Modelling parallel Oracle for performance prediction

E.W. Dempster, N.T. Tomov, M.H. Williams (howard@cee.hw.ac.uk), H. Taylor (hamish@cee.hw.ac.uk), A. Burger (ab@cee.hw.ac.uk) and P. Trinder (trinder@cee.hw.ac.uk)
Department of Computing and Electrical Engineering, Heriot-Watt University, Riccarton, Edinburgh, Scotland, EH14 4AS, UK

J. Lü
The School of Computing, Information Systems and Mathematics, South Bank University, Borough Road, London, SE1 0AA, UK

P. Broughton
International Computers Limited, High Performance Technology, Wenlock Way, West Gorton, Manchester, England, M12 5DR, UK

Abstract. The problem of predicting the performance of a parallel relational DBMS for a set of queries applied to a particular data set on a shared nothing parallel architecture without transferring the application to a parallel system is a challenging one. An analytical approach has been developed to assist with this task and has been applied to the ICL GoldRush machine, a parallel machine with a shared-nothing architecture. This paper describes how the Oracle Parallel Server and the Parallel Query Option are modelled by the method and compares the predictions of the model against actual measurements obtained.

Keywords: Analytical modelling, parallel Oracle, performance prediction, parallel database

1. Introduction

The inherent parallelism in relational databases is well suited to parallel computer technology and commercial interest in the use of parallel computers for running relational database systems has been growing [16]. A number of parallel database systems running on different parallel machines have become available from vendors such as Oracle [18], Informix [14], Ingres [1], Sybase [19] and IBM [12]. However, the cost of migrating to a parallel platform is high and the improvement in performance that can be achieved for individual applications on particular configurations is not easy to estimate. For this reason, there is growing interest in performance prediction tools, both for assessing what parallel database platform configuration is required and for tuning the performance of an existing application running on a parallel system.

STEADY [26, 24] (System Throughput Estimator for Advanced Database sYstems) is an analytical parallel database performance estimation tool which can aid a user in selecting a data placement strategy [23], in determining the performance achievable from different DBMS platform configurations, in investigating effects which changes to the system configuration might have on performance and assisting with tuning a particular DBMS configuration to handle a specific application. To do this it estimates performance in terms of system throughput, resource utilisation and response time. Currently, the tool models parallel Oracle [17, 11, 15], Informix XPS [14] and Ingres [20].

This paper discusses how parallel Oracle is modelled. Section 2 provides a review of related work followed in section 3 by a description of the architecture of the STEADY tool. Section 4 contains an overview of Oracle, discusses its various types of parallelism, and introduces Oracle's execution plan through an example query. The modelling of Oracle is then discussed showing how an execution plan is translated into the task block formalism of the tool. Its locking policies, cache management and database background processes are described, followed by a brief comparison between the modelling of Oracle and Informix. Finally some results of measured and estimated performance are compared and some of the problems encountered are discussed.

2. Related Work

Commercially available performance measuring tools can be divided into three categories: measurement tools, combined measurement/prediction tools and performance prediction tools. Measurement tools are used to take measurements on existing systems. These tools monitor and profile the operation of DBMSs and provide information on the way in which the systems are being used in terms of resource utilisation. Examples of such tools are Digital's ECO Tools [7], the Patrol DB-Log master by BMC software [3] and the Ingres

DB_Maximiser by DB LAB PTY Ltd [5]. Unfortunately they are of little value to users who do not have access to a parallel database system.

Tools which combine monitoring and prediction are useful to users who have a parallel DBMS and would like to modify their existing set up. An example of such a tool set is BEZPlus [2], which comprises the Investigator and Strategist tools. It monitors and predicts the performance of NCR Teradata and Oracle environments on MPP machines. The Investigator tool monitors resource utilisation to highlight potential bottlenecks. The Strategist evaluates hardware and software alternatives in order to identify the effect on performance and workload of business growth. Another example is the computer monitoring tool suite Athene from Metron Technology [10] [9]. It comprises five core applications, one of which is the Planner, which uses queueing theory techniques to predict performance. Performance models generated from recorded system performance data, can be held in a database for future reference.

The tools for performance prediction alone vary in complexity from a simple set of cost formulae to a detailed simulation of the DBMS. An example of a simulation-based tool is SMART(Simulation and Model of Application based Relational Technology) [13] developed by Ifatec. It has recently been superceded by its re-engineered successor, SWAP. SMART/SWAP is an advanced tool which is able to model complex Oracle 7 & 8 applications running on GoldRush and Ncube platforms. An example of a tool based on an analytical approach is the DB2 Estimator [8] from IBM, designed specifically for the relational database system DB2 for OS/390 V5 & V6. It runs on a PC and calculates estimated costs using formulae obtained from an analysis of real DB2 code and performance measurements. It aims to have errors, (i.e. differences between actual and predicted values) of less than 20%. Another example is the Oracle System Sizer V3.0 [6], produced by HP, Dell and Oracle. The tool sizes hardware configurations for Oracle database applications. At present there are versions for HP NetServers and Dell PowerEdge servers

under Windows NT. Oracle are planning versions for additional hardware types in the future.

A user who has existing database applications running on a simple sequential machine will in general, be unsure of the performance and cost implications of migrating such applications to a parallel system. If they don't have access to a similar existing parallel system, then measuring and combined measuring and prediction tools are of little relevance. In this case performance prediction tools offer the only solution. Of these, simulation tools, although potentially more accurate, are time-consuming and costly. On the other hand analytical performance prediction tools provide a quicker and cheaper solution although not necessarily as accurate. An ideal solution would be to use analytical tools in the first stage to provide an idea of the most suitable machine configuration and data placement strategies, followed by simulation to provide a more accurate prediction. The tool set described in this paper, STEADY, was designed as an analytical tool for use in this first stage.

3. STEADY

STEADY is a tool set which can be used to predict the performance which would be obtained when a particular set of transactions is executed on a specific parallel database laid out following some particular data placement strategy.

The tool set takes as input details of the relations (database schema), the data placement strategy to be used, the DBMS configuration and the execution plans of SQL queries, represented as annotated query trees. The query trees capture the order in which relational operators are executed and the method for computing each operator. From these parameters it predicts average resource utilisation, maximum transaction throughput and average transaction response time given a transaction arrival rate. The maximum throughput value is derived by analysing the workload and identifying the

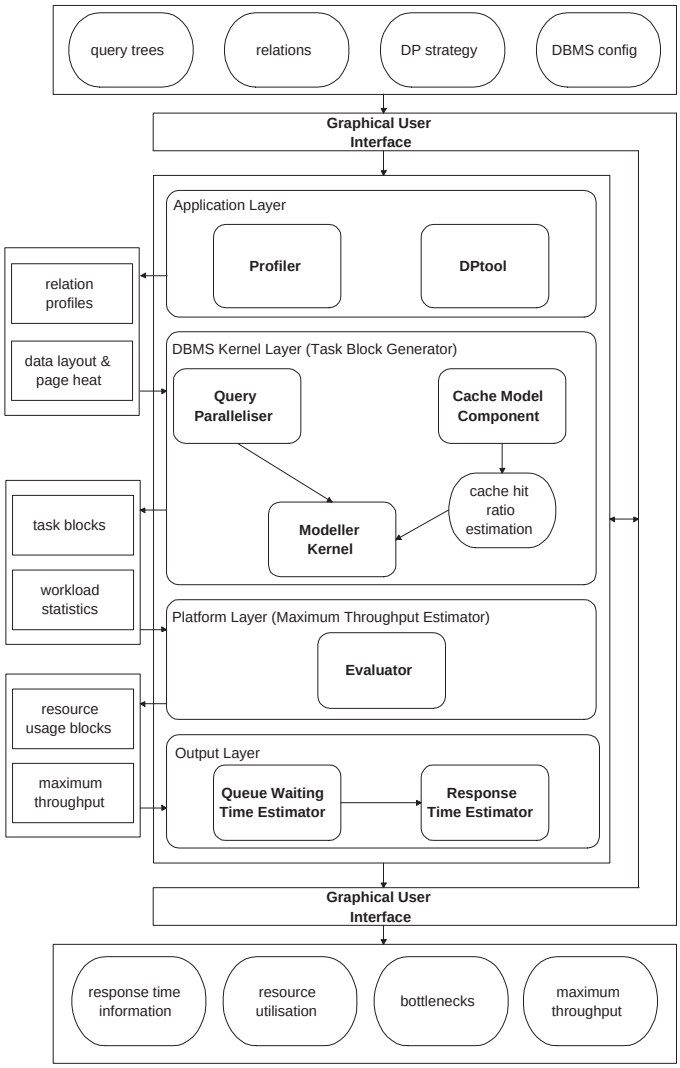

Figure 1. STEADY architecture

system bottlenecks. Given a transaction arrival rate, lower than the maximum throughput, the average response time is derived using an analytical queuing model.

Fig. 1 illustrates the architecture of STEADY. It has four layers connected by a graphical user interface. Each layer comprises one or more modules, as follows:

1. The *application layer* consists of the Profiler and the DPTool modules. The Profiler is used to generate statistical profiles on the base relations. It also generates information on intermediate relations, such as number of tuples, resulting from particular data operations performed in the queries. The DPTool is a data placement tool which produces data placement schemes for parallel databases. For a chosen strategy, DPTool takes information about the relations and the operations to be performed on them and decides how the relations should be fragmented and divided among the different nodes and discs. DPTool also estimates the access frequency of different pages in each relation. It supplies this information along with the generated data layout to the Cache Model Component.

2. The *DBMS kernel layer* consists of the Cache Model Component, the Query Paralleliser and the Modeller Kernel. The Cache Model Component estimates the cache hit ratio for pages from the different relations [25]. The Query Paralleliser is used to generate parallelised query execution plans, incorporating a functionality which captures the different parallel strategies of modelled parallel DBMSs. It transforms the query tree into a *task block* structure. Each task block represents one or more phases in the execution of the relational operators within the query trees. An example of a phase is the building of a hash table in the first part of a hash join operator. The Modeller Kernel takes as input the relation profiles, data layout, estimated cache hit ratios and the task block profile of the query produced by the Query Paralleliser. It produces workload profiles in terms of the numbers of basic operations to be executed on each node in the course of a transaction. This results in a set of workload statistics. Together with this, the Modeller Kernel fills in the details of the task blocks by expanding each execution phase within the block into a corresponding sequence of basic operations. The DBMS kernel layer represents the DBMS specific behaviour of the system.

3. The *platform layer* consists of the Evaluator module. The task block profiles of the queries are mapped by the Evaluator into resource usage profiles. The Evaluator also gives an upper limit to the system throughput value, based on estimating the usage of that resource which is a bottleneck to performance in an average transaction. This layer represents the platform specific behaviour of the system, although there are elements of the Query Paralleliser which are platform specific as well.

4. The *response time layer* consists of the Queue Waiting Time Estimator and the Response Time Estimator modules which use queueing theory to compute the response time for a transaction, from its resource usage profile. Further details on the approach used for response time estimation are given in [21].

4. Oracle Parallel Server and Parallel Query Option

In Oracle, parallelism is exploited in two ways: through the Oracle Parallel Server (OPS) and through the Parallel Query Option (PQO) [17].

OPS allows multiple Oracle instances to share a common database. Each instance is a separate user of Oracle. An instance can be run on one or more nodes of the parallel machine. Each instance has its own processes and log files, but the data and control files are common to all instances. Parallel Server is designed to allow an instance to be shut down, intentionally or otherwise, without affecting the other running instances and also allows multiple users to access the same database.

The primary purpose of PQO is to improve the performance of the system in terms of both throughput and response time. It involves using more than one process(server) to carry out a task. It only comes into action when a query contains a full table scan, otherwise the query is processed sequentially by a single server.

In both OPS and PQO there is a shared database system at the disc level. Its form depends on the architecture of the underlying machine. In a system with multiple nodes and a shared disc this is straightforward, but a shared nothing system requires a coherent virtual file system. Oracle employs a distributed lock manager (DLM) to maintain the status of distributed locks to coordinate access to resources required by different database instances. Requests for locks or I/O may be satisfied by a node remote from the one issuing the request (host).

Assuming that there is at least one full table scan in the query, the operations which take advantage of PQO, are:

- Full table scans.
- Sorts for GROUP BY, ORDER BY and joins.
- Sort merge, hash and nested loop joins.
- Aggregation including GROUP BY, MIN, MAX, and AVG.
- CREATE TABLE.

Operations which are not parallelised include:

- UPDATE, DELETE and INSERT.
- UNION, INTERSECT and MINUS.
- Any queries which do not contain a full table scan.

The flow of a query through the PQO starts with the user process issuing a query or transaction. The dedicated server process parses and executes it. It assigns work to a number of query servers depending upon the degree of parallelism. The query servers split the workload and return the result data back to the dedicated server process. The dedicated server assembles the data and returns the results to the user process. The process of parsing the query includes optimisations, the details of which are not published by Oracle. A node may have several query servers running at the same time. The maximum number of query servers available is set as an initialisation parameter.

The main types of parallelism used by the PQO are *inter-query*, *intra-query* and *pipeline* parallelism. *Inter-query* parallelism is the execution of different queries or transactions in parallel. *Intra-query* parallelism is parallelisation

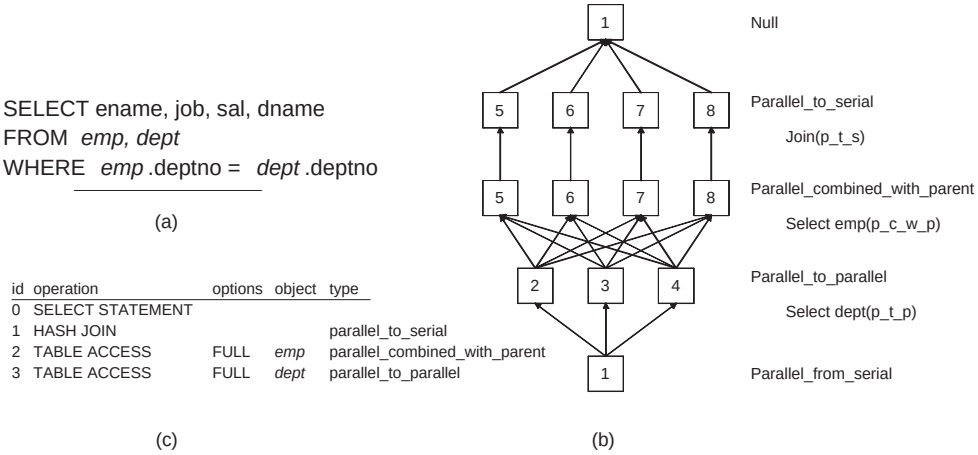

Figure 2. (a) Example query (b) Server allocation (c) Execution plan

within a query and involves two forms: *inter-operator* and *intra-operator* parallelism. In *inter-operator* parallelism different component operations of a query (sub-query) are carried out in parallel. In *intra-operator* parallelism a single operation such as a hashjoin is divided up and executed by multiple servers in parallel. Two successive operations exhibit *pipeline* parallelism when their execution is overlapped in such a way that the second is activated every time the first produces and sends a single tuple. In order to show some of these types of parallelism consider the example query in Fig. 2a.

The query takes two tables, containing employee details and department details, joins them on the department number and returns the employee's name, job, salary and department. Fig. 2b shows a possible server allocation for the example query. The default number of servers for scanning the *dept* table is three and for the *emp* table it is four. Three servers (2,3,4) scan the *dept* table and broadcast all of the tuples to each of the four servers (5,6,7,8) which scan the *emp* table. Each tuple read from the *emp* table is then joined with the broadcast *dept* tuples if the department numbers are the same. The results are then sent to the query dedicated server(1).

The execution plan for the query is shown in Fig. 2c. It shows the actions carried out to execute the query. Some of the operations have options. For example, accessing the data in operations 2 and 3 requires a full table scan

on both the *emp* and *dept* tables. The last column specifies the type of Oracle parallelism to be used. The type 'parallel_combined_with_parent' indicates that the current operation (2) will be carried out by multiple servers which will also carry out the subsequent operation (1). The rest of the types are self explanatory.

Oracle can join tables in three ways: nested loop, sort merge and hash join. Oracle carries out a nested loop join when one of the tables has an index on the joining attribute. It is done by reading the non-indexed table and broadcasting the qualifying tuples to each of the join servers. The join servers receive the tuples and then select tuples from the larger table using the index and join each qualifying tuple against the non-indexed table tuples. Sort merge joins are carried out by scanning and sorting the two tables at the same time, which is an example of *inter-operator* parallelism, i.e. two separate tasks within the same query being carried out at the same time. Sorted partitions are sent to the appropriate merging servers. To show how Oracle carries out a hash join, consider the join query example shown in Fig. 2b. The smaller table is read and all of its tuples are broadcast to each of the servers, which are to read the larger table, and they build a hash table with these tuples. Tuples from the larger table are used to probe the hash table and are joined if their department numbers are the same.

An optimisation feature of Oracle is that all full scans are always read from disc, except for a small number of pages which can be set by the user (default is five pages), so as not to empty the cache each time a full scan is conducted. When scanning large tables, Oracle employs a load balancing technique to share the load of the operation amongst all of the servers involved. The table is split into thirteen parts and these are grouped into three groups containing nine, three and one part, respectively. The first group is allocated to the available servers and execution commences. As soon as a server finishes processing its allotted parts, it is allocated the largest of the remaining parts until all thirteen have been processed.

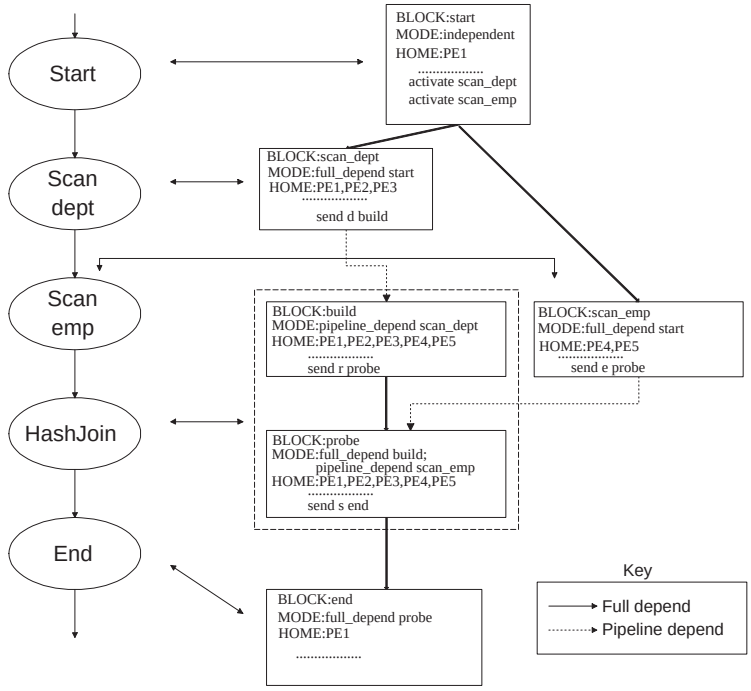

Figure 3. Task Execution Sequence

5. Modelling Parallel Oracle

In developing a model of parallel Oracle both the Parallel Server and the PQO features need to be catered for. The first basic assumption relating to Parallel Server concerns the node which will act as host to the query since the choice of host node can have a significant effect on the performance of a query. For this reason, each node in turn is treated as the host and the performance estimated, after which the results are totalled and an average figure is used for modelling the Parallel Server feature of Oracle, as this effectively means that there are as many users as there are nodes.

The rest of this section describes the modelling of PQO, locking, and background processes.

```
BLOCK:  scan_emp
MODE: full_depend    start
HOME: pe1 1.0, pe2 1.0, pe3 1.0
OPERATION_DEFINITION
      loop { pe1:  x1pe2:   x2, pe3: x3 } {
            read { disc1(1:    p1), disc2(1:  p2) } { pe1: h1, pe2: h2, pe3: h3 };
             loop {    TpP } {
                    predicate_check;
                     send     hash_join e n
             }
       }
END_DEFINITION
```

Figure 4. scan_emp block

5.1. Modelling Parallel Query Option

To show how the different forms of parallelisation employed in Oracle are modelled, consider the join query example shown in Fig. 2a. The parallelised query execution plan can be schematically represented as shown in Fig. 3. It shows the sequence control graph of the query (left-hand side) and the corresponding structure of the task block formalism of the model (right-hand side). There is a task block for each SQL operation and a start and end task block. The *start* block activates the two scan blocks. The *scan_dept* block is fully dependent on block *start*, which means that it can only be started after the execution of *start* has completed. The hash join operation is made up of two blocks (tasks), one to *build* the hash table and the other to *probe* the hash table. The *build* block is pipeline dependent on the *dept* scan block, which means that as the table is being scanned, the hash table is being built with the tuples that were scanned. This represents *pipeline* parallelism which is indicated by a dotted line. The *probe* block is fully dependent upon the hash table being built (indicated by the solid line between the blocks), but pipelines the scanning of the *emp* table and the probing of the hash table, i.e. each tuple read is then used to probe the hash table. The *end* block receives the result tuples and presents the results.

To show how the details of the task block format are modelled, consider the *scan_emp* block, shown in Fig. 4. The first line gives the block's name

(*scan_emp*) and the second line its mode. The *scan_emp* block is fully dependent on block *start*. The block is executed by query servers on nodes PE_1, PE_2 and PE_3 with equal probability, an example of *intra-operator* parallelism. On each server it reads a number of pages locally from relation *emp* where x_i pages are read from node PE_i, as indicated by the keyword loop. The probabilities of the page being read from disc 1 or disc 2 are p_1 and p_2 $(= 1 - p_1)$. The probability attached to the disc read operation (h_i) represents the cache miss ratio for a page of *emp* on node PE_i. This is followed by a loop of TpP iterations where TpP is the number of tuples per page of *emp*. In this loop a predicate check is carried out on each tuple and the selected tuples are sent through the interconnect to the block *hash_join*. Here e and n are the tuple length and the number of tuples sent per iteration.

As Oracle's load balancing technique is designed to balance the workload across the nodes of a parallel machine, the model considers each full table scan to be spread across all the nodes involved equally.

5.2. Oracle Transaction Locking and DLM Policies

For Oracle installed on a GoldRush machine each relation (table) comprises a number of tuples (rows) and these tuples are stored in pages, which is the unit of storage at the disc level. In general, a page may contain a number of tuples. GoldRush Oracle has two distinct types of locks: parallel cache management (PCM) and transaction locks.

PCM locks are used at the database page level. They can be held in shared (read only) or exclusive (update) mode. These page locks are assigned to the distributed lock manager (DLM) instances on the nodes according to a hashing scheme, so that each page lock has an owner. The owner handles PCM lock requests for that page lock. When a transaction requests a page, the PCM locking process is used to ensure the safe extraction of the page and the delivery of that page to the requesting transaction. However, once it receives the page, the transaction will want to access a tuple within that page.

At this point the transaction locks come into play. Each row has a transaction lock associated with it. Transaction locks are owned by individual transactions and not by servers. They are only given up when the transaction commits or rolls back. Each transaction has an owner (its host, where the dedicated server resides). A system is used to number the transactions as they start. This number also identifies the node that is hosting the transaction. A server scanning a relation receives a page from disc and locates the row it wants to access. Attached to this row may be a transaction number, indicating that this row has been, or is going to be modified. If no number exists then the transaction is free to set a row lock and update the row. If the server reading the page finds a transaction number, then it must determine the status of the row. This it does by asking its DLM. The DLM contacts the host of the transaction using a lock status request. The host node consults its rollback segment index for the location of the segment and sends this location to the server querying the status of the row. The server reading the row receives the location of the rollback segment and reads it into its buffer. It looks up the version of the row with the largest transaction number, smaller than its own. The corresponding value of the row is returned.

To illustrate PCM locks consider the following example. Suppose the DLM on node PE_3 is the owner of a page lock, which a server on node PE_1 needs in order to update a page. Suppose that this page is held in exclusive mode on node PE_2. The server on node PE_1 asks its DLM for the lock. The DLM is not the owner, and therefore it sends a lock request to the owner. The owner (the DLM on node PE_3) looks up its lock status table and notes that node PE_2 has the page held under an exclusive lock. The DLM on node PE_3 sends a lock-request message for the page to node PE_2, which writes the page to disc and sends a lock-grant message in return to the DLM on node PE_3. Once the DLM on node PE_3 receives it, it updates its lock status table and sends a lock-grant message to the original requester on node PE_1. The server on node PE_1 reads the page from disc into its buffer with an exclusive lock and continues with the update. Note that node PE_2 releases the lock immediately

```
BLOCK:  start
MODE: independant
HOME: pe1 1.0
OPERATION_DEFINITION
      .....
       send  dlm lck_req  v
END_DEFINITION
```

```
BLOCK:  dlm
MODE: full_depend    start
HOME: pe1 0.25, pe2 0.25, pe3 0.25, pe4 0.25
OPERATION_DEFINITION
       obtain_lock;
        send  end lck_rep{ pe1:1.0 } 1.0
END_DEFINITION
```

```
BLOCK:  end
MODE: full_depend    dlm
HOME: pe1 1.0
OPERATION_DEFINITION
      .....
END_DEFINITION
```

Figure 5. PCM lock blocks

upon request. If one or more servers hold the same page under a shared lock (only one copy of a page is allowed in any cache under an exclusive lock) and another server requests the page for updating, all servers holding a shared-lock on the page, are asked by the page lock owner to give up the lock and page. Once they acknowledge that they have done so, the owner grants the lock to the server requesting it. If a server finishes with a page and no other server requests it, the page stays in its buffer until it is required or the buffer is flushed. At this point the owner is informed to update the lock status table.

5.2.1. *Modelling Locks*

A model has been developed to estimate the number of PCM locks held on pages in a particular server's cache. The model is derived following the approach originally developed by Dan and Yu in [4]. The Oracle7 parallel cache management as described above is similar to the Deferred until Transfer or Flushing policy detailed in [4]. The model allows one to estimate the number of DLM exclusive and shared locks held on pages in a particular node's cache. This, in turn, allows one to obtain an estimation of local and remote buffer hit probabilities. The details of the model are described in greater detail in [25].

The example in Fig. 5 shows how PCM locks are handled in the task block notation. A transaction running on node PE_1 requests a lock from the DLM running on node PE_2, which is the owner of the page. An extra block is added for lock request processing by the DLM instance. v is the probability that the page that the lock request is for, is not held in the local cache.

The additional *dlm* block represents the handling by the DLM of a lock request from the transaction running on node PE_1. The owner of the page may be any of the four nodes; thus each has a probability of 0.25 of owning the page lock. lck_req is the message size transmitted from node PE_1 to node PE_2 to request a lock. lck_rep is the message size transmitted from node PE_2 to node PE_1 to grant the lock request.

The modelling of transaction locks uses the same blocks as the DLM (shown above), because it also uses the DLM to communicate with the transaction owner about row locks, but with different frequencies. The estimation of the frequencies is based on the number of modifying queries in the transaction and the calculation of queue lengths for the rows being modified.

5.3. Modelling Background Processes

In Oracle there are many background processes, including a database writer, log writer, session monitor, checkpoint process, archiver and process monitor. The archiver, checkpoint process, session and process monitors are concerned with recovery from a system breakdown, and are not modelled. This leaves the two most important background processes which both write to disc and therefore require to be modelled. They are the database writer (DBWR) and the log writer (LGWR).

The database writer (DBWR) writes buffers to disc. It maintains two lists, the least recently used (LRU) and the modified buffer list. The DBWR writes the modified buffers to disc when one of three things occur: when no free buffer can be found in the LRU list, when a time out occurs or when a checkpoint process is initiated. The total number of pages in the buffer and the page size are set by the user. The cache miss probability is estimated by the cache model. Knowing these, the number of pages in the buffer cache can be estimated.

The redo log buffer, circular in design, records changes made to the database. The log writer (LGWR) writes the redo log buffer to disc. There are three circumstances when it does this:

```
BLOCK:  dbwr
MODE: independent
HOME: pe1 1.0
OPERATION_DEFINITION
      in_parallel {
          write { disc1(   x:p_1), disc2( x:p_2) } 1.0 }
END_DEFINITION
```

Figure 6. dbwr block

- Every three seconds.
- When the buffer is one third full.
- When the DBWR process writes to disc.

The buffer size, row length of the table and the query frequencies are set by the user. Knowing these, the rate of filling of the buffer can be estimated. To model the writes that occur due to the expiry of the time interval, the cost can be added at the end of the transaction.

The block representation of the background processes which are modelled are very similar. For example, the DBWR process can be represented by a block as shown in Fig. 6. The LGWR block is identical except that the value of x is equal to a third of the size of the log buffer instead of the number of modified buffers in cache. Thus background processes are treated as if they were separate transactions run at the same time as user transactions.

5.4. Comparison with Modelling of Informix

There are a number of important differences in modelling the Parallel DBMSs Oracle and Informix. In Informix, data is only read by the node that owns that data whereas in Oracle data can be read from any node. Consequently all reads in Informix are local reads whereas in Oracle they can be local or remote. In Oracle, locking policies are more complex, partly because of the two types of locks and partly because of maintaining the consistancy of the data in the virtual file system employed. By contrast, in Informix all reads are local and the cost of locking can be dealt with more simply. Repeat full table scans in Informix are cached, unlike Oracle where all multiple full table scans are

read from disc except for a few pages. A separate cache model component had to be written for Oracle, detailed in [25]. Background processes are modelled in the same way for both PDBMSs, only the actual amounts written to disc are different. Informix has no load balancing scheme for full table scans, as all data is read from the node that owns the data, but redistribution of qualifying tuples is carried out for aggregation to even the workload amongst nodes.

6. Comparison With Actual Performance Measurements

Having developed the Oracle model, it was calibrated to obtain estimates for the basic costs using an ICL GoldRush parallel server with 6 nodes and 6 discs per node.

Once the basic costs were obtained, a set of queries and tables were used to validate the model. Three of the queries used will be considered here, and are referred to as Query 1, 2 and 3. Each involves a scan of a relation coupled with one or more aggregate operations. Three uniques relations of the AS3AP [22] benchmark were used by all three queries:

Uniques30, 30,000 tuples on 1 disc of node PE_3 (30k1)
Uniques90, 90,000 tuples on 1 disc of node PE_3 (90k1)
Uniques270, 270,000 tuples on 1 disc each of node $PE_0, PE_1, PE_2, PE_3, PE_4$ *and* PE_5 (270k6)

There were two main steps in the validation phase. The first step established the maximum throughput and the second the average response times for transactions. For each step a separate transaction generator was required. The two generators were written in Pro-C. The first generator, used to dtermine throughput, fires queries at a constant rate. Generally a batch of 100 queries was fired for each inter-arrival time considered, although larger batches were tried to check the validity of the results gained from the smaller batches. The start time of each query was recorded as was the time when the result was

returned and the difference between the two taken as the time for the query to return the result, this includes execution and queue waiting time. The times for all 100 queries was totalled and averaged to give the throughput figure. The response time generator fires queries with exponential inter-arrival times, to represent the load of a working database system. Again batches of 100 were found to be sufficient to give as good results as larger batches. The timings were taken in exactly the same way as for the throughput tests. All timings were taken in clock ticks and then converted into seconds.

6.1. Throughput

A selection of the results for the throughput tests are shown in Fig. 7 a, b and c. The x- and y-axis denote query arrival rate and throughput respectively, and are measured in transactions per second. In the top right hand corner of each graph, the value of the maximum throughput predicted by STEADY, which is a single value independent of the arrival rate, and the maximum throughput actually achieved by the system are displayed. The maximum throughput achieved by the system was established by firing batches of 100 queries with increasing inter-arrival rates, calculating the throughput, until the calculated throughput peaked.

The throughput results for all of the queries showed reasonable agreement with the STEADY maximum throughput results. A total of 9 experiments was performed. In 7 of the 9 cases the relative error between predicted and measured maximum throughput was under 10%, and in 5 of these it was within 5%. Only 2 cases produced errors above 10% and here the worst case was less than 17%. The average discrepancy was about 6%. The STEADY figure is always higher than the actual measured value. This may be due to additional operating system overheads which were not taken into account in the STEADY model.

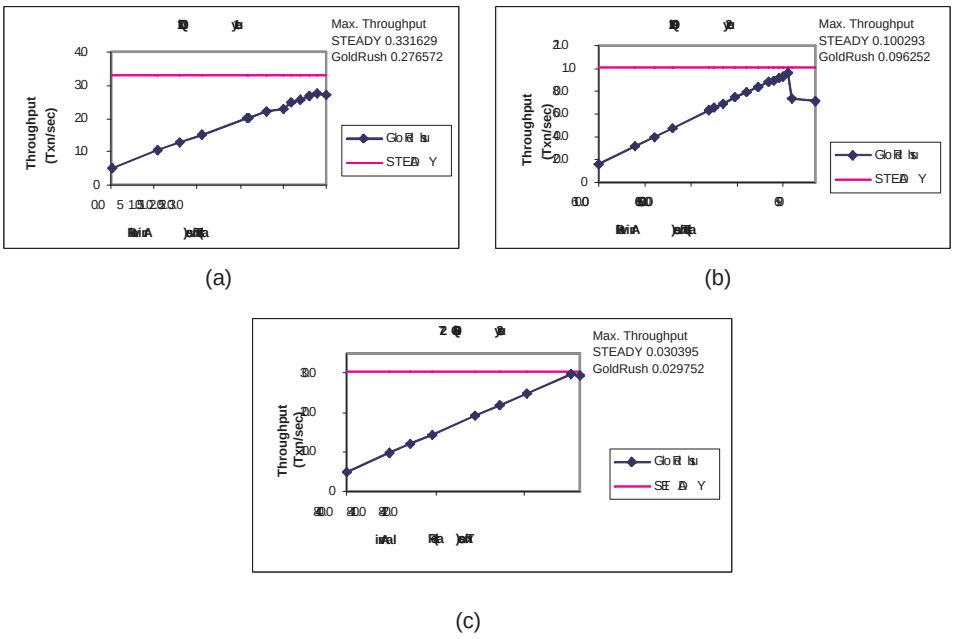

Figure 7. Throughput graphs (a) 30k Query 1 (b) 90k Query 2 (c) 270k Query 3

6.2. Problems with measuring Oracle performance

During calibration the difference in behaviour between Oracle and Informix became apparent. The most obvious difference in behaviour was the variability in measured results exhibited by Oracle compared with the very consistent behaviour of Informix. As a result, each test, which obtains a throughput value for a given arrival rate, had to be repeated a number of times to ensure consistency. As an example Query 1, was run on 12 separate occasions with an inter-arrival rate of 0.27 transactions per second. On each occasion the machine was in a similar state and was dedicated to this application. Each run involved firing 100 queries. The graph of the results of these 12 runs is shown in Fig. 8.

The x-axis of the graph in Fig. 8 represents the identity number of the query, which is allocated as it is fired, while the y-axis represents the execution time of the query in seconds. As can be seen, there are three runs (4, 5 and 6) where the times increased dramatically for no apparent reason and queries

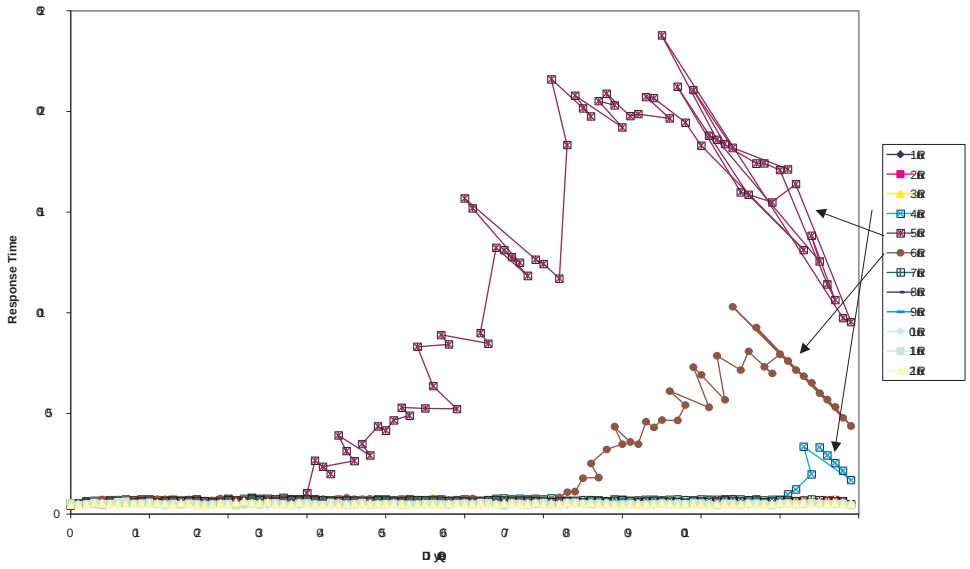

Figure 8. Throughput - 0.27 Txn/sec - 12 Runs

arriving later completed ahead of earlier ones (giving the backward diagonal line effect). The times prior to the sudden increase were as expected and conditions during all runs were identical. The reason for this behaviour was due to the background processes in Oracle, although the monitoring tools did not enable one to isolate and identify these.

If the three runs with anomalously high execution times are removed and the graph is rescaled, the graph in Fig. 9 is obtained. Here a further anomaly is observed. There appear to be two distinct bands of query times, one averaging 5 seconds (band 1), the other averaging just above 6 seconds (band 2). This difference between the bands was attributed to the nondeterminism in the engagement of a background DBMS process, such as garbage collection.

By comparison, Fig. 10 shows the throughput results for three runs at 0.2 transactions per second for Informix on the GoldRush machine using the same type of transaction generator. As can be seen the variation in response times is much less than those of the Oracle graphs, i.e. within 13% either side of the average compared to nearly 40% in Fig. 9. In total more than one hundred runs

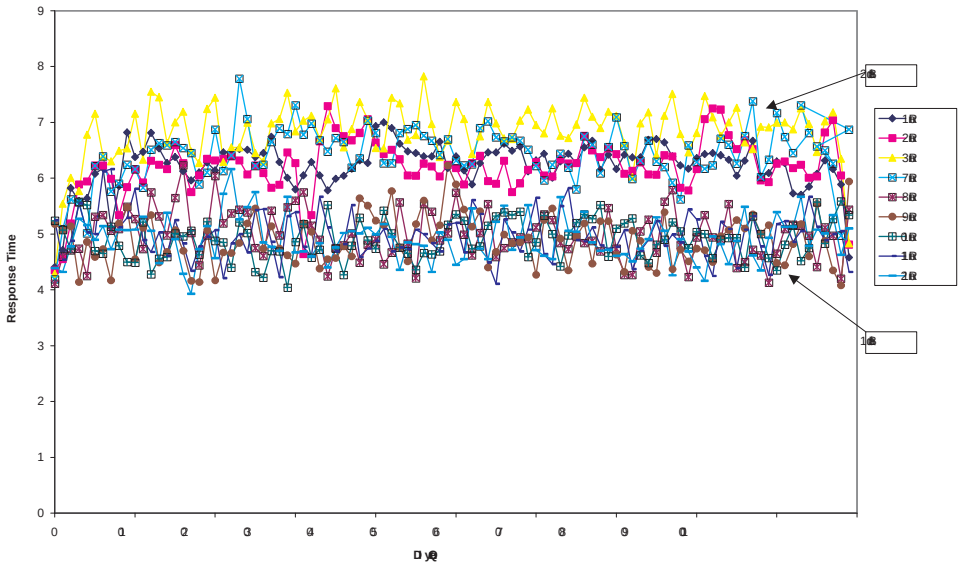

Figure 9. Throughput - 0.27 Txn/sec - 9 Runs

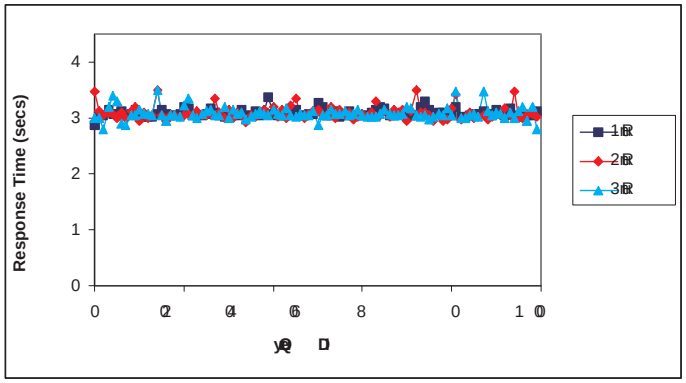

Figure 10. Informix - Throughput - 0.2 Txn/sec

were obtained on Informix and this same level of consistency was observed in all cases. The erratic behaviour seen in Fig. 8 did not arise in Informix at all, possibly due to the different approach to background processes. It should be noted that the band effect observed in Fig. 9 has a minimal effect on the overall maximum throughput figures. However, it does have a significant effect on response time measurements.

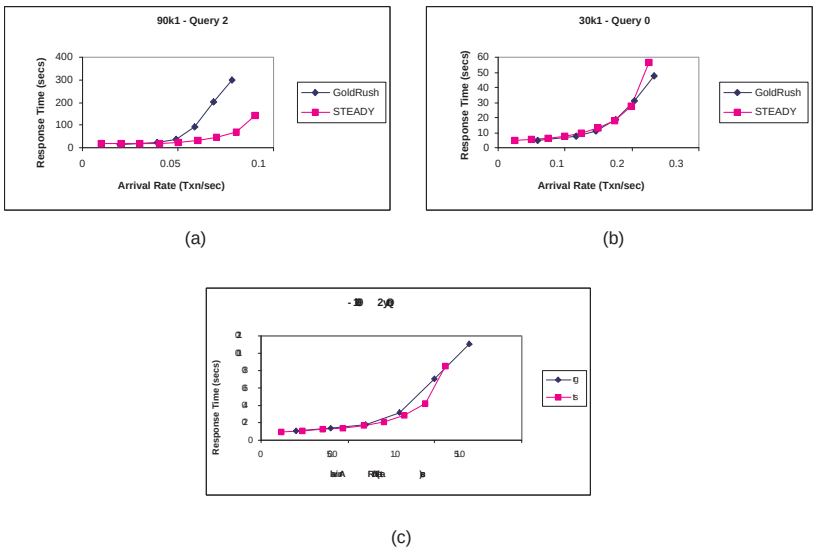

Figure 11. Response Times for (a) 90k query 2 - Oracle (b) 30k query 1 - Oracle (c) 90k query 2 - Informix

6.3. Response Time

The same problems were experienced with the exponential transaction generator, although the discrepancies are more difficult to isolate because of the variabilities in the inter-arrival time. In view of these problems, it was decided to concentrate the response time tests on arrival rates between 10% and 70% of maximum throughput, as this is the resource utilisation that would be considered normal for database work. Two examples of results for the response time tests on Oracle are shown in Fig. 11a & b. These two graphs highlight the variations in the results achieved. Fig. 11a shows a set of results where the error between measured and actual values are close together until 50% of maximum throughput when the curves diverge. However, Fig. 11b shows results where the two curves are closely matched throughout. For comparison purposes the same test as Fig 11a for Informix is shown in Fig. 11c.

For Informix, the average error, between measured and predicted values, for all response time tests was less than 15% for arrival rates up to 70% of maximum throughput. Between 70% and 90% of maximum throughput

measurements for Informix showed errors up to 30%. On the other hand, for Oracle, for transaction arrival rates less than 70% of maximum throughput the average error was 19%, whereas for transaction arrival rates above this it is difficult to predict the response times as additional background processes cause serious problems to overall performance. However, normal database activity occurs below 70% of maximum throughput and it would not be recommended to exceed this value for any significant period of time.

7. Conclusions

Predicting the performance of a parallel database system is a complex task. An analytical approach has been developed for identifying bottlenecks, predicting maximum throughput and estimating response times. This has been applied to three separate parallel relational database systems (Oracle, Informix and Ingres) running on the ICL GoldRush Megaserver. This paper focuses on the Oracle system, discussing the different forms of parallelism, the locking policies, cache management and background processes, showing how these can be mapped into a block formalism and presenting some preliminary results. In validating the model, considerable problems were experienced in repeating results for both throughput and response time tests. The throughput results show very good correlation between STEADY predictions and actual measurements, with an average error of about 6% between the maximum throughput predicted and that measured. The correlation of the response times between STEADY predictions and actual measurements for arrival rates up to 70% of STEADY maximum throughput shows an average error of 19%.

8. Acknowledgements

The authors acknowledge the support received from the Commission of the European Union under the Framework IV programme, for the Mercury project

(ESPRIT IV 20089) and from the Engineering and Physical Sciences Research Council under the PSTPA programme (GR/K40345). They also wish to thank Phil Broughton, Arthur Fitzjohn and Monique Mitchell of ICL and especially Shaoyu Zhou of Microsoft for their contribution to this work.

References

1. The ASK Group Ltd., *ASK Open (INGRES) database administrator's guide* (The ASK Group Limited, Bury St. Edmunds, UK, 1994).
2. BEZ Systems Inc., *BEZPlus for NCR Teradata and Oracle environments on MPP machines* (http://www.bez.com/software.htm, 1997).
3. BMC Software Inc., *PATROL DB-log Master* (http://www.bmc.com/products/trx/index.html, 1998).
4. A. Dan and P. S. Yu, Performance analysis of coherency control policies through lock retention, *Proceedings of ACM SIGMOD Conference* (California, U.S.A.) (June 1992) 114–123.
5. DB LAB PTY Ltd., *Ingres DB_Maximiser* (http://ingres.com.my/db_maximizer.html, 1996).
6. Dell Computer Corp., *Oracle System Sizer* (http://www.dell.com/products/poweredge/partners/db/oracle/wp-sizer.htm, 1998).
7. Digital Equipment Corp., *EcoTools* (http://www.digital.com/alphaserver/products/internet/search/ecotools_3_3.html, 1998).
8. R. Eberhard, IBM Corp., *DB2 Estimator for Windows* (http://www.software.ibm.com/data/db2/os390/db2est-home.html, 1998).
9. T. Foxon, M. Garth and P. Harrison, Capacity Planning in Client-Server Systems, *Journal of Distributed Systems Engineering* **Volume 3** (1996) 32–38.
10. M. Garth, Capacity Planning for Parallel IT Systems, *Computer Bulletin* (November 1996) 16–18.
11. M. Gurry, P. Corrigan, *Oracle Performance Tuning* (O'Reilly & Associates, Inc, California, USA, 1996).
12. IBM Corp., *DB2* (http://www.software.ibm.com/data/db2/, 1998).
13. Ifatec S.A., *SMART2 User Guide, Release 2.1* (Ifatec S.A., 3 Rue Petigny, 78000 Versailles, France, 1994).
14. Informix Software Inc., *Informix-OnLine Dynamic Server & Administrator's Guide* (Informix Press, California, USA, 1994).

15. G. Koch, K. Loney, *Oracle, The complete reference* (Osborne, McGraw-Hill, California, USA, 1997).

16. M. G. Norman, P. Thanisch, *Parallel Database Technology* (Bloor Research Group, Netherlands, 1995) 1–546.

17. Oracle Corporation, *Oracle7 Parallel Server Concepts & Administration, Release 7.3* (Oracle, Server Products, California, USA, 1996).

18. Oracle Corporation, *Oracle7 Server Concepts, Release 7.3* (Oracle, Server Products, California, USA, 1996).

19. Sybase Inc., *Sybase* (http://www.sybase.com, 1998).

20. N. Tomov, E. Dempster, M. H. Williams, A. Burger, H. Taylor, P. J. B. King, P. Broughton, Some Results from a New Technique for Response Time Estimation in Parallel DBMS, *Proceedings of the Seventh International Conference, HPCN Europe 1999, Springer, April 1999* 713–721.

21. N. T. Tomov, E. W. Dempster, M. H. Williams, P. J. B. King, A. Burger, Approximate Estimation of Transaction Response Time, *to appear in The Computer Journal, 1999.*

22. C. Turbyfill, C. Orji and D. Bitton, AS3AP: An ANSI SQL Standard Scaleable and Portable Benchmark for Relational Database Systems, *The Benchmark Handbook, Second Edition, J. Gray (editor), 1993.*

23. M. H. Williams and S. Zhou, Data Placement in Parallel Database Systems, *Parallel Database Techniques* (IEEE Computer Society Press, 1997).

24. M. H. Williams, S. Zhou, H. Taylor, N. Tomov, Decision Support for Management of Parallel Database Systems, *in Proc. of Int. Conf. and Exhibition on High Performance Computers and Networks, Brussels, Belgium* **LNCS Series vol. 1067** (Springer-Verlag, April 1996) 164–169.

25. M. H. Williams, S. Zhou, H. Taylor, N. Tomov, A. Burger, Cache Modelling in a Performance Evaluator for Parallel Database Systems, *Proceedings of the Fifth International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems* (IEEE Computer Society Press, January 1997) 46–50.

26. S. Zhou, M. H. Williams, H. Taylor, Practical Throughput Estimation for Parallel Databases, *Software Engineering Journal, July 1996* 255–263.