DEPARTMENT OF COMPUTER AND MATHEMATICAL SCIENCES

Parallel Query Processing in Object-Oriented Database Systems

C.H.C. Leung and D. Taniar

(47 COMP 16)

December, 1994

(AMS : 68P15)

TECHNICAL REPORT

VICTORIA UNIVERSITY OF TECHNOLOGY
(P O BOX 14428) MELBOURNE MAIL CENTRE
MELBOURNE, VICTORIA, 3000
AUSTRALIA

TELEPHONE (03) 688 4249 / 4492
FACSIMILE (03) 688 4050

Footscray Campus

Parallel Query Processing in Object-Oriented Database Systems

C.H.C. Leung *D. Taniar*

Terabyte Database Research Group
Department of Computer and Mathematical Sciences
Victoria University of Technology
Melbourne, Australia

email: *{clement, taniar}@matilda.vut.edu.au*

Abstract

Parallelization techniques are used to improve performance of Object-Oriented Database (OODB) query processing. Several parallelization techniques, especially those relating to parallelizing path expressions and explicit joins, are examined. Data distribution is a major concern in parallelization, since performance improvement depends on a good data distribution method. Two alternatives are described to overcome load imbalance: re-distribution and replication. General analytic models for a distributed memory parallel architecture are presented, and performance evaluation is carried out.

1 Introduction

The expressiveness of object-oriented data modelling has been one of the strengths of Object-Oriented Databases (OODB), which also gives rise to highly complex data structures and access patterns, with a consequent adverse impact on database performance. It is the aim of this paper to study improvements in OODB query processing through parallelization.

Parallel architecture for database systems is often classified into three categories; *shared memory*, *shared disk* and *shared nothing* architectures [1, 3]. Shared memory architecture is an architecture where all processors share a common main memory and secondary memory. Processor load balancing is relatively easy to achieve in this architecture, because data is located in one place. However, this architecture suffers from memory and bus contention, since many processors may compete for access to the shared data. In a shared disk architecture, the disks are shared by all processors, each of which has its own local main memory. As a result, data sharing problems can be minimized, and load balancing can largely be maintained. On the other hand, this architecture suffers from congestion in the interconnection network when many processors are trying to access the disks at the same time. A shared nothing architecture, which also known as a *distributed memory* architecture, provides each processor with a local main memory and disks. The problem of competing for access to the shared data will not occur in this system, but load balancing is difficult to achieve even for simple queries, since data is placed locally in each processor, and each processor may have unequal load. Because each processor is independent of others, it is easy to scale up the number of processors without adversely affecting performance.

A number of research prototypes and products that utilize parallelization in database systems have been produced. For example, Oracle has been ported to nCube parallel machines, while Informix and Sybase have had their products run on parallel machines [2, 3]. Most of these prototypes and products mainly deal with relational databases.

Researchers have recently proposed several parallelization techniques for OODB [5, 6, 14]. Both [6] and [14] concentrated on parallelization of path expressions. Parallelization of path expressions can be achieved by applying either forward or reverse traversal technique to the query graph [6]. Because the result of this traversal is linear, [6] claims that this linearity becomes the bottleneck of parallel processing. This approach views the degree of parallelization as coarse-grained with the sequential processing being the major part. In practice, the degree of parallelization can vary from coarse-grained to fine-grained parallelization, depending on the application environment. Although fine-grained parallelization can be very complex, it can be made to deliver good performance.

[5] describes a system that has been implemented on a transputer network. Their result is very similar to the result in [6] where it is suggested that the performance of parallel processing was hardly able to improve efficiency because sequential processing was the major part of overall processing. This work also considers the

coorporation between index maintenance with parallel processing, and is shown that update processing remains almost constant as the number of indices increased. Their data was partitioned across a number of disks, and a combination of vertical and horizontal partitioning is used in this study.

Another implication of the traversal technique in [6] is that there is no access plan, since there is only one way of implementing parallelization. In contrast, we have identified that there is more than one way in applying parallelization to the tree path expression queries. Therefore, an access plan can be formulated and an efficient execution can be chosen.

In this study we identify that inter-class parallelization (known as node parallelization in [6]) is not feasible, since not all objects of classes along the path must be read and evaluated, and the scope of classes is limited by the predicates of the previous classes. As we find that inter-class parallelization does not improve parallel processing, we concentrate on intra-class parallelization. To improve data filtering, we propose a data distribution scheme where objects of classes along the path that are not reachable from the root will not be distributed. Consequently, objects that will not participate in the output are discarded from the beginning, and so improving the efficiency of query processing.

[6] has introduced the access scope of a query that includes generalization or class-hierarchy. He has also shown that parallelization of class-hierarchy performed better than others when all parallelization techniques are applicable. However, the traversal techniques that were proposed did not include traversing subclasses. In this study, we consider several traversal techniques for different types of path expressions.

The query model presented in [14] is mainly dividing the query evaluation into two stages. In the first stage, a class projection and a set of selection predicates are specified. At the end of the first stage a sub-database is produced. Because the form of the sub-database is in an object schema form, it can be further processed when necessary. The second stage is providing a set of attribute projections. This results in a tabular or nested tabular form with primitive values. Parallelizing a query using query graph permits the initiation of query execution at several nodes simultaneously, and the intersection of all paths will provide the final result of the query. The data partitioning mechanism adopted was vertical partitioning. A relationship between classes is represented by two partitions, each representing a one-way relationship. This will result in redundant data, and data anomaly can occur. However, it claims that this mechanism suits well to update parallelization, since updating redundant data can be performed in parallel without interfering each other.

In terms of performance, [14] concluded that processor speed does not have much impact on overall performance, but I/O speed does have a significant effect. Communication speed does not affect performance for applications with low correlation, and the higher the correlation, the more impact it will have on performance. Performance was measured on application domains, not on single queries, and the applications used for experiments cover a number of mixture of different types of queries, such as queries with aggregation and queries with inheritance.

In the rest of this paper we will address several issues. Section 2 examines the kind of parallelization available. Section 3 describes how these parallelization techniques in OODB query processing may be used. Section 4 investigates how data are best distributed to processors. Section 5 describes cost models for query parallel processing, and section 6 presents performance evaluation.

2 Forms of Parallelization

There are different forms of parallelization depending on the scope of the problem. In this paper we are going to concentrate on parallelization within a query. To speed up the query response time, a query consisting of many operations in one or more classes, is divided into sub-queries, which are then executed in parallel. In this case, there are three forms of parallelization available to OODB systems: *intra-class* parallelization, *inter-class* parallelization and *hybrid-class* parallelization.

These parallelization techniques are similar to the techniques adopted by parallel RDBMS. Parallelization in OODB has been influenced by the parallelization techniques used widely in the parallel relational systems. However, when we go deeper, parallelization in OODB is significantly different from parallel RDBMS. This is mainly because the base data structure of OODB is much richer, as it includes complex relationship among objects, such as generalization and aggregation hierarchies.

2.1 Intra-Class Parallelization

Intra-class parallelization is a method where a query consisting of one or more predicates in a single class is evaluated in parallel. When the query has one predicate only, the objects are partitioned to all available processors. These processors perform the same predicate evaluation for different collection of objects. On the other hand, when the

query involves multiple predicates, each processor evaluates all predicates for different collection of objects. An alternative way is where each processor evaluates one predicate for the full number of objects.

As an illustration, consider the class schema in Figure 1. This class schema forms an aggregation hierarchy through the association between classes. From this class schema, one can invoke the following query.

Query 1:

```
select JOURNAL
where JOURNAL.organization = "IEEE";
```

To answer the query, each processor is assigned the same predicate, that is checking whether the organization is "IEEE". The processor is also allocated part of data. In this case, data partitioning becomes crucial, because it must guarantee that all processors are equally balanced.

Two data partitioning models exist in parallel database systems: *vertical* and *horizontal data partitioning* [3, 14]. Vertical partitioning partitions the data vertically across all processors. Each processor has a full number of objects of a particular class, but with partial attributes. Because each processor has different attributes, when invoking a query that evaluates a particular attribute value, only processors that hold that attribute will participate in the process. Therefore, processors that do not hold that particular attribute become idle. This model is more common in distributed database systems, rather than in parallel database systems. The motivation to use parallelization in database systems is to divide the processing tasks to all processors, so that the query elapsed time becomes minimum. Processor participation in the whole process is crucial. Even more important, the degree of participation must be as even as possible.

Horizontal partitioning is a model where each processor holds a partial number of complete objects of a particular class. A query that evaluates a particular attribute value will require all processors to participate. Hence, the degree of parallelization improves. This data partitioning method has been used by most existing parallel relational database systems.

When a query involves many predicates on a single class, like in Query 2, each predicate is done by an operator, and each operator has a full copy of the objects to work with. In this case, the number of processors is determined by the number of predicates in the query. Furthermore, all objects may be replicated to all processors.

Query 2:

```
select JOURNAL
where (JOURNAL.title="*parallel*" and
      JOURNAL.year>=1990 and
      JOURNAL.organization="IEEE");
```

Since evaluating more than one predicate on a single class is not much different from evaluating one predicate, each processor is allocated with all predicates and with different collection of objects. If Query 2 is implemented in shared nothing or shared disk architectures, objects of class JOURNAL may be partitioned to all processors. In a shared memory architecture, since the data is located at one place, no data partition is necessary, but there must be a mechanism that guarantees for each processor to work independently and continuously, without any waiting for locks to be released on a particular shared object. The locking mechanism must not interfere with parallelization; otherwise, parallelization will not produce much improvement.

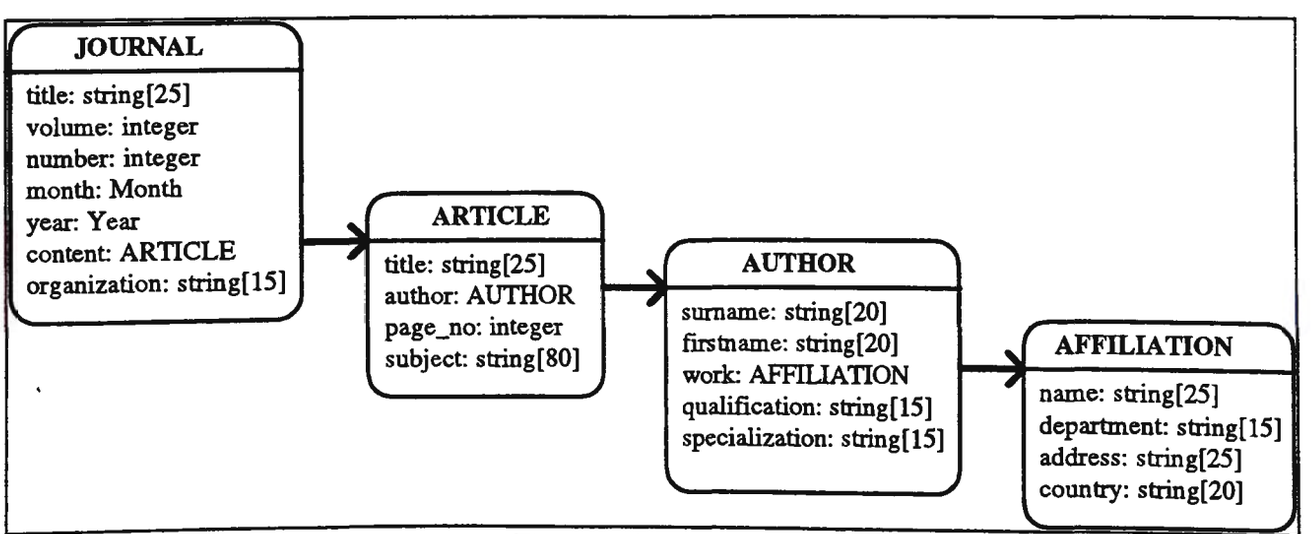

Figure 1: Class Schema

2.2 Inter-Class Parallelization

Inter-class parallelization is a method where a query involving multiple classes and each class appeared in the query predicate is evaluated simultaneously. Consider Query 3 as an example.

Query 3:
select JOURNAL
where (
JOURNAL.organization = "IEEE" and
JOURNAL.content.title = "*object-oriented*" and
JOURNAL.content.author.qualification="PhD" and
JOURNAL.content.author.work.country="UK");

There are four classes involved in Query 3, each with its own predicate. The query shows that each predicate is considered as an independent task, and the objects of a particular class are attached to the predicate to be evaluated. As a result, the entire query is composed of many independent tasks, which run concurrently.

The execution of a query may be divided into a number of sequential phases [9]. Within each phase, a number of operations are executed in parallel, and the results from one phase will be passed to the next for further processing. Depending on how the results need to be finally presented, a consolidation operator may be required to arrange the results in an appropriate final form. If necessary, the consolidation operator will re-distribute the output objects for further processing. However, the final consolidation operation is not parallelisable so it involves the bringing together of parallel results for final presentation.

The task of the consolidation operator can vary from collecting the result of two operators at a time to collecting the result of all operators at once. Thus, the degree of inter-class parallelization can be classified into four categories; *left-deep tree* parallelization, *bushy tree* parallelization, *right-deep tree* parallelization, and *flat-tree* parallelization [4]. Figure 2 illustrates these four types of trees, where a node represents a predicate evaluation of a class. Using Query 3 above, node A can be regarded as the first predicate evaluation (organization="IEEE"), node B as the second (title="*object-oriented*"), and so on. Furthermore, the result of each predicate is subsequently joined. For example, AB indicates the result of joining process (implicitly or explicitly) between the first and the second predicates.

It is obvious from the parallelization trees shown in Figure 2, that the purpose of parallelization is to reduce the height of the tree. The height of a balanced bushy-tree is equal to $log_2 N$, where N is the number of nodes. When each predicate evaluation is independent of each other, bushy-tree parallelization is the best, since the reduction of the height of the tree is quite significant. However, in the case where each predicate evaluation is dependent on the previous ones (e.g., in path expressions), bushy-tree parallelization is inapplicable.

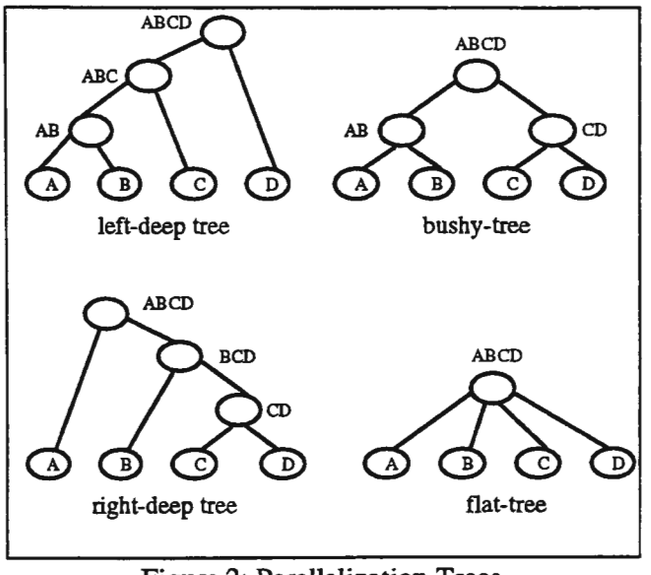

Figure 2: Parallelization Trees

Left-deep tree and right-deep tree are similar to sequential processing with a reduction of one phase only. These parallelization techniques are suitable for predicate evaluations that must follow sequential order; that is, the result of a predicate evaluation will become an input to the next predicate evaluation. This mechanism is like a pipeline-style parallelization. Left-deep trees are not much different from right-deep trees, except for the order of processing the predicates. When a query follows a particular direction to process the predicates for efficiency reasons, only one of these methods can be used. In contrast, when the query disregards the direction, the query optimizer must be able to decide which method will be used that will produce a minimum cost.

Flat-tree parallelization is a tree of height one. Here, the consolidation operator can be very heavily loaded, since the result of all predicate evaluations is collected at the same time. Hence, parallelization will not effect much improvement. However, this technique works well for queries with a single class and many predicates, because no join operation is needed.

2.3 Hybrid-Class Parallelization

Hybrid-class parallelization is actually the combination between inter and intra-class parallelization, where a query involving many classes and predicates, and each predicate of each class is evaluated by many different processors. In

practice, this form of parallelization is more likely to happen.

Processing Query 3 above will result each of four predicates to be executed in parallel. Furthermore, each predicate can be evaluated by many processors. If, for example, 12 processors are available to execute Query 3, each predicate will be executed by 3 processors, e.g., processors 1 to 3 will be concentrating on the first predicate (organization="IEEE"), processors 4 to 6 on the second predicate (title="*object-oriented*"), etc. Additionally, each predicate will be allocated one third of the associated class; e.g. processor 1 will be working on the first third of objects from class JOURNAL, processor 2 on the second third, and processor 3 on the last third. It applies to the subsequent predicates as well; e.g., processor 4 will be working on the first third of objects from class ARTICLE, processor 5 on the second third, etc. It can be expected that this form of parallelization will result a major performance improvement.

3 Implicit and Explicit Join Operations

Join operation is one of the most expensive operations in RDBMS, and still is in OODB. Because of the nature of object structure, in OODB join operation is categorized into *implicit* and *explicit* joins [7]. Implicit join, also known as *path expression*, is a kind of joining process between two classes through association connection. The joining itself is actually "pre-computed", since the association is established even before the query is invoked. This is the consequence of object data modelling, which permits this kind of link.

Explicit join in OODB is similar to that of RDBMS, where objects are joined based on one or several common attributes, but with more complexity as it is possible to join objects based on not only simple primitive attributes, but also objects. Thus, comparison operators to check the equality or non-equality of simple attributes and complex objects will be more complicated [11].

3.1 Path Expressions

Before we discuss how parallelization of path expression can be done, first we define several types of path expressions. The classification is based on the query schema, not the database schema. Figure 3 shows three types of path expressions; linear, tree, cyclic/semi-cyclic path expressions. Each node represents a class and the link between nodes represents the relationship between classes through aggregation hierarchies. The properties of each type of path expressions are as follows:

Linear path expression:

- properties:
  - maximum out-in degree = 1
  - irreflexive
  - antisymmetric
- traversals:
  - forward
  - backward

Tree path expression:

- properties:
  - out-in degree >= 1
  - irreflexive
  - antisymmetric
- traversals:
  - pre-order
  - post-order
  - level-order

Cyclic/Semi-Cyclic path expression:

- property:
  - closed walk, or semi closed walk
- traversal:
  - parallel traversal

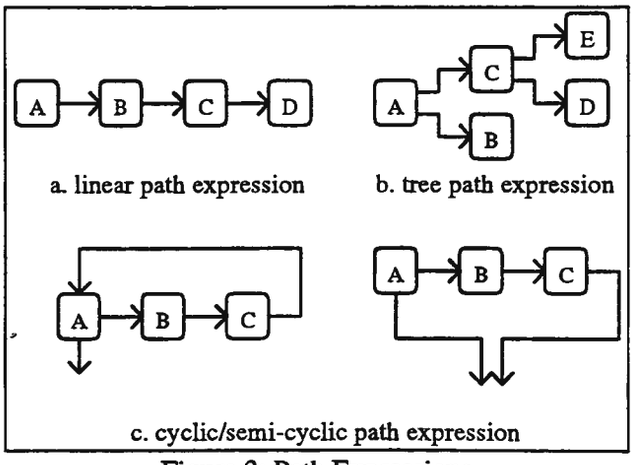

Figure 3: Path Expressions

It can be seen that linear path expression is a special case of tree path expression with maximum out-in degree being 1.

In this paper, we are dealing with linear and tree path expressions, as most queries are in these forms. Parallelizing these queries is done by choosing the right form of parallelization and the degree of parallelization, in order to gain the maximum benefit.

For linear path expression queries, intra-class parallelization in the form of left-deep tree is the most suitable, while bushy-tree and flat-tree parallelizations are not appropriate. To see this, consider Query 3 as an example. Suppose that class JOURNAL has 2,000 objects. If on average each journal has 6 ARTICLEs and each article has an average of 2 AUTHORs, the numbers of objects of

classes ARTICLE and AUTHOR that are connected to a particular journal are 12,000 and 24,000 respectively. The overall numbers of articles and authors are more than the above figures, since many articles do not appear in any journal (e.g., in conference proceedings), and there are many authors of non-journal's articles. Using either bushy-tree or flat-tree parallelization, each predicate is evaluated concurrently. Therefore, the first predicate (organization="IEEE") is evaluated against 2,000 objects of class JOURNAL. The second and the third predicates (title = "*object-oriented*" and qualification = "PhD") are not evaluated against 12,000 and 24,000 objects of classes ARTICLE and AUTHOR, but more than these. This is certainly not efficient, because there are many unnecessary readings and evaluations. To avoid unnecessary reading, left-deep tree parallelization can be used.

In left-deep tree parallelization, predicate evaluation starts from the first class and then follows the link to cover the whole path. Consequently, reading the subsequent classes will narrow to those objects that are selected from the previous classes only. Thus, the numbers of readings and evaluation of the second and the third predicates will be less than 12,000 and 24,000, because some of them are discarded by the previous evaluations. The efficiency of this method relies on the size of the root class. If the last class of the path expression is smaller than the root class, the right-deep tree parallelization can be used, provided that the links between classes are bi-directional links.

To parallelize a query with tree path expression, we can use either *level* parallelization or *left-deep tree* parallelization. Level parallelization is in the form of bushy-tree parallelization where each level indicates one phase. Additionally, consolidation operator combines the result of each branch of the tree to form the final result. Level parallelization is based on the query tree. Each level processes pairs of adjacent nodes. A query like in Figure 3b requires four phases. This is shown in Figure 4a, where each node in the level-tree parallelization represents a node in the tree-path expression. As in Figure 2, each node represents a local predicate evaluation of a particular class. At the end of phase 1, A and B are combined, and so are A and C. Because they are independent from each other, they can be done in parallel. Phase 2 processes AC, which is obtained from the first phase, with D and E. Again, these two processes are executed in parallel. At the end of the second phase, we get ACD and ACE. Phase 3 combines the two results from the second phase to form ACDE. Finally, phase 4 joins task AB of phase 1 with the result of phase 3, and the final result can be presented to the user.

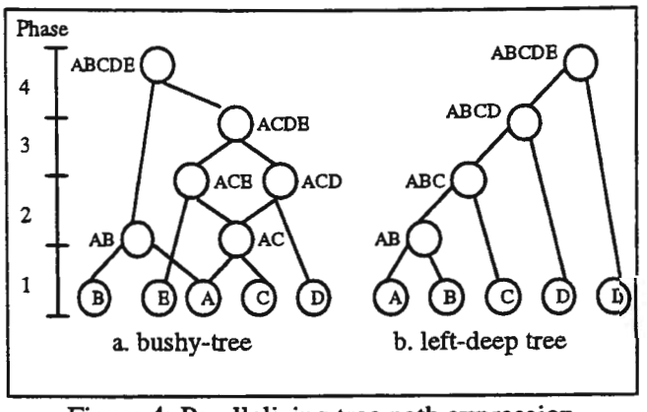

Figure 4: Parallelizing tree path expression

Using left-deep tree parallelization, a tree path expression must be converted into a linear path expression by using one of the available traversal techniques. Using pre-order traversal, the above query can be transformed into A-B-C-D-E. Figure 4b shows how tree path expression can be done in left-deep tree style. It also shows that path expression itself does not improve parallelization at all. Therefore, we must rely on intra-class parallelization, not inter-class parallelization.

3.2 Explicit Joins

Explicit join can be performed between two or more classes based on one or more common primitive attributes or common objects. Explicit join can also be executed within one class. Joining based on primitive attributes is similar to relational join. In this regard, object can be considered as a tuple or a complex/nested tuple. To optimize this kind of join, many joining techniques have been developed, such as nested-loop, hash, and sort-merge joins [3] [4]. Parallelizing explicit joins is best by applying bushy-tree parallelization, where each class involved in the query is first restricted by local predicate evaluation before joining takes place. This is similar to performing select before join in the relational model to minimize the size of the input relations. All local predicate evaluations are done in parallel. Then, each pair of classes is joined together to form the final result. A simple rule in optimizing explicit join based on primitive attributes is to apply local predicate as soon as possible, and to delay joining process as late as possible. The reason being to discard unnecessary objects before joining, because join will result a large number of objects.

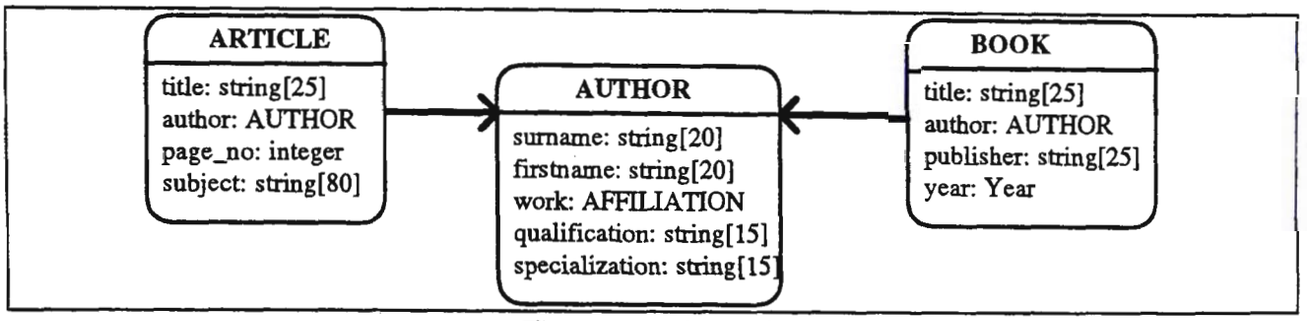

Figure 5: Explicit Join

To illustrate explicit join based on a common object, refer to Figure 5. Both classes ARTICLE and BOOK point to class AUTHOR. Query 4 joins classes ARTICLE and BOOK based on the common class, i.e., class AUTHOR. Each base class is considered as a complex object. This implies that an object ARTICLE together with its associated object AUTHOR is regarded as a unit, while an object BOOK also with its associated object AUTHOR is regarded as another unit. Joining both units is then similar to joining two simple objects.

Query 4:

```
select ARTICLE, BOOK
where (ARTICLE.author = BOOK.author);
```

Because the comparison involves non-primitive attributes, comparison operators must include object comparison, such as object equality and non-equality. A convenient way to compare object equality is by comparing their Object Identifiers (OID) [11].

When a query involves both path expression and explicit join, the path expression is evaluated first, and then the explicit join. Parallelizing this kind of query is the same as parallelizing each part; path expression and explicit join.

4 Data Distribution

Data distribution is one of important aspects in parallel systems. Traditional data distribution scheme for parallel database systems is based on horizontal partitioning. Several data partitioning strategies exist, such as *range*, *hash* and *round-robin* partitioning [3]. These data distribution strategies deal with single relations and are not adequate for parallel OODB systems, since objects are related to other objects through aggregation and inheritance hierarchies. Thus, new data distribution schemes must be defined.

We define two data distribution schemes in OODB: *single class distribution* and *associative distribution*. Single class distribution is a scheme where each class is distributed regardless its relationship to other classes, while associative distribution is a method where objects related to each other are allocated on the same processor to speed up associative search. In this paper we will concentrate on associative distribution. At this stage we consider aggregation hierarchies only, although the same concept can be applied to generalization hierarchies.

In defining our data distribution model, we adopt a distributed-memory architecture as shown in Figure 6 with one coordinator processor and its disk, and a number of worker processors. Each worker processor is equipped with its own local main memory. This structure, configured as a star topology, has been implemented using transputers and further details may be found in [9]. When there is a need to distribute objects from one worker processor to the other, the system configuration can be altered to a fully connected network topology. The processing method is to distribute the process over the available interconnected processing devices in the environment. We assume that the data is already retrieved from the disk. This main memory based structure for high performance databases is increasingly common, especially in OODB, because query processing in OODB requires substantial pointer navigations, which can be easily accomplished when all objects present in the main memory [10, 12].

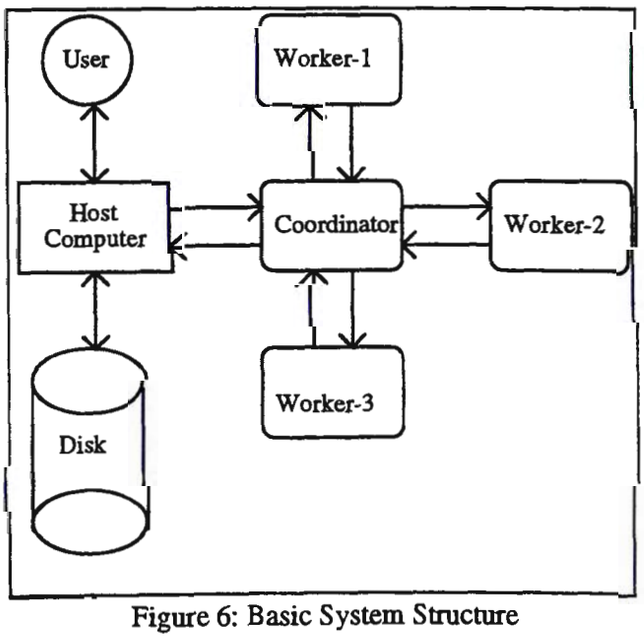

Figure 6: Basic System Structure

The user initiates the process by invoking a query through the host. To answer the query, the coordinator distributes the data from the host to the worker processors. Each worker processor performs local computation, and then sends the result back to the host, which subsequently will present it to the user.

Our associative data distribution strategy is based on query path expression. Data distribution occurs only when a query is invoked by a user. The distribution is based on the root class and its associated classes along the path expression. We call this scheme *filtered data distribution* scheme. It implies that objects along the path that are not reachable from the root class will not be distributed. This method does not require any checking, because distributing a root object and its associated objects is done by navigating pointers from the root to all its associated objects. When there is no pointer left, the scheme will skip to the next root object. In this case, objects that do not appear in the query predicates are discarded from the beginning. For example, using Query 3, ARTICLEs that do not appear in any JOURNAL will not be distributed to the worker processors. Furthermore, AUTHORs of non-JOURNAL ARTICLEs will not be distributed either. Algorithm 1 presents the parallel data distribution algorithm using round-robin method.

```
Parallel Round-Robin Data Distribution:
    LET P be processor number
    LET N be number of worker processors
    LET R be the root class
    PAR FOR P = 1 TO N
        INITIALIZE counter I[P] to P
        WHILE object I[P] of R exists
            GET object I[P] and all its assoc. objects
            DISTRIBUTE to P
            ADD N to I[P]
        END WHILE
    END PAR FOR
```

Algorithm 1: Data Distribution Algorithm

Using filtered data distribution technique, there will occur a problem when processing objects after the first phase. It is possible that after applying the first predicate to the root class, only a proportion of the objects in the next class is selected, resulting in some worker processors being overloaded while others underloaded. There are two possible ways to overcome this problem. First, re-distribute objects from the overloaded processors to the underloaded processors. Second, to have full data replication from the beginning, and on the subsequent phases of query processing, cascade unnecessary objects. In both cases, balance will always be maintained.

4.1 Data Re-Distribution

At the end of each phase, each worker processor informs the coordinator the number of objects it holds. On the basis of this information, the coordinator will calculate the ideal number of objects per processor, and produce two lists: overloaded processors and underloaded processors. Both lists include the number of objects the processor will re-distribute or receive, and these instructions are sent back the workers.

On the basis of the two lists provided by the coordinator, each worker will re-distribute or receive depending on which list the worker processor is on. The ideal re-distribution strategy is to re-distribute the current objects only. In this way, re-distribution cost is kept minimal. Algorithm 2 presents the parallel data re-distribution algorithm.

Because re-distribution is done in parallel, the elapsed re-distribution time depends on the most overloaded processor. The number of objects that must be re-distributed by this processor is the difference between the number of objects it has and the number of objects it should have.

4.2 Full Data Replication

Full data replication method requires each worker processor to have enough memory to hold all objects. If this requirement is fulfilled, this method can be used, although the initial distribution time must be taken into account.

When distributing objects to worker processors, objects that will be processed by a particular worker processor are tagged. Like the re-distribution scheme, each worker processor informs the coordinator the number of objects it has after finishing one phase. The coordinator replies with two lists: overloaded and underloaded lists to all worker processors. Instead of distributing or receiving, the worker processors remove or add tags to allocated objects. Therefore, no physical data re-distribution is needed, and processor load balancing is always achieved.

5 Cost Models

In order to measure the effectiveness of parallelization of OODB query processing, it is necessary to provide cost models that will be used to perform quantitative analysis. The cost models defined in this section are for queries that are expressed in linear path expression fashion using the left-deep tree parallelization. The cost is primarily expressed in terms of the elapsed time taken to answer a query. The notations used are as follows:

```
Parallel Data Re-distribution:
  PAR FOR each overloaded processor P
    WHILE there is object to distribute from processor P DO
      FOR each underloaded processor P'
        LOCK processor P'
        IF locking success THEN
          IF there is object to receive by P' THEN
            IF no. of obj to distribute from P is not less than no. of obj to receive by P' THEN
              LET T be no. of objects to receive by P'
              CALCULATE no. of objects that has still to be distributed from P
              SET underloaded processor P' to full
            ELSE
              LET T be no. of objects to distribute from P
              CALCULATE no. of objects that has still to be received by P'
              SET no. of objects to distribute from P to zero
            END IF
            FOR I=1 TO T
              DISTRIBUTE object from P to P'
            END FOR
          END IF
          UNLOCK processor P'
          IF nothing else to distribute from P THEN
            BREAK FOR
          END IF
        END IF
      END FOR
    END WHILE
  END PAR FOR
```

Algorithm 2: Data Re-distribution Algorithm

- m, number of classes,
- r_1, number of objects of the root class,
- r'_i, number of objects selected from of the i^{th} class,
- n_1, initial number of worker processors,
- n_i, number of worker processors used in the i^{th} phase,
- λ_i, probability of an object of the i^{th} class of having a link to objects of another class,
- f_i, average fan-out degree of the i^{th} class,
- k_i, average skewness degree of the i^{th} class,
- b, variable processor overhead incurred in transmitting of an object,
- c, fixed processor overhead incurred in preparing a stream of data for transmission,
- σ_i, selectivity which gives the probability (or proportion) that a given object of the i^{th} class is selected,
- t_r, time to retrieve an object from buffer,
- t_v, time to evaluate a predicate involving a single attribute,
- t_w, time to form the result and write it to output buffer,
- T_d, total data distribution time,
- T_r, total reading time,
- T_v, total predicate evaluation time
- T_w, total writing time,
- T_e, total elapsed time for an operation.

Total elapsed time for an operation (T_e) is denoted as the sum of total data distribution time, total reading time, total predicate evaluation time, and total writing time.

$$T_e = T_d + T_r + T_v + T_w \tag{5.1}$$

Data Distribution Time

There are two main components in calculating the elapsed data distribution time; variable and fixed processor overhead costs. The variable processor overhead cost depends on the number of objects that is distributed to the worker processors, while the fixed processor overhead cost depends on the number of worker processors used for that particular operation. The fixed cost is related to the cost of opening the channels between the coordinator and the participating worker processors.

Using the *filtered data distribution* scheme, the total data distribution time is:

$$T_d = \left[\frac{r_1}{n_1} + \sum_{i=2}^{m} \frac{r'_i}{n_i} k_i \right].b + n_1.c \quad (5.2)$$

where r_i' is the number of objects in the i^{th} class included for distribution, and is given by the product of the probability of an object of the previous class of having a link to objects of the current class, the size of the previous class, and the fan-out of the previous class. For example, if there are 2,000 journals (r'_1=2,000), each journal must have articles (λ_1=1), and on average each journal has 6 articles (f_1=6), the number of objects in the class ARTICLE (r'_2) can be calculated using equation (5.3) and is given 12,000 objects.

$$r'_i = \lambda_{(i-1)}.r'_{(i-1)}.f_{(i-1)} \quad (5.3)$$

Initially r'_1 is the same as r_1, since all objects of the root class are distributed. The full derivation of the equations presented in this section is given in the Appendix.

Reading Time

The elapsed reading time of each phase of the query processing is equal to number of objects to be read divided by number of participating worker processors. When skewness is present, the maximum number of objects in one processor will determine the reading time.

$$T_r = \left[\frac{r_1}{n_1} + \sum_{i=2}^{m} \frac{\sigma_{(i-1)} r'_i}{n_i} k_i \right].t_r \quad (5.4)$$

Number of worker processors of the subsequent phases of query processing is non-deterministic, since the distribution scheme is not known until run-time. However, it is possible to obtain the average number of busy processors using the following formula [8].

$$n_i = n_{(i-1)} - n_{(i-1)} \left[1 - \frac{1}{n_{(i-1)}} \right]^{\sigma_{(i-1)} r'_i} \quad (5.5)$$

Predicate Evaluation Time

Predicate evaluation time is very similar to the reading time, since all objects read must be evaluated against local predicate. Additionally, the cost model for predicate evaluation also includes the predicate length l.

$$T_v = \left[\frac{l_1 r_1}{n_1} + \sum_{i=2}^{m} \frac{l_i \sigma_{(i-1)} r'_i}{n_i} k_i \right].t_v \quad (5.6)$$

Writing Time

When there is no projection, selected objects of all classes along the path expression must be written to the output buffer. Number of objects selected by the last predicate is less or equal to the number of objects read in the last phase.

$$T_w = \left[\frac{\sigma_m (\sigma_{(m-1)} r'_m) m}{n'_m} k_m \right].t_w \quad (5.7)$$

6 Performance Evaluation

The purpose of the evaluation is to evaluate the effectiveness of data distribution on performance. Using the cost models described earlier, we can estimate the result of an operation. In the evaluation, we use Query 3 as an example. The size of class JOURNAL is 2,000 objects (r_1). On average, each journal has 6 ARTICLEs (f_1), each article has 2 AUTHORs (f_2), and every author has an AFFILIATION (f_3). We assume that the predicate selectivity probability is that only 40% of the journal is IEEE journal (σ_1), 10% of the selected journals are on the area of object-oriented (σ_2), 70% of the author of the selected articles have a PhD (σ_3), and 20% of the selected authors work in the UK (σ_4).

Data skewness has been one of major problems in parallel processing which can manifest as load skew in parallel processing. In this evaluation, we would like to find out the effect of data skew to performance. The degree of skewness ranges from 1.0 (no skew) to 2.8. The skewness degree of 2.0 indicates that there is at least one processor that holds objects twice as much as it should have.

The system parameters used in this evaluation are the same as that in [9], which has successfully experimented a basic relational query processing in a transputer system. In that system, the measurements are obtained from the standard transputer clock that measures the time in *ticks* (equal to 64 microseconds). The basic times for processing Query 3 are t_r=0.41 ticks, t_v=1.69 ticks and t_w=0.46 ticks. We used 10 worker processors, with the re-distribution scheduling time is 4.23 ticks, while the variable and the fixed distribution overhead costs are 1.52 ticks and 0.07 ticks respectively.

Figure 7 shows the result of the evaluation, which also shows the comparison between the

elapsed time for processing the query *without* re-distribution and *with* re-distribution. The result of the data full replication elapsed time is not shown since it is too high and unreasonable to compare with the first two methods. The data full replication strategy will be comparable if the communication cost is low.

As can be seen from the result that *with* re-distribution is better then *without* re-distribution in most cases, except where there is no skew. Although the improvement of re-distribution method is less than 10%, re-distribution is still desirable in most cases. It is expected that the future data distribution scheme will make the re-distribution method more efficient by providing a better processing method.

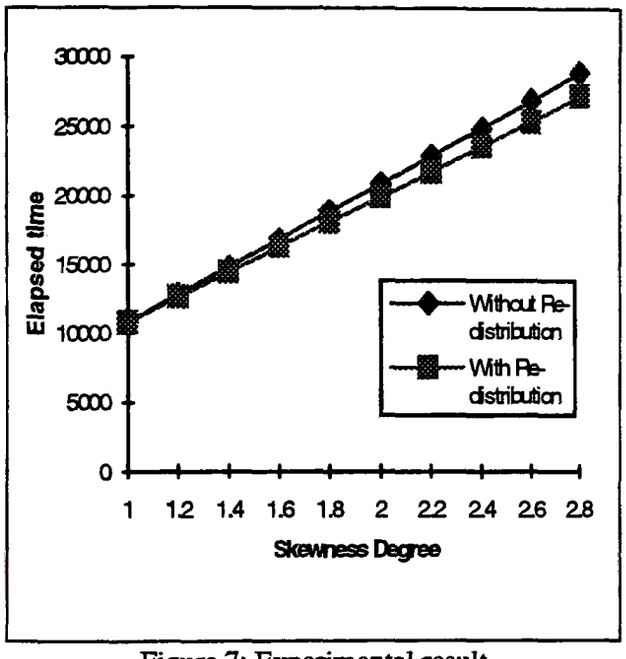

Figure 7: Experimental result

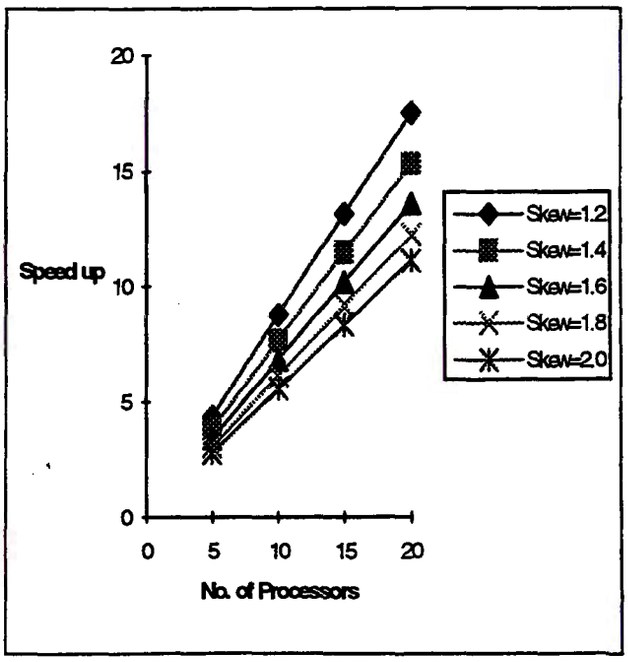

Figure 8: Parallel speedup

The second evaluation strategy was to compare parallel processor performance with uniprocessor performance. The purpose of this evaluation is to show how effective parallel processing is. Linear speed up has been an indicator to show how effective parallel processing can be. In order to achieve this result, the evaluation used 5 to 20 processors, with skewness degree ranges from 1.2 to 2.0.

Figure 8 shows that the skewness effected the improvement greatly. In fact, when a processor has objects twice as much as it should have, the performance downgraded for up to 50%. It also shows that linear speedup was unattainable. It is expected that the future parallel OODB will break the skew barrier, so that a linear can be obtained.

7 Conclusions and Future Works

In this study we have shown how parallelization can be done in OODB query processing. Although in this paper we concentrate on linear path expressions, our parallelization techniques can be applied to other types of path expressions and explicit joins. Sequential processing that is part of overall parallel processing has been known as the major bottleneck in the efficiency improvement of parallel processing performance. In this study, we show that this can be overcome by applying intra-class parallelization. Using this method, the processing strategy does not rely on the association of classes that has been the main source of the sequential processing, but on splitting objects of a class into several partitions that can be accessed simultaneously. The filtered data distribution shows that it can be used efficiently as it allows natural data filtering process.

The result of performance evaluation indicates that considerable benefits can be gained through parallelization. With careful data re-distribution strategy, reasonable performance improvement can be achieved. It also shows that a near linear speed up can be attained using our data distribution technique, although data skewness is still the main limitation.

Future plans for this work include defining a better data distribution scheme that covers not only data distribution, but also index distribution in order to speed up associative search, together with flexible network interconnection. Currently we are investigating parallelization techniques that can be applied to cyclic and semi-cyclic queries. The technique is expected to incorporate explicit joins with path expressions. Optimization strategies for these types of queries are also under investigation.

References

[1] B. Bergsten, M. Couprie, and P. Valduriez. Overview of Parallel Architecture for Databases, *The Computer Journal*, Volume 36, Number 8, pages 734-740, 1993.

[2] R. Bloor. The Coming of Parallel Servers, *DBMS Magazine*, Volume 7, Number 5, pages 14-16, May 1994.

[3] D. DeWitt and J. Gray. Parallel Database Systems: The Future of High Performance Database Systems, *Communications of the ACM*, Volume 35, Number 6, pages 85-98, June 1992.

[4] G. Graefe. Query Evaluation Techniques for Large Databases, *ACM Computing Surveys*, Volume 25, Number 2, pages 73-170, June 1993.

[5] J.P. Gray, et. al. Distributed Memory Parallel Architecture for Object-Oriented Database Application, *Proceedings of the Third Australian Database Conference*, pages 168-181, Melbourne, 1992.

[6] K-C. Kim. Parallelism in Object-Oriented Query Processing, *Proceedings of the Sixth International Conference on Data Engineering*, pages 209-217, 1990.

[7] W. Kim. A Model of Queries for Object-Oriented Databases, *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, pages 423-432, Amsterdam, 1989.

[8] V.F. Kolchin, et. al. *Random Allocations*, Wiley, 1978.

[9] C.H.C. Leung and H.T. Ghogomu. A High-Performance Parallel Database Architecture, *Proceedings of the Seventh ACM International Conference on Supercomputing*, pages 377-386, Tokyo, 1993.

[10] W. Litwin and T. Risch. Main Memory Oriented Optimization of OO Queries Using Typed Datalog with Foreign Predicates, *IEEE Transactions on Knowledge and Data Engineering*, Volume 4, Number 6, pages. 517-528, December 1992.

[11] Y. Masunaga. Object Identity, Equality and Relational Concept, *Deductive and Object-Oriented Databases*, W.Kim, et. al., (eds), pages 185-202, 1990.

[12] J.E.B. Moss. Working with Persistent Objects: To Swizzle or Not To Swizzle, *IEEE Transactions on Software Engineering*, Volume 18, Number 8, pages 657-673, August 1992.

[13] E. Ozkarahan. *Database Machines and Database Management*, Prentice-Hall Inc., 1986.

[14] A.K. Thakore and S.Y.W. Su. Performance Analysis of Parallel Object-Oriented Query Processing Algorithms, *Distributed and Parallel Databases 2*, pages 59-100, 1994.

[15] P. Valduriez. Parallel Database Systems: Open Problems and New Issues, *Distributed and Parallel Databases 1*, pages 137-165, 1993.

Appendix - Derivation of the Cost Models

The distribution time is composed of variable and fixed costs. The variable distribution cost depends on the number of objects being distributed, while the fixed distribution cost depends on the number of processors. Therefore, the distribution fixed cost (*fc*) is:

$$fc = n_1 . c \tag{A.1}$$

The cost to distribute objects of the root class (*rc*) is:

$$rc = \frac{r_1}{n_1} b \tag{A.2}$$

Initially, each processor will receive an equal number of root objects. However, when distributing the objects of the subsequent classes along the path, some processors will likely to have more objects than others, when skew presents. As a result, the most overloaded processor will set the maximum time to transfer objects in that phase. Number of objects being distributed is determined by the fan-out degree and the probability of an object of the previous class to have a link to the current class. Hence, number of objects of the subsequent classes along the path (r'_i) that need to be distributed is:

$$r'_i = \lambda_{(i-1)}.r'_{(i-1)}.f_{(i-1)} \tag{A.3}$$

The variable distribution cost (*vdc*) for the subsequent classes is then:

$$vdc = \left[\frac{\lambda_1 r'_1 f_1}{n_2} k_1 + \frac{\lambda_2 r'_2 f_2}{n_3} k_2 + \ldots + \frac{\lambda_{m-1} r'_{m-1} f_{m-1}}{n_m} k_m \right] b \tag{A.4}$$

The sum of equations (A.1) to (A.4) forms the total distribution cost. That is:

$$T_d = \left[\frac{r_1}{n_1} + \sum_{i=2}^{m} \frac{r'_i}{n_i} k_i \right].b + n_1.c \tag{A.5}$$

The reading cost is similar to the distribution cost, except that it includes the probability of an object to be selected (σ). Number of objects of the current class that needs to be read is restricted by the selectivity of objects from the previous class.

$$T_r = \left[\frac{r_1}{n_1} + \sum_{i=2}^{m} \frac{\sigma_{(i-1)} r'_i}{n_i} k_i \right].t_r \tag{A.6}$$

The predicate evaluation cost is also similar to the reading cost, with an addition of the length of the predicate in each class (l).

$$T_v = \left[\frac{l_1 r_1}{n_1} + \sum_{i=2}^{m} \frac{l_i \sigma_{(i-1)} r'_i}{n_i} k_i \right].t_v \tag{A.7}$$

The writing cost involves all selected objects along the path.

$$T_w = \left[\frac{\sigma_m (\sigma_{(m-1)} r'_m) m}{n'_m} k_m \right].t_w \tag{A.8}$$