Parallel Collection-Equi Join Algorithms for Object-Oriented Databases

David Taniar
Monash University – GSCIT
Churchill, Victoria 3842,
Australia
David.Taniar@fcit.monash.edu.au

J. Wenny Rahayu
La Trobe University
School of Computer Sc. & Comp. Eng.
Bundoora, Vic 3083, Australia
wenny@latcs1.lat.oz.au

Abstract

One of the differences between relational and object-oriented databases (OODB) is that attributes in OODB can be of a collection type (e.g. sets, lists, arrays, bags) as well as a simple type (e.g. integer, string). Consequently, explicit join queries in OODB may be based on collection attributes. One form of collection join queries in OODB is "collection-equi join queries", where the joins are based on collection attributes and the queries check for an equality of both collection operands. Our previous work [7] describes "Parallel Double Sort-Merge" algorithm for collection-equi join queries. Since the publication, we realize that we have overlooked the complexity of collection merging in the algorithm. In this paper, we not only present alternative solutions relating to the collection merging problem, but also introduce a new algorithm called "Parallel Sort-Hash" algorithm. The two algorithms play an important role in parallel object-oriented query processing, due to their superiority over the conventional join methods through relational division and intersection operators.

1 Introduction

In *Object-Oriented Databases* (OODB), although path expression between classes may exist, it is sometimes necessary to perform an explicit join between two or more classes due to the absence of pointer connections or the need for value matching between objects. Furthermore, since objects are not in a normal form, an attribute of a class may have a collection as a domain. Collection attributes are often mistakenly considered merely as set-valued attributes. As the matter of fact, *set* is just one type of collections. There are other types of collection. The Object Database Standard *ODMG* [1] defines different kinds of collections: particularly *set*, *list/array*, and *bag*. Consequently, object-oriented join queries may also be based on attributes of any collection type. Such join queries are called *collection join* queries [6].

An interest in *Parallel OODB* (POODB) among database community has been growing rapidly, following the popularity of multiprocessor servers and the maturity of OODB. The emerging between parallel technology and OODB has shown promising results [3, 4, 9]. However, most research done in this area concentrated on path expression queries with pointer chasing. Explicit join processing exploiting collection attributes has not been given much attention. It is the aim of this paper to present parallel algorithms for collection-equi join queries. The algorithms are non-trivial to POODB, since most conventional join algorithms (e.g. hybrid hash join, sort-merge join) deal with single-valued attributes and hence most of the time they are not capable to handle collection join queries without complicated tricks, such as using loop-division (repeated division operator) and intersection.

The rest of this paper is organized as follows. Section 2 presents the background of this paper, which consists of a brief discussion on collection-equi join queries, and how these queries may be processed using conventional methods through the use of relational division and intersection operators. Section 3 explains our previous work on parallel algorithm for collection-equi join queries. The algorithm is called "Parallel Double Sort-Merge" algorithm. Section 4 describes the complexity of collection merging and provides three alternative solutions to the problem. Section 5 introduces a new algorithm called "Parallel Sort-Hash" algorithm. Section 6 presents performance evaluation of the two parallel algorithms and the comparison with the conventional relational division method. Finally, section 7 draws the conclusions and explains the future work.

2 Background

To provide enough background for the proposed algorithms, we first define collection-equi join queries. Secondly, we describe how conventional processing method using relational division and intersection operators can be used to process such queries.

2.1 Collection-Equi Join Queries

Collection-Equi Join Queries contain join predicates in a form of standard comparison using a relational operator, particularly the equality operator (i.e. the = operator) [6]. The operands of these queries are attributes of any collection types. As mentioned above, ODMG has formulated four kinds of collection types, namely *set*, *list*, *array*, and *bag* [1].

Sets are basically unordered collections that do not allow duplicates. Each object that belongs to a set is unique. *Lists* are ordered collections that allow duplicates. The order of the elements in a list is based on the insertion order or the semantic of the elements. *Arrays* are one-dimensional arrays with variable length, and allow duplicates. The main difference between a list and an array is in the method used to store the pointers that assign the next element in the list/array. Because this difference is mainly from the implementation point of view, lists and arrays will have the same treatment in this paper. A *bag* is similar to a set except for allowing duplicate values to exist. Thus, it is an unordered collection that allows duplicates. For example, an attribute *author* of class *Book* has a collection of *Person* as its domain. Because the order of persons in the attribute *author* is significant, the collection must be of type *list*. In other words, the type of the attribute *author* is *list of Person*. This shows that the domain can be a collection, not only a single value or a single object.

A typical collection-equi join query is to compare two collections for a full equality. Suppose the attribute *editor-in-chief* of class *Journal* and the attribute *program-chair* of class *Proceedings* are of type arrays of *Person*. To retrieve conferences chaired by *all* editor-in-chief of a journal, the join predicate becomes (`editor-in-chief = program-chair`). Only pairs having an exact match between the join attributes will be retrieved. The query expressed in OQL (Object Query Language) [1] can be written as follows:

```
Select A, B
From A in Journal, B in Proceedings
Where A.editor-in-chief = B.program-chair
```

As a running example, consider the data shown in Figure 1. Suppose class *A* and class *B* are *Journal* and *Proceedings*, respectively. Both classes contain a few objects, shown by their OIDs (e.g., objects *a* to *i* are Journal objects and objects *p* to *w* are Proceedings objects). The join attributes are *editor-in-chief* of Journal and *program-chair* of Proceedings; and are of type collection of *Person*. The OID of each person in these attributes are shown in the brackets. For example *a*(*250,75*) denotes a Journal object with OID *a* and the editors of this journal are Persons with OIDs *250* and *75*.

The query results using the sample data shown in Figure 1 depends on the collection type adopted by the join attributes. If the attributes are of type *array*, the query results are a concatenation between object *i* of class *Journal* and object *w* of class Proceedings, since both have the exact match not only on the OIDs of all elements, but also in that order. However, if the join attributes are of type *set*, the query results also include objects *b-p*.

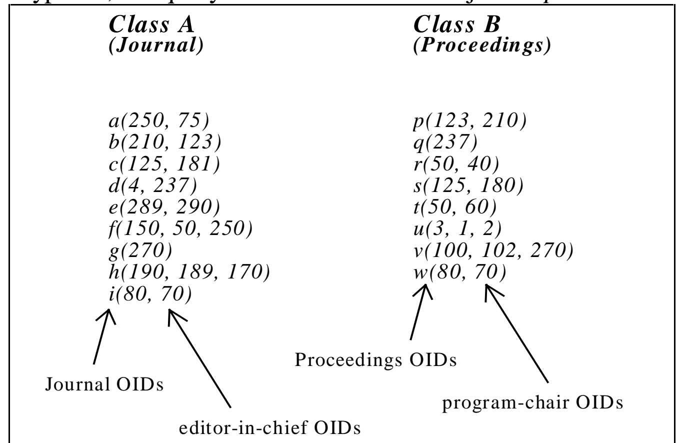

Figure 1. Sample data

Relational operators, like the = operator, are overloaded functions. This feature is not new to object-oriented join queries, because long before OODB exists, relational operators in relational joins have shown this capability. For example, it is permitted to compare an integer with a real number. One of the operand is automatically converted to the type of the other operand (in this case, *integer* to *real*). Casting a collection, however, must be done explicitly in the join predicate. Using the previous example, if *editor-in-chief* is a *list* and *program-chair* is a *set*, the equality predicate becomes (`listtoset(editor-in-chief) = program-chair`), where the editor-in-chief is converted from a list to a set [1]. Comparing two sets/bags can be done easily by sorting them prior to the actual comparison.

2.2 Relational Division: The Conventional Method

Collection-equi join queries may be processed using conventional relational algebra operators. To process collection-equi join queries, conventional partitioned join algorithm (eg., hybrid hash join) will have each class or table normalised prior to joining. Partitioning is then carried out based on the join attribute. For each partition, a hash join is performed [5]. This simple join method will not produce correct results, unless a division operator is applied, because the joining operation must be on collection, not on individual elements.

In this section, we will describe how collection-equi join queries may be processed correctly using relational algebra operators, particularly a *division* and an *intersection* operator. Our previous work reported in Taniar and Rahayu [7] described relational division technique to process collection-equi join queries. The process can be explained as follows.

The first class of the join operand is called a *divisor* table, and each collection of the second class is known as a *dividend* table. Figure 2 shows an example of a relational division operation. In this case, the divisor table is a union of all editors-in-chief, and the dividend table is the program-chairs of the first conference object *p*. The result of this division is the combination of *b* and *p*.

all editors-in-chief

a	250
a	75
b	210
b	123
c	125
c	181
...	...
...	...
...	...
i	80
i	70

dividedby

program-chairs of a conference

p	123
p	210

giving

division result

b	*p*

Figure 2. Relational Division

It is clear from the example in Figure 2 that the division operation must be repeated for each collection of objects from the second class (it is called a *loop division*). The algorithm can be written as follows.

```
// subquery 1
for each collection c in objects of the 2nd class
    all coll. of the 1st class dividedby c giving Temp
    T1 = T1 + Temp
End
```

The results of the loop division (subquery 1) are *b-p*, *d-q*, and *i-w*. The first and the last pairs are produced by the subquery because both collections within each pair are the same, whereas the second pair is because the collection in *q* is a subset of that of *d*.

The division operator is a manifestation of a universal quantifier, which differs from the collection equality. The universal quantifier evaluates whether a divisor object contains all values of the dividend table. This requirement does not ensure that all values within a divisor object must contain all values in the dividend table. Therefore, the another loop division must be carried out to the two classes, but with a reverse role (eg., the division is the second table and the dividend is each collection of the first table). The following pseudo-code is for the second loop division operation.

```
// subquery 2
for each collection c in objects of the 1st class
    all coll. of the 2nd class dividedby c giving Temp
    T2 = T2 + Temp
End
```

The results of subquery 2 (T2) will include the pairs *b-p*, *i-w*, and *q-v*. To obtain the final query result, the results from the first (T1) and the second (T2) loop division are intersected.

Collection-Equi-Join = T1 *intersect* T2.

The intersection of T1 and T2 is given by *b-p* and *i-w*. These pairs show that both collections in each pair are equal.

3 Parallel Double Sort-Merge Join Algorithm: Our Previous Work

In this section, we describe our previous work on parallel join algorithm for collection-equi join queries. The details of this algorithm can be found in Taniar and Rahayu [7]. Parallel Double Sort-Merge join algorithm for collection-equi join queries proceeds in two steps. The first step is the data partitioning step which produces disjoint partitions, and the second step is the joining step. Since the partitions are disjoint, the join operation in each partition can be done independently. In the joining step the sort-merge operation are used twice: one to the collection attribute, the other to the objects of both classes. The pseudocode of the algorithm is shown in Figure 3.

```
Parallel-Double-Sort-Merge-Collection-Equi-Join:
Begin
    // step 1: partitioning step
        partition the objects of both classes based on their
        first elements (for lists/arrays), or their minimum
        elements (for sets/bags).
    // step 2: joining step (in each processor)
    // a. sort phase
    (i) sort the elements of each collection (sets/bags
        only).
    (ii) sort the objects based on the 1st element of the
        collection.
    // b. merge phase
    (iii) merge the objects of both classes based on their
          first element on the join attribute.
    (iv) if matched, merge the two collection attributes
         based on their individual elements (starting from
         the second element).
End
```

Figure 3. Parallel Double Sort-Merge Algorithm

The data partitioning method for parallel collection-equi join is much influenced by common practices of arrays/sets comparison in programming. An array can be compared with another array by evaluating each pair of elements from the same position of the two arrays. A characteristic of arrays comparison is that once an element is found to be different from its counterpart (i.e., element of the same position from the other array), the comparison stops and returns a negative result. Unlike arrays comparison, sets comparison is not based on the position of each element in the collection, since the order of the elements is not significant. For example, *array*(2,3,1) ≠ *array*(3,2,1), but *set*(2,3,1) = *set*(3,2,1). In comparing two sets, it will become easier if the two sets are alphabetically/numerically pre-sorted. For instance, *set*(2,3,1) is sorted to be *set*(1,2,3), and so is the second set. Comparison can then be carried out as per array comparison.

It is clear that an array comparison very much depends on the position of each element in an array. The first element will open the gate for further element comparisons, if the first pair is evaluated to be true. In contrast, set comparison depends on the smallest element in a set, which is the first element after sorting. This element acts like the first element in the array. Based on these characteristics, the first element of an array and the smallest element of a set play an important role in data partitioning. Common horizontal data partitioning methods, such as range or hash, can be used to produce disjoint (non-overlap) partitions. If the collection is an array or a list, partitioning is based solely on the first element of the list/array, since list/array comparison operates on the original element composition of the collection. If the partitioning attribute is a set or a bag, partitioning is based on the smallest element of the collection, because a set/bag comparison requires the collections to be sorted.

The joining step is further decomposed into the sorting and the merging phases. The sorting operation is applied twice: to the collections, and to the class. Sorting each collection is needed only if the collection is a set or a bag, and sorting the objects is based on the first element (if it is an array or a list) or on the smallest element (if it is a set or a bag). The sorting phase is not carried out before data partitioning, as sorting done in parallel in each processor after data partitioning will minimize the elapsed time.

Like the sorting phase, the merging phase consists of two operations: class-level merging and collection-level merging. Merging the objects of the two classes is based on the first element of each collection. If they are matched, a subsequent element comparison can proceed. Two cases are presented as an example. Case 1 is where the two collections are arrays, and case 2 is where the collections are sets. Figure 4 gives an illustration of the result of the algorithm.

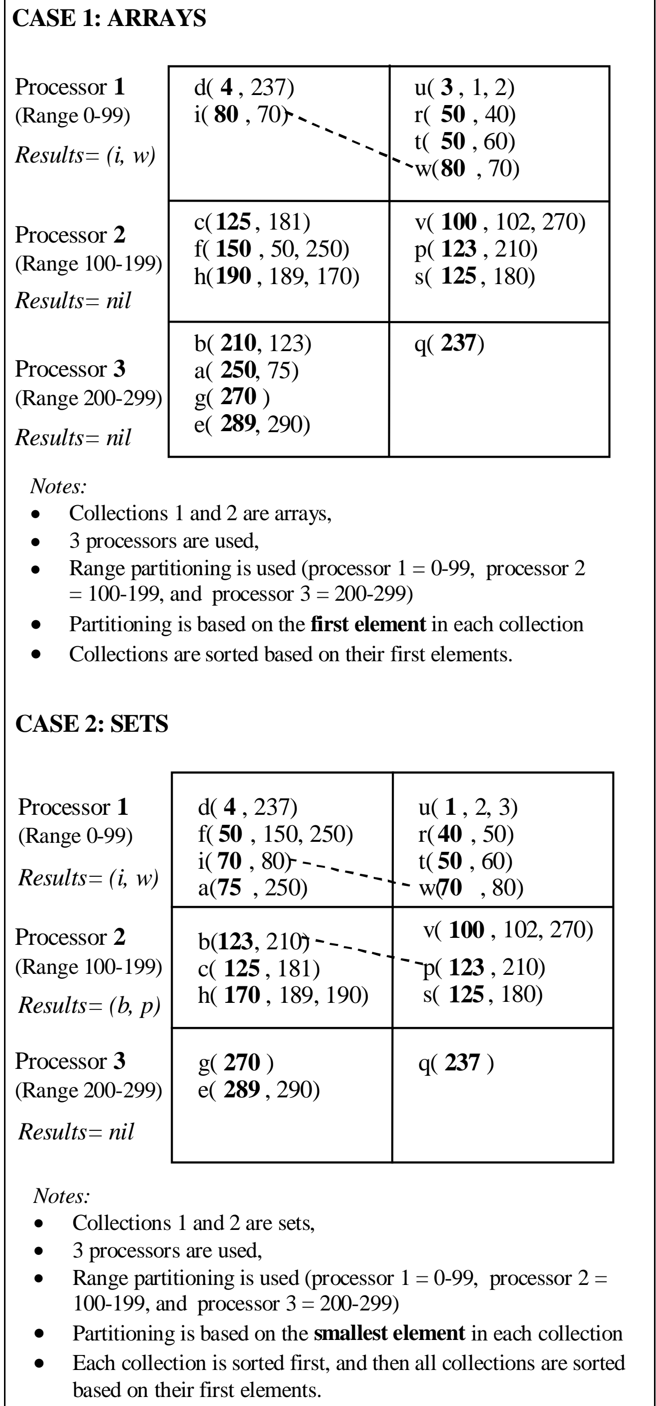

Figure 4. Results of Parallel Double Sort-Merge algorithm

4 Issues Relating to Collection Merging

Since the publication of our previous work [7] presented in the previous section, particularly Figure 3

(i.e. the pseudocode for the “Parallel Double Sort-Merge” join algorithm), we realize that we have overlooked the complexity of the algorithm, especially the merging phase (step 2*b*). Merging in parallel double sort-merge join is tricky. Merging in this algorithm is done at two levels: *class level* and *collection level*. Class level merging is based on the first element on each collection. If the result is positive, meaning that the result of this comparison is true, collection merging is then pursued. Collection merging is simply performed as done in simple arrays merging.

The two-level merging process can be re-iterated as follows. If the first element of the first collection is greater than that of the second collection, the first collection is ignored and the counter of the second class is incremented by one object. If the opposite happens, the counter of the first class is incremented instead. If the first elements of two objects are the same, collection merging is then carried out. The problem of a two-level merging can be explained as follows. Regardless of the result of the collection merging, complexity occurs regarding whether or not the counter is to be incremented. If the counter is to be incremented, which counter(s) are to be incremented. In this section, we describe three alternative solutions to the collection merging problem.

Solution 1: Nested-Loop at Class Level

The problem of two-level merging is similar to, but more complicated than, the problem of simple merging of two arrays where duplicates are allowed. To overcome this problem, a simple nested loop can be applied. A nested-loop construct is applied at a class level, and a sort-merge is carried out at a collection level. This is a naive solution to the collection merging problem. With a nested-loop construct, there is no need to keep track the class counter. This modified algorithm can then be called “*Parallel Nested-Loop Sort-Merge*” join algorithm, instead. Since a nested loop is used for merging at a class level, class sorting becomes unnecessary. Collection sorting is still needed, as merging at a collection level is carried out as per normal merging operation.

Solution 2: Nested-Loop at Collection Level

The second alternative solution is to employ a nested-loop, not at a class level, but at a collection level. The way it works is similar to the problem description stated earlier in section 4. That is, when the first elements of the two collections to compare are the same, collection merging can be pursued. A nested-loop is applied to all collections from both classes, where they have the same first elements. To clarify the matter, consider an example given in Figure 5. A nested-loop is carried out by the collections having the same first elements.

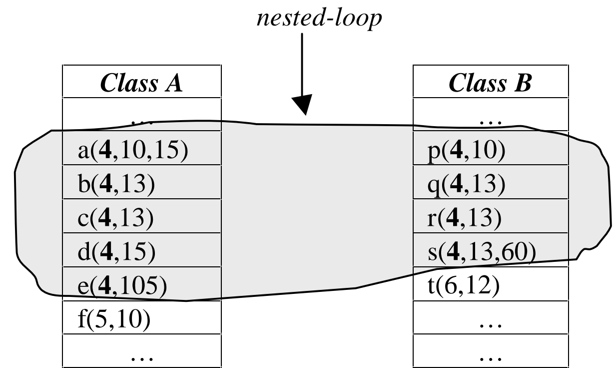

Figure 5. Nested-Loop at a Collection Level

The snapshot of the merging process in a modified version of the algorithm can be seen in Figure 6. From the algorithm, it is clear that the algorithm still maintains the two-level merging (i.e. merging at a class level, and at a collection level). Therefore, the name of the join algorithm is kept unchanged (i.e. parallel double sort-merge) to reflect the two levels of sort-merge. In the pseudocode, the function *is_merged* is assumed to be a collection merging function, in which two collections are compared for a full equality through a sort-merge process. The function takes two parameters, that is the two collections two be compared. Also notice that in order to keep track the starting point of the nested-loop, a tag (in this case, variables *m* and *n*) is used to label the starting point of the loop.

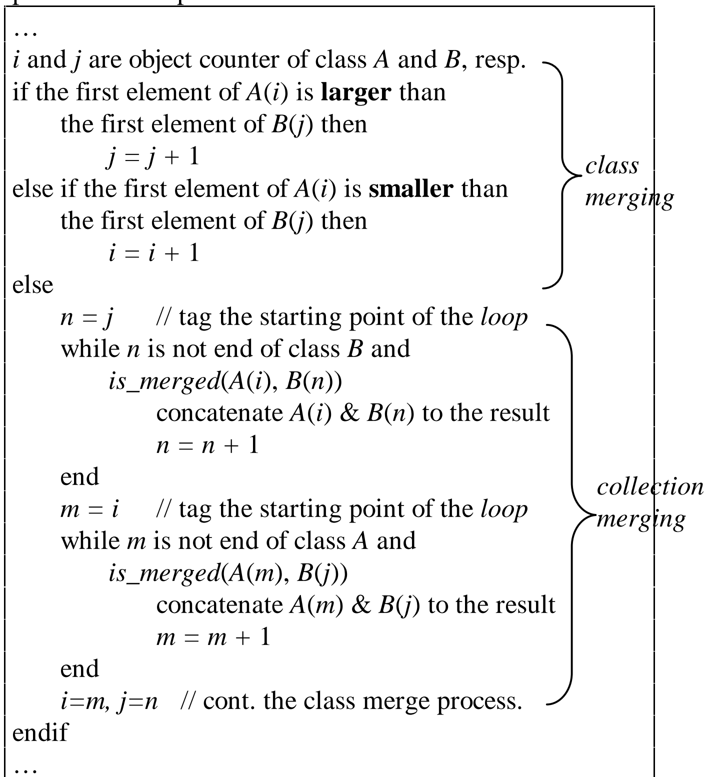
```
…
i and j are object counter of class A and B, resp.
if the first element of A(i) is larger than
    the first element of B(j) then
        j = j + 1
else if the first element of A(i) is smaller than
    the first element of B(j) then
        i = i + 1
else
    n = j    // tag the starting point of the loop
    while n is not end of class B and
        is_merged(A(i), B(n))
            concatenate A(i) & B(n) to the result
            n = n + 1
    end
    m = i    // tag the starting point of the loop
    while m is not end of class A and
        is_merged(A(m), B(j))
            concatenate A(m) & B(j) to the result
            m = m + 1
    end
    i=m, j=n   // cont. the class merge process.
endif
…
```


Figure 6. Nested-Loop at Collection Level

Solution 3: Nested-Loop at Matched Collection Level

The third alternative solution to the merging problem is achieved by having a nested-loop construct to the matched collections only. This nested-loop construct is necessary since all matched collections must be paired up each other. To clarify the matter, consider the example shown in Figure 7.

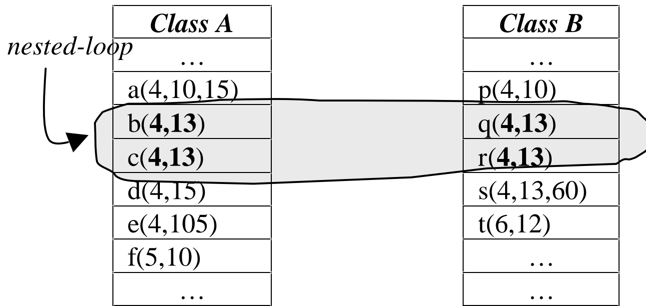

Figure 7. Nested-Loop at a *Matched* Collection Level

The order of the comparison is as follows: *a-p, a-q,* ***b-q, b-r, b-s, c-q, c-r, c-s****, d-s, d-t, e-t,* and *f-t*. The bold printed pairs indicate the nested-loop. Other than these, the increment of a class counter is done by comparing the whole collections, not the first elements only. Therefore, if the first collection, as a whole, is larger than the second collection, the second counter is incremented. The first counter is incremented, otherwise. When both collections are the same, a nested-loop is applied to find out whether the subsequent collections from both sides are the same. In other words, nested-loop at a matched collection level is accomplished by not splitting class merging from collection merging. The border between the two-level merging becomes unclear, since class merging is now not based on the first elements of both collections, but based on the whole collections.

The snapshot of the merging process in a modified version of the algorithm can be seen in Figure 8. In the algorithm, the function *coll_merge* is a merging of two collections. It is somehow similar to merging two strings. Since strings are similar to lists/arrays, merging two lists/arrays are also similar to merging strings. However, set merging is a slightly different from string merging, since sets are preprocessed by means of sorting, whereas strings are not.

Based on the three proposed alternative solutions to the merging problem, it is obvious that the third solution is the best, since the impact of the nested-loop construct is minimized. An important lesson drawn from the modification of the original Parallel Double Sort-Merge algorithm is that the merging process for class and collection should not be separated as initially proposed. In contrast, class and collection merging should be combined into one step, as specified by the third alternative solution, which is adopted as a solution for the merging problem faced by the initial Parallel Double Sort-Merge algorithm.

```
…
i and j are object counter of class A and B, resp.
if coll_merge (A(i), B(j)) = +1 then       // +1 is A(i) > B(j)
     j = j + 1
else if coll_merge (A(i), B(j)) = -1 then   // -1 is A(i) < B(j)
     i = i + 1
else
     concatenate A(i) and B(j) to the query result
     n = j + 1       // tag the start point of the loop
     while n is not end of class B and
          coll_merge(A(i), B(n)) = 0
          // 0 is A(i) = B(n)
               concatenate A(i) and B(n) to query result
               n = n + 1
     end
     m = i + 1  // tag the start point of the loop
     while m is not end of class A and
          coll_merge(A(m), B(j)) = 0       // 0 is A(m) = B(j)
               concatenate A(m) and B(j) to query result
               m = m + 1
     end
     i=m, j=n   // cont. class merging process.
endif
…
```

class and collection merging.

Figure 8. Nested-Loop at *Matched* Collection Level

5 Parallel Sort-Hash Join Algorithm: A New Algorithm

In this section we introduce a new algorithm based on a combination of *sort* and *hash* methods for collection-equi join queries. In the hashing part, hash tables are used. Hash tables for collection joins are different from those for relational hash join queries. For collection join queries, *multiple hash tables* are used. The following section describes multiple hash tables and probing mechanism. The discussion on the join algorithm will then follow.

5.1 Multiple Hash Tables and Probing Function

Each hash table contains all elements of the same position of all collections. For example, entries in hash table 1 contain all first elements in the collections. The number of hash tables is determined by the largest collection among objects of the class to be hashed. If the collection is a *list/array*, the position of the element is as the original element composition in each collection. If the collection is a *set/bag*, the smallest element within each collection will be hashed into the first hash table, the

second smallest element is hashed to the second hash table, and so on. Set/bag hashing will be enhanced if the set/bag is *preprocessed* by means of *sorting*, so that the hashing process will not have to search for the order of the elements within the set/bag. Figure 9 shows an example where three objects are hashed into multiple hash tables. Case 1 is where the objects are arrays, and case 2 is where the objects are sets.

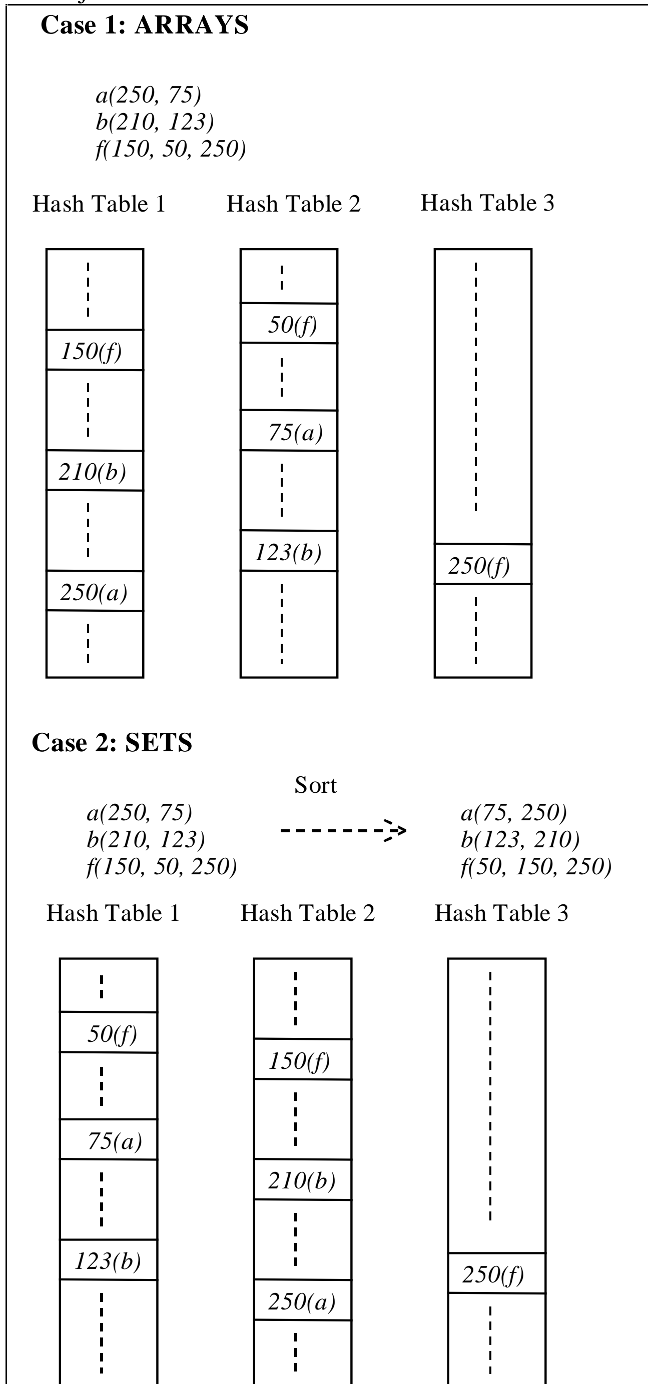

Figure 9. Multiple Hash Tables

Once the multiple hash tables are built, the probing process begins. The probing process is basically the central part of collection join processing. The probing function for collection-equi join is called *function universal*. It recursively checks whether a collection exists in the multiple hash table and the elements belong to the same collection. Figure 10 shows the pseudocode of the probing function.

```
Function universal (element i, hash table j): Boolean
Begin
    Hash and Probe element i to hash table j
    If matched Then
    // match the element and the object
        Increment i and j
        If end of collection is reached Then
        // check for end of collection of the
            Return TRUE            // probing class.
        End If
        If hash table j exists Then
        // check for the hash table
            result = universal (i, j)
        Else
            Return FALSE
        End If
    Else
        Return FALSE
    End If
    Return result
End Function
```

Figure 10. Probing Function *Universal*

5.2 Parallel Sort-Hash Collection-Equi Join Algorithm

Like the sort-merge version of parallel collection-equi join algorithm, the data partitioning is a disjoint partitioning which makes use of the first elements (for lists/arrays) or the smallest elements (for sets/bags).

The local joining process in each processor consists of several steps. The first step (step 2a) is the preprocessing and is only applicable to sets and bags. This step is actually a sorting process for each collection. The second step (step 2b) is to create multiple hash tables. The third step (step 2c) is the probing process where the function *universal* is called. Since this function acts like a universal quantifier where it checks only whether all elements in a collection exist in another collection, it does not guarantee that the two collections are equal. In order to check for the equality of two collections, it has to check whether collection of class *A* (collection in the multiple hash tables) has reached end of collection. This can be done by checking whether the size of the two matched collections is the same. Figure 11 shows the pseudo-code

for the sort-hash version of parallel collection-equi join algorithm.

```
Parallel-Sort-Hash-Collection-Equi-Join:
Begin
    // step 1 (disjoint partitioning):
        partition the objects of both classes based on their
        first elements (for lists/arrays), or their minimum
        elements (for sets/bags).
    // step 2 (local joining):
    In each processor
    // a. preprocessing (sorting)      // sets/bags only
        For each collection of class A and class B
            Sort each collection
        End For
    // b. hash
        For each object of class A
            Hash the object into multiple hash table
        End For
    // c. hash and probe
        For each object of class B
            Call universal (1, 1) // element 1,hash table 1
            If TRUE AND the collection of class A has
                reached end of collection Then
                    Put the matching pair into the result
            End If
        End For
    End
End
```

Figure 11. Parallel Sort-Hash Collection-Equi Join Algorithm

6 Performance Evaluation

The experimental environment was a DEC Alpha 2100 model with 4 CPUs running at 190MHz. It is a shared-memory based multi-processor system. The total performance of the system is around 3000Mips and 8Gflops. The size of main memory was 2Gb, and each CPU was equipped with 4Mb cache. The processors are all based on the same 64-bit RISC technology. The 64-bit technology breaks the 2-gigabytes limitations imposed by conventional 32-bit systems. Subsequently, the usage of very large memory is common to Digital Alpha servers. Very large memory systems significantly enhance the performance of very large database applications by caching key data into memory. The underlying operating system was Digital UNIX, and the algorithms were implemented in C++.

6.1 Implementation Issues

A number of aspects emerged from the implementation of collection-equi join algorithms.

Information on data distribution is kept in a distribution table

The distribution table is a two-dimensional array of number of processing elements x number of objects. The row represents the number of child processes (each child process is allocated a processor; the parent process is the coordinator), and the column is the maximum number of objects that can be allocated to one process (or processor). The distribution table could have been implemented in a single dimensional array, and each element in the array is a linked-list. The difference is just a matter of dynamic versus static data structure. For simplicity, a two-dimensional array is used.

Data distribution is carried out by assigning an OID to an appropriate row in the distribution table. A counter for each row in the distribution table is needed to keep track of the number of objects allocated to a particular process. For example, an object with OID 175 is to be distributed to processor 0. If the counter for processor 0 is equal to 16; meaning that there are 16 objects allocated to processor 0 so far, OID 175 will be allocated to row 0 column 16 (row and column start from 0) and the counter for this row is incremented by one. Apart having a counter for each row in the distribution table, a lock must be used every time the counter is updated. The checking of each object can then be done in parallel using a round-robin scheduling.

Hash tables are shared

For the sort-hash version of parallel collection-equi join, each partition does not employ a separate hash table. The consequence of having shared hash tables is that the distribution for hash join is disjoint using a round-robin partitioning. A hash table is implemented in a two-dimensional array. The row indicates the hash index, whereas the column is to accommodate collisions. The decision to use an array representation is merely for programming convenient. A linked-list-based representation may have been used instead.

Linked multiple hash tables are used for collection-equi join

Multiple hash tables have been implemented in a cube array, where the additional dimension to the normal two-dimensional array is to accommodate all elements of a collection in which its first element has been hashed. Each row in a cube array contains OIDs belonging to the same hash values (i.e., collision). For each collection in a row, all elements of the collection are attached to it. Hence, once the first element is probed to a particular row, probing for further elements of the same collection is simple done by merging all elements attached to the root cell. Using this mechanism, collection-equi join operation

is simplified, as probing is done once; that is to the first/smallest element only.

6.2 Experimentation Results

Double Sort-Merge and Sort-Hash versions of parallel collection-equi join algorithms are examined. A comparison with the conventional method (i.e. relational division) is also presented. Factors considered in the experimentation include the size of collections, the size of classes, and the join selectivity degree. Some of the results are presented as follows.

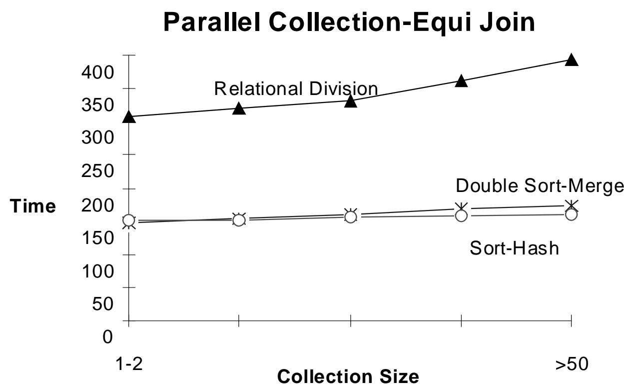

Figure 12. Varying the collection sizes

Figure 12 shows a comparative performance between the proposed algorithms with the relational division, by varying the collection size. When the collection size is small, the sorting cost for the collection is cheap, resulting in the overall performance of the double sort-merge version to improve. As the collection size grows, the sorting cost for the collections also increases. However, the overall performance of the double sort-merge version is quite steady although the collection size is increased. This is because sorting each collection is relatively small compared to the other cost components, such as for the sorting of the objects. In the experiments, the size of the collection varies from 2 to over 50 elements. For the same number of objects per class, the difference between sorting 50 elements and sorting 4 element is relatively insignificant, unless the number of objects is increased dramatically. Performance of the sort-hash version is slightly better than (in general) that of the double sort-merge version, especially when the collection size is large. The processing cost for the sort-hash version is quite comparable with that of the double sort-merge version because the sort-hash version, in some cases (especially for sets/bags), incurs a collection sorting cost. Furthermore, the hashing and the probing processes have to be repeated. The sort-hash version is however saved from the objects collection sorting cost imposed by the double sort-merge version.

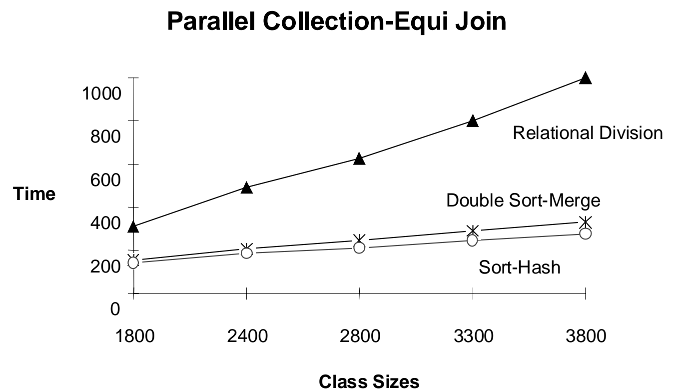

Figure 13. Varying the class sizes

Figure 13 shows another comparative performance against the size of the operand. Performance of the sort-hash version is shown to be better than that of the double sort-merge version. Processing cost for the double sort-merge version increases as the size of the operand expands. This is due to the objects sorting cost. Processing cost for the sort-hash version is not affected by the size of the operand more than the double sort-merge version. The hashing and the probing processes are linear in complexity, which is much simpler than the *NlogN* complexity for the objects sorting.

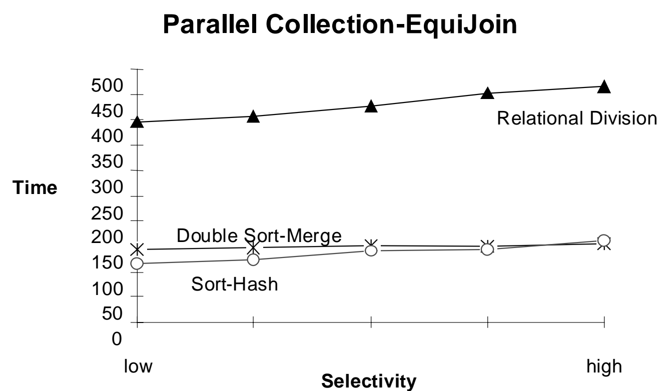

Figure 14. Varying the selectivity degree

Figure 14 incorporates the selectivity degree for each algorithm. It shows that the join selectivity factor does not affect the degradation of performance significantly. For the double sort-merge version, it appears that the merging cost for the matched collections is only a small component of the overall cost. For the sort-hash version, the increase is due to the repetition of the hashing and the probing processes, which can be expensive when the selectivity

degree is high. And for the relational division method, intersection cost component seems to be small, compared with the loop division. Hence, the join selectivity factor does not play a significant role in the overall performance.

The performance graphs shown above prove that the proposed algorithms (double sort-merge version and sort-hash version) are always better than the conventional relational loop division algorithm for parallel processing collection-equi join queries. The efficiency of the proposed algorithms can be more than 100% compared to the relational loop division. The cost for the relational loop division increases sharply especially for large operands. This is due to the expensive loop division cost.

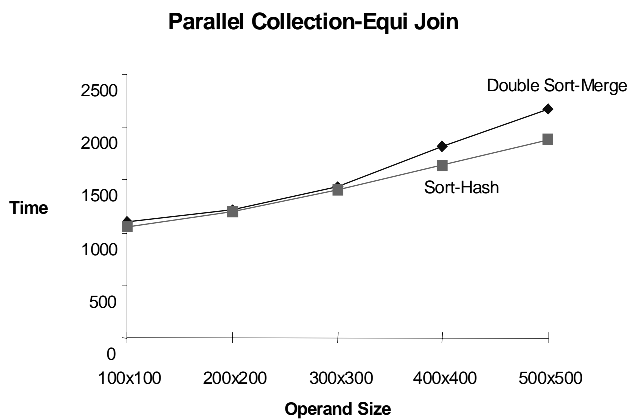

Figure 15. Double Sort-Merge vs. Sort-Hash

Figure 15 shows the difference between the double sort-merge version and the sort- hash version. The class size varies from 100 to 500 objects. The average collection size is 3 objects. A number of observations are made. First, performance of the double sort-merge version is quite comparable with the sort-hash-version. Second, when the size of the operand is getting larger, the sort-hash version shows its superiority to the double sort-merge version. This indicates the reliability of the sort-hash version of parallel collection-equi join algorithm. Finally, the fixed cost for parallel processing is shown to be large, since the increase in the elapsed time is far from linear, which is caused by the major proportion (in the case of small operand) is dominated by the processor set-up overheads.

In general, performance of the sort-hash version for collection-equi join queries is demonstrated to be superior to that of the double sort-merge version. The degree of improvement may vary from one system to another, depending on the system architecture. Therefore, it can be expected that the sort-hash version of parallel collection-equi join algorithm will become the basis for processing object-oriented collection join queries. This algorithm may also be used in other non-relational systems (such as nested relational systems) where collection types are supported.

7 Conclusions and Future Work

The need for join algorithms especially designed for collection-equi join queries is clear, as the conventional parallel join algorithms were not designed for collection types. In this paper, we present a modified version of *Parallel Double Sort-Merge* algorithm by providing three alternative solutions to the collection merging problem found in our previous work, and propose a new algorithm called *Parallel Sort-Hash* algorithm. These two algorithms are designed especially for collection-equi join queries in object-oriented databases. Performance of these algorithms are promising compared to the traditional and complicated relational division and intersection operators, since the division operation must be applied repetitively (i.e. loop division) and an intersection must also be used to obtain the final results. Our future plan includes investigating the possibility to use a double hash method, instead of a combination of sort and hash like in the Parallel Sort-Hash algorithm.

References

[1] Cattell, R.G.G. (ed.), *The Object Database Standard: ODMG-93*, Release 1.1, Morgan Kaufmann, 1994.

[2] DeWitt, D.J. and Gray, J., "Parallel Database Systems: The Future of High Performance Database Systems", *Communication of the ACM*, vol 35, no 6, pp. 85-98, 1992.

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

[4] Leung, C.H.C., and Taniar, D., "Parallel Query Processing in Object-Oriented Database Systems", *Australian Computer Science Communications*, vol. 17, no. 2, pp. 119-131, 1995.

[5] Mishra, P. and Eich, M.H., "Join Processing in Relational Databases", *ACM Computing Surveys*, vol. 24, no. 1, pp. 63-113, March 1992.

[6] Taniar, D., and Rahayu, W., "Object-Oriented Collection Join Queries", *Proceedings of TOOLS Pacific'96 International Conference*, Melbourne. pp. 115-125, 1996.

[7] Taniar, D., and Rahayu, W., "Parallel Double Sort-Merge Algorithm for Object-Oriented Collection Join Queries", *Proceedings of International Conference on High Performance Computing HPC ASIA'97*, IEEE Computer Society Press, Seoul, Korea, 1997.

[8] Taniar, D., and Rahayu, W., "Parallelization and Object-Orientation: A Database Processing Point of View", *Proceedings of the TOOLS Asia'97 International Conference*, Beijing, China, pp. 301-310, 1997.

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