TRANSPUTER COMMUNICATIONS, VOL. 3(4), 233–260 (OCTOBER 1996)

# PARDB: Design, Algorithms and Performance of a Transputer Based Parallel Relational DBMS

THEOHARIS THEOHARIS AND AGGELOS PARASKEVOPOULOS

*Dept. of Informatics, University of Athens, TYPA Buildings, Panepistimioupolis, 157 71 Athens, Greece*

**SUMMARY**

**The main design choices involved in parallel RDBMSs are discussed and the design choices made for the authors' system, PARDB, are justified. The parallel architecture, the process structure and the parallel algorithms used to implement each of the relational operators in PARDB are described. Performance measurements show that near-linear speedup is achieved in uniscan operators (such as select), whereas multiscan operators (such as join) perform slightly worse because of the communication involved in exchanging relation data between the processors.**

KEY WORDS database management system; relation; parallelism; transputer

## 1 INTRODUCTION

Commercial and scientific database sizes are ever increasing. Applications such as banking, government (and particularly tax departments), geographical information systems, large engineering designs and scientific data visualization require database response times that cannot usually be provided within the required time frame given their large size. The exploitation of parallelism in database management systems (DBMS) has been a goal of DBMS designers for almost as long as these systems have been commercially available[1]. The demand for performance constantly outstrips the limits of existing technology, and parallel processing is the only means available to bridge this gap. Parallelism can improve a DBMS's performance in both the time and space dimensions by reducing processing time and by making the efficient control of multiple mass storage units possible. A parallel DBMS employs parallel processing techniques with the aim of achieving better performance and should not be confused with a distributed DBMS which has the aim of correctly managing geographically distributed data. Nevertheless, the two areas do share many techniques since parallelism requires some form of data distribution between processors.

A number of parallel relational database management systems (RDBMS) are currently under construction[2]. Most of those publicized are academic research projects, and we outline in this paper what we think are some representative systems. The authors' system, PARDB[3], is described in detail. The aim of the PARDB (PArallel Relational DataBase) project was to provide the operators of an RDBMS on a Transputer[4] based parallel machine running under the operating system Helios[5] in a form that is transparent to

*Received 11 June 1995*
*Revised 3 March 1997*

the user. In other words, the interface routines are the same as those of a sequential RDBMS, and the user only observes the increased performance and the existence of some extra routines which may be used in order to fine tune the parallel system. This implementation takes advantage of intra-query parallelism only; inter-query parallelism should be addressed in the future. Transputer technology provides the means to build cheap multiple instruction multiple data (MIMD) stream parallel processing systems (see Section 4.1) and has been available for over a decade. Unix-like operating systems, such as Helios, enable the relatively easy porting of existing software onto Transputer systems.

The rest of this paper is structured as follows. Section 2 outlines the major design alternatives that are open to the designer of a parallel RDBMS while Section 3 presents a survey of parallel RDBMS projects for comparison. Section 4 describes PARDB in detail; a description of the parallel processing platform is followed by the design choices that were made, an outline of the system's process structure and hardware architecture, a description of the parallel algorithms that implement each major class of relational operators in a CSP-like notation, and finally a presentation of the measured performance results. Section 5 concludes this paper and outlines our future research plans. A brief description of the CSP notation is given in Appendix 1 and the CSP algorithms for relational operators are presented in Appendix 2.

## 2 PARALLEL RDBMS DESIGN ALTERNATIVES

The following list of design alternatives is a result of the experience produced by a number of parallel RDBMS projects; see Section 3. It is by no means exhaustive, but it highlights the main decisions that a parallel RDBMS designer has to make.

### 2.1 Level of parallelism

As is the case in all parallel processing projects, the first question to answer is at what level the RDBMS will be parallelized. 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 first involves the processing of different queries in parallel, possibly on different partitions of the database, whereas the latter involves deciding how to best exploit the parallelism within a query. If a query involves the join of four relations R1, R2, R3 and R4 for example, R1 and R2 may be joined in parallel with the join of R3 and R4 by a system exploiting intra-query parallelism.

Operator parallelism arises when the function of a relational operator is distributed among different processors. This involves the partitioning of the database over multiple secondary storage devices and leads to the consideration of data distribution and load balancing issues.

Tuple parallelism, at the lowest level, attempts to increase processing power and secondary storage I/O bandwidth by dividing the bits making up each tuple of a relation among several secondary storage units connected to different processors. This is certainly effective and has few overheads, but can only result in a limited amount of speedup and requires expensive interprocessor communication in order to put a tuple together. A large number of systems use tuple parallelism.

### 2.2 Machine granularity

Machine granularity refers to the complexity of the individual processors of a parallel machine. Simple bit-processors, such as those found on the Goodyear MPP or the Connection Machine, usually come in the thousands and it seems attractive to assign a tuple per processor as is the case in [6]. Such *fine*-grained systems require a completely new RDBMS design and indexing in particular becomes difficult. Fewer, but more complex processors, such as the Inmos Transputer or the Intel iPSC, provide *coarse*-grain parallelism and allow more conventional RDBMS design and indexing. Stone[7] concludes that indexing, by significantly reducing I/O traffic, gives such a great advantage that serial algorithms using indexing can sometimes outperform massively parallel algorithms that do not use it.

### 2.3 Network topology

The topology of the parallel processor network should reflect design requirements. *Hypercubes*[8,9] minimize network diameter and are suitable when interprocessor communication is expected to be heavy. *Trees* minimize the length of the maximum broadcast path. For modest communication or when there is not a sufficient number of connections (e.g. T8xx Transputer links are limited to four) a *grid*, *ring* or *pipe* may suffice.

There are two main communication requirements of a parallel RDBMS. First there is the necessity for communication between the processors to exchange relation data in the parallel implementation of certain operators. Second is the need to broadcast commands to all processors and to receive the results of operations at a central site before presenting them to the user. The first requirement suggests minimization of the network diameter (hypercube) whereas the second one suggests minimization of the maximum broadcast path (tree). However, the type of interprocessor communication required here usually involves the distribution of each processor's inner relation partition to all other processors and this can be handled with reasonable efficiency on a ring architecture.

### 2.4 Sharing

Main memory and secondary storage are two resources that may or may not be shared among the processors of a parallel RDBMS and the following system classes exist: *shared-nothing* (SN), *shared-memory* (SM),[1] *shared-disks* (SD),[2] and *shared-everything* (SE)[10,11].

In the SN class each processor has its own local memory and disk(s) and thus owns a portion of the database which it can manipulate directly. The database is thus partitioned among the processors. SN architectures are easily scalable and can efficiently use cheap off-the-shelf disk units, but they suffer from data distribution and load-balancing problems. The SM approach attempts to make interprocessor communication and synchronization easier through the use of common memory, and furthermore avoids replication of the RDBMS code. Common memory, however, makes this architecture difficult to scale. SD on the other hand tries to alleviate the data distribution and load balancing problems by making all disks accessible to all processors (and thus loses in scalability). Finally SE shares both memory and hard disks and is the least scalable.

[1] Memory will refer to main memory.

[2] Disk will refer a secondary storage device.

### 2.5 Partitioning

There are two main methods for partitioning a database among a number of secondary storage devices; *horizontal partitioning* (or *declustering*) and *vertical partitioning*. The former distributes a relation's tuples (or rows) whereas the latter distributes the attributes (or columns) among the secondary storage devices. Horizontal partitioning enables the system processors to use normal RDBMS algorithms when operating on a partition of the relation. Interprocessor communication is only necessary when implementing multiscan[3] operators. In the case of vertical partitioning interprocessor communication is necessary in order to form any complete tuple by joining the distributed attribute values. However, vertical partitioning does not suffer from the uneven data distribution that may arise in horizontal partitioning.

Tuples can be partitioned horizontally using one of three main strategies; *hashing*, where a hash function of a tuple's attribute values determines where it shall go, *round robin*, where the tuples are arbitrarily divided among the storage units, and *range partitioning*, where each secondary storage unit stores only tuples with a certain range of attribute values. Hashing and range partitioning allow the implementation of multiscan operators without interprocessor communication when the hashing or range partitioning attribute is involved. However, multiscan operators based on other attributes require the repartitioning of the relations involved among the processors. In on-line transaction processing (OLTP), hashing and range partitioning are advantageous because transactions for a particular record can be directed to the secondary storage device upon which the record is known to be stored[12].

## 3 PARALLEL RDBMS SURVEY

Table 1 lists six parallel RDBMSs which, with the exception of Bubba, are academic research projects. It can be seen that certain design alternatives have been selected in most systems. This is a preview of the characteristics of future RDBMSs.

Bubba[13], Gamma[14], MDBS[15] and PARDB (Section 4) share a significant number of characteristics including operator parallelism, coarse grained processors, shared-nothing architecture and horizontal tuple partitioning. Coarse grained processors allow the use of existing RDBMS techniques and code (including the significant advantage of indexing) whereas the shared-nothing architecture and horizontal partitioning both serve the purpose of system expandability. Operator parallelism does not exclude query or tuple parallelism and one can envisage systems taking advantage of all three types of parallelism. The hypercube topology is also a common choice because it minimizes network diameter, and is expandable and theoretically appealing.

XPRS[16] follows a different philosophy; it uses a shared-memory architecture to aid interprocessor communication and load-balancing, and proposes a different relation partitioning scheme which is a combination of horizontal and vertical partitioning. It mainly addresses query parallelism but there are plans to exploit all three types of parallelism.

Finally the MPP approach[6] is an attempt to implement relational operators on a fine-grained SIMD machine by assigning a tuple to the memory of each processing element; in such a fine-grained architecture it is not viable to provide a secondary storage unit for each processing element. Though elegant, it is not easy to see how all aspects of a RDBMS can

[3] As defined in [8] a multiscan operator is one in which 'the processing of an individual tuple involves comparing its attribute value(s) against other tuples'.

Table 1. Parallel RDBMS

| | Bubba | Gamma | MDBS | PARDB | XPRS | MPP |
|---|---|---|---|---|---|---|
| Level of parallelism | Operator | Operator | Operator | Operator | Query & Operator & Tuple | Operator |
| Granularity | Coarse | Coarse | Coarse | Coarse | Coarse | Fine |
| Topology | HY | HY/Ring | Bus | Ring/HY | ? | Grid |
| Sharing | SN | SN | SN | SN | SM | SD |
| Partitioning | HO | HO | HO | HO | HO & VE | HO |
| Partitioning algorithm | HA/RA | HA/RA/RR | ? | RR | HA/RA | RR |

**HY:** Hypercube. **SM:** Shared-memory. **HO:** Horizontal. **HA:** Hashing. **SD:** Shared-disks. **SN:** Shared-nothing. **VE:** Vertical. **RA:** Range. **RR:** Round Robin.

be efficiently implemented on such a system.

Other attempts at Transputer based RDBMSs have been made in the past. Akaboshi[17] presents experimental Transputer based hardware for the Join (only) operator, and Bryan[18] reports on the design of a Transputer-based database machine called MEDUSA for the storage of GIS information. England[12] compares two commercial transaction processing modes (OLTP, MIS) in the banking and financial sector.

## 4 PARDB

The following subsections describe the authors' parallel RDBMS which was built at the National Technical University of Athens and the University of Athens.

### 4.1 Parallel processing platform

The parallel hardware used was a Parsytec Multicluster-2. It consists of 16 T800[4] Transputers, an interconnection network, a large external hard disk and a SPARCstation host (see Figure 1). Each Transputer is a floating point processor with 4 Kbytes of on-chip memory, four on-chip links which enable it to be connected to other Transputers, and 2 Mbytes of external RAM. At present, all links of all 16 Transputers are connected to the interconnection network which can join any link to any other and thus allows the construction of a MIMD system of arbitrary topology consisting of up to 16 nodes. The interconnection network is made up of a hierarchy of Inmos C004 link switch chips.

The Helios operating system controls the above hardware; it is Unix-like but unfortunately not Unix compatible. Producing a parallel application under Helios involves the following steps:

1. The physical network interconnections are defined in a special file (resource map file).

[4] The floating point capabilities of the T800 Transputer are not extensively used in database processing and, in a commercial environment, such a system could be built out of lower-end Transputers.

2. The application is written as a number of separate C processes communicating between them via streams.
3. A logical network topology is requested for the C processes using a special Component Description Language (CDL).

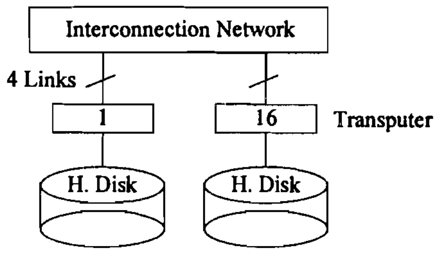

**Figure 1.** Parallel processing platform

Helios undertakes to map the logical network onto the physical one as well as possible. Virtual links that cannot be mapped onto physical links, are implemented transparently to the user by Helios using message routing through the network.

The C processes defined in the CDL script will usually execute on separate Transputers (true parallel processing). However Helios allows multiple processes to run on the same Transputer (pseudoparallelism). This arises out of a specific user request (multithreading) or when hardware resources are insufficient (multitasking).

## 4.2 Design choices

The design of PARDB was guided by the following objectives:

1. interface compatibility with serial RDBMS
2. scalability and speedup
3. Transputer based
4. code reliability
5. short development time

The choices that have been made for each of the design alternatives described in Section 2 are presented in this section. The system exploits operator parallelism on a shared nothing (SN) architecture where the tuples are horizontally partitioned among the processors; initially only the simpler round robin distribution strategy is considered.

The SN architecture can easily be scaled by replication; it does not involve any shared resource which would be a potential performance bottleneck. It is assumed that each Transputer has access to a local hard disk (see Figure 1) where it can store its partition of the database (disk I/O is the usual performance limiting factor). The existing hardware has only one hard disk. To circumvent this problem a simulator was built[19] which accurately estimates the performance that would be achieved if a hard disk per processor really existed. Transputer hardware is ideal for implementing SN systems; distributed memory and communication via messages is part of the philosophy behind the Transputer parallel processing model.

Another advantage of the SN system is that every one of its processor nodes can be seen as an independent sequential RDBMS. A conventional RDBMS can thus be taken off-the-shelf and run at every processor node, resulting in a significant saving in development effort and exploiting the reliability of a thoroughly tested RDBMS. Under this scheme each processor node runs the RDBMS on a partition of the database and holds the corresponding sub-indices. An important goal was interface compatibility i.e. making the application interface identical to that of the original off-the-shelf RDBMS.

PARDB was built around the concept of operator parallelism which naturally fits the goal of interface compatibility. Tuple parallelism has been avoided in order to keep the original RDBMS code. Horizontal partitioning is therefore used for tuple distribution between the processors; if multiple hard disks were connected to each processor, it would be possible to use vertical partitioning in order to increase the I/O bandwidth into the hard disks. Query parallelism can be added at a higher level (see Section 5).

## 4.3 System structure

A number of sequential RDBMS's were considered by the authors, who finally selected 'UMDBASE'[20] as the basis for the project, for the following reasons:

1. Availability of C source code.
2. Nicely layered structure (paged file, heap file & access method layers).
3. Informal support by T. Sellis of the University of Maryland.
4. Free of charge.

UMDBASE was ported onto the Helios and subsequently onto the Parix operating systems and runs on every Transputer that holds a partition of the database. Such a Transputer, along with the software that it hosts, is called a *slave*. A slave independently manages its local database partition and stores all the relevant data (such as subindices and RDBMS code).

One Transputer is reserved to provide a unique interface to applications (which is almost identical to the interface of the original sequential RDBMS) and for delegating the work to the appropriate slaves through message passing. This Transputer and the associated software is the *master*. The master does not take part in the execution of relational operators and thus does not constitute a bottleneck. PARDB can comprise many slaves but only one master (see Figure 2).

The master interfaces with the application via a script file that contains relational commands. The current list of commands includes Create, Destroy, Append, Delete, Project, Select, Union, Intersection, Difference, Join, Load (load a database onto the disk of each slave), Collect (dump a database 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).

A table of the number of tuples per relation per slave is kept by the master (see Figure 3). This important table is called the *relation table*. The relation table is maintained by a service routine, *reltable_service*, which is called by the master to create a new entry, delete an existing one, change the number of tuples of a particular slot or get statistical information necessary for the efficient functioning of PARDB. This table gives the master a picture of

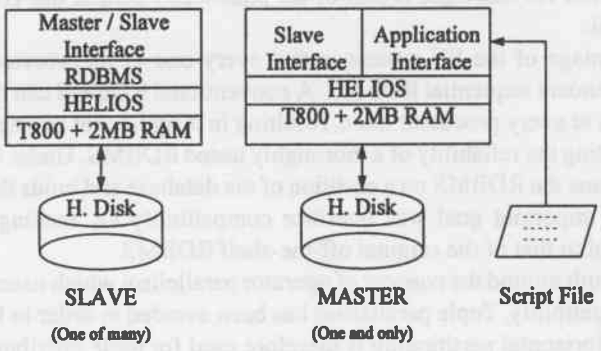

**Figure 2.** Parallel processing platform

the size of each partition and is always kept up to date; all changes to tuple distribution are recorded immediately in the central relation table.

After a script file command is read by the master, messages are sent to the slaves which execute the command (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 any results and report successful termination of a command. The master finally records any changes in the relation table.

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 database(s), it then exchanges partition data with other slaves (necessary for multiscan 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[21].

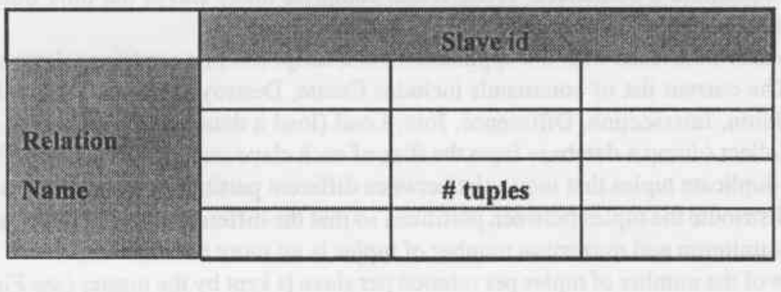

**Figure 3.** Relation table

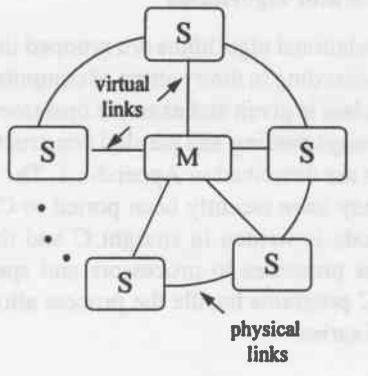

**Figure 4.** Communication structure

Global partitioning algorithms redistribute the tuples of the relation to be operated on 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. So, for example, the partitions of all relations resident on processor 0 may contain only tuples whose field name is in the range AAAAAAAA to AZZZZZZZ. With global partitioning, each processor need only perform the relational operators on its local tuples, after redistribution. The redistribution itself is usually based on sorting or hashing techniques. Global partitioning avoids the exhaustive comparison of every tuple of one relation with every tuple of the other, but incurs the extra cost of partitioning. The sorting or hashing of the global partitioning method is made more efficient by minimizing the network diameter; the hypercube is then a suitable choice, and good sorting and hashing algorithms exist for this topology. Our 16 T800s can be configured so as to form a 4D hypercube of diameter 4.

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. It was chosen to opt for the simpler ring architecture (in which all of our available processors can easily be connected using physical – as opposed to virtual – links). Correspondingly our algorithms were implemented using the cycling technique. Direct (virtual) links also exist between the master and each slave for the transmission of control (relational commands and arguments) and housekeeping information; see Figure 4. One *thread* per link handles communication, thus enabling the parallel execution of link I/O and relational processing on each node (pseudo-parallel in the case of virtual links). Threads (lightweight parallel processes) are provided by the Helios and PARIX operating systems. In the future it is hoped to implement the hypercube/global partitioning version and compare the two.

### 4.4 Parallel relational operator algorithms

In this section, the parallel relational algorithms are grouped into three major classes *uniscan*, *multiscan* and *append* according to their pattern of computation and communication. A general description of each class is given and example operators are described in a superset of C enriched with CSP message passing and parallel constructs[22]; see Appendix 1. The remaining parallel operators are described in Appendix 2. The actual implementations are in Helios C and CDL[5]; they have recently been ported to C under the PARIX OS[23]. Under Helios, sequential code is written in straight C and the CDL language acts as a 'parallel glue' that allocates processes to processors and specifies the interconnections between them. In PARIX, C programs handle the process allocation and the interprocess communication using OS libraries.

#### *4.4.1 Uniscan operators*

A uniscan operator is an operator whose processing does not require the comparison of each tuple with other tuples. This class comprises the *select*, *project*, *union*, *delete* and *aggregate* relational operators. The last operator computes an aggregate of a specific attribute of a relation based on a selection filter; it requires slightly different processing and is treated separately.

##### 4.4.1.1 Description

Uniscan operators are implemented in three steps as shown in Figure 5:

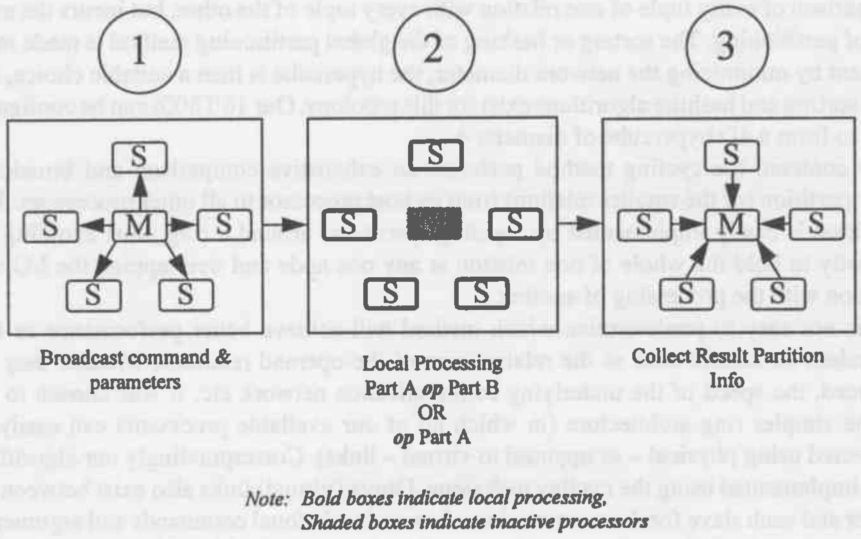
1
2
3
S
S
M
S
S
S
S
S
S
S
S
S
S
M
S
S
S
Broadcast command & parameters
Local Processing
Part A *op* Part B
OR
*op* Part A
Collect Result Partition Info
*Note:* **Bold boxes indicate local processing, Shaded boxes indicate inactive processors**

**Figure 5.** Uniscan processing

Step 1. The uniscan operator and associated parameters are broadcast from the master to the slaves.
Step 2. All slaves execute in parallel the sequential relational operator on their local database partitions (one or two operand partitions depending on the operator).
Step 3. Information on each resulting partition (size etc.) is collected by the master in order to update central data structures.

In the case of aggregate an extra step is performed by the master:

Step 4. Computation of global aggregate from partial aggregates (Figure 6).

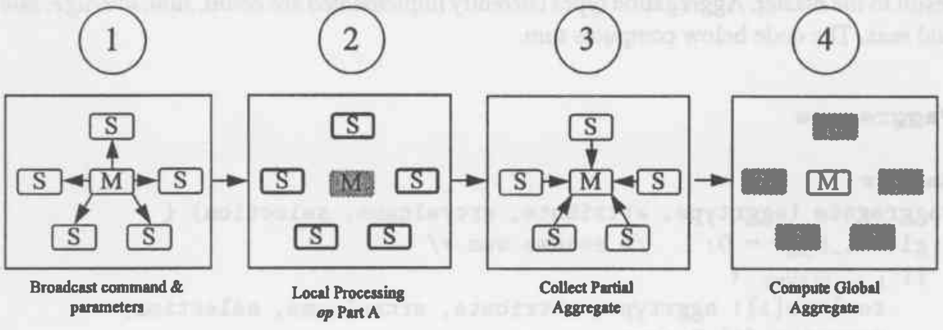
1
2
3
4
S
M
Broadcast command & parameters
Local Processing *op* Part A
Collect Partial Aggregate
Compute Global Aggregate

**Figure 6.** Aggregate processing

4.4.1.2 Example operator – select and aggregate
The select operator acts as a selector of tuples from the source relation based on a selection filter. The resulting tuples are placed in a new relation.

The master creates a new entry in the relation table for the relation resulting from the selection, it sends the select operator arguments to the slaves, receives the size of each newly created partition from them and updates the relation table. The slaves operate correspondingly.

**Pselect**

```
Master
Pselect (resrelname, srcrelname, selection) {
  reltable_sevice(create, resrelname, -, -,);
  || i in SLAVES  {
     toslave[i] ! resrelname, srcrelname, selection);
     fromslave[i] ? numtuples;
     reltable_service (add, resrelname, i, numtuples);
  }
}
```

**Slave**

```
frommaster ? resrelname, srcrelname, selection ;
n_tup=select (resrelname, srcrelname, selection);
tomaster ! n_tup
```

In the case of aggregate, the master sends the aggregation operator arguments to the slaves, gets the aggregation result from each slave and computes the global aggregate over all the slaves. This is computed as an atomic operation; the master I/O threads use a binary semaphore for this purpose. Correspondingly each slave receives the aggregation arguments from the master, computes the aggregate on its own partition of the relation and sends the result to the master. Aggregation types currently implemented are *count*, *sum*, *average*, *min* and *max*. The code below computes sum.

**Paggregate**

**Master**
```
Paggregate (aggrtype, attribute, srcrelname, selection) {
  global_aggr = 0;     /* assume sum */
   || i in SLAVES  {
     toslave[i]! aggrtype, attribute, srcrelname, selection;
     fromslave[i] ? aggr;
     wait (binary_sema);
     global_aggr = global_aggr + aggr;   /* assume sum */
     signal (binary_sema);
  }
}
```

**Slave**

```
frommaster ? aggrtype, attribute, srcrelname, selection;
aggr = aggregate (aggrtype, attribute, srcrelname,
                                        selection);
tomaster ! aggr;
```

### *4.4.2 Multiscan operators*

A multiscan operator is an operator whose processing requires the comparison of each tuple with other tuples. This class comprises join, difference and intersection.

#### 4.4.2.1 Description

A multiscan operator is implemented in four steps, as shown in Figure 7.

Step 1. The multiscan operator and associated parameters are broadcast from the master to the slaves.

Repeat *P* times[5]

Step 2. All slaves execute in parallel the sequential relational operator on their local database partitions.

Step 3. The partitions of the smaller relation are circulated by one hop[6] around the ring.

End

Step 4. Information on each resulting partition (size etc.) is collected by the master in order to update central data structures.

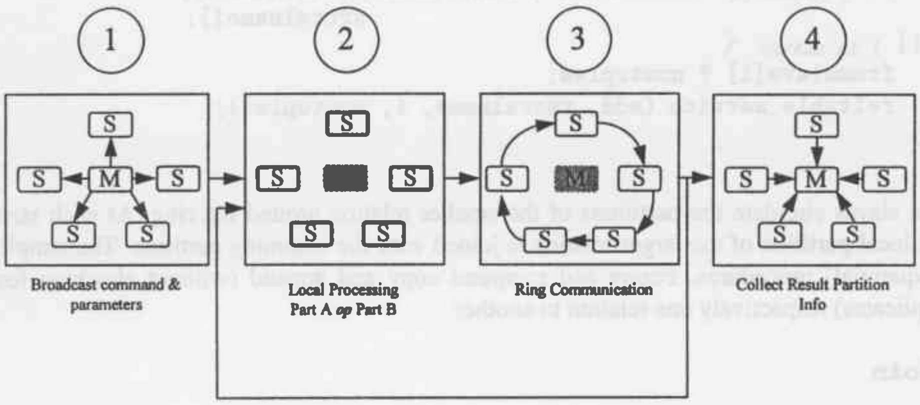
1
2
3
4
S
M
Broadcast command & parameters
Local Processing Part A *op* Part B
Ring Communication
Collect Result Partition Info

**Figure 7.** Multiscan processing

4.4.2.2 Example operator – join

A distributed version of the nested loop join algorithm was implemented. The smaller of the two relations is cycled around the ring. Join is commutative; this implies that the algorithm is the same irrespective of which relation is smaller (contrast that to the Difference operator in Appendix 2). The master broadcasts the identity of the larger relation to the slaves, along with the other join parameters, and then waits until the numbers of tuples in each partition of the resulting relation are transmitted back from the slaves; this information is then used by the master to update the relation table.

The join algorithm does not attempt to balance the operand relations nor does it guarantee that the result relation will be balanced. Rather, the task of balancing is left to the user who can explicitly call the balancing routine. The performance of the join algorithm can be significantly improved by providing it with balanced operand relations.

Duplicate tuples would not arise as a result of the join operation provided that they did not exist before in the source relations.

[5] *P* is the number of processors in the ring

[6] That is, each slave receives a partition from its predecessor in the ring.

```
Pjoin

Master
Pjoin (resrelname, srcrelname1, srcrelname2) {
  reltable_service (create, resrelname, -, -);
  size1 = reltable_service (total, srcrelname1, -, -);
  size2 = reltable_service (total, srcrelname2, -, -);
  if (size1 > size2)
    || i in SLAVES  {toslave[i] ! resrelname, srcrelname1,
                                            srcrelname2};
  else
    || i in SLAVES  {toslave[i] ! resrelname, srcrelname2,
                                           srcrelname1};
  || i in SLAVES  {
    fromslave[i] ? numtuples;
    reltable_service (add, resrelname, i, numtuples);
  }
}
```

The slaves circulate the partitions of the smaller relation around the ring. At each step the local partition of the larger relation is joined with the incoming partition. The simple (sequential) procedures, rcopy and rappend copy and append (without checking for duplicates) respectively one relation to another.

```
Pjoin

Slave
frommaster ? resrelname, bigsrcrel, smallsrcrel;
create(temprel);
create (relbuf1);
create(relbuf2);
rcopy (relbuf1, smallsrcrel);
n_tup = join (resrelname, bigsrcrel, relbuf1);
for (i = 1; i < P; i++) {
  if odd(i) {
    {fromleftslave ? relbuf2 || torightslave ! relbuf1};
    n_tup = n_tup+join (temprel, bigsrcrel, relbuf2);
    rappend (resrelname, temprel);
  }
  else {
    {fromleftslave ? relbuf1 || torightslave ! relbuf2 };
    n_tup = n_tup+join (temprel, bigsrcrel, relbuf1);
    rappend (resrelname, temprel);
  };
};
destroy (temprel);
destroy (relbuf1);
destroy (relbuf2);
tomaster ! n_tup;
```

### 4.4.3 *Append*

Append is the procedure that inserts a new tuple to a relation and is implemented in three steps (Figure 8).

Step 1. The master uses the relation table service routine to determine the processor identifier (pid) of the slave with the minimum number of tuples for the given relation and then transmits the relation name and the tuple to that slave.
Step 2. The selected slave appends the tuple to its local partition.
Step 3. The master receives acknowledgement from the slave and updates the relation table.

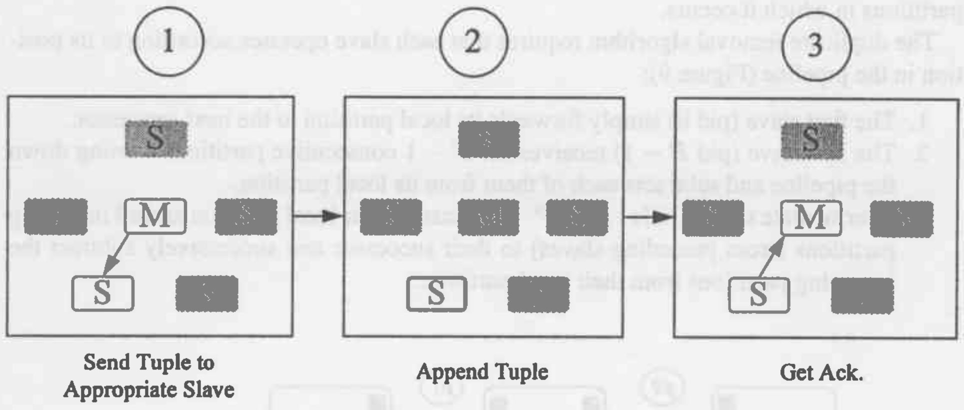

**Figure 8.** Append processing

The code for this operation follows.

```
Pappend

Master
Pappend (relname, newtuple){
  minpid = reltable_service (min,relname,-,-);
  toslave[minpid] ! relname, newtuple;
  reltable_service (add,relname,minpid,1);
}

Slave[minpid]
frommaster ? relname, newtuple;
append (relname, newtuple);
```

### 4.4.4 *Duplicate removal*

The duplicate removal utility eliminates duplicate tuples that may exist between different partitions of a relation. Duplicates may arise from the execution of the project, union or append operators. Local duplicates (within a partition) are not dealt with, as these are supposed to be cleared up as part of the execution of any sequential operator that may give rise to such duplicates.

The duplicate removal utility is under the explicit control of the user. It is the only algorithm that does not exploit the symmetrical properties of the ring, but instead, requires pipeline connectivity (i.e. the link between slave $P$ and slave 0, is unused). It achieves its objective by passing the partitions through the pipeline and, at each stage, subtracting the incoming partition from the local partition using the local difference operator. In designing the algorithm care was taken to ensure that a duplicate tuple is not subtracted from all the partitions in which it occurs.

The duplicate removal algorithm requires that each slave operates according to its position in the pipeline (Figure 9):

1. The first slave (pid 0) simply forwards its local partition to the next processor.
2. The last slave (pid $P-1$) receives the $P-1$ consecutive partitions coming down the pipeline and subtracts each of them from its local partition.
3. Intermediate slaves (pids $1, \ldots, P-2$) transmit their local partition and all incoming partitions (from preceding slaves) to their successor and successively subtract the incoming partitions from their local partition.

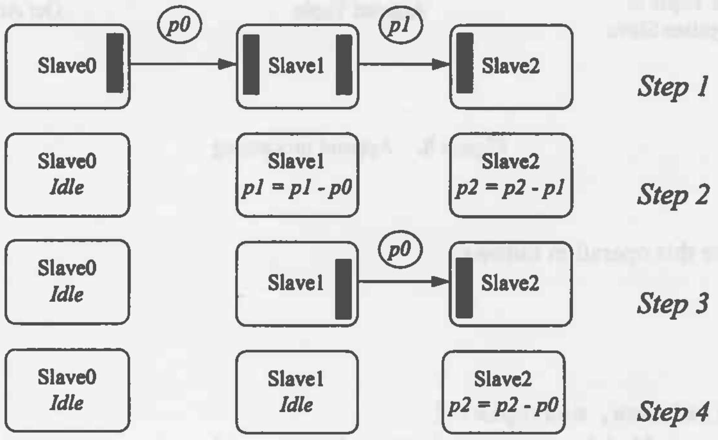

**Figure 9.** Duplicate removal utility

The master simply broadcasts to the slaves the name of the relation and then waits for the final partition size of each slave so that it can update the relation table. Each slave implements the algorithm discussed above using two buffers in order to process input and output in parallel.

**Prm_dupl**

```
Master
Prm_dupl (relname) {
  || i in SLAVES{
     toslave[i] ! relname;
     fromslave[i] ? numtuples;
     reltable_service (set, relname, i, numtuples);
}
```

**Prm_dupl**

```
Slave
from master ? relname;
create(relbuf1);
create(relbuf2);
rcopy(relbuf1, relname);
toggle = TRUE;

for(i = 1; i <= pid; i++) {
  if toggle {
    if pid < (P-1) {
       fromleftslave ? relbuf2 ||
       torightslave ! relbuf1
    }
    else fromleftslave ? relbuf2;
    n_tup = difference (relname, relname, relbuf2);
  }
  else {
    if pid < (N-1) {
       fromleftslave ? relbuf1 ||
       torightslave ! relbuf2
    }
    else fromleftslave ? relbuf1;
    n_tup = difference (relname, relname, relbuf1);
  }
  toggle = ~toggle;
}

if pid < (P-1) {
   if toggle torightslave ! relbuf1
   else torightslave ! relbuf2
}
destroy (relbuf1);
destroy (relbuf2);
tomaster ! n_tup;
```

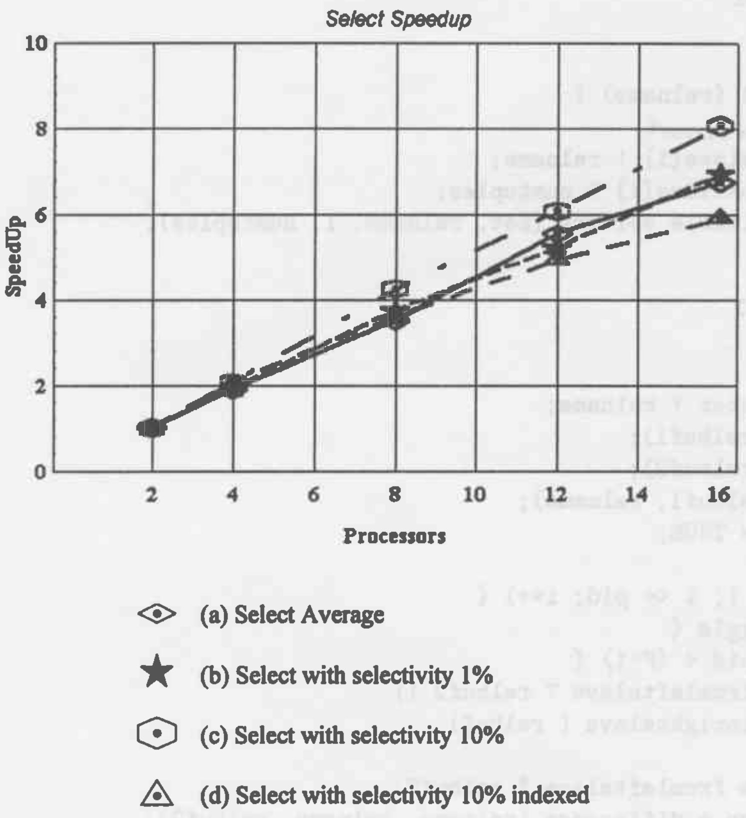

**Figure 10.** Select speedup

## 4.5 Performance results

Performance measurements for two representative operators, select (uniscan) and join (multiscan) are presented. Other operators within each class (uniscan, multiscan) exhibit similar performance.

Figure 10 shows the select speedup in the case of an aggregation (uniscan) operator (a), a selection with 1 per cent selectivity (b), a selection with 10 per cent selectivity (c) and an indexed selection with 10 per cent 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 database processing is disk I/O and this is divided among the multiple hard disks. The query used involved a single selection filter; it is expected that speedup figures will improve further if complex select queries are used because the resulting higher computational complexity will be divided among the processors.

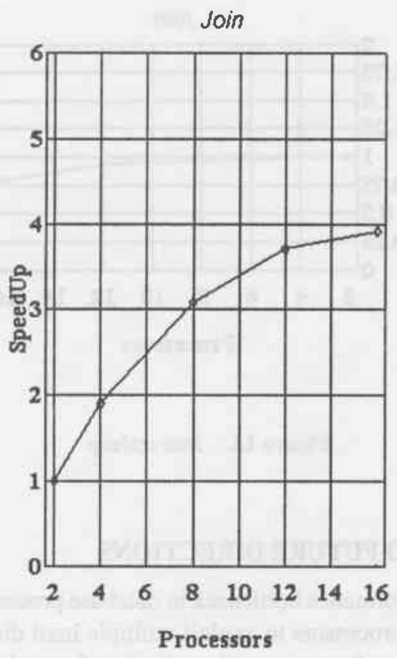

Figure 11. Join speedup

Figure 11 shows join operator speedup on partitioned 1K× 10K-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. Complexity analysis (to be published) has shown that it should be possible to improve multiscan operator performance by pre-sorting each database partition (if an index on required attribute does not exist). A merge-type algorithm with linear complexity can then be used to execute the operation.

Finally Figure 12 shows the results of a join scaleup experiment with relation sizes 1K×10K, 2K×10K, 4K×10K, 6K×10K and 8K×10K. The computational complexity is proportional to the number of processors used in each case. The results again reflect the fact that database operators are heavily I/O based, thus increasing the I/O throughput in line with relation size results in near-linear scaleup.

Measurements involving 16 processors have slightly worse performance than expected because in this case there are not enough processors in the system for a one-to-one process to processor allocation. Thus one slave is run on the same processor as the master process and another slave on the same processor as the multiple hard disk simulator.

Unfortunately system resources (single hard disk also accessible by other users) did not allow experimentation with very large databases.

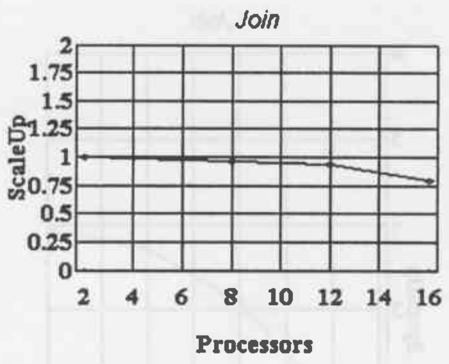

**Figure 12.** Join scaleup

## 5 CONCLUSIONS AND FUTURE DIRECTIONS

I/O is usually the main performance bottleneck in database processing. The shared-nothing approach allows multiple processors to exploit multiple hard disks and thus increase I/O (as well as computational) performance with no limit as far as the system is concerned.

Given the relational operator algorithms which we have presented in this paper, such systems can be built very cheaply from off-the-shelf processors with built-in communication capability (e.g. Transputers) and cheap hard disks. The authors' Transputer implementation showed that the most significant performance gains are in the case of uniscan operators.

The area of parallel RDBMS's lacks significant complexity results which can reveal the theoretical limits of such technology. The authors are currently working on formal specifications and complexity analysis for the parallel algorithms involved (in cooperation with Dr. A. E. Abdallah of the University of Reading).

Further speedup is possible by exploiting intra-query parallelism. The authors are at present investigating intra-query parallelism in conjunction with the possibility of avoiding the use of hard disk storage at internal node of the query tree.

Alternative connectivity topologies (e.g. hypercube) and partitioning strategies (e.g. hashing) should be implemented on the same system and compared.

Commercial application of academic research results such as those presented in this paper, requires that some standard database facilities be provided. These include concurrency control for OLTP, referencial integrity, security and recovery.

## APPENDIX 1: CSP NOTATION BASICS

The notation used for the description of the parallel algorithms is blackboard C enriched with CSP[22] parallel constructs. In CSP a number of independent processes communicate via channels. The parallel constructor symbol '||' indicates the parallel execution of processes and it is extended to accept a set index where necessary, e.g. $||_{e \text{ in set}}$ {process with parameter e}. The '?' symbol indicates reception of a message on a channel and the '!' symbol indicates transmission on a channel. For example channelname ? variable, channelname !

value. Reception and transmission are point-to-point, synchronized, and unbuffered. The sequential process combination symbol is the semicolon.

## APPENDIX 2: PARALLEL RELATIONAL OPERATORS

### A2.1 Uniscan operators

#### *A2.1.1 Delete*

To delete a selection of tuples, the master broadcasts the relation name and selection filter to all slaves, then receives from them the number of tuples deleted and finally updates the relation table. Each slave receives the relation name and selection filter from the master, deletes the required tuples using the sequential RDBMS delete operator and returns the number of deleted tuples to the master.

**Pdelete**

**Master**
```
Pdelete (relname, selection) {
  || i in SLAVES  {
    toslave[i] ! relname, selection;
    fromslave[i] ? numdeleted;
    reltable_service (subtract, relname, i, numdeleted);
  }
}
```

**Slave**
```
frommaster ? relname, selection;
n_tup = delete (relname, selection);
tomaster ! n_tup
```

#### *A2.1.2 Project*

The master creates a new table entry for the result relation, sends the project operator arguments to the slaves, receives the size of each partition of the new relation from the slaves and updates the relation table.

Each slave receives the project operator arguments from the master, creates a local relation for the storage of the result, executes the sequential project command and sends the number of tuples in the resulting relation to the master. Duplicate tuples that arise as a result of the project operation within any one partition are removed by the sequential project operator. However duplicates between different partitions may arise and these are not automatically removed. The user must explicitly request their removal by calling the RmDupl operator, described in Section 4.4.4.

**Pproject**

**Master**
```
Pproject (resrelname, srcrelname, project_attributes) {
  reltable_service (create, resrelname,-, -, );
  || i in SLAVES {
     toslave[i] ! resrelname, srcrelname,
                  project_attributes;
     fromslave[i] ? numtuples;
     reltable_service (add, resrelname, i, numtuples);
  }
}
```

**Slave**

```
frommaster ? resrelname, srcrelname, project_attributes;
create (resrelname);
n_tup = project(resrelname, srcrelname, project_attributes);
tomaster ! n_tup
```

*A2.1.3 Union*

The master creates a new table entry for the result relation, sends the union arguments to the slaves, receives the size of each new partition of the result relation from the slaves and updates the relation table. The slaves act correspondingly on their local partitions. In a similar manner to the project operator, duplicate tuples may exist between different partitions (processors) of the result relations. It is left to the user to execute the RmDupl operator on the result relation.

**Punion**

**Master**

```
Punion (resrelname, srcrelname1, srcrelname2) {
  reltable_service (create, resrelname, -, -,);
  || i in SLAVES {
     toslave[i] ! resrelname, srcrelname1, srcrelname2 ;
     fromslave[i] ? numtuples;
     reltable_service (add, resrelname, i, numtuples);
  }
}
```

**Slave**

```
frommaster ? resrelname, srcrelname1, srcrelname2;
create (resrelname);
n_tup = union (resrelname, srcrelname1, srcrelname2);
tomaster ! n_tup;
```

### A2.2 Multiscan operators

*A2.2.1 Intersection*

The master creates a new table entry for the new relation, determines the sizes of the two operand relations and sends the names of all three relations to the slaves (larger operand before smaller). It then receives the size of each partition of the result relation from the slaves and updates the relation table accordingly.

**Pintersection**

**Master**

```
Pintersection (resrelname, srcrelname1, srcrelname2) {
  reltable_service (create, resrelname, -, -);
  size1 = reltable_service (total, srcrelname1, -, -);
  size2 = reltable_service (total, srcrelname2, -, -);
  if (size1 > size2)
    || i in SLAVES  {toslave[i] ! resrelname, srcrelname1,
                                   srcrelname2 };
  else
    || i in SLAVES  {toslave[i] ! resrelname, srcrelname2,
                                   srcrelname1};
    || i in SLAVES  {
     fromslave[i] ? numtuples;
     reltable_service (add, resrelname, i, numtuples);
  }
}
```

The slaves implement the intersection by ‘cycling’ the partitions of the smaller of the two operand relations around the ring. Intersection is commutative and the algorithm remains the same irrespective of which relation is larger. At each loop execution a slave receives a partition of the smaller relation from its left neighbour, intersects it with its local partition of the larger relation and appends the result to the result relation. The number of tuples in each partition of the resulting relation is then sent to the master by every slave. The code fragments below make use of two buffers (`relbuf1` and `relbuf2`) which aid the circulation of the smaller relation around the ring. Each of them is shown to be big enough to store a whole partition of the smaller relation. In practice however, communication message sizes close to the ideal message size are used and these also determine the size of these buffers.

**Pintersection**

**Slave**

```
frommaster ? resrelname, bigsrcrel, smallsrcrel;
create (resrelname):
create (temprel);
create (relbuf1);
create (relbuf2);
rcopy (relbuf1, smallsrcrel);    /* relbuf1 = smallsrcrel */
n_tup = intersection (resrelname, bigsrcrel, relbuf1);
for (i = 1; i < P; i++) {
  if odd (i) {
    {fromleftslave ? relbuf2 || torightslave ! relbuf1};
    n_tup = n_tup + intersection (temprel, bigsrcrel,
                                  relbuf2);
    rappend (resrelname, temprel);
  }
  else {
    {fromleftslave ? relbuf1  ||  torightslave ! relbuf2};
    n_tup = n_tup + intersection (temprel, bigsrcrel,
                                  relbuf1);
    rappend (resrelname, temprel),
  };
} ;
destroy (temprel);
destroy (relbuf1);
destroy (relbuf2);
tomaster ! n_tup;
```

### A2.2.2 *Difference*

Difference is also handled in a way which allows the smaller of the two operand relations to rotate around the ring of slaves. The master compares the sizes of the two operands and sends their names to the slaves along with a message indicating which is larger. It then waits for the slaves to return the number of tuples in each partition of the result relation and updates the relation table.

**Pdifference**

**Master**

```
Pdifference (resrelname, srcrelname1, srcrelname2){
  reltable_service (create, resrelname, -, -);
  size1 = reltable_service (total, srcrelname1, -, -);
  size2 = reltable_service (total, srcrelname2, -, -);
  if (size1 > size2) bigger = 1 else bigger = 2;

  || i in SLAVES {
       toslave[i] ! resrelname, srcrelname1, srcrelname2,
                    bigger;
     fromslave[i] ? numtuples;
     reltable_service (add, resrelname, i, numtuples);
  }
}
```

Difference is not commutative and the operation of the slave depends on which operand is smaller. If relation 2 is smaller (the one being subtracted) then its partitions are circulated around the ring, at each step being subtracted from the local partition of relation 1 of each slave. If relation 1 is the smaller then it is the one circulating around the ring; at each step the local partition of relation 2 is subtracted from the circulating partition of relation 1.

**Pdifference**

**Slave**

```
frommaster ? resrelname, srcrelname1, srcrelname2, bigger;
create (resrelname);
create (relbuf1);
create (relbuf2);
if (bigger == 1) {
  rcopy (relbuf1, srcrelname2);
  rcopy (resrelname, srcrelname1);
}
else rcopy (relbuf1, srcrelname1);
for(i = 0; i < P; i++) {
  if even (i) {
    {fromleftslave ? relbuf2 || torightslave ! relbuf1};
    if (bigger == 1) n_tup = difference ( resrelname,
                                    resrelname, relbuf2)
    else n_tup = difference(relbuf2, relbuf2, srcrelname2);
  }
  else {
    {fromleftslave ? relbuf1  || torightslave ! relbuf2};
    if (bigger == 1) n_tup = difference (resrelname,
                                    resrelname, relbuf1)
    else n_tup = difference(relbuf1, relbuf1, srcrelname2);
  };
};
if (bigger == 2) {
  if even(P) rcopy(resrelname, relbuf1);
  else rcopy(resrelname, relbuf2);
};
destroy (relbuf1);
destroy (relbuf2);
tomaster ! n_tup;
```

## A2.3 Auxiliary

### *A2.3.1 Create and destroy*

These operators simply create and destroy a relation respectively over the whole network of processors. Each slave creates/destroys its local partition of the relation and the master creates/deletes an entry in the relation table.

### *A2.3.2 Balance*

Balance is a user oriented auxiliary operator which ensures that the tuples of a relation are evenly distributed among the slaves. It ensures that the sizes of the partitions of any two slaves differ by no more than one tuple.

To this effect, the master computes the average number of tuples per slave by using information stored in the relation table. If this average is not an integer, it is rounded up (in some slaves) and down (in the rest of them) in order to achieve the correct sum. The precise distribution is stored in an array, '`average`'. This array, along with the relation name and the existing number of tuples in each slave's partition, are transmitted to the slaves. Finally, the master waits for confirmation that the correct number of tuples has been added to or removed from the partition of each slave before updating the relation table.

If a slave has fewer tuples than average, it enters a loop in which it acquires new tuples coming round the ring and appends them to its local partition. It subsequently creates two pseudoparallel threads. The first thread deletes tuples from the slave's partition and sends them into the ring, while the slave possesses more tuples than average. When finished, it sends a termination message with the slave's identifier into the ring indicating that this slave has reached the desired number of tuples. The second thread counts the number of termination messages it receives and terminates when this is equal to the number of slaves. Termination messages are passed on down the ring (except the one that originated in the local slave).

Termination messages are a way of terminating the parallel algorithm smoothly. Informally, this algorithm is deadlock free, because the sum of the tuples 'missing' from all slaves is equal to the sum of 'spare' tuples (above average). It is intented to construct a formal proof of absence of deadlock and livelock.

**Pbalance**

**Master**
```
Pbalance (relname) {
  || i in SLAVES {
    reltable_service (average, relname, i, average[i]);
    reltable_service (current, relname, i, current[i]);

      /* average[i] = no. of tuples slave i must contain */
      /* current[i] = no. of tuples slave i contains */

      toslave[i] ! relname, average[i], current[i];
      fromslave[i] ? difference[i];
      reltable_service (add, relname, i, difference[i]);
  }
}
```

**Pbalance**

**Slave**
```
tc = 0;                 /* termination message count */
diff = 0;               /* no. tuples appended/deleted count*/
frommaster ? relname, average, current;
while (current < average) {
  fromleftslave ? message;
  if termination (message) {   /* termination or tuple? */
```

```
    torightslave ! message;
    tc++;
  }
  else {                    /* tuple */
    append (relname, message); /* append to local
                                partition */
    diff++;
  };
};
{ while (current > average) {
    get_one (relname, tuple);  /* pick an arbitrary local
                                tuple */
    torightslave ! tuple;
    del_one (relname, tuple);  /* delete the specified
                                tuple */
    diff--;
  };
  torightslave ! embed_pid (message, pid);
                             /* send termination */
}                            /* message with local pid */
||
{ while (tc < P) {
    fromleftslave ? message;
    if termination (message) {
      tc++;
      if (extract_pid (message) <> pid)
                  /* not local termination message */
        torightslave ! message;
    }
    else
      torightslave ! message;
  };
};
tomaster ! diff;
```

## REFERENCES

[1] D. DeWitt and J. Gray, 'Parallel database systems: the future of high performance database systems', *C. ACM*, **35**(6), 85–98 (1992).

[2] T. Theoharis and A. Paraskevopoulos, 'Parallel relational data base management system design aspects', *Proc. Nato Advanced Research Workshop (NARW)* Calabria, Italy, June 1992.

[3] T. Theoharis, G. Papakonstantinou and P. Tsanakas, 'The design of PARDB; a parallel relational database management system', *Proc ISCIS VII* Antalya, Turkey, November 1992.

[4] Inmos Ltd., *Transputer Reference Manual*, Prentice Hall International, 1988.

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

[6] E. Davis, 'Application of the massively parallel processor to database management systems', *National Computer Conference*, 1983, pp. 299–307.

[7] H. Stone, 'Parallel quering of large databases: a case study', *IEEE Computer*, October 1987, pp. 11–21.

[8] O. Frieder, 'Multiprocessor algorithms for relational-database operators on hypercube systems', *IEEE Computer*, November 1990, pp. 13–28.

[9] N. Rishe, D. Tal and Q. Li, 'Architecture for a massively parallel database machine', *Microprocessing and Microprogramming*, **25**, 33–38 (1989).

[10] M. Stonebraker, 'The case for shared nothing', *IEEE Database Engineering*, March 1986, pp. 4–9.

[11] H. Pirahesh, C. Mohan, J. Cheng, T.S. Liu and P. Selinger, 'Parallelism in relational data base systems: architectural issues and design approaches', *Proc. 2nd International Symposium on Databases in Parallel and Distributed Systems*, IEEE Computer Society Press, Dublin, Ireland, July 1990.

[12] R. England, R. Guiton, S. Hanson, G. Jones, J. Kerridge, J. Krug, G. Miller, P. Thompson and D. Walter, 'The performance of the idioms parallel database machine', in M. Valero *et al.* (eds), *Parallel Computing and Transputer Applications*, IOS Press / CIMNE, Barcelona 1992.

[13] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith and P. Valduriez, 'Prototyping Bubba, a highly parallel database system', *IEEE Transactions on Knowledge and Data Engineering*, **2**(1), 4–24 (1990).

[14] D. DeWitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H. Hsiao and R. Rasmussen, 'The Gamma database machine project', *IEEE Transactions on Knowledge and Data Engineering*, **2**(1), 44–61 (1990).

[15] J. E. Hall, D. K. Hsiao and M. N. Kamel, 'The multibackend database system (MDBS): a performance study', *PARBASE–90 Conference Proceedings*, IEEE Computer Society Press, 1990, pp. 139–143.

[16] M. Stonebraker, R. Katz, D. Patterson and J. Ousterhout, 'The design of XPRS', *Proc. 14th VLDB Conference*, Los Angeles California, 1988, pp. 318–330.

[17] N. Akaboshi, Y. Noguchi, R. Take and H. Yokota, 'A performance evaluation of join operations on a transputer based parallel database machine', in S. Noguchi and M. Yamamoto (eds), *Transputer/Occam Japan 5*, IOS Press, 1993.

[18] G. M. Bryan and W. E. Moore, 'Designing a transputer based parallel database machine', in M. Valero *et al.* (eds), *Parallel Computing and Transputer Applications*, IOS Press / CIMNE, Barcelona 1992.

[19] A. Paraskevopoulos and T. Theoharis, 'A distributed resource simulation facility for transputer systems', submitted to *Parallel Processing Letters*.

[20] T. Sellis *et al.*, 'UMDBASE', *Technical Report CMSC 424–0201*, University of Maryland, Spring 1989.

[21] K. C. Baru and Sriram Padmanabhan, 'Join and data redistribution algorithms for hypercubes', *IEEE Transactions on Knowledge and Data Engineering*, **5**(1), 161–168 (1993).

[22] C. A. R. Hoare, *Communicating Sequential Processes*, Prentice Hall International, 1985.

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