FlexParDB - an RDBMS employing intra-query and operator parallelism

T. Theoharis & J. Y. Cotronis
Department of Informatics, University of Athens,
Panepistimioupolis, 15771 Athens, Greece
Email: {theotheo, cotronis}@di.uoa.gr

Abstract

FlexParDB, a relational algebra query execution system is presented, which combines intra-query and operator parallelism. Intra-query parallelism is expressed in the wavesets, which are a partition of the set of query-tree operators; valid wavesets are consistent with the flow of relational data from the leaves to the root of the query-tree. The wavesets represent the query execution plan. A simple script language for the description of a query-tree and its wavesets has been developed. Wavesets are executed by parallel multiple executions of PARDB, a system supporting operator parallelism. FlexParDB has been implemented on a massively parallel Transputer architecture.

1. Introduction

There are three levels at which parallelism can be introduced in an RDBMS, resulting respectively in *query*, *operator* and *tuple* parallelism. *Query* parallelism can be divided into inter- and intra-query parallelism. The former involves the processing of different queries in parallel, possibly on different partitions of the data base, while the latter involves deciding how to best exploit the parallelism within a query. For example, if a query involves the join of four relations R1, R2, R3 and R4, in a system exploiting intra-query parallelism the join of R1 and R2 may be done in parallel with the join of R3 and R4 (followed by a join of the two results). *Operator* parallelism arises when the function and data of a relational operator is distributed among different processors. This involves the partitioning of the data base over multiple secondary storage devices and leads to the consideration of data distribution and load balancing issues.

In previous work operator parallelism and intra-query parallelism have been separately investigated. Operator parallelism was employed in PARDB [7], a parallel RDBMS developed at the University of Athens based on Transputer hardware [5].

An intra-query parallelism execution model which exploits pipelining, the Tree Pipelining query execution model (TP) [8], has been developed, whereby a query is transformed into a query-tree representation and pipelining is determined by the query-tree structure. For query execution the nodes of the tree represent processes of relational algebra operators and the arcs represent communication of intermediate results from children to parent processes. Processes are allocated to a number of processors, utilising the intrinsic parallelism of the query tree: (i) leaf-processes, as they have the base relations available, start execution immediately, they produce tuples of intermediate relations and then propagate them to their parent processes; (ii) inner-processes start execution in pipeline mode as soon as they have sufficient tuples to operate on. The query tree is optimised before being submitted for execution. A cost model for the estimation of communication and I/O costs in parallel execution spaces according to TP has been used in exhaustive parallel optimisation and in parallelized enhanced iterative improvement.

A query execution environment for TP [2] has been developed on the Parix message passing environment running on Parsytec GC 3/512 massively parallel Transputer architecture [5]. The query tree is annotated with implementation specific information, such as the allocation of processes to processors. The annotated query tree structure is interpreted by a Loader program, which composes a message passing program out of reusable library components, each implementing a relational algebra operator. The composed message passing program implements the pipeline execution of the query tree. This composition is a special case of the composition of message passing programs by the Ensemble methodology on Parix [3].

We would like to combine the advantages of operator parallelism in PARDB and pipeline parallelism in TP to achieve more efficient parallel query execution. By adding operator parallelism to TP we could control the flow of relation tuples from the leafs to the root. As this flow depends on the complexity of operators and the data volume to be processed, we could allocate more processors to processes which slow the pipeline processing.

In TP the maximum number of processors required is equal to the number of query-tree nodes and, if a sufficient number of processors is available, low processor utilisation will not influence the overall speedup. In contrast, if operator parallelism is added, there is no bound on the number of processors (even for a single operator) and we should seek the optimal allocation to maximise utilisation of the available processors. Extending the optimiser to cope with the extra parameter of operator parallelism is a complex task in terms of its design and its execution complexity.

In the pipeline execution of a query tree there appear to be “waves of activity” which begin at the query tree leaf nodes and, as intermediate results become available, internal nodes are activated; if a full query tree is therefore mapped onto parallel processes, a large proportion of them will be idle and processor utilisation will be low.

It therefore seems more efficient to allocate processors according to waves of activity which are essentially sets of query-tree nodes (operators) called *wavesets*. We have developed the FlexParDB system which provides a mechanism for the definition of wavesets with operator parallelism and their execution. This entails the partitioning of the query-tree nodes into wavesets and the allocation of the available processors to the operators in each waveset. For the execution of wavesets, the PARDB system has been appropriately extended and used as the basic building block of FlexParDB.

The rest of this paper is structured as follows. Section 2 gives an outline of PARDB and its extensions for FlexParDB, section 3 describes the specification and the execution mechanism of alternative execution plans in FlexParDB. System performance is discussed in section 4 and the conclusions and future work are given in section 5.

2. PARDB: Operator Parallelism Unit

PARDB [7] provides operator parallelism within each node of the query tree. All relations are horizontally partitioned among the Transputers. The original implementation was on a Parsytec Multicluster under the Helios operating system [6] but PARDB has now been ported to a Parsytec GC machine under PARIX [5].

One Transputer is reserved to provide a unique external interface to query processing applications and for delegating the work to the appropriate slaves through message passing. This Transputer and the associated software is the master. All other Transputers hold a partition of the database and are called slaves; a slave independently manages its local data base partition and stores all the relevant data (such as subindices and sequential RDBMS code-which has to be duplicated at every slave). PARDB can comprise many slaves but only one master (see fig. 1).

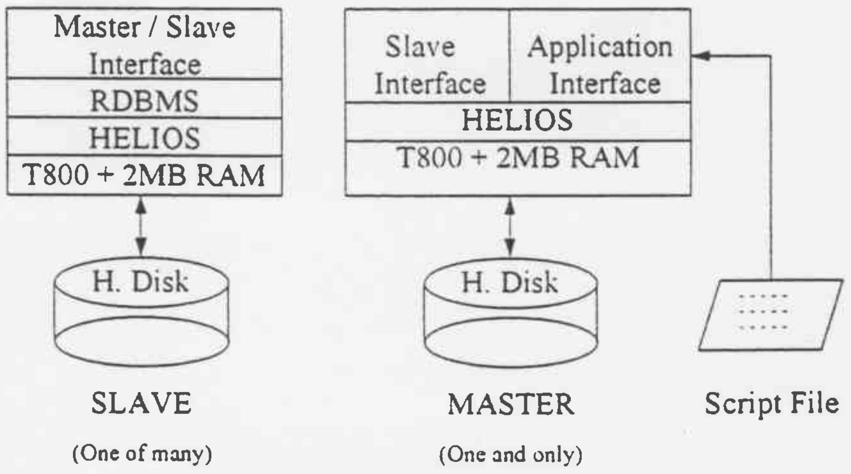


Figure 1. PARDB System Architecture

The master reads the relational commands to be executed from a script file. The current list of commands includes Create, Destroy, Append, Delete, Project, Select, Union, Intersection, Difference, Join, Load (load a data base onto the disk of each slave), Collect (dump a data base from the disk of each slave onto a central disk), RmDupl (remove duplicate tuples that may exist between different partitions of a relation) and Balance (redistribute the tuples between partitions so that the difference between the partitions with the minimum and maximum number of tuples is no more than 1).

The master reads script file commands, and sends messages to the slaves which execute the commands (usually that simply involves broadcasting the command and any arguments to the slaves). The master then waits for messages from the slaves; the messages contain results and report successful termination of a command. The master finally records any changes in a central data structure, the relation table, which keeps the number of tuples per partition per relation. Correspondingly upon receiving a command from the master, a slave calls the appropriate sequential RDBMS routine to perform the necessary processing on the local partition of the data base(s), it then exchanges partition data with other slaves (for multiscan [4] operators) and sends a termination message to the master.

The network architecture is determined by the pattern of interprocessor communication requirements and the capabilities of the underlying hardware. Most multiscan relational operations can be implemented on a parallel system either by the global partitioning method or by the cycling method [1]. Global partitioning algorithms redistribute the tuples of the operand relations, so that the partitions of all operand relations that reside on the same processor have the same range of values for the attribute of interest; the required sorting or hashing step is made more efficient by minimising the network diameter (e.g. hypercube). By contrast, the cycling method performs an exhaustive comparison and broadcasts every partition (of the smaller relation) from its host processor to all other processors. This broadcast is easily implemented by "cycling" partitions around a ring, thus avoiding the necessity to hold the whole of one relation at any one node and overlapping the I/O of a partition with the processing of another.

It is not easy to predetermine which method will achieve better performance as it is dependent on factors such as the relative size of the operand relations, whether they are balanced, the speed of the underlying communication network etc. We have chosen to opt for the simpler ring architecture and the cycling technique. Communication links between the master and each slave also exist.

PARDB is fully scalable and can be configured to use any number of slaves at run time (subject to system resources). This allows us to allocate a number of processors to each query tree node which is proportional to the complexity of the node; this number is currently a rough guess but in the future it should be based on the output of a query optimiser (cost functions).

PARDB is the basic building block in FlexParDB. Each operator of the query-tree will be performed by one execution instance of PARDB; for this reason PARDB has been generalised to partition its output relation parametrically to be used by parent-nodes, as each node in the query-tree may be using a different numbers of processors (fig. 2). A node (execution of

PARDB) must therefore be able to produce its output relation in any number of partitions **m** which is possibly different to the number of partitions **n** of its input relations. This is achieved by having all slaves send their output tuples to the master who appends them (by round robin) to the **m** partitions of the output relation.

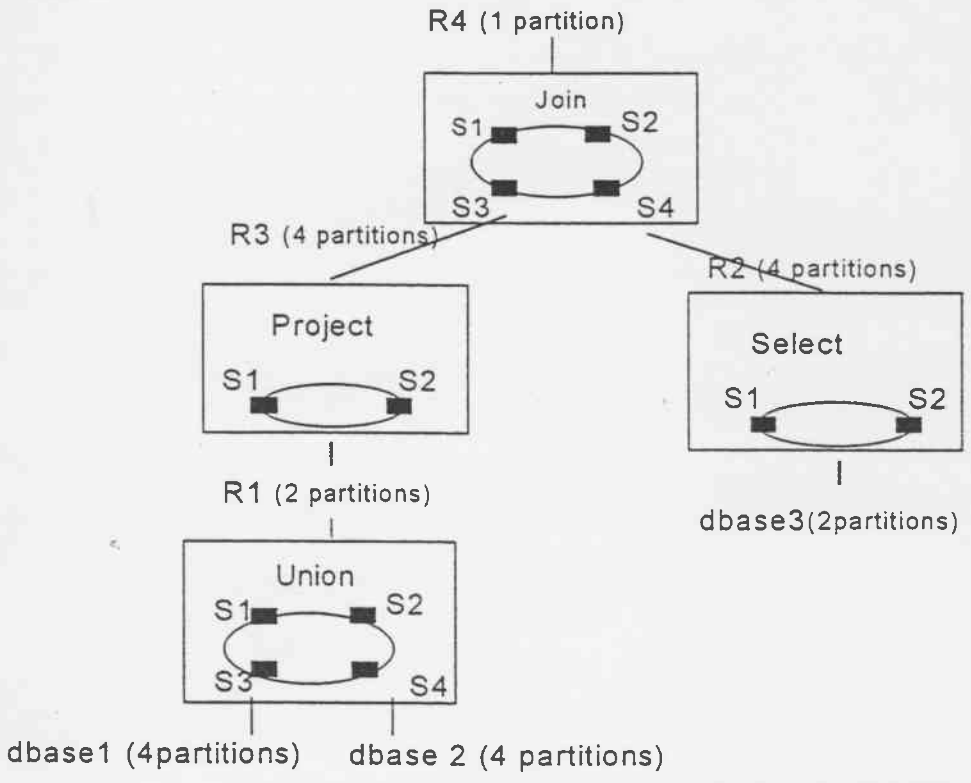


Figure 2. Query Tree with Operator Parallelism

3. Query Execution Plans in FlexParDB

For the representation of queries in query-tree forms we have used recursive list structures. For example the query-tree of figure 2 may be expressed by the list structure

(Join [f1, f1] (Project [f1, f2] (Union [dbase1, dbase2]))
(Select [condition(dbase3.f2)]))

in which the root of the query-tree is the Join operator on attribute f1 of the output relation from its left child with attribute f1 of the output from its right child; the left child is a Project operator and its right child a Select operator. For these list structures to become query execution plans in FlexParDB two elements have to be specified for each operator: the order of execution and the number of its allocated processors. For example we would like to specify the execution plan that corresponds to figure 2:

step1: Union [dbase1, dbase2]	using 4 slaves + 1 master
step2: Select [condition(dbase3.f2)]	using 2 slaves + 1 master
Project [f1, f2]	using 2 slaves + 1 master
step3: Join [f1, f1]	using 4 slaves + 1 master

We maintain the recursive list structure above but for each operator we also indicate its order of execution and the processor allocation following its parameters "OPERATOR [parameters] ORDER:ALLOCATION", where ORDER and ALLOCATION are positive integers, indicating the execution step at which an operator will be initiated and the number of processors allocated in the form (slaves + master), respectively. In this notation, the above execution plan will be expressed by the script S1:

```
(Join [f1, f1] 3:(4+1)
        (Project [f1, f2] 2:(2+1) (Union [dbase1, dbase2] 1:(4+1)))
        (Select [condition(dbase3.f2) 2:(2+1)]) )
```

We may allocate 3 and 1 slave processors to the Project and Select operators, respectively, by script S2:

```
(Join [f1, f1] 3:(4+1)
        (Project [f1, f2] 2:(3+1) (Union [dbase1, dbase2] 1:(4+1)))
        (Select [condition(dbase3.f2) 2:(1+1)]) )
```

or, we may specify that Project and Select are performed in different steps allocating to each of them 4 slave processors by the script S3:

```
(Join [f1, f1] 4:(4+1)
        (Project [f1, f2] 3:(4+1) (Union [dbase1, dbase2] 1:(4+1)))
        (Select [condition(dbase3.f2) 2:(4+1)]) )
```

Operators having the same order are said to belong to the same *waveset*. Let O1 and O2 be the order of operators OP1 and OP2; then O1=O2 iff there exists a waveset W such that OP1, OP2 belong to W. For a script to be valid its wavesets and the operators in them must satisfy a number of properties:

1. *the partition property*: Wavesets are a partition of the set of the query tree operators, Nodeset, that is to say they are pairwise disjoint and their union is equal to Nodeset.

2. *the strict wave form property*: As we do not permit pipelining in FlexParDB no ancestors or descendants of any operator must coexist in the same waveset. This property implies that the number of wavesets is greater than or equal to the height of the query tree.

3. *the query tree compatibility property*: The order of execution of operators must be compatible with the order of the flow of intermediate relations from the leafs to the root. If operator OP1 is an ancestor of operator OP2 then O1>O2, where O1, O2 are the respective orders of OP1 and OP2.

4. *the allocation property*: The total number of processors allocated to operators in the same waveset should be less than or equal to the number of available processors.

Query execution is performed in two phases, script pre-processing and waveset executions.In phase A the script is pre-processed in order to prepare the explicit parameters for each PARDB execution, as follows:

1. Ordered wavesets are constructed and validated according to the above properties. 2. Intermediate relations, between nodes of the query-tree, are stored in temporary files whose names are internally generated. A producing operator prepares its output relation in as many partitions as the processor allocation of its consuming operator. The number of output partitions is implicit in the query-tree script, determined by the nesting of subtrees. However, each PARDB execution requires explicit parameters for the input file name(s), the output file name and the number of output partitions. Therefore, for each waveset we construct simple individual waveset execution plans which explicitly specify all parameters of each PARDB execution as follows:

```
OPERATOR [parameters] Allocation          /* as in script */
         [Input File Name(s)]
         [Output File Name, #partitions ]   /* generated file names */
```

For example, the complete waveset execution plans for script S1 are the following:

```
Waveset 1    UNION [dbase1, dbase2] 5 [dbase1, dbase1] [R1, 2]
Waveset 2    PROJECT [f1, f2] 3 [R1] [R3, 4]
             SELECT [condition(dbase3.f2)] 3 [dbase3] [R2, 4]
Waveset 3    JOIN [f1, f1] 5 [R3, R2] [R4, 1]
```

The sequence of waveset executions implements a particular execution plan of a query. PARIX reserves a fixed number of processors throughout the execution of a user program, e.g. FlexParDB executing a query plan. Therefore all wavesets have the same number of processors at their disposal. We next describe the execution of a waveset on N processors.

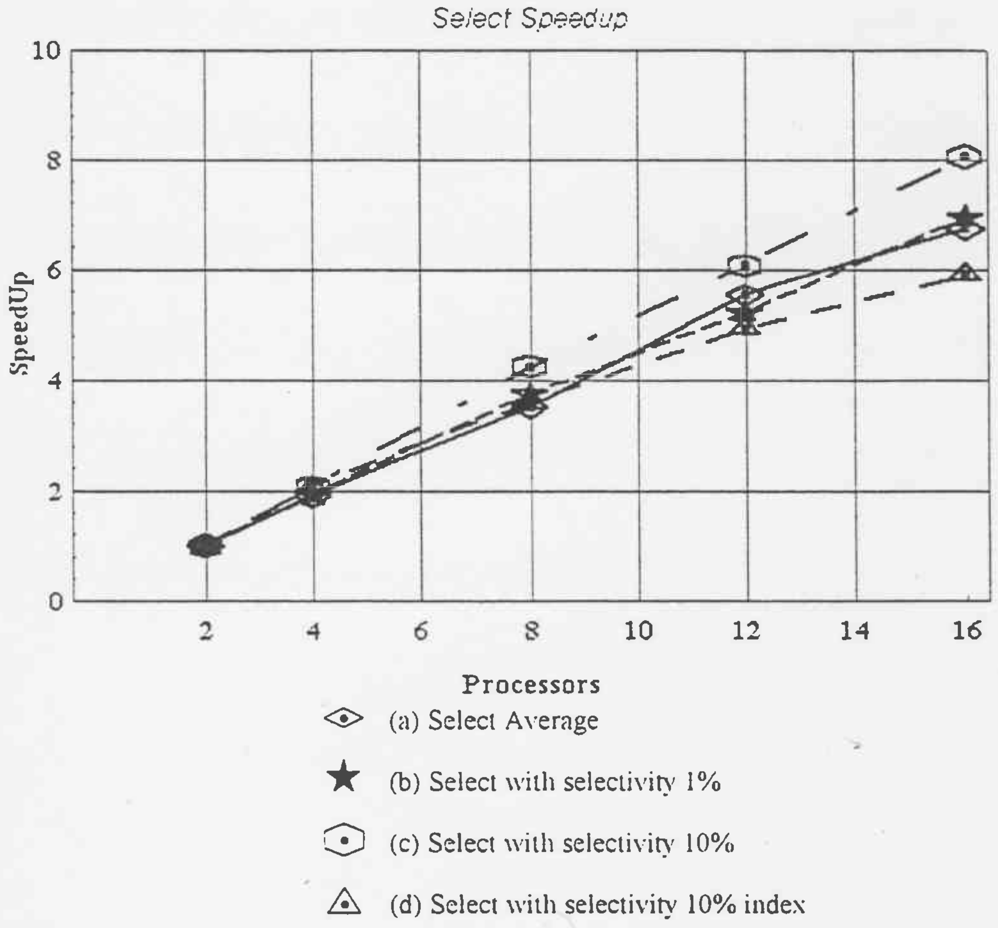
In phase B, the Waveset Execution, we initiate as many parallel PARDB executions as the number of elements in each waveset. Each PARDB executes one operator in the waveset. A termination mechanism decides when all PARDB executions of a waveset have terminated, so that the next waveset can be activated.

4. Performance

The performance of FlexParDB strongly depends on the performance of its building block, PARDB. PARDB executions for the same waveset have no data dependencies and do not require any communication. Therefore the time required for the execution of a waveset is equal to the time required for the slowest PARDB execution. The optimiser should ensure that processor numbers are allocated to the operators in proportion to their complexities and data volumes. In the rest of this section we shall therefore deal with the performance of individual PARDB executions.

We present performance measurements for two representative operators, select (uniscan) and join (multiscan). Graph 1 shows the select speedup in the case of an aggregation operator (a), a selection with 1% selectivity (b), a selection with 10% selectivity (c) and an indexed selection with

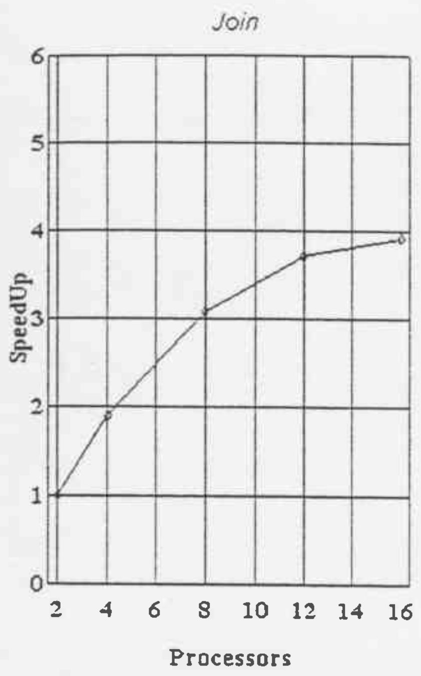
10% selectivity (d). They were all executed on a partitioned 100K-tuple relation and it can be seen that speedup is almost linear in all cases except the indexed one; even in this case, where the sequential algorithm is very efficient, there is significant speedup as the major cost in data base processing is disk I/O and this is divided among the multiple hard disks.



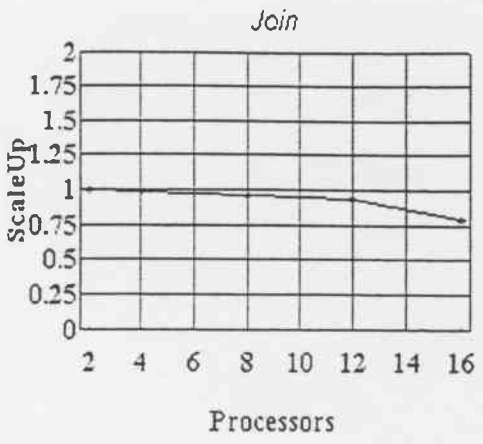
Graph 1. Select Speedup

Graph 2 shows join operator speedup on partitioned 1Kx10K-tuple relations. Here the high communication cost involved in exchanging partitions (join is a multiscan operator) limits the value of the speedup. Hash or bucket-based join algorithms could eliminate the need for communication during the join but it is doubtful whether they could significantly improve performance as the sorting step necessary before the join requires heavy communication.

Finally Graph 3 shows the results of a join scaleup experiment with relation sizes 1Kx10K, 2Kx10K, 4Kx10K, 6Kx10K and 8Kx10K giving join computational complexity proportional to the number of processors used in each case. The results again reflect the fact that data base operators are heavily I/O based; thus increasing the I/O throughput in line with relation size results in near-linear scaleup. The above measurements were conducted on a Parsytec Multicluster with 16 T800 Transputers.



Graph 2. Join Speedup



Graph 3. Join Scaleup

5. Conclusions and Future Work

FlexParDB combines intra-query and operator parallelism. Intra-query parallelism is expressed in the wavesets, which are a partition of the set of query-tree operators; valid wavesets are consistent with the flow of relational data from the leaves to the root of the query-tree. The wavesets represent the query execution plan. We have developed a script language for the description of a query-tree and its wavesets. Wavesets are executed by parallel multiple executions of PARDB, a system supporting operator parallelism.

FlexParDB fully controls the utilisation of a given number of processors. Any valid waveset and associated processor allocation can be described through our script language. A query optimiser can readily be integrated in FlexParDB through our flexible script language. The optimiser should produce the script and the associated wavesets which minimise the total query execution time.

The minimisation of total execution time in FlexParDB may be achieved by deciding on two interdependent factors: the wavesets and the allocation of processors to the operators within a waveset. Two criteria may be used: the processor utilisation criterion, requiring that all operators in a waveset terminate at the same time and the collective speed-up criterion, requiring that utilised processors be used efficiently to obtain the maximum speed-up. These are currently under investigation.

As a future step we envisage allowing pipeline processing within a wave set. We may describe such wavesets by the present script language; these are not currently valid due to the strict wave form property of section 3. We may relax this property by the complete sub-path property, stating that if an operator and one ascendant coexist in a waveset, then all ascendants of the operator in-between must also exist in the waveset. No other changes are necessary to the pre-processing phase of query execution. The PARDB system should be extended to deal with pipeline processing, by having as input and/or output either communication channels or files. A PARDB execution of an operator would therefore pass its output directly, via communication channels instead of files, to its parent PARDB execution, if they coexist in the same waveset. The technical feasibility has been demonstrated in the implementation of the Tree Pipelining query execution model [2]. We do not envisage that the extension of FlexParDB to include pipelining will raise significant problems, the design of an optimiser for a pipelined FlexParDB system is a challenging task. However, a solution should be obtained by combining results in Tree Pipelining optimisation and ongoing optimisation work for FlexParDB.

Acknowledgements: we would like to thank A. Paraskevopoulos, F. Lazarinis, S.Sfakianakis and N. Drivas for their help in coding FlexParDB.

References

[1] Baru K.C., & Sriram Padmanabhan, Join and Data Redistribution Algorithms for Hypercubes, *IEEE Transactions on Knowledge and Data Engineering*, 1993, 5(1), 161-168.

[2] Cotronis.J.Y., A Methodology for Initiating Arbitrary Structured Programs in Parix by Interpreting Graphs, in ZEUS 95 (eds P. Fritzon and L. Finmo), *Parallel Programming and Applications*, IOS Press 1995.

[3] Cotronis.J.Y., Efficient Program Composition on Parix by the Ensemble Methodology, in Proceedings *Euromicro Conference 96*, IEEE Computer Society, Prague 1996.

[4] Frieder O., Multiprocessor Algorithms for Relational-Database Operators on Hypercube Systems, *IEEE Computer*, November 1990, 13-28.

[5] Parsytec Computer GmbH, *Parsytec GC, Technical Summary*, Parsytec Computer GmbH, Julicher Strasse 338, D-5100 Aachen, 1991.

[6] Perihelion Software Ltd, *The Helios Operating System*, Prentice Hall International, 1989.

[7] Theoharis T., G. Papakonstantinou and P. Tsanakas. The design of PARDB; a parallel relational data base management system, in *Proceedings ISCIS VII,* Antalya, Turkey, November 1992.

[8] Spiliopoulou M., M. Hatzopoulos, J. Cotronis, Parallel Optimisation of Large Join Queries with Set Operators and Aggregates in a Parallel Environment Supporting Pipeline, *IEEE Transactions on Knowledge and Data Engineering* , 1996, 8(3), 429-445.