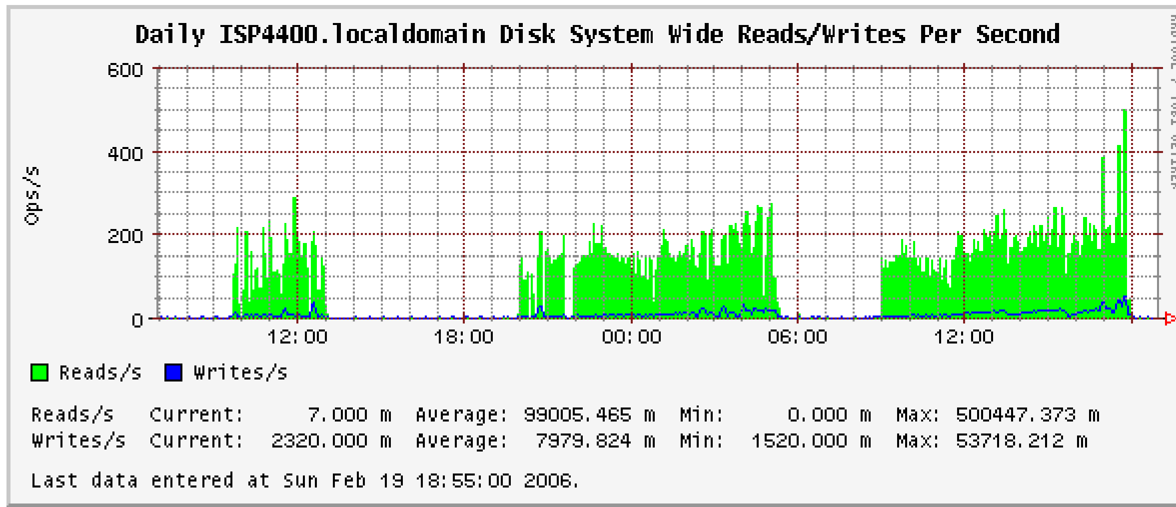
Graph 24 –Run Queue – ISP4400 4 CPUs – HJ 11Gb table to 170Mb tables – 1-12 concurrent sessions / DOP 1-4



[3] Although I recall that the extra System CPU may be because direct path reads are reported this way, but I was unable to confirm this before publication

The Run Queue graph highlights the detrimental effect on the server as a whole caused by multiple concurrent PX users. Again, the significant bit of the graph is on the far right. When PX is not being used (first spike), the run queue only starts to go above the number of CPUs as the number of sessions increases above about 7 or 8, halfway through the test. Compare that to the other three spikes, which exhibit long run queues throughout the tests.

Graph 25 – No. of disk ops per second – ISP4400 4 CPUs – HJ 11Gb table to 170Mb tables – 1-12 concurrent sessions / DOP 1-4



Looking at the I/O statistics, there was a danger that the multi-user tests were also I/O bound to some extent and although this might represent a typical system, what I was really looking for was to test CPU load and inter-slave communications. I decided to hash join two of the smaller tables in each session so that all of the data was likely to be read from the database buffer or filesystem caches.

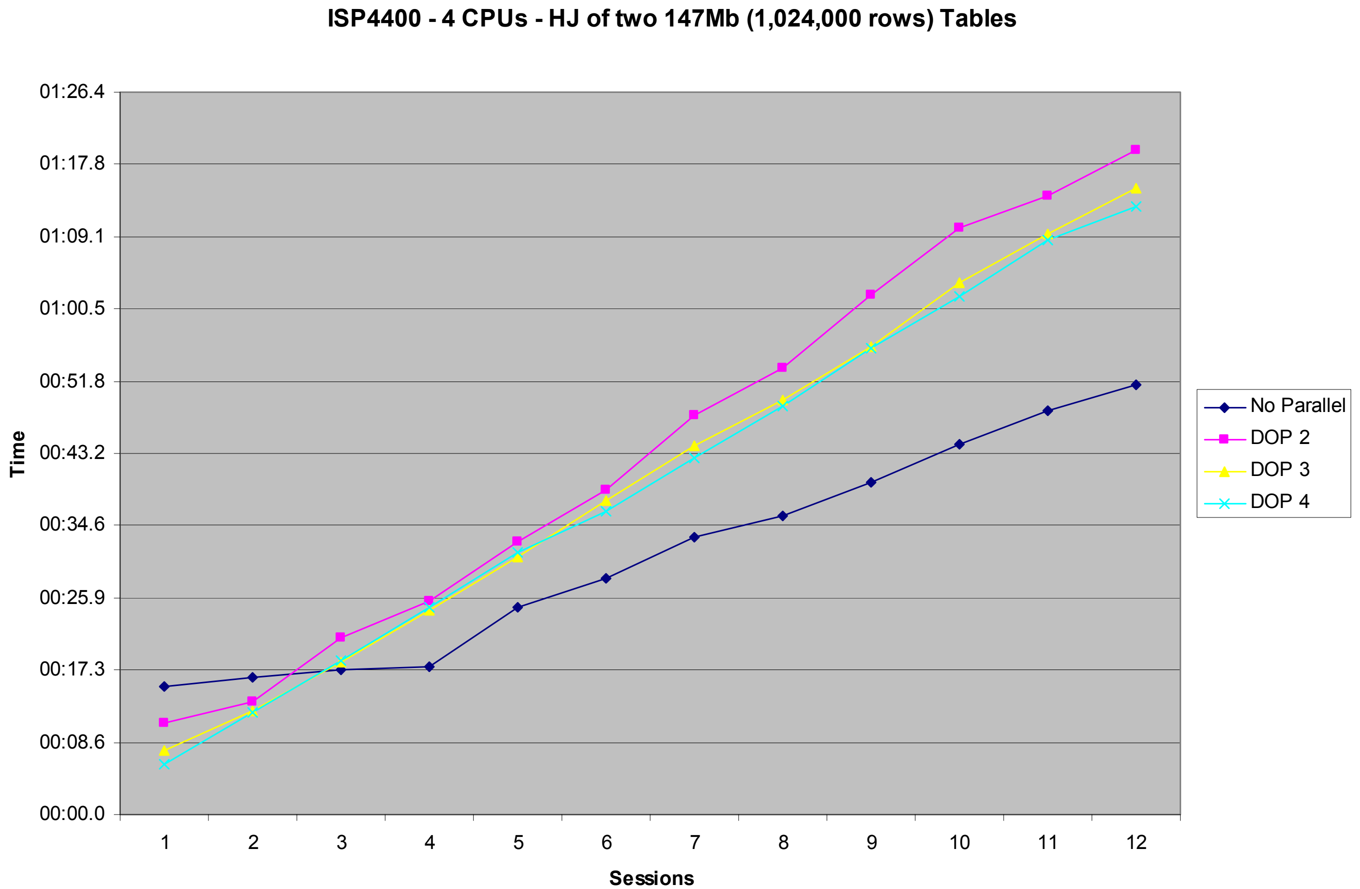
Although this succeeded in eliminating disk I/O and the run times reduced, the PX wait events didn't increase as much as I expected. I suspected that was because the blocks were only about 10% full and there were only 128,000 rows in each table because of the PCTFREE setting of 90. That was designed to increase I/O volume but, as the size of the tables was scaled down and I was more interested in how data moved through the system, the scripts simply weren't generating enough work for the slaves[4].

I addressed this by changing PCTFREE on the small tables to 10 and inserting 1 million rows, resulting in 147Mb data segments (see setup4.sql).

---

[4] **In fact, it would be worth re-running the single user/volume tests (i.e. rolling.sh) with the same test_tab1 and test_tab2 data segments, but with a percent free setting of 10 so that the tables contain many more rows. This would change the balance of the jobs from I/O activity towards more intensive PX and CPU usage. Although I suspect that the I/O bottlenecks would still be the main limiting factor, the most efficient DOP might be higher.**

Graph 26 – ISP4400 4 CPUs – Hash Join two 147Mb tables – 1-12 concurrent sessions / DOP 1-4

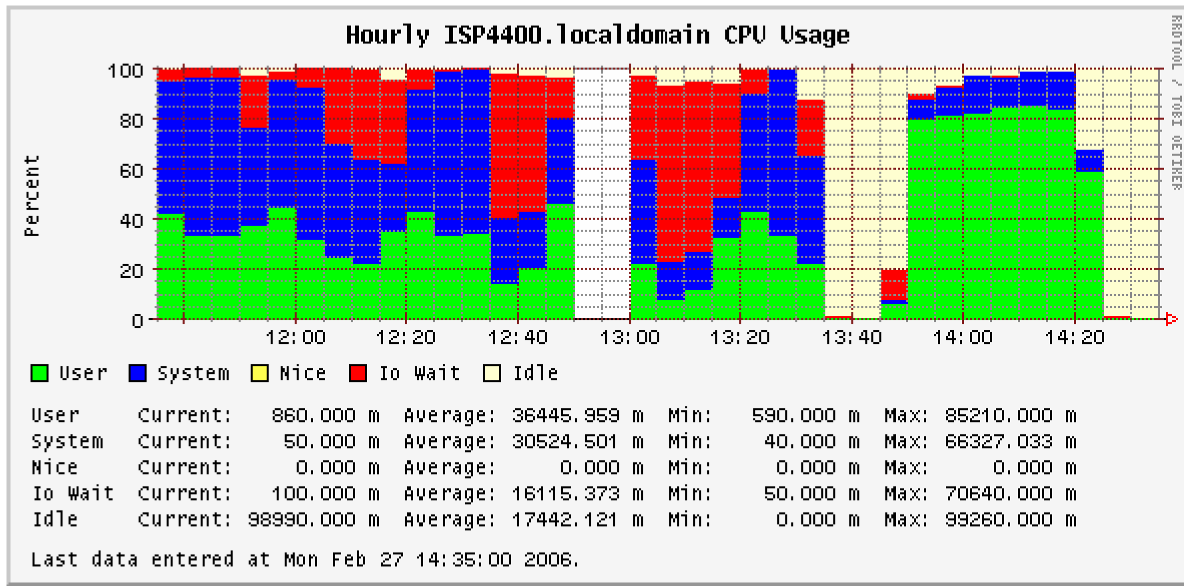


With little or no I/O activity, the drop-off in performance when using PX over the single-threaded approach is lower. However, the minor benefits are still only apparent when there are less than 3 active sessions!

Operating System Statistics

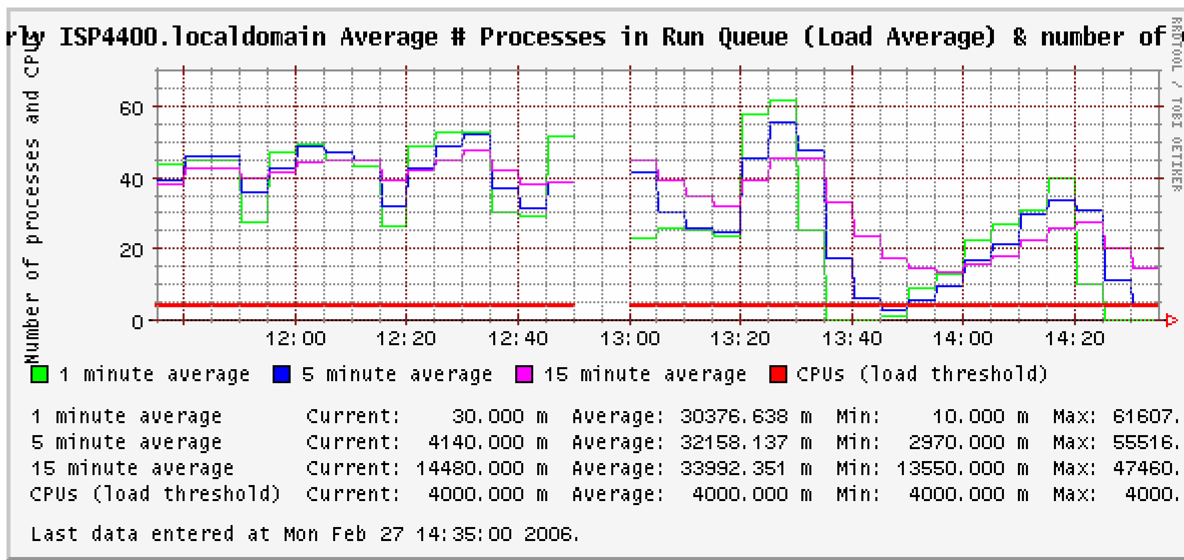
Although I would have liked to have kept the same scale for the x axis as the ISP4400 volume tests, the multi-user tests completed so quickly that I've used hourly graphs again here.

Graph 27- CPU Usage – ISP4400 4 CPUs – Hash Join two 147Mb tables – 1-12 concurrent sessions / DOP 1-4



Although this is a much shorter range of time, the only significant part of the graph is on the far right, from 13:45 – 14:25. With the change to much smaller tables that can be cached effectively, the IO Wait percentage of CPU is now insignificant although even a few non-parallel sessions are capable of using almost all of the available CPU.[5]

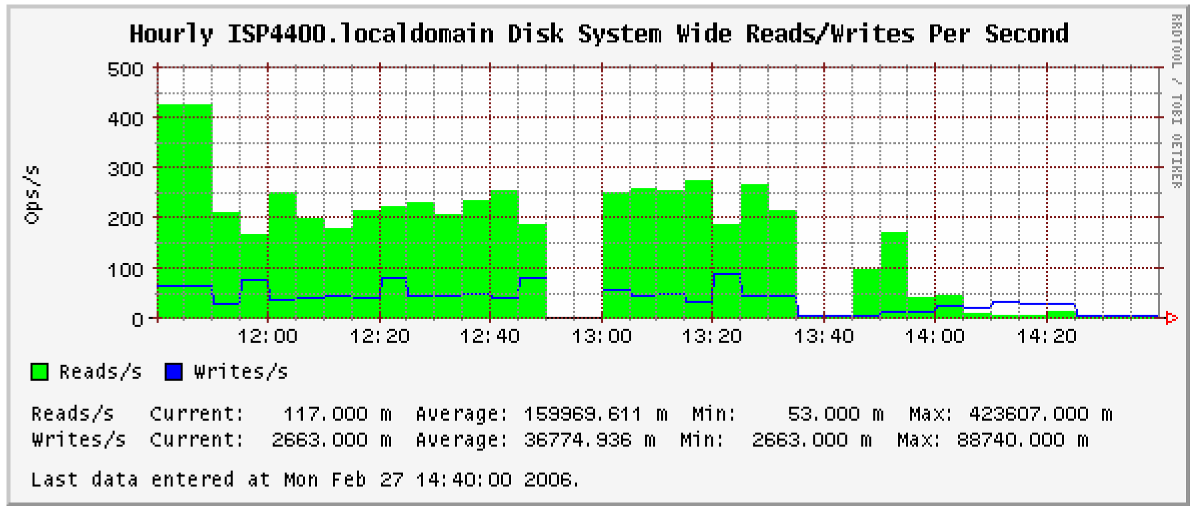
Graph 28- Run Queue – ISP4400 4 CPUs – Hash Join two 147Mb tables – 1-12 concurrent sessions / DOP 1-4



Once again, everything prior to 13:45 was related to another test. Although this run queue curve looks similar to the other tests, if you compare it to graph 24 and compare the y axes, you'll see that the run queue was much shorter during this test and, whilst the server performance did suffer as the run queue grew, it didn't feel quite so unusable!

[5] The smaller proportion of System CPU is another indication that this is how Direct Path Reads are recorded.

Graph 29- No. of disk ops per second – ISP4400 4 CPUs – Hash Join two 170Mb tables – 1-12 concurrent sessions / DOP 1-4



Looking at 13:45 onwards, although there was some initial disk activity (some of it generated by me copying some trace files) it soon disappeared. In earlier tests, I witnessed no disk activity at all.

Oracle Timed Events

Here's the tkprof output for the longest-running session when 10 concurrent sessions were running without using PX

tkprof output – ISP4400 – 4 CPUs – 10 sessions - No Parallel – Run Time – 44.28 seconds

```
SELECT /*+ parallel(tt1, 1) parallel(tt2, 1) */ MOD(tt1.pk_id + tt2.pk_id, 113),
 COUNT(*)
FROM test_tab3 tt1, test_tab8 tt2
WHERE tt1.pk_id = tt2.pk_id
GROUP BY MOD(tt1.pk_id + tt2.pk_id ,113)
ORDER BY MOD(tt1.pk_id + tt2.pk_id ,113)

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        9     16.61      43.21       8070      36970          0         113
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total       11     16.61      43.21       8070      36970          0         113

Misses in library cache during parse: 0
Optimizer mode: ALL_ROWS
Parsing user id: 22

Rows     Row Source Operation
-------  ---------------------------------------------------
    113  SORT GROUP BY (cr=36970 pr=8070 pw=0 time=43209091 us)
1022997   HASH JOIN  (cr=36970 pr=8070 pw=0 time=43088222 us)
```

```
1024000    TABLE ACCESS FULL TEST_TAB3 (cr=18484 pr=1 pw=0 time=133121899 us)
1024000    TABLE ACCESS FULL TEST_TAB8 (cr=18486 pr=8069 pw=0 time=5122033 us)


Elapsed times include waiting on following events:
  Event waited on                             Times   Max. Wait  Total Waited
  ----------------------------------------   Waited  ----------  ------------
  SQL*Net message to client                       9        0.00          0.00
  db file sequential read                      1531        0.00          0.09
  latch: cache buffers chains                     4        0.18          0.43
  latch free                                      2        0.10          0.19
  db file scattered read                        674        0.01          0.41
  latch: cache buffers lru chain                  1        0.00          0.00
  SQL*Net message from client                     9        0.00          0.00
********************************************************************************
```

There are a few more buffer cache latch wait events and these were common to the smaller table/multi-user tests when many sessions were trying to access the same blocks more frequently, particularly as there were many more rows per block at PCTFREE 10. A number of I/O waits were also apparent, but if you look at the very short duration and contribution to the overall response time and combine that with the I/O graphs we've just seen, it looks like the data was being cached in the O/S filesystem cache.

Next is the output for a DOP of 2. Remember that two slave sets will be used for this query, which implies (2 * 2) + 1 = 5 processes.

tkprof output – ISP4400 – 4 CPUs – 10 sessions – DOP 2 – Run Time – 70.26 seconds

```
SELECT /*+ parallel(tt1, 2) parallel(tt2, 2) */ MOD(tt1.pk_id + tt2.pk_id, 113),
 COUNT(*)
FROM test_tab3 tt1, test_tab3 tt2
WHERE tt1.pk_id = tt2.pk_id
GROUP BY MOD(tt1.pk_id + tt2.pk_id ,113)
ORDER BY MOD(tt1.pk_id + tt2.pk_id ,113)

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        5      0.00       0.00          0          0          0           0
Execute      5     24.30     265.97      36920      37158          0           0
Fetch        9      0.03      65.40          0          0          0         113
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total       19     24.34     331.38      36920      37158          0         113

Misses in library cache during parse: 0
Optimizer mode: ALL_ROWS
Parsing user id: 22

Elapsed times include waiting on following events:
  Event waited on                             Times   Max. Wait  Total Waited
  ----------------------------------------   Waited  ----------  ------------
  os thread startup                               5        0.84          2.31
  PX Deq: Join ACK                                4        0.00          0.01
  PX qref latch                                 236        0.85         14.91
  PX Deq: Parse Reply                             2        0.07          0.07
  SQL*Net message to client                       9        0.00          0.00
  PX Deq: Execute Reply                          94        2.15         60.89
  PX Deq: Execution Msg                          96        4.26         19.94
  direct path read                              344        0.35          0.36
  PX Deq Credit: send blkd                     2886        2.18        102.20
```

```
  PX Deq Credit: need buffer                       479          1.98          10.93
  PX Deq: Table Q Normal                          3386          4.10          59.17
  PX Deq: Table Q Sample                             2          0.00           0.00
  PX Deq: Table Q Get Keys                           2          0.00           0.00
  PX Deq: Table Q qref                               2          0.00           0.00
  SQL*Net message from client                        9          0.00           0.00
  PX Deq: Signal ACK                                 4          0.09           0.10
  latch free                                         1          0.00           0.00
  enq: PS - contention                               2          0.00           0.00
********************************************************************************
```

- It's clear that time is being lost here waiting on several PX-specific wait events. Many of the events are the same idle events described for the volume tests, but much more time is being spent on **PX Deq Credit: send blkd** and **PX Deq Credit: need buffer** (which have already been described on page 29) and …
- **PX qref latch** (*Metalink Note 240145.1*). Although it did appear in the volume test job profiles, the time spent was insignificant and it's a much bigger factor here. Now that the I/O bottleneck has disappeared, this new bottleneck has appeared. I discussed this in my previous paper and suggested that there are two main techniques for reducing the impact of this event. First, you could try increasing parallel_execution_message_size. I gave this a quick test and found that there was a small improvement from increasing it to 16Kb. The second approach you could take is to decrease the DOP (or stop using PX for this statement?)

Testing Tips

Before offering some conclusions, here are a few tips to help you if you decide to perform similar tests yourself based on my experiences of these tests.

1. Start small while writing the scripts and then increase the size of the tests once you're comfortable everything is working as you'd hoped. However, be wary of the results at this stage, because small volumes are likely to lead to some strange results.
2. Script everything if possible so that you can run the tests unattended. Feel free to take the test scripts from this paper and plug your own SQL statements into them to test what suits your application.
3. Capture *everything* you can during the tests. There's nothing more frustrating than having a successful test run, seeing a strange result and having insufficient data to identify the underlying cause. There were also a number of cases where looking at some of the apparently unnecessary data I'd captured highlighted a problem to be fixed. However, you should always be conscious of whether the measurement activity is impacting the tests.
4. You may need to revisit earlier configurations as you refine the scripts based on your results. These tests were an iterative process and I would have preferred further iterations, given more time.
5. There is lots of cheap equipment to be had on eBay. Businesses cycle through equipment so quickly that what is deemed surplus to requirements is probably still suitable equipment for testing. However, it you are going to run these tests at home, try to buy a server that doesn't sound like an aircraft taking off unless you have a spare room!

Areas for Further Investigation

One of the benefits of setting up and running these tests is that it would be easy to re-run with different configurations and focussing on different types of workload. As I was working on the tests I kept thinking of other areas I would have liked to test, given endless amounts of time. Here are just a few but I'm sure you could think of your own ideas too.

- As I mentioned earlier, the focus for this exercise was to push PX on various platforms to discover the sensible limits and their causes. It would be interesting to use Oracle's default PX settings on each server and allow the server to scale back the DOP as the number of sessions increase to see how this would change both the run times and the server resource utilisation.

- More complex and CPU-intensive SQL statements could be embedded in the test scripts.
- The tests could be repeated with the data spread across a larger number of hard disks to see what relieving that bottleneck would achieve and what new bottlenecks would become apparent.
- There are many opportunities to try different storage characteristics such as block sizes, partitioning, partition-wise joins, ASSM etc. I think my next test will probably be a repeat of the volume tests with more tightly packed blocks, as described earlier.

Conclusions

Based on three different hardware configurations and a large number of tests, there are certain suggestions I would make. (Note the use of the word *suggestions* – the test results from these servers can't prove for certain what will happen on your system, but you can prove that yourself by running similar tests.)

1. On an SMP server with sufficient CPU and I/O bandwidth, there will be significant benefits from using a DOP of two (and possibly a few degrees higher) for long-running jobs when you are the only user of the server.
2. The benefits diminish rapidly and, taking into account the extra resource usage, it could be argued that a DOP of anything greater than two has a poor ratio of costs to benefits. I emphasise – ***not a DOP of two per CPU, but for the system as whole***. Remember that, because Oracle can use 2*DOP number of slaves for even a simple query, the number of slaves grows quickly. Of course, you may find that your I/O subsystem is capable of much higher throughput or that your jobs are much more CPU-intensive but the biggest benefits come from moving from noparallel to a DOP of two in these tests and I suspect this would be true on other configurations.
3. It's not very sensible to use Parallel Execution for multiple online users. If you do, make sure that parallel_adaptive_multi_user is enabled (to prevent the server as a whole from becoming overloaded) and that your users understand that response times will be variable depending on the current server workload.
4. If you're implementing a new system, trying running tests to establish what is the most appropriate DOP for your environment. Run through the tests, increasing the DOP whilst monitoring the system Run Queue. When you reach the shortest execution time, my advice would be to STOP and choose a DOP slightly lower than the apparent optimum. For example, if a DOP of 4 gives the fastest run times, I'd be tempted to use a DOP of two or three, to prevent exhausting system resources when the system goes live.
5. You can never have too much hardware when running Parallel Execution but, no matter how much equipment you throw at the problem, a single bottleneck can render it all pretty worthless. Note the 12 CPUs on the Sun E10K server I had at my disposal. Most of that expense was worthless to me because I didn't have sufficient I/O bandwidth to keep the CPUs busy. They spent most of their time waiting on I/O.

Everything I've said here is based on the configurations that I tested. In particular, none of the test configurations had more than five available Hard Disk Drive spindles, so I/O bottlenecks were a constant underlying performance problem.

I'll close with a couple of off-the-wall suggestions.

- To borrow a phrase - 'It's the *disks*, stupid!' When deciding the best DOP, so much attention is paid to the number of CPUs, when perhaps CPUs are so powerful and plentiful in modern servers that we should really be basing these decisions on the I/O bandwidth available? If you reconsider all of the test results here, it would appear that, rather than higher DOPs being appropriate when the I/O subsystem is slow relative to the CPU resource, lower DOPs yield better results. (Of course, the results might well be different when I test again with more tightly-packed blocks)
- Maybe there's a different 'Magic of Two' that's specific to Parallel Execution – that the maximum cost/benefit point for parallel jobs is a DOP of 2, resulting in 3-5 processes per job. I'd be interested to hear of people who have had success with higher DOPs or multiple users or who try similar tests. Drop me an email at dougburns@yahoo.com

Bibliography and Resources

The primary inspiration for this paper was Cary Millsap's earlier paper, Batch Queue Management and the Magic of '2' - http://www.hotsos.com/e-library/abstract.php?id=13

Cary was also good enough to let me take a look at his presentation material from his Miracle Master Class in 2006 which contained some of his more recent thoughts on the subject. I'm unaware whether this material is published yet.

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! There are references to PX in the Concepts manual, Administrator's reference etc, but the Data Warehousing Guide contains a particularly useful chapter, Using Parallel Execution -
http://download-east.oracle.com/docs/cd/B19306_01/server.102/b14223/usingpe.htm#i1009828

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.

184417.1 – Where to track down the information on Parallel Execution in the Oracle documentation!

203238.1 – Summary of how Parallel Execution works.

280939.1 - Checklist for Performance Problems with Parallel Execution (but also contains a useful explanation of DFO effects on number of PX slaves used)

191103.1 – Parallel Execution Wait Events (contains links to event-specific information)

201799.1 – init.ora parameters for Parallel Execution

240762.1 - pqstat PL/SQL procedure to help monitor all of the PX slaves running on the instance or for one user session

202219.1 – Script to map PX slaves to Query Co-ordinator (An alternative to using the procedure in note 240762.1)

238680.1 – Investigating ORA-4031 errors caused by lack of memory for queues.

242374.1 – Discusses some of the issues around PX session tracing (but not in any great depth)

237328.1 – Direct Path Reads (brief discussion)

Tom Kyte has referred to the 'Magic of 2' in at least one question and answer thread at asktom.oracle.com

http://asktom.oracle.com/pls/ask/f?p=4950:8:18113549788223702501::NO::F4950_P8_DISPLAYID,F4950_P8_CRITERIA:37335239610842

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.

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

Orca and Procallator

Both are available from http://www.orcaware.com/orca/ My thanks to Blair Zajac and Guilherme Chehab for their contribution.

I've also written a brief blog entry on how I configured Orca on the two Linux servers used in the tests.

http://doug.burns.tripod.com/2006/02/orca.html

Acknowledgements

Thanks to the following people for their help in preparing this paper.

- Mike Scott – Hindsight IT (http://www.hindsight.it/) - for locating the high-end kit for me and constant assistance in system configuration and performance matters. I wouldn't have completed the paper without his help.
- Jonathan Lewis (http://www.jlcomp.demon.co.uk/) – for some characteristically intelligent and timely suggestions and clarifications.
- Peter Robson for last minute detailed editorial comments and interesting perspectives.
- Gerrit Van Wyk – for allowing me to use work equipment.
- The Dog – for late night monitoring

About the author

Doug Burns is an independent contractor who has 15 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.

Appendix A – Test Scripts

setup.sql

```
REM setup.sql - to be run using a DBA-privileged account

REM NOTE THAT THIS WILL DROP ANY EXISTING TESTUSER ACCOUNT!!
drop user testuser cascade;

REM LIKEWISE, THIS WILL DROP THE TEST_DATA TABLESPACE
drop tablespace test_data including contents and datafiles;

REM YOU'LL WANT YOUR OWN FILE LOCATIONS IN HERE
REM 24GB FOR BIGGER TESTS, 6GB FOR SMALLER
create tablespace TEST_DATA
datafile '/u01/oradata/TEST1020/test_data01.dbf' size 24000m
extent management local uniform size 1m
segment space management auto;

create user testuser identified by testuser
default tablespace test_data
temporary tablespace temp;

alter user testuser quota unlimited on test_data;

REM MAKE SURE AUTOTRACE IS SET UP AND THEN GRANT PRIVS TO TESTUSER
@?/sqlplus/admin/plustrce

grant create session, create table, create sequence, plustrace,
gather_system_statistics select any table to testuser;
grant execute on dbms_monitor to testuser;
grant execute on dbms_stats to testuser;

REM SETUP STATSPACK FOR ANOTHER POSSIBLE MEASUREMENT APPROACH
@?/rdbms/admin/spcreate
```

setup2.sql

```
connect testuser/testuser
set echo on
set lines 160

alter session set recyclebin = off;
alter session set max_dump_file_size=unlimited;

alter session enable parallel dml;
alter session enable parallel ddl;

execute dbms_random.seed(999);

drop sequence test_seq1;
drop table test_tab1;
drop sequence test_seq2;
drop table test_tab2;

create sequence test_seq1 cache 1000;
create sequence test_seq2 cache 1000;

REM Create 1000 row tables with a sequence-generated primary key, repeating values of NUM_CODE
between 1 and 10 and some random character data as padding.

create table test_tab1
```

```
pctfree 90 pctused 10
as select test_seq1.nextval PK_ID, MOD(test_seq1.nextval, 10) NUM_CODE,
        dbms_random.string('A', 100) STR_PAD
from all_objects
where rownum <= 1000;

create table test_tab2
pctfree 90 pctused 10
as select test_seq2.nextval PK_ID, MOD(test_seq2.nextval, 10) NUM_CODE,
        dbms_random.string('A', 100) STR_PAD
from all_objects
where rownum <= 1000;

REM Increase number of rows to just over 8 million.

begin
        for i IN 1 .. 13 loop
                insert /*+ parallel(test_tab1, 2) append */
                into test_tab1
                select /*+ parallel(test_tab1, 2) */
                        test_seq1.nextval, NUM_CODE, STR_PAD
                from test_tab1;

                commit;

                insert /*+ parallel(test_tab2, 2) append */
                into test_tab2
                select /*+ parallel(test_tab2, 2) */
                        test_seq2.nextval, NUM_CODE, STR_PAD
                from test_tab2;

                commit;

        end loop;
end;
/

REM Generate CBO statistics on the tables even though these are not really necessary given the
nature of the queries that will be run and the lack of indexes.
begin
dbms_stats.gather_table_stats(user, 'test_tab1', cascade => false,
                                        estimate_percent => 1,
                                        method_opt => 'for all columns size 1');
dbms_stats.gather_table_stats(user, 'test_tab2', cascade => false,
                                        estimate_percent => 1,
                                        method_opt => 'for all columns size 1');
end;
/

REM Sanity-check of row counts and segment sizes
select table_name, num_rows from user_tables;
select segment_name, bytes/1024/1024 MB from user_segments;
```

rolling.sh

```
#!/bin/ksh

# Note that the next line will be CPU_COUNT=num_cpus for the Intel tests as this
# script is run once for a given CPU configuration on those servers. i.e. Because
# CPU availability can only be modified online on the E10K, that was the only
# platform that uses the outer 'CPU' loop.

CPU_COUNT=12
DOP_LIST="1 2 3 4 5 6 7 8 9 10 11 12 16 24 32 48 64 96 128"
```

```
# Clear out any existing trace files to make things simpler
rm /ora/admin/TEST1020/bdump/*.trc
rm /ora/admin/TEST1020/udump/*.trc

# As noted earlier, this loop is only used in the E10K version
while [ $CPU_COUNT -gt 1 ]
do
        for DOP in $DOP_LIST
        do

# Jonathan Lewis suggestion - gather system statistics for each run
sqlplus / as sysdba << EOF
exec dbms_stats.gather_system_stats('START');
exit
EOF

# Connect as testuser, enable autotrace and start spooling to log file
sqlplus testuser/testuser << EOF

set autotrace on
break on dfo_number on tq_id
set echo on
spool ${CPU_COUNT}_$DOP.log


REM Enable tracing at the session level and set identifier to help trcsess later
exec dbms_session.set_identifier('${CPU_COUNT}_$DOP');
alter session set tracefile_identifier='${CPU_COUNT}_$DOP';
exec dbms_monitor.client_id_trace_enable(client_id => '${CPU_COUNT}_$DOP');

set timing on

REM Plug the SQL to be tested in here.

SELECT /*+ parallel(tt1, $DOP) */ COUNT(*)
FROM test_tab1 tt1;

SELECT /*+ parallel(tt1, $DOP) parallel(tt2, $DOP) */ MOD(tt1.pk_id + tt2.pk_id, 113), COUNT(*)
FROM test_tab1 tt1, test_tab2 tt2
WHERE tt1.pk_id = tt2.pk_id
GROUP BY MOD(tt1.pk_id + tt2.pk_id ,113)
ORDER BY MOD(tt1.pk_id + tt2.pk_id ,113);

set timing off
set autotrace off
exec dbms_monitor.client_id_trace_disable(client_id => '${CPU_COUNT}_$DOP');

REM For additional confirmation of what just occurred, look at v$pq_tqstat
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 , process;

exit
EOF

# Get the results of the System Statistics gather
sqlplus / as sysdba << EOF
exec dbms_stats.gather_system_stats('STOP');

spool ${CPU_COUNT}_$DOP.sysstats
select  pname, pval1
from    sys.aux_stats$
where   sname = 'SYSSTATS_MAIN';
```

```
exit
EOF

# Generate a consolidated trace file for the query coordinator and all of the slaves
# using trcsess …

trcsess output='/ora/admin/TEST1020/udump/${CPU_COUNT}_$DOP.trc' clientid='${CPU_COUNT}_$DOP'
/ora/admin/TEST1020/udump/test*.trc /ora/admin/TEST1020/bdump/test*.trc

# … and tkprof it.
tkprof /ora/admin/TEST1020/udump/${CPU_COUNT}_$DOP.trc
/ora/admin/TEST1020/udump/${CPU_COUNT}_$DOP.out sort=prsela,fchela,exeela

        done

# The following section is only relevant on the E10K, where CPUs can be taken offline
# dynamically

        CPU_COUNT=$(($CPU_COUNT -1))
        sudo psradm -f $(($CPU_COUNT +11))
        psrinfo

        CPU_COUNT=$(($CPU_COUNT -1))
        sudo psradm -f $(($CPU_COUNT +11))
        psrinfo
done
```

setup3.sql

```
REM This script is a cut-and-paste disgrace ;-)

connect testuser/testuser
set echo on
set lines 160

spool setup3.log

alter session set recyclebin = off;
alter session set max_dump_file_size=unlimited;

alter session enable parallel dml;
alter session enable parallel ddl;

execute dbms_random.seed(999);

drop sequence test_seq3;
drop table test_tab3;
drop sequence test_seq4;
drop table test_tab4;
drop sequence test_seq5;
drop table test_tab5;
drop sequence test_seq6;
drop table test_tab6;
drop sequence test_seq7;
drop table test_tab7;
drop sequence test_seq8;
drop table test_tab8;
drop sequence test_seq9;
drop table test_tab9;
drop sequence test_seq10;
drop table test_tab10;

create sequence test_seq3 cache 1000;
```

```
create sequence test_seq4 cache 1000;
create sequence test_seq5 cache 1000;
create sequence test_seq6 cache 1000;
create sequence test_seq7 cache 1000;
create sequence test_seq8 cache 1000;
create sequence test_seq9 cache 1000;
create sequence test_seq10 cache 1000;

create table test_tab3
pctfree 90 pctused 10
as select test_seq3.nextval PK_ID, MOD(test_seq3.nextval, 10) NUM_CODE,
        dbms_random.string('A', 100) STR_PAD
from all_objects
where rownum <= 1000;

create table test_tab4
pctfree 90 pctused 10
as select test_seq4.nextval PK_ID, MOD(test_seq4.nextval, 10) NUM_CODE,
        dbms_random.string('A', 100) STR_PAD
from all_objects
where rownum <= 1000;

create table test_tab5
pctfree 90 pctused 10
as select test_seq5.nextval PK_ID, MOD(test_seq5.nextval, 10) NUM_CODE,
        dbms_random.string('A', 100) STR_PAD
from all_objects
where rownum <= 1000;

create table test_tab6
pctfree 90 pctused 10
as select test_seq6.nextval PK_ID, MOD(test_seq6.nextval, 10) NUM_CODE,
        dbms_random.string('A', 100) STR_PAD
from all_objects
where rownum <= 1000;

create table test_tab7
pctfree 90 pctused 10
as select test_seq7.nextval PK_ID, MOD(test_seq7.nextval, 10) NUM_CODE,
        dbms_random.string('A', 100) STR_PAD
from all_objects
where rownum <= 1000;

create table test_tab8
pctfree 90 pctused 10
as select test_seq8.nextval PK_ID, MOD(test_seq8.nextval, 10) NUM_CODE,
        dbms_random.string('A', 100) STR_PAD
from all_objects
where rownum <= 1000;

create table test_tab9
pctfree 90 pctused 10
as select test_seq9.nextval PK_ID, MOD(test_seq9.nextval, 10) NUM_CODE,
        dbms_random.string('A', 100) STR_PAD
from all_objects
where rownum <= 1000;

create table test_tab10
pctfree 90 pctused 10
as select test_seq10.nextval PK_ID, MOD(test_seq10.nextval, 10) NUM_CODE,
        dbms_random.string('A', 100) STR_PAD
from all_objects
where rownum <= 1000;
```

```
begin
        for i IN 1 .. 7 loop
                insert /*+ parallel(test_tab3, 2) append */
                into test_tab3
                select /*+ parallel(test_tab3, 2) */
                        test_seq3.nextval, NUM_CODE, STR_PAD
                from test_tab3;

                commit;

                insert /*+ parallel(test_tab4, 2) append */
                into test_tab4
                select /*+ parallel(test_tab4, 2) */
                        test_seq4.nextval, NUM_CODE, STR_PAD
                from test_tab4;

                commit;

                insert /*+ parallel(test_tab5, 2) append */
                into test_tab5
                select /*+ parallel(test_tab5, 2) */
                        test_seq5.nextval, NUM_CODE, STR_PAD
                from test_tab5;

                commit;

                insert /*+ parallel(test_tab6, 2) append */
                into test_tab6
                select /*+ parallel(test_tab6, 2) */
                        test_seq6.nextval, NUM_CODE, STR_PAD
                from test_tab6;

                commit;

                insert /*+ parallel(test_tab7, 2) append */
                into test_tab7
                select /*+ parallel(test_tab7, 2) */
                        test_seq7.nextval, NUM_CODE, STR_PAD
                from test_tab7;

                commit;

                insert /*+ parallel(test_tab8, 2) append */
                into test_tab8
                select /*+ parallel(test_tab8, 2) */
                        test_seq8.nextval, NUM_CODE, STR_PAD
                from test_tab8;

                commit;

                insert /*+ parallel(test_tab9, 2) append */
                into test_tab9
                select /*+ parallel(test_tab9, 2) */
                        test_seq9.nextval, NUM_CODE, STR_PAD
                from test_tab9;

                commit;

                insert /*+ parallel(test_tab10, 2) append */
                into test_tab10
                select /*+ parallel(test_tab10, 2) */
                        test_seq10.nextval, NUM_CODE, STR_PAD
                from test_tab10;
```

```
                commit;

        end loop;
end;
/

begin
dbms_stats.gather_table_stats(user, 'test_tab3', cascade => false,
                                        estimate_percent => 1,
                                        method_opt => 'for all columns size 1');
dbms_stats.gather_table_stats(user, 'test_tab4', cascade => false,
                                        estimate_percent => 1,
                                        method_opt => 'for all columns size 1');
dbms_stats.gather_table_stats(user, 'test_tab5', cascade => false,
                                        estimate_percent => 1,
                                        method_opt => 'for all columns size 1');
dbms_stats.gather_table_stats(user, 'test_tab6', cascade => false,
                                        estimate_percent => 1,
                                        method_opt => 'for all columns size 1');
dbms_stats.gather_table_stats(user, 'test_tab7', cascade => false,
                                        estimate_percent => 1,
                                        method_opt => 'for all columns size 1');
dbms_stats.gather_table_stats(user, 'test_tab8', cascade => false,
                                        estimate_percent => 1,
                                        method_opt => 'for all columns size 1');
dbms_stats.gather_table_stats(user, 'test_tab9', cascade => false,
                                        estimate_percent => 1,
                                        method_opt => 'for all columns size 1');
dbms_stats.gather_table_stats(user, 'test_tab10', cascade => false,
                                        estimate_percent => 1,
                                        method_opt => 'for all columns size 1');
end;
/

select table_name, num_rows from user_tables;
select segment_name, bytes/1024/1024 MB from user_segments;

spool off
```

setup4.sql

```
REM This script is even more of a cut-and-paste disgrace ;-)
REM It's just setup3.sql with a couple of value changes that should be parameters

connect testuser/testuser
set echo on
set lines 160

spool setup3.log

alter session set recyclebin = off;
alter session set max_dump_file_size=unlimited;

alter session enable parallel dml;
alter session enable parallel ddl;

execute dbms_random.seed(999);

drop sequence test_seq3;
drop table test_tab3;
drop sequence test_seq4;
drop table test_tab4;
drop sequence test_seq5;
drop table test_tab5;
```

```
drop sequence test_seq6;
drop table test_tab6;
drop sequence test_seq7;
drop table test_tab7;
drop sequence test_seq8;
drop table test_tab8;
drop sequence test_seq9;
drop table test_tab9;
drop sequence test_seq10;
drop table test_tab10;

create sequence test_seq3 cache 1000;
create sequence test_seq4 cache 1000;
create sequence test_seq5 cache 1000;
create sequence test_seq6 cache 1000;
create sequence test_seq7 cache 1000;
create sequence test_seq8 cache 1000;
create sequence test_seq9 cache 1000;
create sequence test_seq10 cache 1000;

create table test_tab3
pctfree 10 pctused 10
as select test_seq3.nextval PK_ID, MOD(test_seq3.nextval, 10) NUM_CODE,
        dbms_random.string('A', 100) STR_PAD
from all_objects
where rownum <= 1000;

create table test_tab4
pctfree 10 pctused 10
as select test_seq4.nextval PK_ID, MOD(test_seq4.nextval, 10) NUM_CODE,
        dbms_random.string('A', 100) STR_PAD
from all_objects
where rownum <= 1000;

create table test_tab5
pctfree 10 pctused 10
as select test_seq5.nextval PK_ID, MOD(test_seq5.nextval, 10) NUM_CODE,
        dbms_random.string('A', 100) STR_PAD
from all_objects
where rownum <= 1000;

create table test_tab6
pctfree 10 pctused 10
as select test_seq6.nextval PK_ID, MOD(test_seq6.nextval, 10) NUM_CODE,
        dbms_random.string('A', 100) STR_PAD
from all_objects
where rownum <= 1000;

create table test_tab7
pctfree 10 pctused 10
as select test_seq7.nextval PK_ID, MOD(test_seq7.nextval, 10) NUM_CODE,
        dbms_random.string('A', 100) STR_PAD
from all_objects
where rownum <= 1000;

create table test_tab8
pctfree 10 pctused 10
as select test_seq8.nextval PK_ID, MOD(test_seq8.nextval, 10) NUM_CODE,
        dbms_random.string('A', 100) STR_PAD
from all_objects
where rownum <= 1000;

create table test_tab9
pctfree 10 pctused 10
```

```
as select test_seq9.nextval PK_ID, MOD(test_seq9.nextval, 10) NUM_CODE,
        dbms_random.string('A', 100) STR_PAD
from all_objects
where rownum <= 1000;

create table test_tab10
pctfree 10 pctused 10
as select test_seq10.nextval PK_ID, MOD(test_seq10.nextval, 10) NUM_CODE,
        dbms_random.string('A', 100) STR_PAD
from all_objects
where rownum <= 1000;

begin
        for i IN 1 .. 10 loop
                insert /*+ parallel(test_tab3, 2) append */
                into test_tab3
                select /*+ parallel(test_tab3, 2) */
                        test_seq3.nextval, NUM_CODE, STR_PAD
                from test_tab3;

                commit;

                insert /*+ parallel(test_tab4, 2) append */
                into test_tab4
                select /*+ parallel(test_tab4, 2) */
                        test_seq4.nextval, NUM_CODE, STR_PAD
                from test_tab4;

                commit;

                insert /*+ parallel(test_tab5, 2) append */
                into test_tab5
                select /*+ parallel(test_tab5, 2) */
                        test_seq5.nextval, NUM_CODE, STR_PAD
                from test_tab5;

                commit;

                insert /*+ parallel(test_tab6, 2) append */
                into test_tab6
                select /*+ parallel(test_tab6, 2) */
                        test_seq6.nextval, NUM_CODE, STR_PAD
                from test_tab6;

                commit;

                insert /*+ parallel(test_tab7, 2) append */
                into test_tab7
                select /*+ parallel(test_tab7, 2) */
                        test_seq7.nextval, NUM_CODE, STR_PAD
                from test_tab7;

                commit;

                insert /*+ parallel(test_tab8, 2) append */
                into test_tab8
                select /*+ parallel(test_tab8, 2) */
                        test_seq8.nextval, NUM_CODE, STR_PAD
                from test_tab8;

                commit;

                insert /*+ parallel(test_tab9, 2) append */
                into test_tab9
```

```
                select /*+ parallel(test_tab9, 2) */
                        test_seq9.nextval, NUM_CODE, STR_PAD
                from test_tab9;

                commit;

                insert /*+ parallel(test_tab10, 2) append */
                into test_tab10
                select /*+ parallel(test_tab10, 2) */
                        test_seq10.nextval, NUM_CODE, STR_PAD
                from test_tab10;

                commit;

        end loop;
end;
/

begin
dbms_stats.gather_table_stats(user, 'test_tab3', cascade => false,
                                        estimate_percent => 1,
                                        method_opt => 'for all columns size 1');
dbms_stats.gather_table_stats(user, 'test_tab4', cascade => false,
                                        estimate_percent => 1,
                                        method_opt => 'for all columns size 1');
dbms_stats.gather_table_stats(user, 'test_tab5', cascade => false,
                                        estimate_percent => 1,
                                        method_opt => 'for all columns size 1');
dbms_stats.gather_table_stats(user, 'test_tab6', cascade => false,
                                        estimate_percent => 1,
                                        method_opt => 'for all columns size 1');
dbms_stats.gather_table_stats(user, 'test_tab7', cascade => false,
                                        estimate_percent => 1,
                                        method_opt => 'for all columns size 1');
dbms_stats.gather_table_stats(user, 'test_tab8', cascade => false,
                                        estimate_percent => 1,
                                        method_opt => 'for all columns size 1');
dbms_stats.gather_table_stats(user, 'test_tab9', cascade => false,
                                        estimate_percent => 1,
                                        method_opt => 'for all columns size 1');
dbms_stats.gather_table_stats(user, 'test_tab10', cascade => false,
                                        estimate_percent => 1,
                                        method_opt => 'for all columns size 1');
end;
/

select table_name, num_rows from user_tables;
select segment_name, bytes/1024/1024 MB from user_segments;

spool off
```

session.sh

```
#!/bin/ksh

CPU_COUNT=$1
SESSION=$2
# Join to a different table depending on a mod division of the session number passed in
TABLE=`expr $2 \% 8 + 3`
DOP=$3

sqlplus testuser/testuser << EOF
```

```
set autotrace on
break on dfo_number on tq_id
set echo on
set pages 9999
spool ${CPU_COUNT}_${SESSION}_${DOP}.log

exec dbms_session.set_identifier('${CPU_COUNT}_${SESSION}_${DOP}');
alter session set tracefile_identifier='${CPU_COUNT}_${SESSION}_${DOP}';
exec dbms_monitor.client_id_trace_enable(client_id => '${CPU_COUNT}_${SESSION}_$
{DOP}');

set timing on

REM Note that I edited this statement and replaced test_tab1 with test_tab3 for the
REM second set of multi-user tests

SELECT /*+ parallel(tt1, $DOP) parallel(tt2, $DOP) */ MOD(tt1.pk_id + tt2.pk_id,
 113), COUNT(*)
FROM test_tab1 tt1, test_tab$TABLE tt2
WHERE tt1.pk_id = tt2.pk_id
GROUP BY MOD(tt1.pk_id + tt2.pk_id ,113)
ORDER BY MOD(tt1.pk_id + tt2.pk_id ,113);

set timing off
set autotrace off
exec dbms_monitor.client_id_trace_disable(client_id => '${CPU_COUNT}_${SESSION}_
${DOP}');

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 , process;

exit
EOF
trcsess output="${CPU_COUNT}_${SESSION}_$DOP.trc" clientid="${CPU_COUNT}_${SESSI
ON}_$DOP" /home/oracle/product/10.2.0/db_1/admin/TEST1020/udump/test*.trc /home/
oracle/product/10.2.0/db_1/admin/TEST1020/bdump/test*.trc

tkprof ${CPU_COUNT}_${SESSION}_$DOP.trc ${CPU_COUNT}_${SESSION}_$DOP.out
sort=prsela,fchela,exeela
```

multi.sh

```
#!/bin/ksh
LOOP=0
while [ $LOOP -lt $2 ]
do
       LOOP=`expr $LOOP + 1`

#Run child session.sh script in background

       ./session.sh $1 $LOOP $3 &
       echo $LOOP
done
# Wait for all child sessions to complete
wait
```