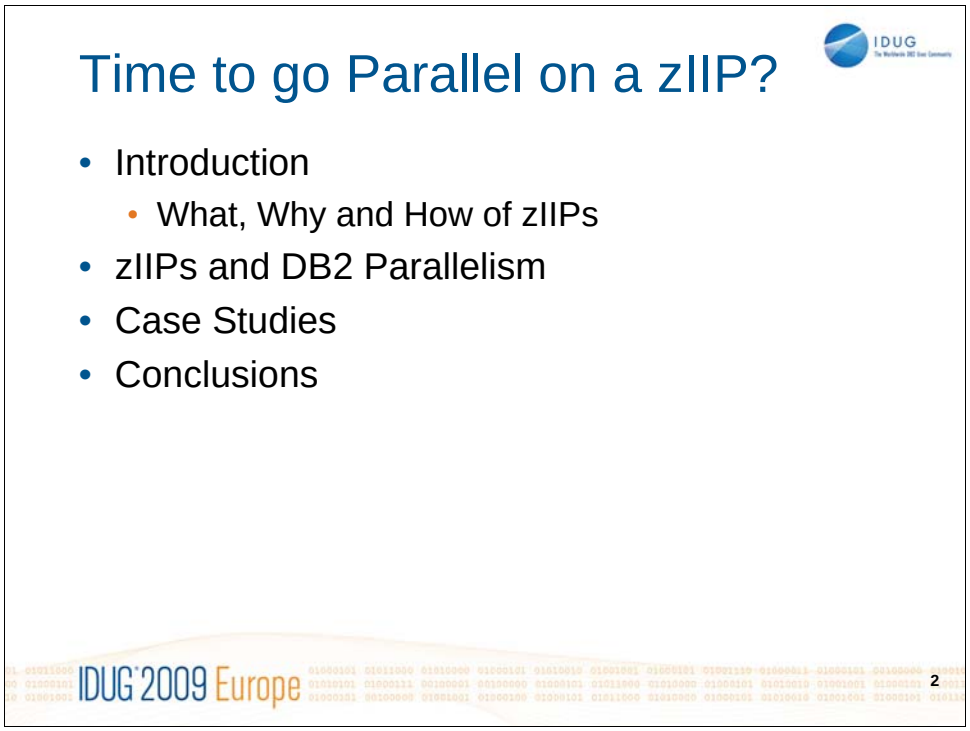

zIIP processors have provided great benefit for distributed processing; however, most people seem to overlook the fact that *the* biggest beneficiary of zIIP offload is actually parallel processing.

So, with cost-cutting all the rage in today's financial crisis, is there a case for turning on parallelism in our main non-BI(Data Warehouse) production systems to benefit from zIIP offload? And if there is, how can we identify the processes and queries where parallel processing will benefit without creating overhead and how much offload can we expect?

This presentation examines 2 separate non-BI environments where we have tried to increase zIIP usage by introducing Parallel Processing. Not only will it provide real figures on the actual zIIP offload obtained for parallel processing in non-BI environments, but it will also examine some of the difficulties involved in identifying candidate workloads and provide some real-life horror stories of what happens when parallelism is used inappropriately!



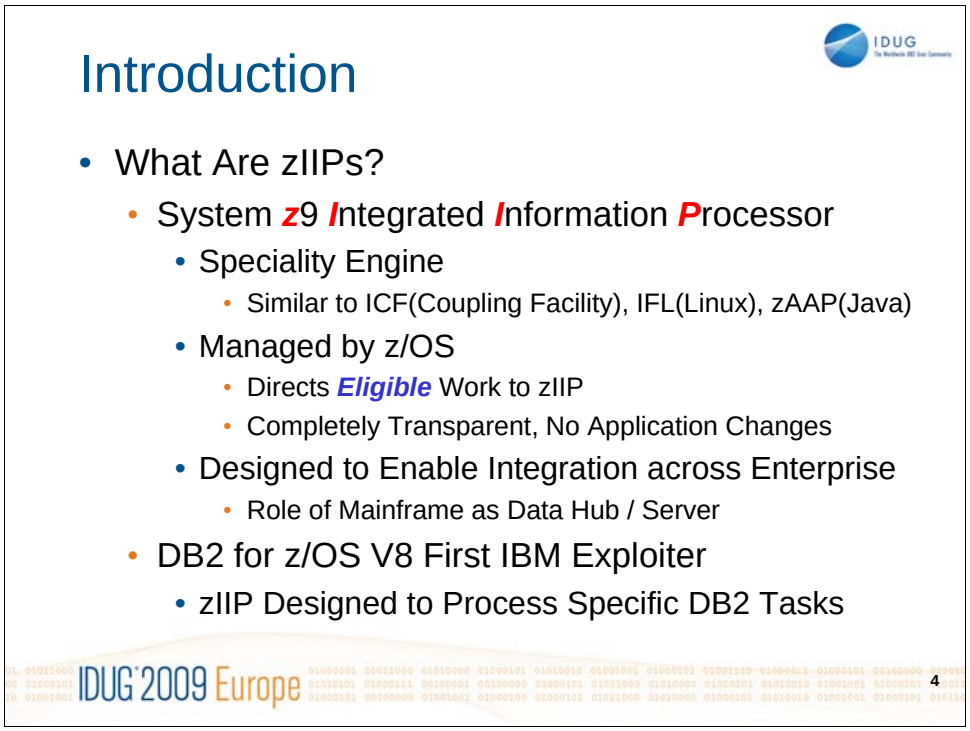

After a brief overview of the basics of zIIP processors and the types of workload that can exploit them, the presentation will concentrate on how the zIIP can be exploited by parallel processing in DB2 for z/OS.

Through real-life case studies of both complex and non-complex queries the presentation will provide ***real*** figures on the actual zIIP offload obtained by activating parallel processing in two different non-BI(Data Warehouse) production environments(V8).

Through the case studies the presentation will also provide hints and tips on how to identify candidate queries and workloads, highlighting the importance of partitioning and the choice of an appropriate partitioning criteria together with the overheads and pitfalls involved in parallel processing, especially when used for inappropriate workloads.

# Time to go Parallel on a zIIP?

- Introduction
  - What, Why and How of zIIPs
- zIIPs and DB2 Parallelism
- Case Studies
- Conclusions



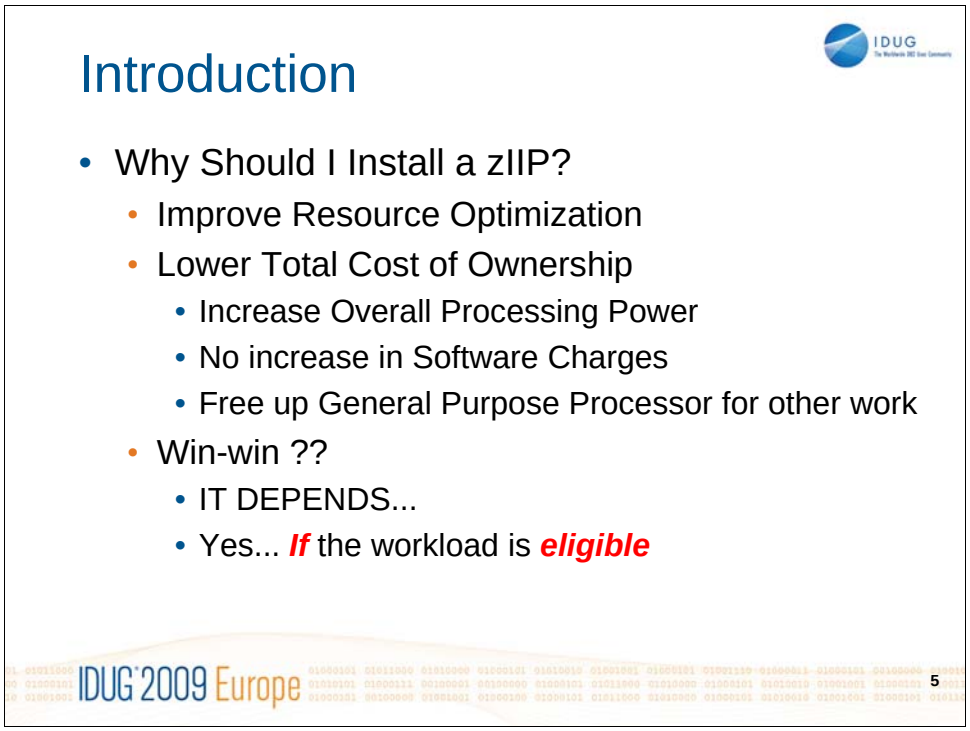

To start with let's just take a very quick look at what a zIIP actually is.

A zIIP is what IBM calls a "speciality" engine. In other words a processor, just like the ICF, IFL & zAAP before it, which is only available to certain (*eligible*) tasks or workloads, as opposed to a General Purpose Processor, which is available for ***all*** types of processing.

The operating system manages and directs eligible work between the GPP and the zIIP meaning that its use is completely transparent, and uncontrollable, to existing applications i.e. if they qualify as being eligible the operating system will decide when and how much of the work to offload to the speciality engine.

One thing to note here is that not ALL of the eligible work will be offloaded, just certain "portions" of it. The sizes of these "portions" for the various workloads are not documented publicly!!

The main purpose of the zIIP is, as its name suggests, is to enhance the role of the Mainframe as the data hub or server across the Enterprise by providing cheaper processing power for Information integration and Data Serving workloads across the enterprise.




So *why* should we install a zIIP? Well, the answer to this really is a no-brainer:

The purpose of the zIIP is to improve resource optimization and ***lower the cost of ownership*** for eligible workloads.

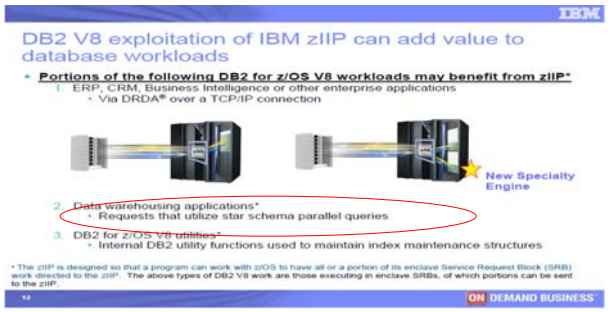
With a zIIP you can increase your overall processing power with cheaper hardware costs, but above all, without increasing your software charges. Also, by redirecting work to the zIIP, capacity is freed up on the GPP allowing it to perform other tasks or workloads at no extra cost!

So it would appear that we have a win-win situation...well, this is DB2, so of course the answer really is...IT DEPENDS... And in this case, as already mentioned several times so far, it depends on whether our processing is ***eligible*** for offload to the zIIP.

So, what work is eligible...

# Introduction

- How Can We Use a zIIP?
  - IBM Initial Announcement 2006:




  - V9 Native SQL Procedures
  - V9 Some XML and Cryptography processing

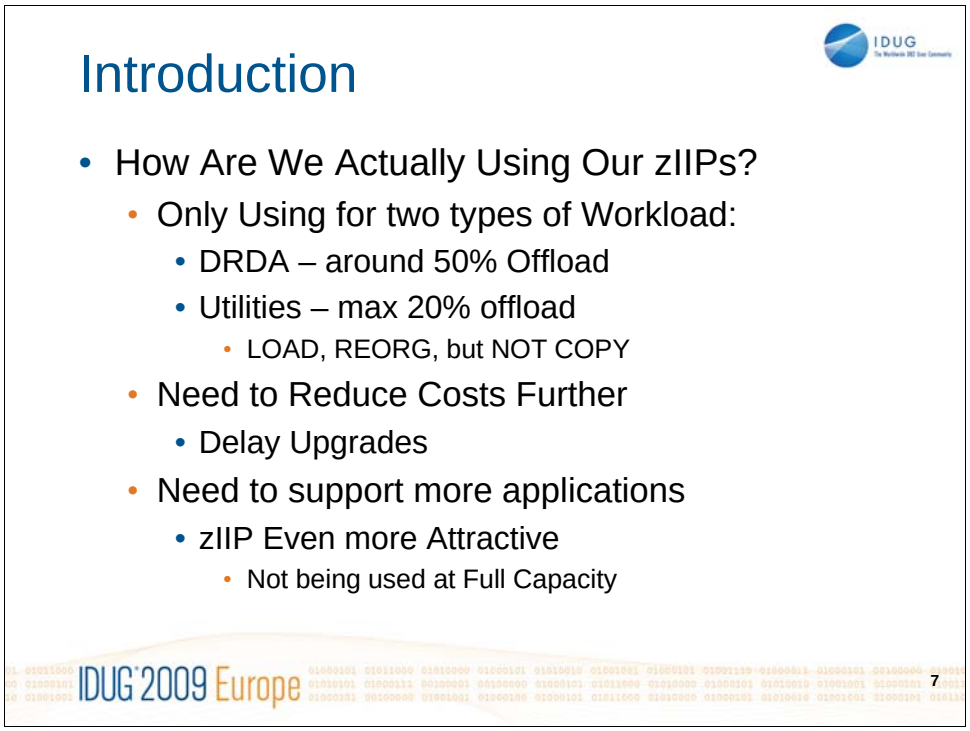


According to the IBM announcement in 2006 the initial eligible workloads for exploiting the zIIP were:

- DRDA Applications over a TCP/IP connection;
- IBM Utilities – internal processes for Index Maintenance;
- Star Schema Parallel Queries.

Or more precisely, “the zIIP is designed so that a program can work with z/OS to have all or a portion of its enclave Service Request Block (SRB) work directed to zIIP.” In fact the above types of work all run in enclave SRBs, and therefore portions of which can be sent to zIIP.

Since the initial announcement IBM have announced that in Version 9 Native SQL procedures will be eligible for the zIIP and also some XML and Cryptography processing is also eligible.




So that's the theory, what about in practice?

Are shops using zIIPs ? And if so, how are they using them? For what types of workload? And what is the effect ?

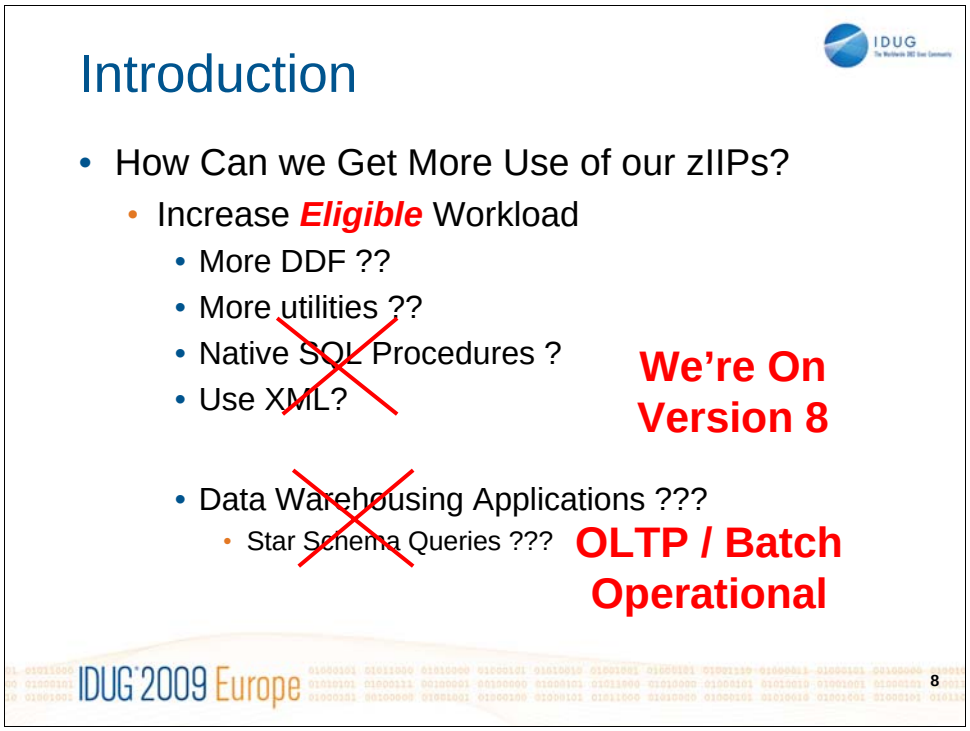
Well, I have a few customers who have installed zIIPs and overall they are fairly happy with the results. However, all of them have only used their zIIPs for two types of workload:

- IBM Utilities
- Distributed Processing via DRDA

Now, with today's current financial climate, my customers are all being asked to reduce their costs further. Normally by means of delaying a processor upgrade, but at the same time they are also being asked to provide more processing power to support more applications or increased throughput of existing applications.

This situation makes the zIIP ever more attractive, especially as in most of my customers for large periods, include peak times, the zIIP is under utilized!

So how can we increase use of our zIIPs ?




To get more work directed to the zIIP, the solution is quite simple, we just have to increase the types of workload that are ***eligible*** to be redirected. So what does that actually mean...

We could try and run more processing via DDF – new applications would still need General Purpose Processor capacity to run the portion of work not offloaded to zIIP, but existing applications will gain, however, moving an existing application to run via DRDA is neither a trivial nor a quick exercise.

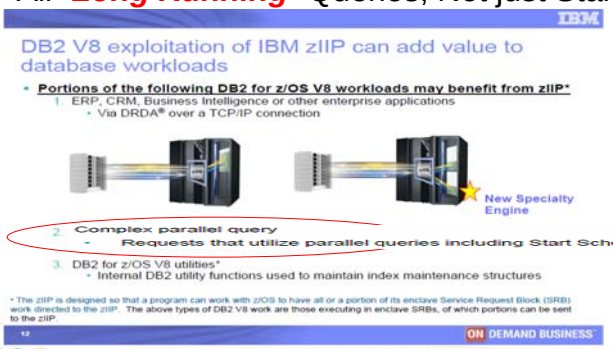
If we were on Version 9 we could think about writing some Native SQL procedures or using some XML, but we're not, we're still on Version 8.

That leaves Data Warehousing Applications and Star Schema Queries... Again no good to my customers who are standard OLTP/Batch Operational shops.

So it would appear that we have no chance....Or does it....??

# Introduction

- How Can We Use zIIPs - revisited
  - PK27578 – July 2006
    - Extend Exploitation to ***ALL*** CP Parallel Queries
    - All "***Long Running***" Queries, Not just Star Join






About 6 months after the initial announcement of the zIIP, IBM came out with a new APAR – PK27578 – which extended zIIP exploitation to ***ALL*** CP Parallel Queries including Star Join. In fact, the original APAR – PK18454 – only specified "Star Join parallel child processing" as being eligible for offload to the zIIP.

The new APAR then further qualified this by stating that eligibility for zIIP offload was for "long running" queries; however, there has never been an official definition of exactly what is meant by the term "long running"...

Also, after this APAR IBM changed the wording on its zIIP presentations, with the terms "Data Warehouse" & "Business Intelligence Workloads" being replaced by "Complex Parallel Queries".

# Introduction

- Who Is Using Parallelism?
  - "Tried when it was introduced V3/V4"
    - Goal to reduce ***elapsed*** times
    - ***Increase in CPU***
- Version 3 November 1993
  - 16 years ago...Time to rethink?
- New Paradigm?
  - Can parallelism actually ***reduce*** CPU ?
  - Increase zIIP usage, Free up GPP
  - Delay Upgrade, Reduce Costs!

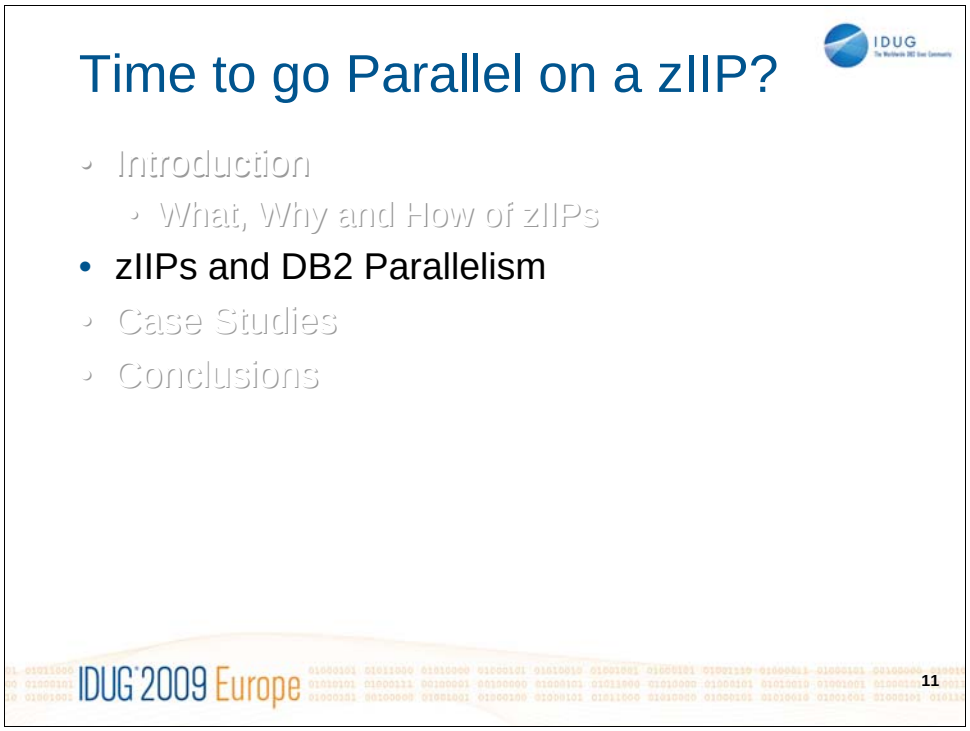



Who is using parallelism? And, more importantly, who is using it a non-DW/BI environment? Well, my customers certainly weren't and according to Willie Favero's Blog, not many other sites were!!
In fact, there appears to be some hostility towards the use of parallelism: from Willie's Blog almost 30% of respondents replied "No, you have to be crazy"!!

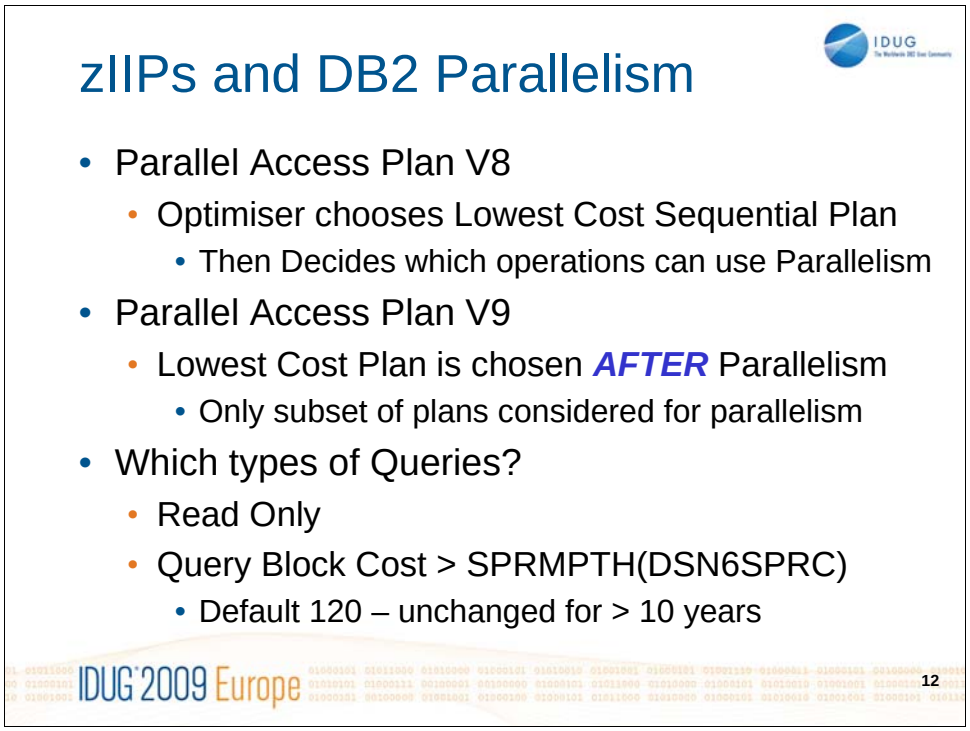
Probably, like many of my customers, most people tried parallelism when it was first introduced in V3/V4, and decided, for various reasons, that it wasn't for them. The main reason for this was that the goal of Parallelism is to ***reduce Elapsed*** times and normally that means a slight ***increase*** in ***CPU***.

Is this reticence to use Parallelism still justified ?? After all, V3 was almost 16 years ago...maybe it's time to give it another chance? And now with a zIIP we may even be able to ***reduce GPP*** use by using parallelism!!

Our goal was precisely this, to evaluate whether we could increase our usage of the zIIP whilst actually reducing usage of the General Purpose Processor and thus reduce our costs by delaying an upgrade.



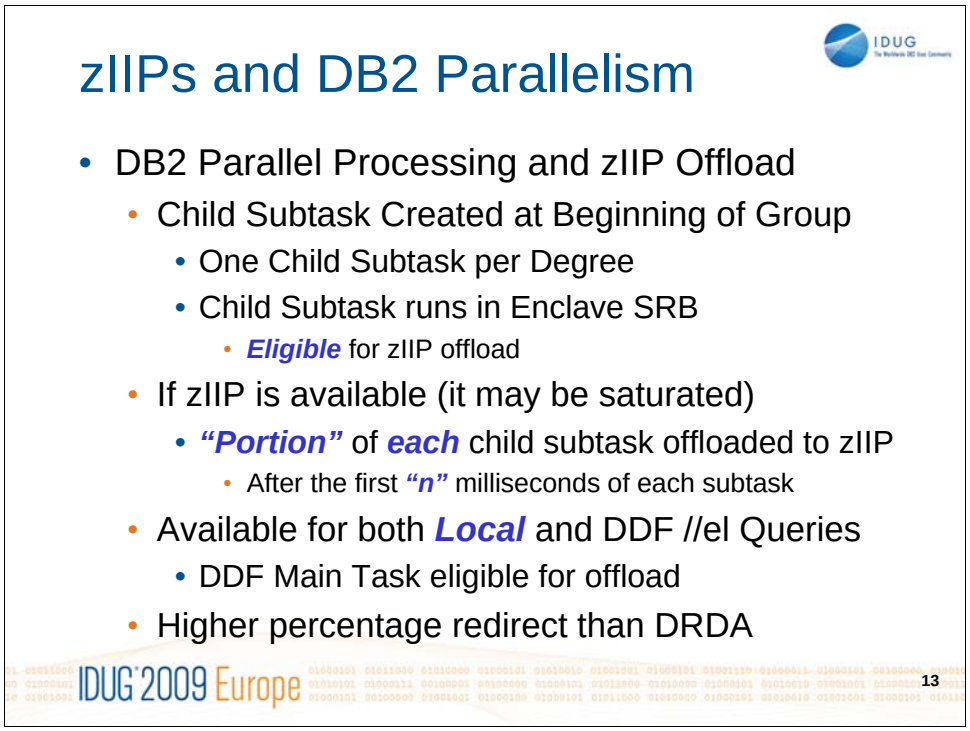

So, now let’s take a look at how the zIIPs work with DB2 Parallelism...




Quick review of how the Optimiser chooses a Parallel Access Plan.

In Version 8 the optimiser will first choose the lowest sequential Plan and then evaluate which operations can be performed in parallel. Whereas in Version 9, the optimiser chooses the lowest cost plan ***after*** having considered plans that exploit parallelism. It is worth mentioning here that only a subset of parallel plans are evaluated so as to reduce the bind/prepare time.

The other important factor when considering parallelism is the impact of the zParm SPRMPTH in macro DSN6SPRC. This has a default value of 120, which has been the default since its introduction more than 10 years ago. However, in those 10 years I/O subsystems and Processors have become considerably faster, and we have 64-bit addressing for memory management... so it is reasonable to question whether 120 still an acceptable value... Unfortunately that is beyond the scope of this presentation.



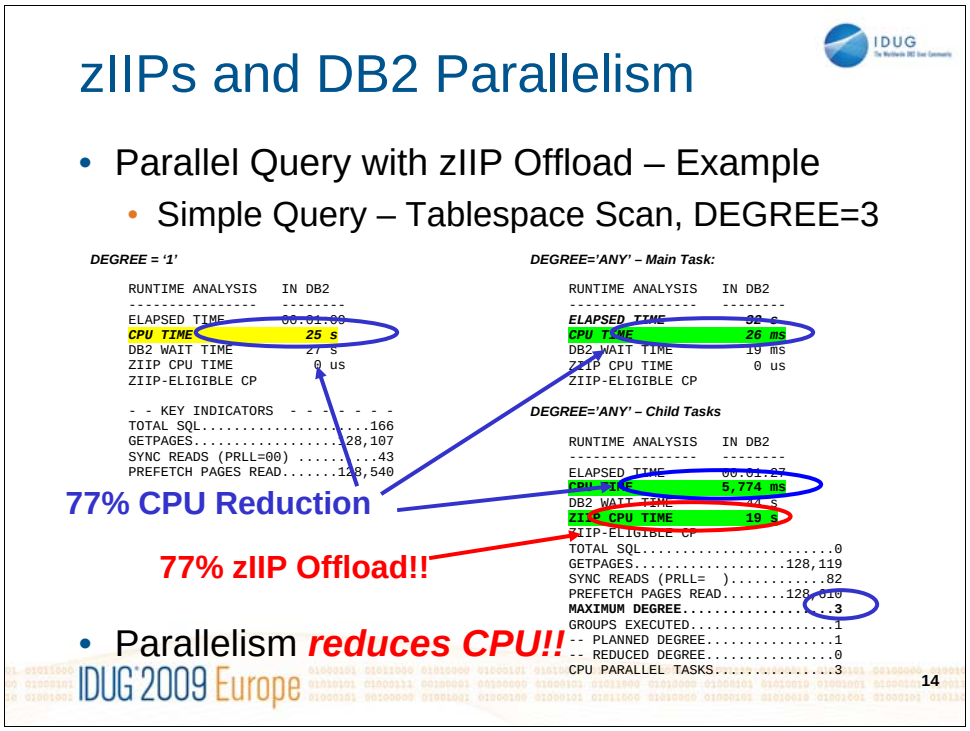

Once we actually get a parallel Access Plan then how does that fit in with the zIIP?

Well, as you know, each parallel group has its own Parallel Degree, and at the start of the Parallel Group one child subtask per degree is created. i.e. a parallel group with DEGREE=10 would create 10 child subtasks. Now, as each of these child subtasks runs in an Enclave SRB then they become eligible for offload to the zIIP.

If a zIIP is available to be used, i.e. it's not being fully utilised, then after the first "n" milliseconds of processing of each individual child subtask, z/OS will redirect a "portion" of the work from the GPP to the zIIP. The size of this portion offloaded and the threshold of the number of milliseconds that do not get redirected are not officially documented by IBM.

It is worth remembering that the zIIP offload is available to ALL queries and not just queries coming in over DRDA. Also, for a parallel query coming in over DDF the Main task will also get a portion redirected to the zIIP as it is eligible for offload as a DRDA process.

Lastly, it should be emphasised that Parallel child subtasks will get a higher percentage redirect than DRDA processes.




So, that's the theory lets look at it in practice...

We tried a very quick test to try to see what the actual offload would be like. We used a very simple query and ran it with DEGREE(1) and then DEGREE(ANY).

When ran with DEGREE(ANY) we got an actual degree of 3.

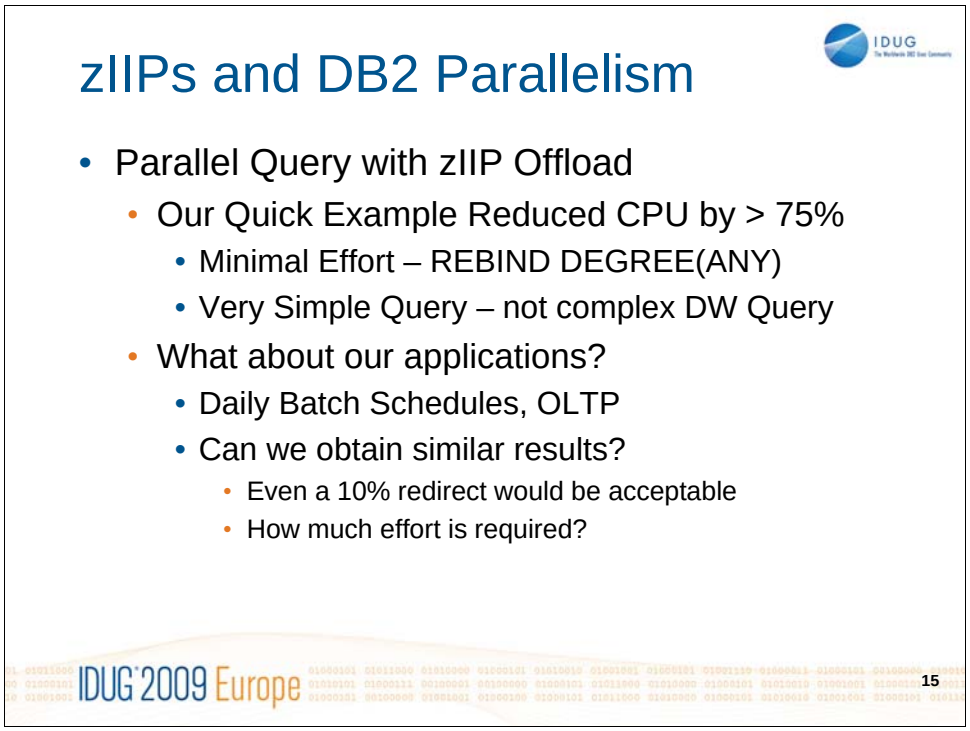
The query used was:

```
SELECT  COUNT(*), COD_CITO
FROM    DBA1.TVPIF2TG
GROUP   BY COD_CITO
WITH UR ;
```

And as can be seen by the figures on the slide we got some fantastic results which confirmed all the theory:

- The Parallel Child Subtasks got an offload of almost 80% to the zIIP;
- The net overall impact on the GPP was a 77% reduction!!

Running parallel had ***reduced*** our CPU !!!




So if in our simple test we obtained such good results with minimal effort and a non-complex, non DW style query, what would happen if we tried parallelism on our applications?

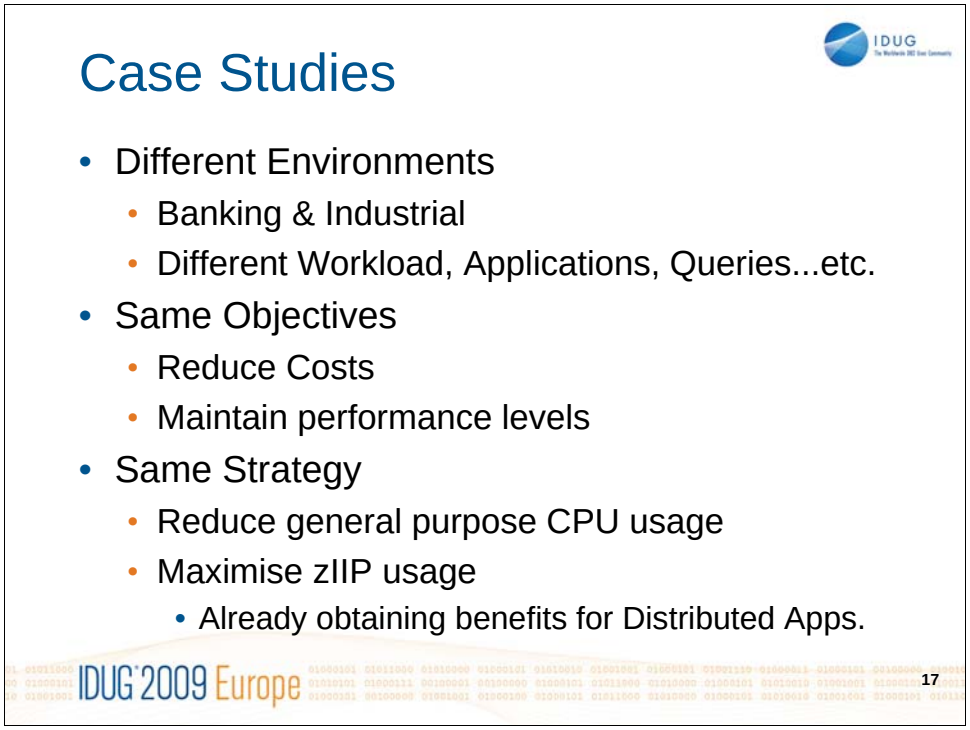
Could we achieve the same magnitude of savings on our regular Batch/OLTP applications? And if so, how much effort would be involved – would it be just as simple as rebinding with DEGREE(ANY)?

# Time to go Parallel on a zIIP?

- Introduction
  - What, Why and How of zIIPs
- zIIPs and DB2 Parallelism
- Case Studies
- Conclusions



So lets see what we did…



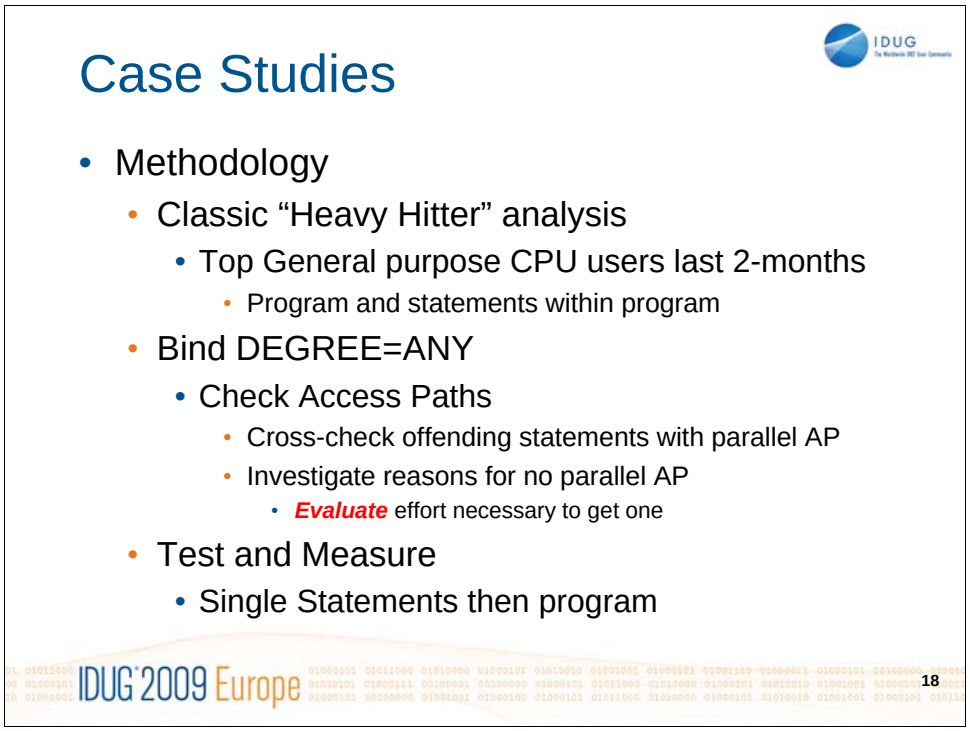

The following case studies come from two of my customers from completely different industries with completely different environments.

Hence, the workload, applications, database designs and more importantly, the types of queries are completely different.

The workload and applications analysed are from OLTP / BATCH applications as neither of the customers had a significant amount of Data Warehouse or Business Intelligence type of applications running on the mainframe. Also, one of the customers was using 2-way data sharing.

However, the two customers both shared the same objective: REDUCE COSTS whilst maintaining the current satisfactory performance levels.

And as both customers had already seen the significant benefits of the zIIP processor for their distributed applications they both wanted to ensure they were maximising their zIIP usage.




Before going further just a few words on the methodology that we used.

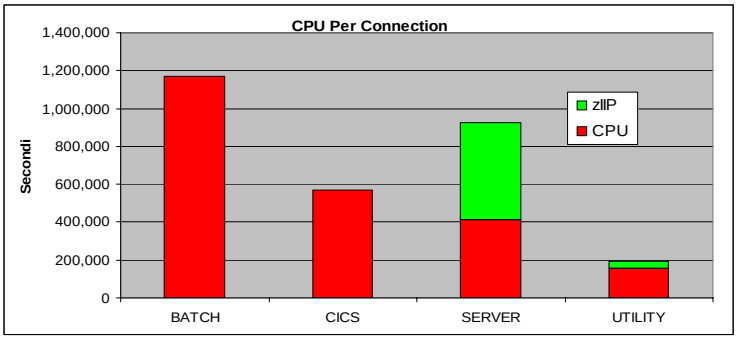
We used a classical Heavy Hitter analysis to identify the top general purpose CPU users over a 2 month period. For these programs we then identified the offending statements within the program.

After rebinding the program into a dummy collection with DEGREE(ANY) we verified whether the offending statements obtained a parallel Access Path; if so we tested each statement individually, otherwise we investigated the reasons for the optimiser not choosing a parallel Access Plan and evaluated the effort necessary to generate one i.e. query rewrite, table design etc. Ideally, we wanted to obtain as much parallelism as possible with the least effort, just by rebinding; however, in certain cases we would also be willing to invest more effort if we thought the return would be worth it.

Once we were happy with each individual statements performance we then tested the whole program.

# Case Studies

- Customer 1 – Current CPU / zIIP Usage




| CONNECTION | CPU | zIIP | % Total CPU | % zIIP |
|---|---|---|---|---|
| BATCH | 1,172,677 | 3 | 50.79% | 0.00% |
| CICS | 568,851 | 0 | 24.64% | 0.00% |
| SERVER | 411,610 | 510,591 | 17.83% | 55.37% |
| UTILITY | 155,926 | 36,538 | 6.75% | 18.98% |



So, this was the situation with the first Customer.

As can be seen by the graph, this customer was already getting great use out of the zIIP for their distributed applications.

In fact, over 55% of their CPU usage was redirected to the zIIP and when factoring in the zIIP conversion factor (the zIIP processor is faster than the general CPU processor so one second on the zIIP is worth more on the main processor) this customers distributed application usage is almost equivalent to their Batch workload.

So, our first goal for this customer was to try to redirect some of their batch workload off on to the zIIP.

You can also see from the graph that we were also getting a respectable 18% offload for utility work – as documented by IBM.

# Case Studies

- Customer 1
  - Top 10 Batch Programs:

| Program | CPU | % Total CPU | Cumulative CPU |
|---|---|---|---|
| ProgramX | 70,982 | 2.97% | 2.97% |
| ProgramY | 54,638 | 2.29% | 5.26% |
| **Program1** | 37,325 | 1.56% | 6.82% |
| ProgramZ | 36,353 | 1.52% | 8.34% |
| **Program2** | 29,380 | 1.23% | 9.57% |
| **Program3** | 26,087 | 1.09% | 10.66% |
| **Program4** | 25,836 | 1.08% | 11.74% |
| **Program5** | 24,381 | 1.02% | 12.76% |

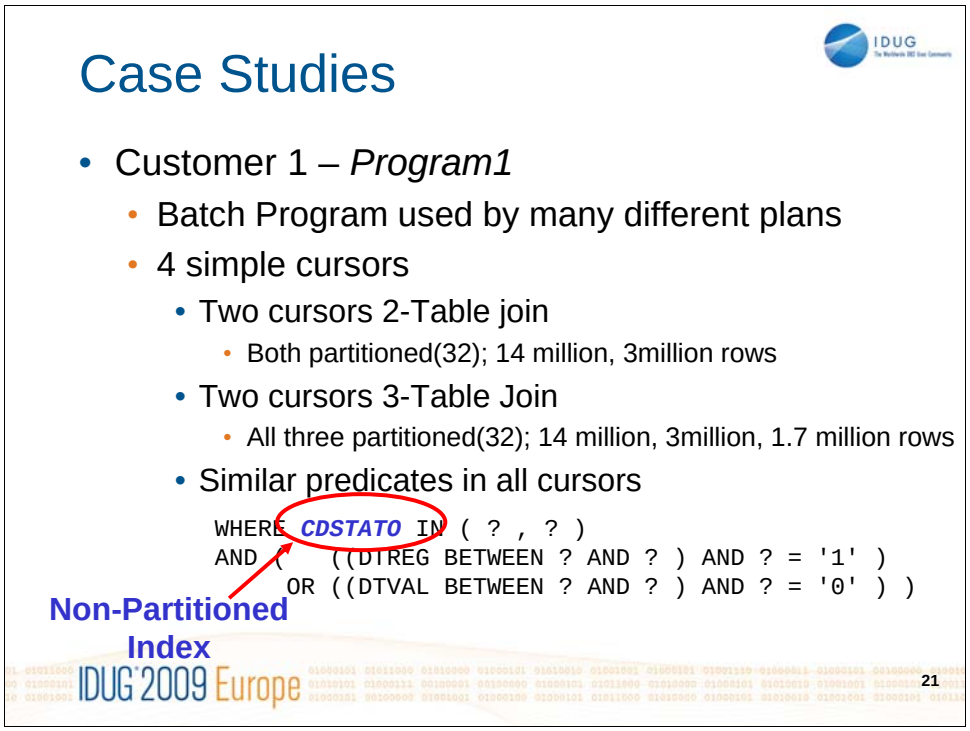


So, here are the top 8 users of General Purpose CPU for this customer.

Three of the programs in the list, programs X, Y & Z, were already under analysis and had other problems which could be fixed without resorting to a parallel Access Path, so our analysis concentrated on the remaining 5 programs which we thought had already been tuned as well as possible.

Remember our objective is to see whether we can offload some of the CPU usage to the zIIP.

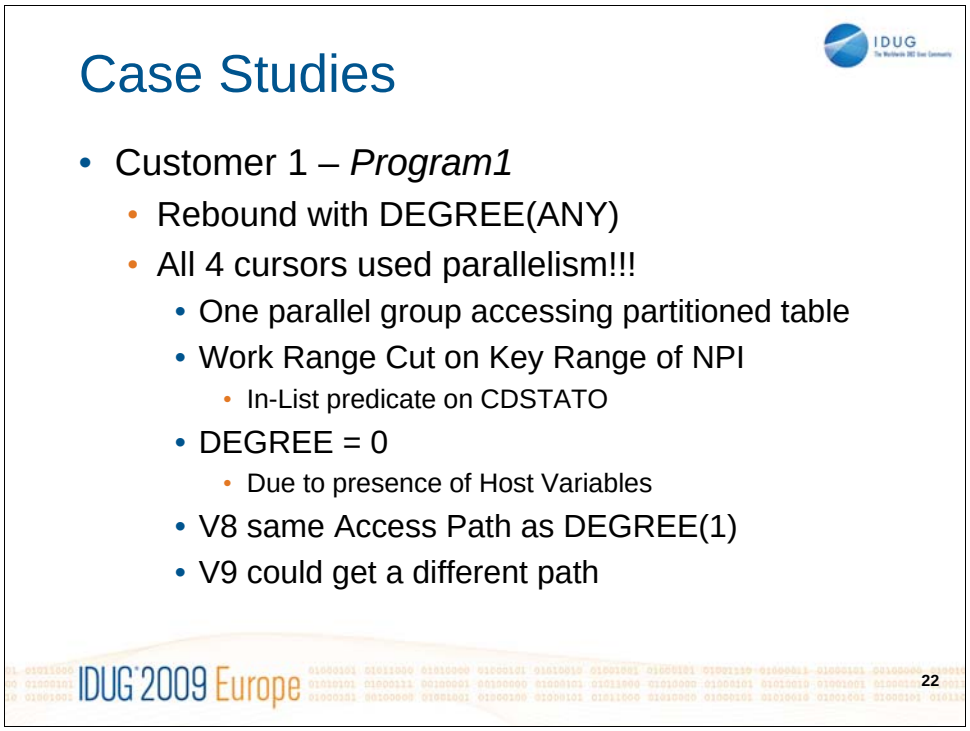
So lets start by taking a look at Program1




Luckily for us, Program1 was relatively straightforward: it only had four cursors which were all very similar and contained the same predicates:

- All of the queries contained an IN predicate on the leading column of a non-partitioned index;
- Two of the cursors were a 2-way join of 2 partitioned tables;
- The other 2 cursors were a 3-way join of the same 2 tables plus a non-partitioned table.

One last thing of interest to note is that the program was used by many different plans.




So the first thing we did was to rebind the program into a dummy collection with DEGREE(ANY) and amazingly all four of the cursors used Parallelism for their Access Paths!!!

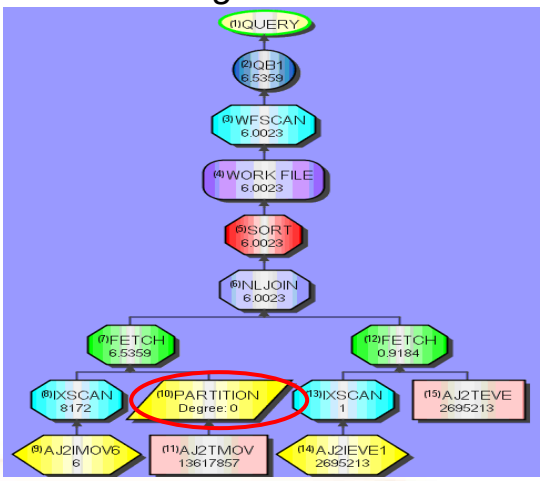
All the cursors had one parallel group accessing the leading partitioned table in the join and the work ranges were cut based on the key values of the NPI on the column CDSTATO – identifying the status of the row.

One thing to note is that due to the fact that the IN-List predicate used host variables and not constants, the actual DEGREE of parallelism will be determined at run-time and is not externalised at Explain – hence the DEGREE = 0.

And of course, as was to be expected as we are in V8, the Access Path was identical to the Access Path with DEGREE(1) apart from the use of parallelism; in Version 9 this may not be the case as the optimiser determines the plan with the lowest cost ***after*** considering parallelism for a subset of the possible plans.

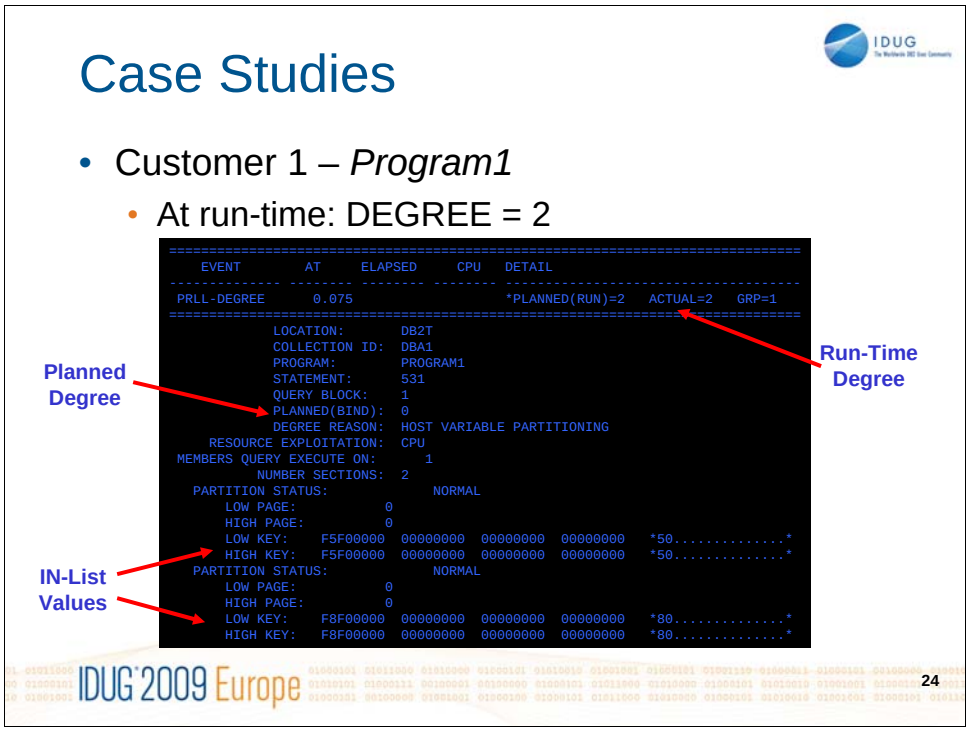
# Case Studies

- Customer 1 – *Program1*






... And here's the access path of one of the cursors from Visual Explain where you can see the DEGREE = 0 due to the host variables in the static SQL.




As could be expected, at actual run-time we got a Degree of 2.

In fact from IFCID 221 we can see confirmation and reason of the DEGREE determination both at Bind Time and at Run-Time:

PLANNED(BIND) = 0

REASON = Host variable partitioning

PLANNED(RUN) = 2

ACTUAL = 2

From IFCID 221 we also get information about each partition or parallel task with its page or key range; in our case this was key range based on the actual values used in our IN-LIST predicate. We also see any information about degradation of parallelism(PARTITION STATUS) , for example if the partition group is empty.

One last thing, remember that if we have a *planned* DEGREE > 0 then the page or key ranges for each partition group are externalised by EXPLAIN in the DSN_PTASK_TABLE.

# Case Studies

- Customer 1 – *Program1*
  - Results: daily ***PACKAGE*** data for one Plan

| DATE | SQL | Elapsed | CPU | zIIP CPU | % zIIP | Getpages |
|---|---|---|---|---|---|---|
| 12/02/2009 | 20,779 | 319.99 | 98.64 | 0 | 0.00% | 1,835,030 |
| 13/02/2009 | 26,533 | 313.28 | 98.85 | 0 | 0.00% | 1,839,299 |
| 14/02/2009 | 23,892 | 305.08 | 97.95 | 0 | 0.00% | 1,842,356 |
| 17/02/2009 | 21,118 | 308.75 | 98.39 | 0 | 0.00% | 1,844,773 |
| 18/02/2009 | 25,800 | 363.02 | 99.73 | 0 | 0.00% | 1,848,969 |
| 19/02/2009 | 18,628 | 457.89 | 100.18 | 0 | 0.00% | 1,851,253 |
| 20/02/2009 | 22,115 | 325.48 | ***1.5*** | 0 | 0.00% | 1,726 |
| 21/02/2009 | 23,703 | 366.03 | ***1.7*** | 0 | 0.00% | 1,852 |
| 24/02/2009 | 25,460 | 317.39 | ***1.85*** | 0 | 0.00% | 1,982 |



So, after testing, we bound the program in production with DEGREE(ANY) and here are the results!

As you can see from this data we have had an amazing result!! From close to 100 seconds down to just 1 second of CPU!!

And look at the Getpages, down from almost 2 million to almost 2000!!

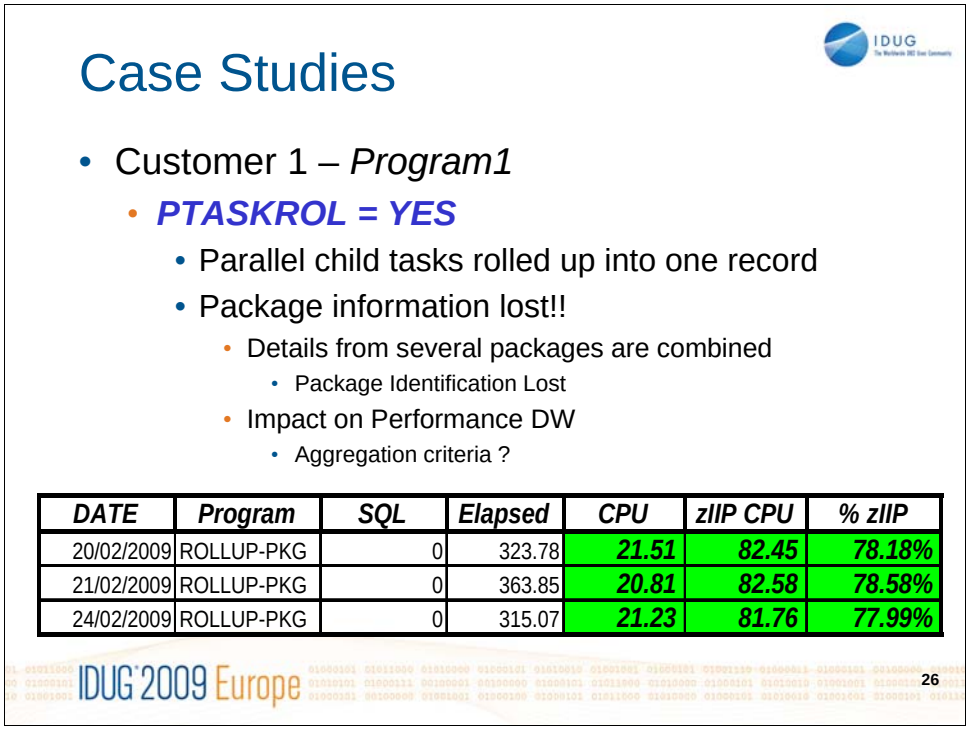
Results like this are wonderful!!!

But hold on, that's almost a 99% reduction, IBM claims up to 75-80% offload, so where has all the CPU gone ?

We had the same Access Path as before...

Also from this data we can see that the "missing" CPU usage hasn't apparently been offloaded to the zIIP!!!

So what's happened ?? The clue lies in the fact that this is ***package*** data...




The answer is simple: We had the zparm PTASKROL = YES!!

That meant that all the parallel child task accounting records were all rolled up into one record. In other words details from several packages may be combined and therefore we lose some package related information, most importantly the name of the package!

If, like us, you have a historical DW of all your package performance data then you should be aware that when using PTASKROL=YES you will get a “new” package for roll-up records; our monitor uses “ROLLUP-PKG” and I think DB2PM uses “*ROLLUP*” .

So, unless you are grouping your package data by some other criteria, for example: PLAN, CORRID etc, by using PTASKROL=YES you will lose a certain level granularity and your performance data will become meaningless. Obviously, one solution to this is to use PTASKROL=NO. But that of course will generate one record per parallel task and if parallelism is to be used extensively may not be a viable solution due to the volume of SMF data.

## Case Studies

- Customer 1 – *Program1*

| DATE | SQL | Elapsed | CPU Main | CPU ROLL-UP | zIIP CPU | % zIIP |
|---|---|---|---|---|---|---|
| 18/02/2009 | 25,800 | 363.02 | 99.73 | 0 | 0 | 0.00% |
| 19/02/2009 | 18,628 | 457.89 | 100.18 | 0 | 0 | 0.00% |
| 20/02/2009 | 22,115 | 325.48 | ***1.5*** | ***21.51*** | ***82.45*** | ***78.18%*** |
| 21/02/2009 | 23,703 | 366.03 | ***1.7*** | ***20.81*** | ***82.58*** | ***78.58%*** |
| 24/02/2009 | 25,460 | 317.39 | ***1.85*** | ***21.23*** | ***81.76*** | ***77.99%*** |

  - Bottom Line:
    - ***79% offload of parallel child tasks to zIIP!!***
    - ***78%*** reduction in main CPU usage!!
    - No significant reduction in Elapsed time
      - Skewed Data Distribution



So, now we have "found" where our CPU went to, we can see the complete results:

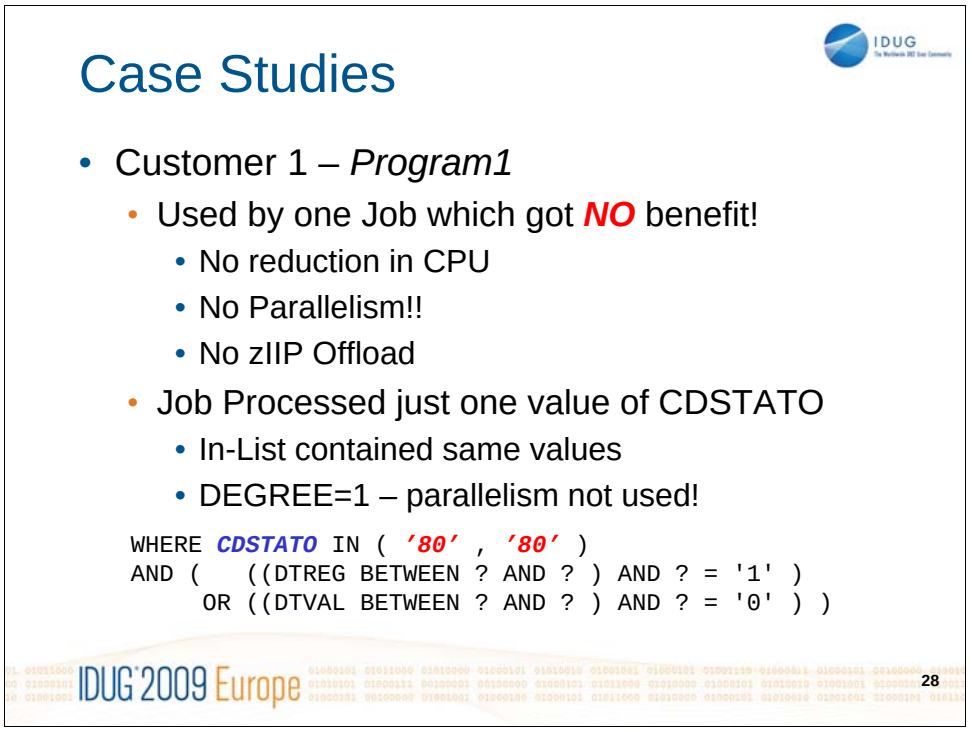
- 79% offload of parallel child task to zIIP
- 78% reduction in general purpose CPU usage!!

All with very little effort – just a rebind and some testing!!

Note 1: In this case the 79% offload to zIIP also translated into a 78% reduction in General Purpose CPU because most of the work was done by the parallel child tasks; we'll see later that this is not always the case.

Note 2: There was no significant reduction in elapsed times due to the skewed data distribution of the column CDSTATO. Remember, a Parallel Group cannot be quicker than its slowest subtask, and in our case, one of the values had 80% of the data and therefore our overall elapsed times did not obtain any noticeable benefit due to parallel processing.

As stated at the start of the presentation, our main objective was to reduce CPU not necessarily to reduce elapsed times, which we have done, and it is reasonable to assume that if there had been a uniform distribution we would have also had a very significant reduction in elapsed times.




After a while, though, one particular job started to puzzle us - we noticed that it was getting no benefit whatsoever from parallel processing:

- Its CPU times showed no change;
- There were no parallel sub tasks;
- And consequently there was no zIIP offload.

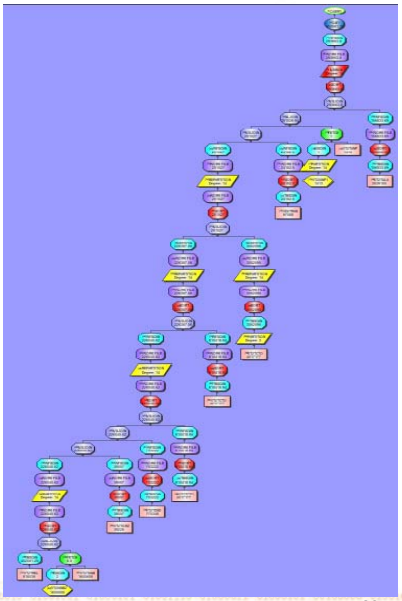
The reason was very simple: that particular job *only processed* just one value of CDSTATO!! So the IN-LIST predicate contained the same value for both host variables and at run-time DB2 could only choose a DEGREE of 1 – i.e. parallelism is not used!

N.B. This will also happen when using a BETWEEN predicate.

To get around this problem it may be worth setting the second variable to a non-existent value to ensure that the optimiser always "sees" two distinct values and will always choose a DEGREE of parallelism > 1.

## Case Studies

- Customer 1 – *Program2*
  - Complex Query
  - 10-Way Join
  - Mix of Join Methods
    - Nested Loop & Merge
  - Several Parallel Groups
    - DEGREE = 14
      - Key range on Join Column
    - DEGREE = 3
      - Page Range





Moving on to the second program in our analysis, *Program2*, this is again a batch program but this time we have a completely different situation.

This time the culprit is a complex query:

- 10 way join – including both inner and outer joins
- Different join methods are also used: Nested Loop and Merge Scan

When the program was Rebound with DEGREE(ANY) we saw 5 parallel groups used in 8 out of the 10 Access Plans in the query:

- 4 of the parallel groups were used for the Joins and were partitioned according to the key ranges of the Join columns;
- The other parallel groups was used to access one of the tables with a degree of 3 based on the page ranges.

So on the surface of it, this would appear to be exactly the type of Complex query that should benefit from zIIP offload ...

# Case Studies

- Customer 1 – *Program2*
  - Results: daily ***PACKAGE*** data

| DATE | SQL | Elapsed | CPU Main | CPU Child | zIIP CPU | % zIIP Child | % zIIP Total |
|---|---|---|---|---|---|---|---|
| 12/02/2009 | 19,109,152 | 4,469 | 1,446 | 0 | 0.00 | 0 | 0 |
| 14/02/2009 | 19,157,744 | 4,922 | 1,437 | 0 | 0.00 | 0 | 0 |
| 18/02/2009 | 19,167,776 | 9,637 | 1,565 | 0 | 0.00 | 0 | 0 |
| 21/02/2009 | 19,185,776 | 3,650 | 1,113 | ***99.30*** | ***262.36*** | ***72.54%*** | ***17.79%*** |
| 24/02/2009 | 19,194,656 | 4,198 | 1,198 | ***104.92*** | ***268.65*** | ***71.91%*** | ***17.10%*** |
| 26/02/2009 | 19,191,968 | 3,629 | 1,184 | ***106.08*** | ***276.01*** | ***72.24%*** | ***17.62%*** |

  - Again good redirect of Child task
  - But lower overall impact



As can be seen by the results, again the parallel child task processing obtains great offload to the zIIP: as with the previous program, more than 70%.

However, in this case, this offload has not had such a big impact on the overall CPU usage of the program. In fact we see that the zIIP offload is only around 17% of the Total CPU.

Why is this??? Let's see....

# Case Studies

- Customer 1 – *Program2*
  - Query and Access Plan seemed ideal Candidates
    - Complex query, lots of parallel groups
  - Intricacies of Parallel Processing
    - Majority of Work done by main task!

| TASK | ELAPSED | CPU | zIIP | #STMTS | GETPAGE |
|---|---|---|---|---|---|
| CHILD | 18 s | 424 ms | 1,429 ms | | 21,164 |
| CHILD | 44 s | 1,822 ms | 6,781 ms | | 71,794 |
| CHILD | 48 s | 1,938 ms | 7,033 ms | | 75,922 |
| CHILD | 54 s | 2,956 ms | 11 s | | 122,705 |
| CHILD | 55 s | 2,490 ms | 9,237 ms | | 98,097 |
| CHILD | 1,123 us | 864 us | 0 us | | |
| CHILD | 2,062 us | 805 us | 0 us | | |
| CHILD | 1,745 us | 602 us | 0 us | | |
| PARENT | 00:57:59 | 00:33:45 | 0 us | 20,671K | 6,196k |



Well, the reason for this is that although the query and Access Plan seemed ideal candidates for zIIP offload, most of the actual processing was performed by the Main Task and not the Parallel Child Subtasks

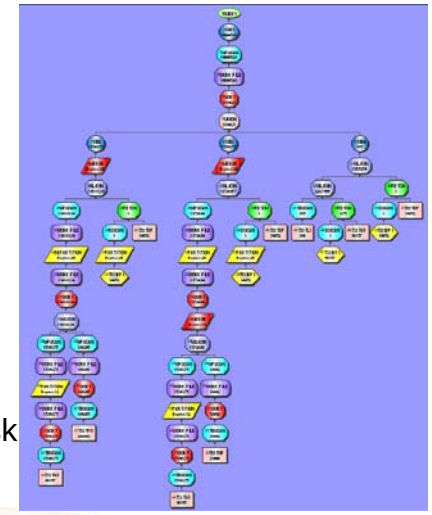
In fact if we look at the breakdown of the single child sub tasks, we can see that some of them actually do quite a bit of work. One of them performs more than 100K getpages, and as expected gets most of the work offloaded to the zIIP.

However, some of the sub-tasks don't do anything at all and as we can see, it is the main task which performs the bulk of the work: more than 6million Getpages... And unfortunately that will not get offloaded.

So the lesson to be learned from this is that even though we may get lots of parallel Access Plans in our explain output, due to the intricacies of parallel processing we may not get as much benefit as expected.

# Case Studies

- Customer 1 – *Program4*
  - Similar to Program2
    - Complex Query
      - Union + 3-Table Joins
      - Several Parallel Groups
  - Child tasks
    - Good Offload
  - Low overall impact
    - <10% Main CPU
    - Again, Mostly Main Task
    - No regression

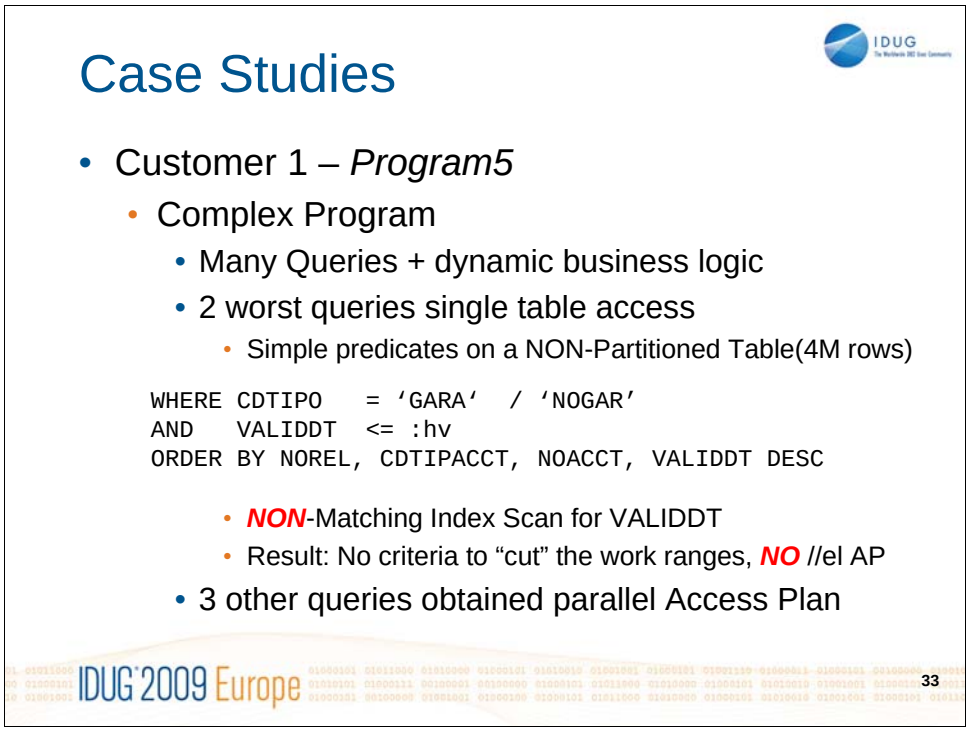




*Program4* was very similar to *Program2*, in the fact that it also had one very complex query which again appeared to be ideal for zIIP offload.

But, as with *Program2*, we got the expected offload to zIIP for the Child Sub Tasks but, again, most of the work for the query was performed by the Main Parent Task and therefore we again had fairly disappointing results getting less than a 10% reduction main CPU use.

However, another important thing to note is that although the actual work offloaded was disappointing we did not suffer any regression in performance.




The last program we analysed, Program5, was a slightly more complex program; it had many more queries and was executed both in Batch and from CICS many times during the day, therefore the business logic meant that different queries were executed each time the program ran.

The two worst queries in the program were very simple queries to a non-partitioned table. Unfortunately, however, the queries could not benefit from a parallel access plan:

- The table was not partitioned so DB2 could not "cut" the work ranges according to the page ranges of the data;
- The access path was a non-matching index scan so DB2 could not "cut" the work ranges according to the key ranges of an Index.

Obviously, the solution would be to create an Index on the columns involved in the query both to obtain a Matching access but also to enable DB2 to "cut" the work ranges on the index keys. However, so far it has not been possible to do that.

Three other queries in the program obtained a parallel Access Plan, so we still decided to Bind the program with DEGREE(ANY).

# Case Studies

- Customer 1 – *Program5*
  - Results: daily ***PACKAGE*** data

| DATE | SQL | Elapsed | CPU Main | CPU Child | zIIP CPU | % zIIP Child | % zIIP Total |
|---|---|---|---|---|---|---|---|
| 16/02/2009 | 7,118,099 | 1,361 | 445 | 0 | 0.00 | 0 | 0 |
| 17/02/2009 | 7,519,176 | 1,436 | 473 | 0 | 0.00 | 0 | 0 |
| 18/02/2009 | 7,351,206 | 1,501 | 465 | 0 | 0.00 | 0 | 0 |
| 19/02/2009 | 7,379,232 | 1,708 | ***471*** | ***15.17*** | ***46.84*** | ***75.54%*** | ***8.79%*** |
| 20/02/2009 | 7,399,912 | 1,465 | ***465*** | ***15.49*** | ***49.81*** | ***76.28%*** | ***9.40%*** |
| 23/02/2009 | 7,426,542 | 1,502 | ***448*** | ***14.87*** | ***46.42*** | ***75.74%*** | ***9.12%*** |

  - Good child redirect to zIIP – around 75%
  - Overall net main CPU ***increased***!!
    - Parent task + parallel child tasks



These are the results of binding *Program5* with DEGREE(ANY).

As can be seen, again the work done by the parallel child tasks gets the expected offload to the zIIP – around 75%. However, this offload accounts for less than 10% of the main CPU use.

The other interesting thing to note here, is that when summing the main CPU use for both the parent tasks and the child tasks we actually get a net ***increase*** in CPU use compared to binding the program with DEGREE(1)!!

# Case Studies

- Customer 1 – *Program5*
  - 3 queries used Parallelism
    - Short running & executed many times
      - Child tasks not getting offloaded to zIIP
        - First 'n' ms of work on child task is not offloaded
        - Cost of running and coordinating parallel child tasks outweighs benefit

| TASK | ELAPSED | CPU | zIIP | #STMTS | GETPAGE |
|---|---|---|---|---|---|
| CHILD | 16 ms | 721 us | 0 | | 6 |
| CHILD | 21 ms | 1,022 us | 0 | | 7 |
| CHILD | 35 ms | 954 us | 0 | | 7 |
| CHILD | 35 ms | 848 us | 0 | | 7 |
| CHILD | 49 ms | 1,014 us | 0 | | 7 |
| CHILD | 50 ms | 865 us | 0 | | 6 |



The queries that obtained a Parallel Access Plan were, most of the time, short running queries which were executed hundreds of times during the day.

The result of this was that although the query was run in parallel the work done by the child tasks was not exceeding the limit for zIIP offload. Remember work is offloaded to the zIIP after the first 'n' milliseconds. And the net result was that the overhead in running and coordinating the child tasks outweighed any benefit we got from the zIIP offload.

# Case Studies

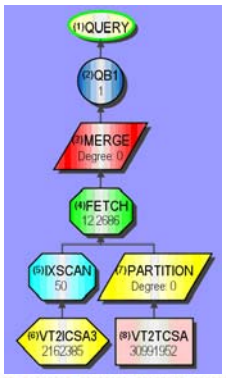
- Customer 1 – *Program3*

```
SELECT  SUM(...)
WHERE   NOCARTA = :hv
  AND   TIPOREC = 'P'
  AND   IMPORTO <> 0
  AND   SWITCH = 'S'
```

| Attribute | Value |
|---|---|
| Input RIDs | 3.0991952E7 |
| Index Leaf Pages | 89907 |
| Matching Predicates | Filter Factor |
| NOCARTA=(EXPR) | 1.606188561709132E-6 |
| Scanned Leaf Pages | 1 |
| Output RIDs | 50 |
| Total Filter Factor | 1.6061886E-6 |
| Matching Columns | 1 |

| Attribute | Value |
|---|---|
| Type | SELECT |
| CPU Cost (ms) | 2 |
| CPU Cost (su) | 18 |
| Cost Category | A |

SPRMPTH=120






The more observant amongst you will have noticed that I skipped Program3...well, most of the CPU used by Program3 was due to one simple SELECT:

- Matching Index access to a 30 million row, 26-partition table.

As can be seen from the Visual Explain report, the index provided excellent filtering, in fact the optimiser expected to qualify 50 rows from the index, and the overall CPU Cost forecast was 2ms... So a very low-cost, short running query.

One thing we should note at this point is that in our zparms we had SPRMPTH set to the default value of 120, yet despite that, surprisingly we still managed to obtain a Parallel Access Plan- with DEGREE=0 due to the presence of the host variable.

So how did it perform ???

# Case Studies

- Customer 1 – *Program3*
  - Program ran Several times during the day
    - Query executed a few times
    - ***No benefit***: child task not doing enough work
      - ***No regression***

| TASK | ELAPSED | CPU | #STMTS | GETPAGE |
|---|---|---|---|---|
| PARENT | 0.02.38 | 2,335 ms | 40,197 | 4,621 |
| CHILD | 283 ms | 19 ms | | 107 |
| PARENT | 0.02.35 | 2,221 ms | 40,204 | 4,708 |
| CHILD | 518 ms | 55 ms | | 382 |

  - Night run normally 3 hours... With Parallelism:

| TASK | ELAPSED | CPU | #STMTS | GETPAGE |
|---|---|---|---|---|
| PARENT | 5.46.55 | 0.45.22 | 3,988K | 11,597K |
| CHILD | 3.52.09 | 0.52.12 | | 24,232K |



Here are the results of the first day for Program3:

During the day the program ran several times, however, our query was only executed a few times during each run. Due to the fact that the child tasks had very little work to do, the program obtained little or no benefit. Also, in this case, the overhead of parallelism had a less pronounced effect than the previous program we looked at due to the fact that each run was very short. So, although we obtained no benefit we didn't suffer any noticeable regression.

However, during the night batch schedule, the program behaved completely differently...in fact, normally, the program ran in around 3 hours using about 45 minutes of CPU performing millions of statements....with our query running in parallel however, the program now lasted almost 6 hours and used twice as much CPU !!!

# Case Studies

- Customer 1 – *Program3*
  - Night Run: Query Executed > 100,000 times
    - Generated > 4.5million Parallel Child Tasks

```
RUNTIME ANALYSIS   IN DB2     IN APPL.      TOTAL      %IN DB2(=)     TOTAL(*)
----------------   --------   --------   --------     0 ...25...50...75..100%
ELAPSED TIME       03:51:27      42 s    03:52:09     ! ==================* !
CPU TIME           00:51:42      30 s    00:52:12     ! ====                !
DB2 WAIT TIME      02:18:53                           ! ===========         !
ZIIP CPU TIME      00:01:36              00:01:37     ! <                   !
ZIIP-ELIGIBLE CP                         3,425 ms
- - - - - - ACTIVITY - - - - - -      - - - - - KEY INDICATORS  - - - - - - -
TOTAL SQL.....................0     I/O RSP: SYNC= 1,212 us, ASYNC= 7,723 us
GETPAGES.................24,232K     LOCK SUSPENSIONS = 17,533
SYNC READS (PRLL=  ).......4,537K     PARALLELISM ROLL UP RECORD, TASKS=4,783K
```

    - Excessive service wait in Parent Task

```
UNIT SWITCH EVENTS                                    !                    !
..OTHER          5,213K  1,858 us 02:41:24   46.52 ! *********            !
```



The reason for this excessive regression in the performance of the program was due to the fact that our query was executed several hundreds of thousands of times and generated more than 4.5 million parallel child tasks, each one of which providing little or no benefit to the processing, as can be seen by the low zIIP offload:

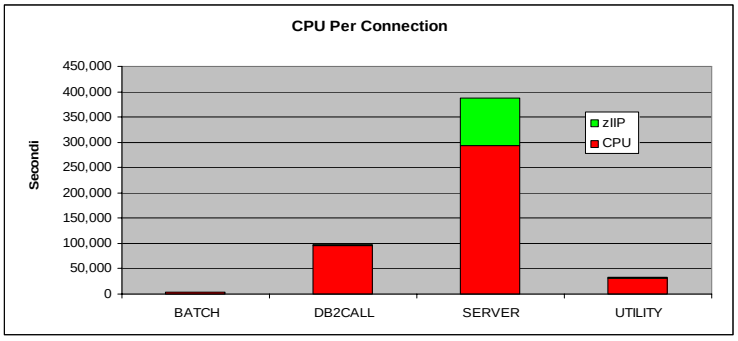
- only 1.5 minutes of CPU actually offloaded to the zIIP.

Apart from the increase in CPU, the other major impact was the doubling of the elapsed time. Looking at the Parent Task accounting data most of this can be accounted for by the excessive Service Wait time due to the enormous number of parallel child tasks that had been generated.

Needless to say.....we decided ***against*** running this program with parallelism and rebound with DEGREE=1!!!

# Case Studies

- Customer 2 – Current CPU / zIIP Usage




| CONNECTION | CPU | zIIP | % Total CPU | % zIIP Offload |
|---|---|---|---|---|
| BATCH | 3,745 | 0 | 0.87% | 0.00% |
| DB2CALL | 95,281 | 2,642 | 22.10% | 2.70% |
| SERVER | 292,627 | 94,460 | 67.88% | 24.40% |
| UTILITY | 30,620 | 3,301 | 7.10% | 9.73% |



Now let's take a look at the second case study.

As can be seen by the graph, this customer is predominantly a Distributed shop In fact more than 3/4 of all CPU use (main + zIIP) is for Distributed applications.

What may seem strange however, is that we are only getting a measly 24% offload to the zIIP. Remember, the first customer was getting more than 55% redirect. Well, the reason for this is that this customer is using Stored Procedures for most of its applications. And as the Stored Procedures are all in COBOL and the customer is on Version 8, unfortunately none of the processing is eligible for offload to the zIIP.

So, our goal for this customer was to try to redirect some of their Stored Procedure work off to the zIIP.

Also, note that in this customer we observed only a 10% offload to zIIP for Utility work compared to almost 20% for the first customer.

# Case Studies

- Customer 2
  - Top 10 Stored Procedures:

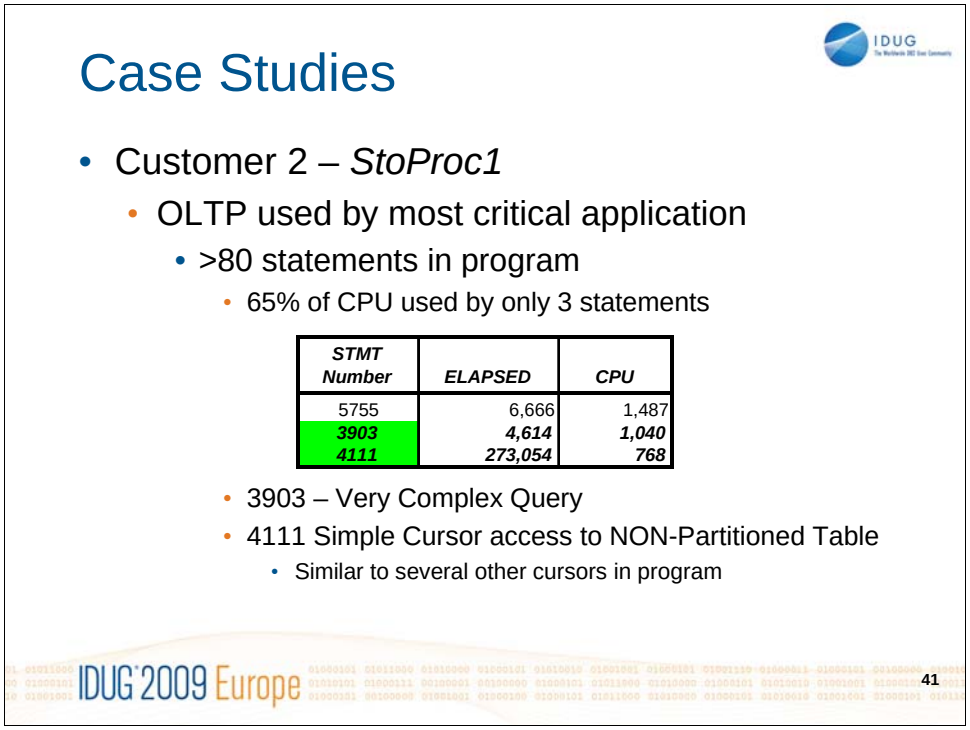
| Program | CPU | % Total CPU | Cumulative CPU |
|---|---|---|---|
| StoProc1 | 75,736 | 17.91% | 17.91% |
| StoProc2 | 16,396 | 3.88% | 21.79% |
| StoProc3 | 14,094 | 3.33% | 25.12% |
| StoProc4 | 11,939 | 2.82% | 27.94% |
| StoProc5 | 11,684 | 2.76% | 30.70% |
| StoProc6 | 11,131 | 2.63% | 33.34% |
| StoProc7 | 10,887 | 2.57% | 35.91% |
| StoProc8 | 10,142 | 2.40% | 38.31% |
| StoProc9 | 5,797 | 1.37% | 39.68% |
| StoProc10 | 5,658 | 1.34% | 41.02% |



So again, we analysed our data to identify the worst Stored procedures – here are the top 10 users of General Purpose CPU for this customer.

And as you can see, in this environment, one Stored Procedure in particular accounts for almost 1/5 of all General Purpose CPU Consumption.

So let's take a look at it...




First of all *StoProc1* was used by the most critical OLTP application at this customer and contained more than 80 SQL Statements.

However, 3 of these 80 statements accounted for almost 70% of all the CPU used by the Stored Procedure.

The first of these statements, stmt number 5755, was already being reviewed by the application programmers so was not part of our analysis. The second, stmt number 3903, was a very complex query performing a Left Outer Join on 21 tables. And the third, stmt 4111, was a fairly simple cursor access to a non-partitioned table. There were also several other cursors, very similar to this last one, accessing the same table but with different predicates to guarantee use of different indexes.

So our analysis concentrated on statements 3902 and 4111...

# Case Studies

- Customer 2 – *StoProc1*
  - REBIND DEGREE(ANY)
    - ONLY one query used parallel Access Plan
      - Stmt 3903 – Complex Query
      - Used in 7 out of the 23 Access Plans
        - 3 //el Groups with DEGREE = 3, 3 & 6
    - Most other queries simple single table Access
      - Low Estimated costs in DSN_STATEMNT_TABLE
      - No access to partitioned tables



A rebind of the program with DEGREE(ANY) showed that just one query obtained a Parallel Access Plan – our statement 3903 the Complex Query; in fact this managed to get 3 parallel groups in 7 out of its 23 Access Plans (see next slide) !!

None of the other queries in the program obtained a parallel access plan.

The main reason for this was that all of the accesses were to NON-partitioned tables, and most of them to the same table as our second query under analysis, stmt 4111.

# Case Studies

- Customer 2 – *StoProc1*
  - Explain Complex Query(3903) – DEGREE(ANY)

| PLN | MET | TNAME | ACC | MTC | IXNAME | IX | SNEW UJOG | SCOM UJOG | PF | ACCS DGID | JOIN DGID | SOSO CINI | MOD | MERGE JOIN COLS |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | OI9006T | R | 0 | | N | NNNN | NNNN | S | 0 0 | 0 0 | 0 0 | | 0 |
| 2 | 1 | PDT771 | I | 1 | PDX771A | N | NNNN | NNNN | | 0 0 | 0 0 | 0 0 | | 0 |
| 3 | 1 | CIT102 | I | 3 | CIX102A | N | NNNN | NNNN | | 0 0 | 0 0 | 0 0 | | 0 |
| 4 | 1 | PD64 | I | 2 | PD64X | N | NNNN | NNNN | | 0 0 | 0 0 | 0 0 | | 0 |
| 5 | 1 | PGT050 | I | 2 | PGX050A | N | NNNN | NNNN | | 0 0 | 0 0 | 0 0 | | 0 |
| 6 | 1 | PD62 | I | 2 | PD62X | Y | NNNN | NNNN | | 0 0 | 0 0 | 0 0 | | 0 |
| 7 | 1 | PD62 | I | 3 | PD62X | N | NNNN | NNNN | | 0 0 | 0 0 | 0 0 | | 0 |
| 8 | 1 | PD62 | I | 3 | PD62X | N | NNNN | NNNN | | 0 0 | 0 0 | 0 0 | | 0 |
| 9 | 1 | PD62 | I | 3 | PD62X | N | NNNN | NNNN | | 0 0 | 0 0 | 0 0 | | 0 |
| 10 | 1 | PD62 | I | 3 | PD62X | N | NNNN | NNNN | | 0 0 | 0 0 | 0 0 | | 0 |
| 11 | 1 | PD62 | I | 3 | PD62X | N | NNNN | NNNN | | 0 0 | 0 0 | 0 0 | | 0 |
| 12 | 1 | PD62 | I | 3 | PD62X | N | NNNN | NNNN | | 0 0 | 0 0 | 0 0 | | 0 |
| 13 | 1 | PD62 | I | 3 | PD62X | N | NNNN | NNNN | | 0 0 | 0 0 | 0 0 | | 0 |
| 14 | 1 | PD62 | I | 3 | PD62X | N | NNNN | NNNN | | 0 0 | 0 0 | 0 0 | | 0 |
| 15 | 1 | PD62 | I | 3 | PD62X | N | NNNN | NNNN | | 0 0 | 0 0 | 0 0 | | 0 |
| **16** | **2** | **PD63** | **I** | **0** | **PD63Y** | **N** | **NYNN** | **NYNN** | | **3 2** | **- -** | **- -** | **C** | **3** |
| **17** | **2** | **PD63** | **I** | **0** | **PD63Y** | **N** | **NYNN** | **NYNN** | | **3 4** | **- -** | **- -** | **C** | **2** |
| **18** | **1** | **PD63** | **I** | **4** | **PD63X** | **N** | **NNNN** | **NYNN** | | **6 6** | **6 6** | **- -** | **C** | **-** |
| **19** | **1** | **PDT742** | **I** | **1** | **PDX742A** | **N** | **NNNN** | **NNNN** | | **6 6** | **6 6** | **- -** | **C** | **-** |
| **20** | **1** | **PGT050** | **I** | **2** | **PGX050A** | **N** | **NNNN** | **NNNN** | | **6 6** | **6 6** | **- -** | **C** | **-** |
| **21** | **1** | **MTT070** | **I** | **1** | **MTX070A** | **N** | **NNNN** | **NNNN** | | **6 6** | **6 6** | **- -** | **C** | **-** |
| **22** | **1** | **PDT767** | **I** | **1** | **PDX767A** | **N** | **NNNN** | **NNNN** | | **6 6** | **6 6** | **- -** | **C** | **-** |
| **23** | **3** | | | **0** | | **N** | **NNNN** | **NNYN** | | **- -** | **- -** | **6 -** | **-** | **-** |



Here is the output of the PLAN_TABLE highlighting the parallel accesses for our complex query.

As can be seen there are 3 separate parallel groups 2, 4 & 6 with DEGREEs of 3, 3 & 6 respectively.

As this cursor was one of the worst statements in the program we were expecting some fairly good results from this use of parallelism...

## Case Studies

- Customer 2 – *StoProc1*
  - Results of REBIND DEGREE(ANY):
    - NO IMPACT whatsoever!!
      - No noticeable CPU/Elapsed Reduction
      - Insignificant offload to the zIIP
      - Most of work done by main task not parallel child tasks
  - Reason for no Parallelism in other queries:
    - No access to PARTITIONED Tables
    - Nearly All Queries Access one table – PRT740
    - What happens if we PARTITION it??



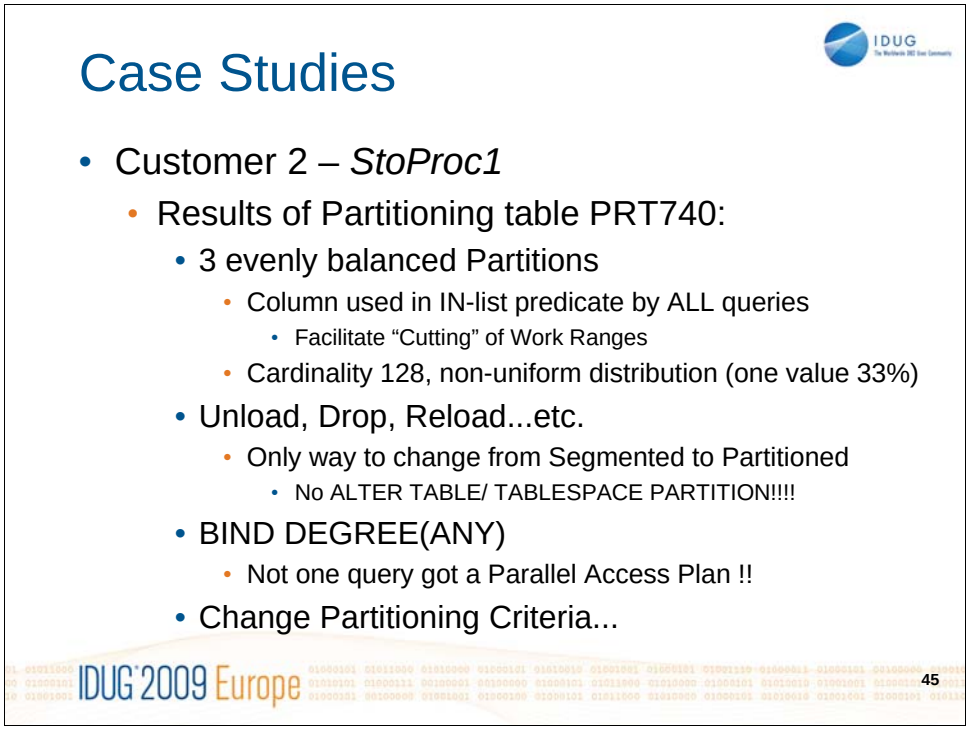
However, despite having a parallel Access Plan, when we ran the Stored Procedure we didn't see any noticeable impact whatsoever in performance measures:

- no reduction in CPU, no offload to zIIP;
- no increase in CPU or Elapsed times;

Luckily we didn't see any performance regression either, but that was small consolation. The reason for this was, again, the fact that the parallel child tasks performed only a small fraction of the work of the overall query, so any gain they provided was balanced out by their overhead.

So, after examining stmt 4111, we realised that the only way we were going to get a Parallel Access Plan was if we partitioned the table!! After all, the query had lots of predicates on indexed columns which would provide plenty of possibilities for the optimiser to "cut" the work ranges and the same was true for most of the other queries in the program.

So we decided to partition our most heavily accessed table within the program – table PRT740.

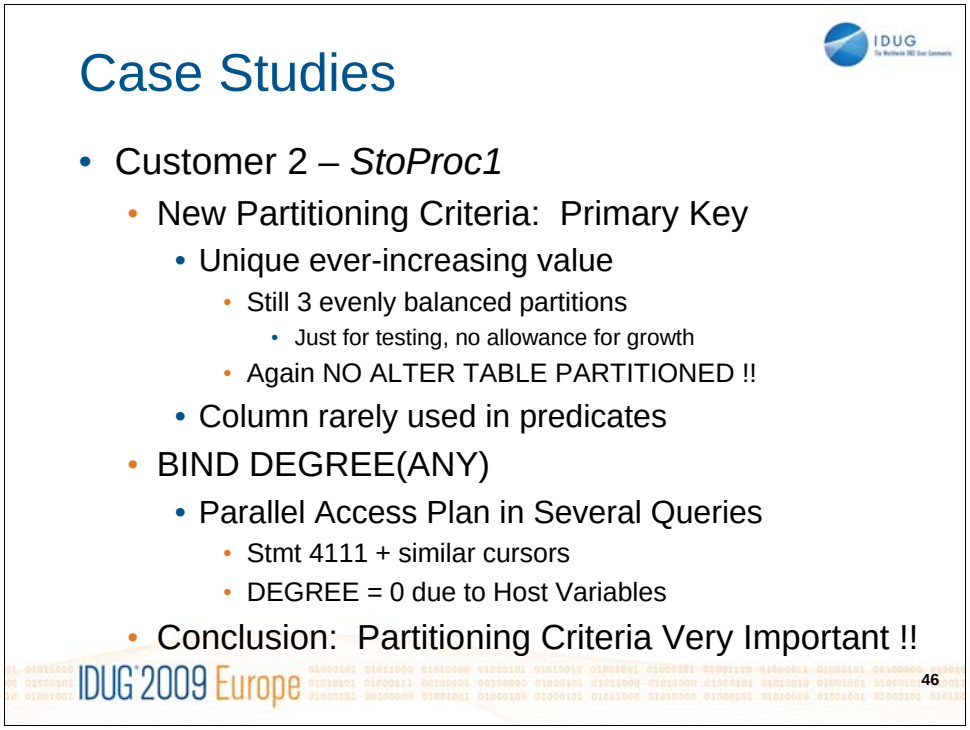



We chose to partition the table PRT740 into 3 partitions (see below) using a column that was used in all of the queries of the program. This column had a fairly low cardinality, 128 in a table of 1.5million rows, and was always used in an IN-List predicate. In fact it was this that made us choose the column; we thought that due to the IN-list predicates the optimiser would be able cut the work ranges on our partitions....

That's what we thought...so after unloading, dropping, reloading etc. we bound the program with DEGREE(ANY)...and were very upset to see that not one query got a Parallel Access Plan!!

Why was this?? We're not sure, maybe it was due to the non-uniform distribution, maybe too few partitions ?? We didn't investigate further... Instead we decided to see whether a different partitioning criteria would have a better impact...

N.B. We only chose three partitions because our goal was to offload to the zIIP, and partitioning was only to facilitate a parallel Access Plan. One last point, wouldn't it be nice to be able to just ALTER a non-partitioned table to be partitioned ?? Instead of having to check existing grants, generate the DDL for views, unload, drop, reload, recreate views, grants etc.




For our new partitioning criteria we chose the Primary Key of the table; a single unique column with ever-increasing values.

Again we created 3 evenly balanced partitions; this was just for the initial testing, to see if we could get any parallel Access Plans at all. Obviously, if successful, due to the ever-increasing nature of the Primary Key we would have to rethink this to allow for growth and ease of maintenance.

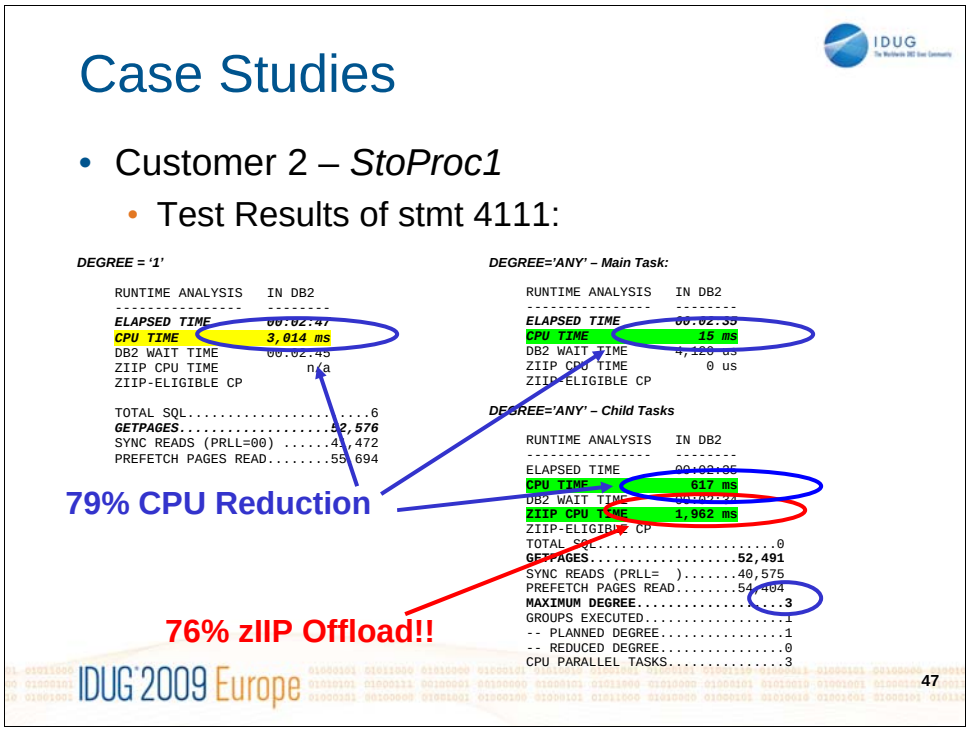
As before, we had to Unload, Drop, Reload etc. ... We *still* haven't got an ALTER TABLE PARTITIONED command...:-(

This time, after BINDING the Stored Procedure with DEGREE(ANY), several of the queries got a Parallel Access Plan, including our stmt 4111. Of course, due to the presence of host variables they all had a DEGREE=0.

N.B. The complex query(3903), wasn't affected as that didn't access this table.

So the main conclusion we can draw from this is just how important the Partitioning Criteria is for enabling Parallelism; as you have seen, just by changing the criteria we had a drastic change in the number of queries that could obtain a Parallel Access Plan.

Now lets see how it performs...




So here are the results of our stand-alone tests of stmt 4111.

As can be seen by the data, at runtime DB2 chose a DEGREE = 3 and we got some excellent results:

- 75% Offload of child task to zIIP
- 79% reduction in main CPU!!
- Slight reduction in overall Elapsed Time

Also, the number of Getpages and Synchronous I/Os remained constant.

So, we were very optimistic that this would bring great benefits and went ahead with our tests for the whole program, simulating our production environment...

# Case Studies

- Customer 2 – *StoProc1*
  - Test Results simulating Production:

***DEGREE = 1***

```
---------ACTIVITY----------------------------------
            TOTAL   AVERAGE   MAXIMUM   MINIMUM
         --------  --------  --------  --------
ELAPSED  00:03:51  4,714 ms      31 s     242 ms
ELP-DB2  00:03:07  3,816 ms      30 s   6,725 us
CPU       6,679 ms    136 ms  1,326 ms  6,100 us
CPU-DB2   6,989 ms    131 ms  1,235 ms  5,566 us
WAITS    00:02:59  3,658 ms      29 s     269 us
ZIIP CPU     76 ms  1,535 us  3,432 us    888 us
ZIIP-DB2     64 ms  1,303 us  2,946 us    764 us
ZIIP-EL. 3,272 us     68 us    551 us      0 us
SQL         39,627       808    18,625        36
GETPAGES   124,365     2,538    11,076         1
SYNC RDS    27,637       491     3,483         0
PFCH PGS   369,249     7,536    48,619         0
UPD/COMT       943        19       134         1
```

***DEGREE = ANY***

```
---------ACTIVITY----------------------------------
            TOTAL   AVERAGE   MAXIMUM   MINIMUM
         --------  --------  --------  --------
ELAPSED  00:05:20  6,324 ms      56 s     289 ms
ELP-DB2  00:03:44  4,566 ms      37 s      14 ms
CPU       5,769 ms    118 ms  1,047 ms  1,657 us
CPU-DB2   5,566 ms    114 ms    969 ms  1,635 us
WAITS        12 s    236 ms  3,084 ms    265 us
ZIIP CPU 2,233 ms     46 ms    480 ms      0 us
ZIIP-DB2 2,222 ms     45 ms    480 ms      0 us
ZIIP-EL.    19 ms    388 us  7,914 us      0 us
SQL         39,627       808    18,625        36
GETPAGES   139,738     2,851    12,639         2
SYNC RDS    33,757       688     3,246         0
PFCH PGS   422,384     8,620    79,909         0
UPD/COMT       878        17       134         1
PARALLELISM - - - - - - - - - - - - - - - - - -
                                 TOTAL  AVERAGE
MAX DEGREE (MAX=   6).....294     6.00
GROUPS EXECUTED.....7,848   160.16
-- PLANNED DEGREE.....131     2.67
-- REDUCED DEGREE.....0     0.00
PARALLEL TASKS.....501    10.22
```

**13% CPU Reduction**

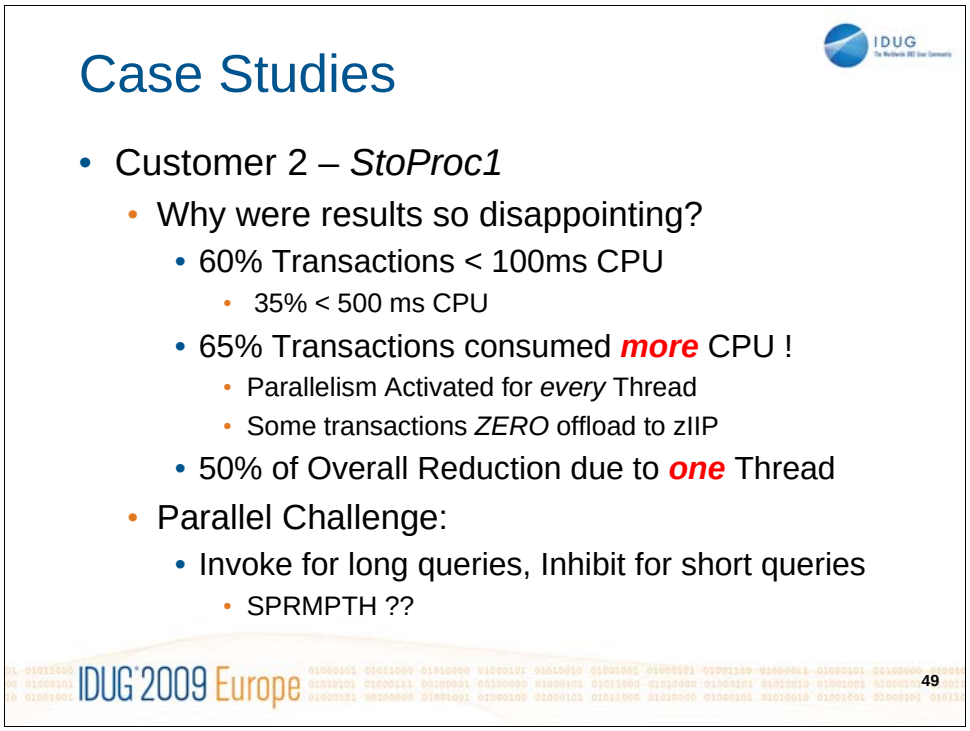
**28% zIIP Offload!!**



When we ran our simulation tests we observed that every single transaction obtained some degree of parallelism executing between 6 and 15 parallel child tasks, with a maximum degree of 6.

However, as can be seen by the data, the results weren't as encouraging as the stand-alone tests; in fact we only managed an average per transaction of:

- 28% offload of child task to zIIP
- 13% reduction in main CPU!
- Slight increase in average Elapsed Times due to higher waits for I/O operations both synchronous and prefetch;



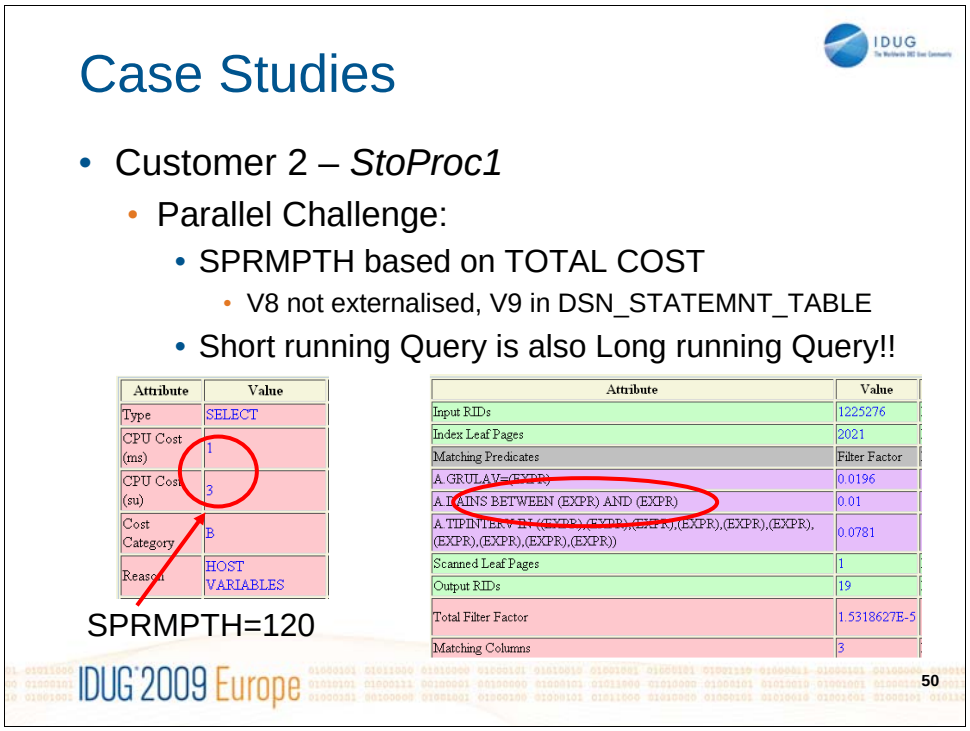

So why were the results so different from our stand-alone tests? Well, one reason was certainly that our stand-alone test was a worst-case scenario whilst the simulation used more realistic search criteria.

However, the main reason was because the Stored Procedure was a "do-everything" program and therefore had drastically varying execution times; in fact, if we look back at the data on the previous chart, we see that the CPU times per thread range from 5ms to over 1 second, however, the majority of the transactions, over 60%, used less than 100ms of CPU.

Despite these short transaction times we still observed Parallelism being invoked for every single thread, in fact it was precisely these sort of transactions that we were hoping to improve by offloading them to the zIIP.

However, unfortunately our results showed that almost 50% of the zIIP offload and reduction in Main CPU was due to just one thread! In fact 65% of the test transactions actually ran using ***more*** main CPU, albeit by very small absolute amounts, and presumably due to the overhead of the parallel processing. The net overall reduction in main CPU use was due to a couple of long running transactions which outweighed these slight increases in the shorter running transactions.

So, ideally we would like parallelism to kick-in for the longer transactions but not for the shorter ones, as in our case...what about SPRMPTH??

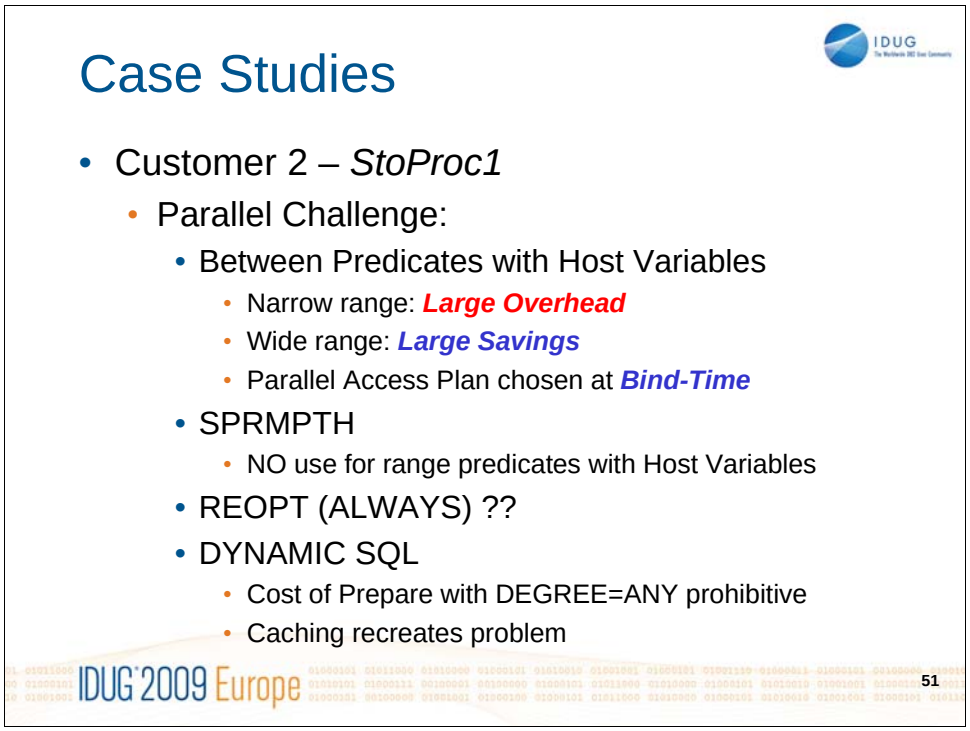



If we take a closer look at our parallel queries the first thing we have to ask is why a Parallel Access Plan was actually chosen in the first place. After all, SPRMPTH was set to the default of 120 and the estimated CPU and SU costs, as seen in the V8 DSN_STATEMNT_TABLE, were extremely low – 1 & 3 respectively, just like the query in ***Program3*** in the first Case Study.

The reason is quite simple, SPRMPTH is actually based on the TOTAL COST of the query, and not on the SU/MS in the V8 DSN_STATEMNT_TABLE!! Unfortunately, we can only guess as to what the TOTAL COST for our queries was as this figure is only externalised in V9.

So, to reduce the overhead of parallelism for the short-running queries we could have increased SPRMPTH. However, in our case, that would also have disabled parallelism for our longer running queries!! Due to the fact that our short running queries were also some of our long running queries!! They were the same query just being run with different values in the host variables!

In fact, all of our queries had BETWEEN predicates which had an enormous effect on the number of rows actually processed by the query: when the range was narrow then the overhead of parallelism increased our processing times; whereas when the range was large then parallelism provided great benefits including the expected offload to the zIIP.




So in our environment, the biggest challenge we faced with trying to use parallelism to offload processing to the zIIP was the fact that most of the queries had several BETWEEN predicates with Host Variables.

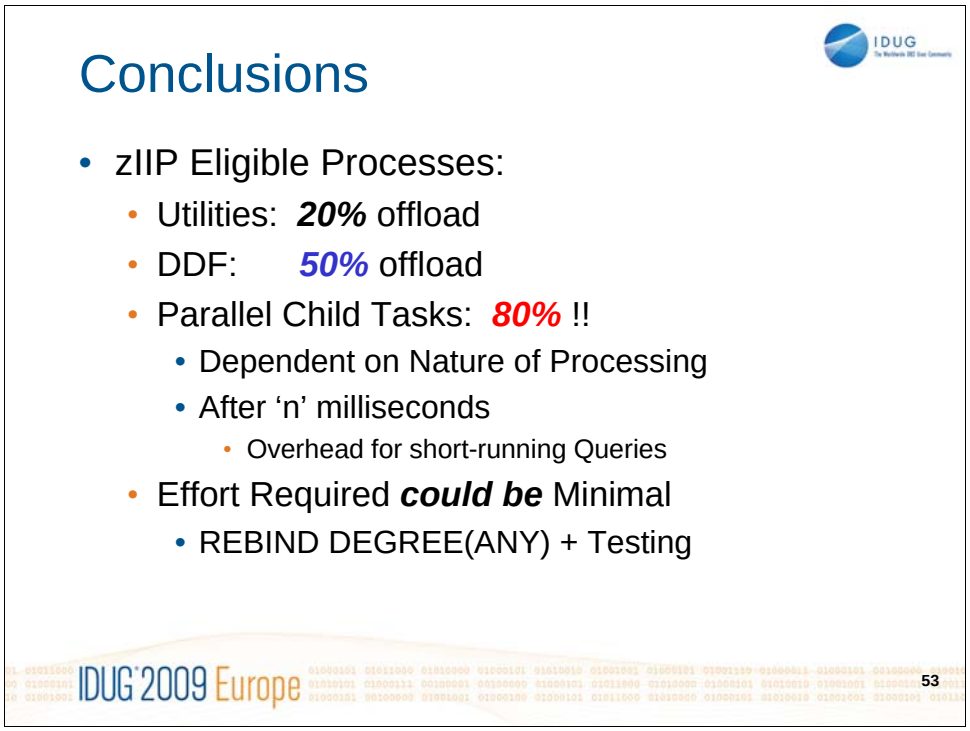
In fact, the main problem with this is that the Optimiser chooses to use a Parallel Access Plan at BIND TIME, when the values of the host variables are unknown. At run-time only the ***degree*** of parallelism is chosen, ***not*** the decision whether to use a parallel Access Plan or not, and for short running queries this causes some overhead. As we have seen on the previous slide, SPRMPTH can be used to inhibit the choice of a Parallel Access Plan, but it has a limited effect on queries with range predicates and host variables, and would also eliminate any benefit for wide range intervals.

One possible solution could be to use REOPT(VARS) but obviously this has its own drawbacks and with our Stored Procedure with more than 80 statements it was not a feasible option.

We also evaluated recoding the queries with Dynamic SQL, but this posed 2 problems; the first was the overhead of DEGREE(ANY) on the PREPARE. For simple queries we saw an overhead of 15-20%; which is great if you get benefit from the Parallel Plan but otherwise it is extremely prohibitive. The second problem is that if dynamic caching is activated then we have just recreated the problem as Parallelism will, or will not, be used based on the values used in the first run of the query.

# Time to go Parallel on a zIIP?

- Introduction
  - What, Why and How of zIIPs
- zIIPs and DB2 Parallelism
- Case Studies
- Conclusions

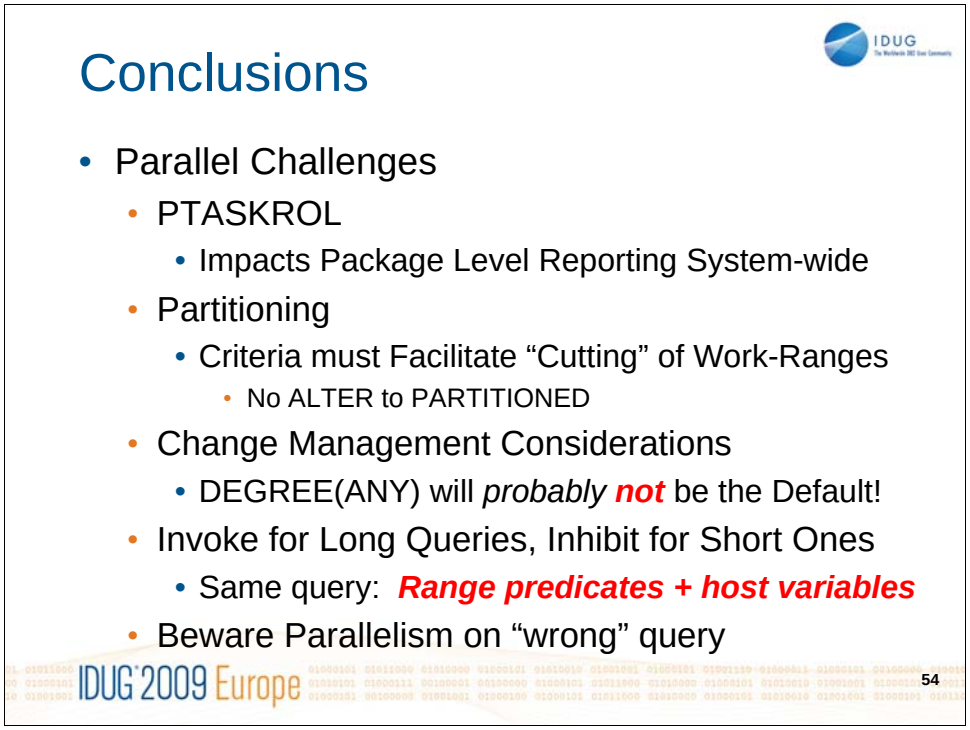



Parallel Processing offers by far the biggest benefit of a zIIP; in fact, as we have seen, we can obtain up to an 80% offload to the zIIP which is far more than we can get for Utility processing or queries coming in through DDF.

The amount of processing actually offloaded to the zIIP is dependent on the amount of processing performed by the parallel child sub-tasks, not on the total processing of the query; in some cases we have seen less than expected offload because most of the processing of the query was performed either sequentially, by other plans within the query, or by the originating parallel task and not the child sub-tasks.

Also, eligible work is offloaded to the zIIP after the first 'n' milliseconds, meaning that even sub-second queries can qualify for the full 80% offload; however, for some short-running queries the overhead of parallelism outweighs any benefit due to the fact that no work is offloaded to the zIIP due to this threshold – again this is at the Parallel Child sub-task level, and therefore is very difficult to predict.

We have also seen that in some cases, the rewards of going parallel and offloading to the zIIP can be obtained with minimal effort – all that may be needed is just a REBIND with DEGREE(ANY); however, in other cases, the effort required may be such that it is not worth pursuing.



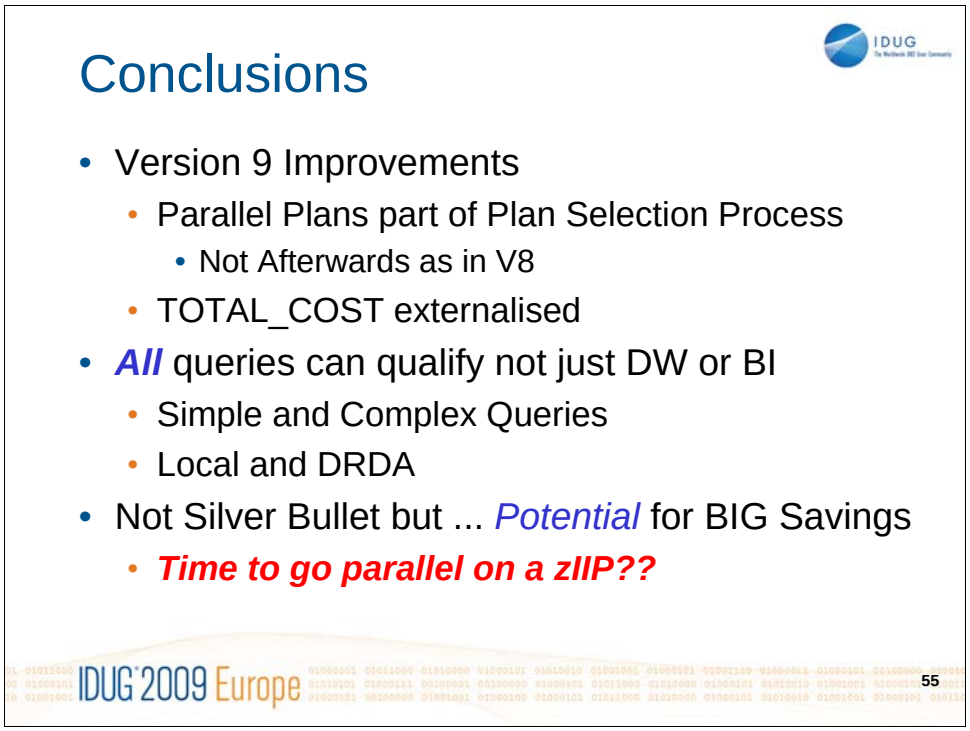

The use of parallelism, however, introduces several new challenges, such as:

• the impact PTASKROL has on monitoring at package level; once we start using Parallelism heavily this will have to be set to YES, to avoid being inundated with irrelevant subtask detail records; this may or may not be a problem, depending on your monitoring methodology;

• we have also seen the importance of the partitioning criteria; with an inadequate criteria the optimiser can't "cut" appropriate work ranges for our queries and will not choose a parallel Access Plan and consequently will not be eligible for offload to the zIIP;

• the use of Parallelism also raises some Change Management issues as it is not a "one-size fits all" solution and DEGREE(ANY) will probably not be applied to all of our programs as the default option, but only to a restricted subset;

• and as we have seen in the case studies, the real challenge is trying to guarantee that parallelism is invoked only for those queries that will gain from it; and as we have seen, difficulties arise if our applications use range predicates with host variables which are set to widely varying values at run-time;




Version 9 offers some interesting improvements; above all the fact that the optimiser will now apply parallelism ***BEFORE*** choosing the lowest cost Access Plan, thus increasing the probability of parallelism being used and hence more possibility to offload to the zIIP; also, the TOTAL_COST is now externalised in DSN_STATEMNT_TABLE which will enable us to identify exactly which queries would be subject to a parallel Access Plan according to our value of SPRMPTH;

We have also seen that all our queries are eligible for offload to the zIIP, not just those coming in from DRDA but local ones as well; and also not just complex long running DW / BI queries but simple and fairly short running ones;

So, to conclude, setting DEGREE=ANY to try to offload all our work onto a zIIP is not a Silver Bullet that is going to solve all our problems, ***however***, it does have the potential to offer some very big savings and provides yet another string to the DBA's "tuning bow"... So if you looked at parallelism 15 years ago, it may be worth taking another look!!

# Thank You!!

Adrian Collett

Expertise4IT s.r.l.

adrian@expertise4IT.com