Inheritance and Parallelization: Emerging Object-Oriented and Parallel Technologies for High Performance Database Systems

David Taniar
Department of Computer and Mathematical Sciences
Victoria University of Technology
P.O. Box 14428 MCMC, Melbourne 8001
Australia
taniar@matilda.vut.edu.au

Abstract

Queries on inheritance hierarchies are known as "inheritance queries". A model for parallelization of inheritance queries is called "intra-class parallelization", in which parallelization is achieved by dividing classes into a number of partitions. Hence, the efficiency of parallelization of inheritance queries depends on its data structure. We analyse parallel processing performance using horizontal and vertical inheritance division structures. We also propose a "linked-vertical division" which is demonstrated to be better than the traditional horizontal and vertical divisions.

1. Introduction

Parallel processing for object-oriented queries has been studied for quite sometime [7,9,12]. However, most of the existing work concentrated on association and composition relationships only. Parallel processing involving inheritance has not been given much attention. Inheritance is one of the most important concepts in object-oriented technology, as it provides a mechanism to reuse some parts of an existing system. It is the aim of this paper to study how inheritance queries can be efficiently parallelized by employing appropriate data structures.

There are different forms of parallelization depending on the scope of the problem. We are particularly interested in 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. This is a common technique adopted by most parallel database systems [4,6]. Parallelization of object-oriented queries has been influenced by the parallelization techniques used widely in parallel relational database systems. However, parallelization of object-oriented queries is significantly different from parallelization of relational queries, due to the richness of the base data structure of objects which includes complex relationship among objects, such as generalization/specialization.

The rest of this paper is organized as follows. Section 2 describes briefly inheritance concepts and inheritance queries. Section 3 explains how parallelization can be applied to inheritance queries. Section 4 presents inheritance data structures for efficient parallel executions. Section 5 presents performance evaluations. And finally, section 6 gives the conclusions.

2. Inheritance and Inheritance Queries: A Brief Overview

Inheritance is a relation between classes that allows for the definition and implementation of one class to be based on other existing classes [11]. Inheritance indicates that the methods and data associated with sub-classes are an extension of the properties associated with super-classes. A sub-class is a class which inherits from, at least one generalized class. This general class is called a super-class. Consequently, a sub-class must have all properties of the super-class, and may have others as well. In other word, a sub-class is more specialized than the super-class. Inheritance is a key feature of the object-oriented paradigm.

Inheritance queries are queries on an inheritance hierarchy. They can be categorized into two types: super-class and sub-class queries. The classification is based on the target class (Target class is a central focus of a query). A super-class query is defined as a query evaluating super-class objects, whereas a sub-class query is defined as a query evaluating sub-class objects. Since all sub-class objects are also super-class objects, a super-class query must also evaluate all of its sub-classes.
Figure 1 shows an example of super-class and sub-class queries. For an ease of visualization, a graphical notation is used to represent a query, in which classes and inheritance relationships are denoted as nodes and dotted lines, respectively. The shaded nodes are the target classes.

(a) Sub-Class Query
"Retrieve the names of lecturers having a PhD".

\[ \pi_{\text{name}} \]

A

B

\sigma_{\text{PhD}} \text{ in qualification}

B = \text{Person} \quad C = \text{Lecturer}

(b) Super-Class Query
"Retrieve all personnel living in suburb 'East Melbourne'."

\[ \pi_{\text{Oid}} \]

\sigma_{\text{suburb='East Melbourne'}}

is_a

A = \text{Person} \quad B = \text{Admin} \quad C = \text{Lecturer} \quad D = \text{Research Student}

Figure 1. Inheritance Queries

3. Intra-Class Parallelization

An object in an inheritance hierarchy is actually one object, although the declaration of the object is split into classes in the hierarchy. Consequently, parallelization of inheritance queries is similar to that of single class. However, it becomes more complex as inheritance queries involve superclasses and subclasses. For instance, a superclass query also includes all of its subclasses. A common model available to single class queries is called an 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 following query.

Oqt. 1.
Select a
From a in Lecturer
Where "PhD" in a.qualityfication

To answer the query, each processor is assigned the same predicate, that is checking whether the qualification is a "PhD". Each processor is allocated a portion 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 [4]. 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 OQL 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.

Oqt. 2.
Select a
From a in Lecturer
Where (a.\text{surname} = "Kim" and
\quad a.\text{age} < 30 and
\quad "PhD" in a.qualification)

Since evaluating multiple predicates 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 OQL 2 is implemented in shared nothing or shared disk architectures, objects of class Lecturer 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.

Parallelization of a single class query (e.g., OQL 2) is quite simple in parallel relational database systems. However, using object-oriented paradigm, since some attributes/methods are declared in superclasses (e.g., attributes surname and age are declared in class Person, not class Lecturer), parallelization of single class has to involve inheritance structures. The execution of such query depends on the adopted data structure. In the next section, it will be described how intra-class parallelization is applied to different inheritance structures.

4. Inheritance Structures

Traditionally, there are two data structures available to inheritance: Horizontal Division and Vertical Division [3,5]. In addition to these structures, a Linked-Vertical Division is proposed.

4.1. Horizontal Division

Horizontal division stores subclass objects as complete units. The values of an inherited attribute from a super-class is stored in the subclass.

If a class is represented as an unnormalized table, an illustration of the above example is displayed in Figure 2. It clearly shows that objects with OIDs 5, 6, 7 and 8 are Persons, whereas the last two objects (OIDS 7 and 8) are also Lecturers.

Regardless the scope of a sub-class query, parallelization can be accomplished by partitioning the sub-class into a number of participating processors to be processed simultaneously. This is possible because the contents of a sub-class are independent to its superclasses. On the other hand, processing a super-class query must include all its sub-classes, because each instance of sub-class is also an instance of its super-class. An algorithm for parallelization of inheritance queries using horizontal division is presented in Figure 3.

Upon receiving of an inheritance query, the algorithm is passed in all classes associating with the query through its parameter. In the case where the query is a subclass query (i.e., m = 1), only one class, that is the subclass itself, will be passed to the algorithm. If it is a superclass query, the first class R1 is the superclass and R2 to Rm are all of its subclasses. The processors are numbered consecutively (1, 2, ..., N), and so are the objects of each class (1, 2, ...). Processing is done in a round-robin fashion. The ith processor initially processes the ih object. The counter Ip (the object counter in processor P) will then be incremented by the number of processors P. Hence, with 2 processors (P=2) available, all odd objects will be processed by the first processor and all even objects by the second processor. These activities are repeated as many times as the number of classes involved in the query.

<table>
<thead>
<tr>
<th>Person</th>
<th>OID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Adams</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Bill</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Lecturer</th>
<th>OID</th>
<th>Name</th>
<th>Subjects</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Ellen</td>
<td>Network, AI</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Fred</td>
<td>Database, OOP</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2. Horizontal Division

Procedure Intra-Class-Horizontal \((R_1, R_2, ..., R_m)\):

\[
\begin{align*}
& // m = \text{number of classes} \\
& // R_1 = \text{objects of class 1} \\
& // R_2 = \text{objects of class 2} \\
& // R_m = \text{objects of class m} \\
& // \text{example: } R_5[5] = \text{the 5th object of class 1}. \\
& \text{Begin} \\
& \text{Let } N \text{ be the number of processors available} \\
& \text{Let } P \text{ be the processor number (id.)} \\
& // \text{processor numbers are consecutive (1, 2, ..., } N) \\
& \text{For } j = 1 \text{ to } m \quad \text{// foreach class} \\
& \text{Parallel For } P = 1 \text{ to } N \\
& // \text{foreach processor (Parallel execution)} \\
& \quad \text{Set } I_p \text{ to } P \\
& \quad // I_p = \text{counter for processor } P \\
& \quad \text{While } R_i[I_p] \neq \text{NULL} \\
& \quad \text{// object } I_p \text{ of } R_i \text{ exists} \\
& \quad \text{Get object } R_i[I_p] \\
& \quad // \text{process it} \\
& \quad \text{Allocate it to } P \\
& \quad \text{Process it in } P \\
& \quad \text{Add } N \text{ to } I_p \\
& \quad // \text{increment the counter} \\
& \text{End} \\
& \text{End} \\
& \text{End.}
\end{align*}
\]

Figure 3. Intra-Class Parallelization Algorithm using Horizontal Division

4.2. Vertical Division

Vertical division partitions a class according to where the attribute is originally declared. A sub-class
object is divided into a number of partitions, as part of the sub-class is declared in an inheritance hierarchy. Objects solely belonging to a super-class, together with some parts of sub-class objects are kept in the super-class. As an object must have a unique OID, sub-class objects which are divided into a number of partitions employ the same OID. This OID refers to a logical object identifier (LOID).

<table>
<thead>
<tr>
<th>Person</th>
<th>LOID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Adams</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Bill</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Ellen</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Fred</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Lecturer</th>
<th>LOID</th>
<th>Subjects</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td></td>
<td>Network, AI</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>Database, OOP</td>
</tr>
</tbody>
</table>

Figure 4. Vertical Division

```
Procedure Intra-Class-Vertical (R1, R2, ..., Rm):
    // m = number of classes
    // R1 = objects of class 1 (superclass)
    // R2 = objects of class 2 (subclass 1)
    // Rm = objects of class m (subclass m-1)
    // example: R1[5] = the 5th object of class 1.

Begin
    Let N be the number of processors available
    Let P be the processor number (id.)
    // superclass query
    If (m = 1) Then
        Parallel For P = 1 to N
        // initialize the counter
        Set Ip to P
        While R1[Ip] ≠ NULL
            // only superclass objects are processed
            Get object R1[Ip]
            Allocate it to P
            Process it in P
            // round-robin processing
            Add N to Ip
        End
    End
    Else // subclass query
        // need an explicit join operation
        Parallel-Join (R1, R2, ..., Rm) using N proc.
    End if
End.
```

Figure 5. Intra-Class Parallelization Algorithm using Vertical Division

Figure 4 gives an illustration of vertical division. It shows that some parts of Lecturer with OIDs 7 and 8 are declared in class Person.

Parallelization of sub-class queries (e.g., queries on Lecturer) can be accomplished by applying a parallel join algorithm to Lecturer and Person on LOID. The join operation is necessary only when the scope of the query is to cover super-classes, as well as the target class (i.e., subclass). If the scope is localized to the target sub-class, there will be no need to involve its super-class in order to minimize the size of the object to be processed.

Parallelization of super-class queries can be efficiently performed by evaluating the content of the target super-class itself. The algorithm is presented in Figure 5.

For superclass queries (m = 1), there is only one class will be passed to the algorithm; that is the superclass itself (i.e., R1). Processing is based on the round-robin partitioning. For subclass queries (m > 1), the subclass and all of its superclasses (in the case of the query involves in multiple superclasses) will be passed to the algorithm through the parameter. Since parallel subquery processing using a vertical division requires an explicit join to perform, a parallel join algorithm must be applied to these classes (i.e., R1× R2× ...× Rm). Conventional parallel join algorithms [6], such as hybrid hash, Grace join, may be employed.

4.3. Linked-Vertical Division

Linked-vertical division is a vertical division with pointers/links. If an object is divided vertically into a number of classes, each partition will have a unique physical object identifier (POID) and a common LOID. To avoid a relational join operation to take place when assembling an object, each POID within the same LOID is linked through a pointer connection Link.

Figure 6 shows an example of a linked-vertical division in unnormalized tables.

Parallelization of sub-class queries can be achieved through partitioning sub-class objects. Then each object traces the link to its super-classes. For super-class queries, because the content of super-class is isolated as in vertical division, the processing model for super-class queries for linked-vertical division is the same as that of vertical division. The algorithm is presented in Figure 7.

The procedure receives the classes involved in the query through the parameter. If it is a superclass query (m=1), only R1 is passed to the procedure. However, if it is a subclass query, the subclass (Rm) and all of its superclasses (R1, ..., Rm-1) are processed. After processing a subclass object, it traverses to its immediate superclass object through a pointer, and processes the object. This traversal is repeated until a superclass object
is reached and processed. These activities are repeated to other subclass objects. Parallel processing is achieved in a round-robin fashion of the subclass objects.

<table>
<thead>
<tr>
<th>Person</th>
<th>LOID</th>
<th>POID</th>
<th>Name</th>
<th>Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>50</td>
<td></td>
<td>Adams</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>60</td>
<td></td>
<td>Bill</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>70</td>
<td></td>
<td>Ellen</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>80</td>
<td></td>
<td>Fred</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Lecturer</th>
<th>LOID</th>
<th>POID</th>
<th>Subjects</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>71</td>
<td></td>
<td>Network, AI</td>
</tr>
<tr>
<td>8</td>
<td>81</td>
<td></td>
<td>DB, OOP</td>
</tr>
</tbody>
</table>

**Figure 6. Linked-Vertical Division**

**Procedure Intra-Class-Linked-Vertical**

\( (R_1, R_2, \ldots, R_m) \):

// \( m \) = number of classes
// \( R_1 \) = objects of class 1 (superclass)
// \( R_2 \) = objects of class 2 (subclass 1)
// \( R_m \) = objects of class \( m \) (subclass \( m-1 \))
// example: \( R_1[5] \) = the 5th object of class 1.

Begin
Let \( N \) be the number of processors
Let \( P \) be the processor number
// foreach processor
Parallel For \( P = 1 \) to \( N \)
Set \( Ip \) to \( P \)
While object \( R_m[Ip] \neq \) NULL
// \( m=1 \) is superclass query,
// \( m>1 \) is subclass query.
Get object \( R[Ip] \)
Allocate it to \( P \)
Process it in \( P \)
If \( m > 1 \) Then
// subclass query
\( j = m - 1 \)
// going to traverse to its superclass
Repeat
Traverse to its superclass
Get the pointed superclass object
Allocate it to \( P \)
Process it in \( P \)
\( j = j - 1 \)
Until \( j = 0 \)
// repeat for all of its superclass objects
End if
Add \( N \) to \( Ip \)
// round-robin partitioning
End
End

**Figure 7. Intra-Class Parallelization Algorithm using Linked-Vertical Division**

5. Performance Evaluation

In order to evaluate the efficiency of each inheritance structure for parallel query processing, performance evaluation was carried out. Experimentations were performed by means of simulation using Transim [8]: a transputer-based simulator. A number of queries was generated with different object sizes and number of objects. Some of the results are presented as follows.

**Figure 8. SuperClass Queries**

**Figure 9. SubClass Queries**

Figure 8 shows performance of superclass queries using the three inheritance structure. The size of the subclass was varied. The performance of horizontal division is poor, compared with the others. This is due to both superclass and subclass are accessed in the horizontal division. On the other hand, using the vertical/linked-vertical division, the access is localized to the superclass only. It is also noted that the performance of vertical and linked-vertical for superclass queries are similar. Another interesting thing is that performance degradation of the horizontal division is quite significant when the subclass size increases, since large amount of
unnecessary information about the subclass is also accessed.

Figure 9 shows performance of subclass queries. Due to a need for an explicit join, vertical division does not perform well. The performance of the horizontal and the linked-vertical division is quite similar, although most of the time the horizontal division shows a slightly better performance. This is due to a need for pointer traversal overhead in the linked-vertical division.

It is demonstrated that horizontal division is best for subclass queries, because of object independence. In contrast, linked-vertical and vertical division are more suitable for super-class queries. As performance depends on the data structures and the query types, a further evaluation based on a frequency of each query type has to be made, in order to determine an efficient data structure for most queries. Let $f_1$ be the frequency of superclass query and $y$ is the speeding factor of the traversal time compared to the processing time. The slower the processing unit time, the less the lower bound for $f_1$. Figure 10 shows that when the ratio of the processing and the traversal time is more than 10, the lower bound for $f_1$ becomes very small. Therefore, it can be concluded that the linked-vertical division is an efficient data structure for inheritance queries.

![Frequency ($f_1$) vs. Ratio](image)

**Figure 10. Frequency ($f_1$) vs. Ratio**

6. Conclusions and Future Plans

Parallelization in object-orientation is provided by intra-class parallelization, inter-object parallelization, and inter-class parallelization. The efficiency of parallelization of inheritance queries depends on its data structure. From our study, it shows that the linked-vertical division is an appropriate data structure for high frequency super-class queries. When the predicate evaluation time is much slower than the traversal time, the lower bound of the super-class query frequency declines. Hence, in most cases, the linked-vertical division outperforms the traditional horizontal and vertical divisions.

References