Implementation of Parallel Collection Equi-Join Using MPI

Nung Kion Lee[1], David Taniar[2], J. Wenny Rahayu[1], and Mafruz Zaman Ashrafi[2]

[1] Monash University
School of Business Systems
Vic 3800, Australia
{David.Taniar,Mafruz.Ashrafi}@infotech.monash.edu.au
[2] La Trobe University
Department of Computer Science and Engineering
Australia
wenny@cs.latrobe.edu.au, csinkl@pop.latrobe.edu.au

Abstract. One of the collection joins types in Object Oriented Database (OODB) is collection equi-join. The main feature of collection joins is that they involve collection types. In this paper we present our experience in implementing collection equi-join algorithms by using Message Passing Interface (MPI). In particular, it layouts the fundamental techniques that are used in the implementation and that may be applicable to other collection joins. Two collection equi-joins discussed here are Double Sort-merge and Sort Hash Join. The implementation was done on a clustered environment and employed a data parallelism concept.

1 Introduction

Object-Oriented Databases (OODB) queries have many unique features as compared to Relational Databases. One of the features is that OODB can contain collection types [1,2]. A collection can be a set, a *list/array*, or a *bag*. The main difference among these collection types is that whether they are structured or unstructured [3]. For example, in a list/array, the ordering semantic is important, whereas in a set it is not. One of the particular queries for OODB is *collection join.* Collection join is very similar to the relational equivalent operator, but in OODB the attributes involved are of collection types. The join queries depend on the collection type and the operator involved (that is whether it is an equi, a sub-collection, an intersect or a proper subset) [3].

As database size becomes very large, query processing time will become significant. In order to reduce the processing time, parallel algorithm is being used. Many parallel algorithms have been developed for implementing queries in Relational Databases (for example *sort-merge, hash based and hybrid hash*). The main objectives of parallel algorithms are to speed up and scale up when more resources are employ. To achieve the same result in OODB, parallel object query processing becomes an active research area.

In this paper we are going to share our experience in developing *Collection Equi-Join queries* based on *Sort-Hash* and *Double Sort-Merge Join* proposed in [4,5]. The implementation outlines some techniques that we used.

2 Background

2.1 Collection Equi-Join Queries

Collection-Equi Join Queries contain join predicates in a form of standard comparison using a relational operator (i.e. the = operator). The operands of these queries are collection attribute types. The standard collection types defined by ODMG are *set, list, array*, and *bag* [1]. Set is unordered collections that do not allow duplicates. List/array is ordered collections that allow duplicates. Where as *bag* is similar to *set* but allow duplicates.

The value of the collection type attribute is a collection of Object Identifier (OID) of the object instance that it references to. An OID is a unique universal identification for an instance of object. As an illustration consider an equi-join query that follows.

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

In the query above, A is object of *Journal* class and B is object of *Proceedings* class respectively. Suppose *editor-in-chief* and *program-chair* are two collection attributes of *Person* class. The OID value of these two attributes will be the OID for *Person* object instance it's referred to (example editor-in-chief = {15,20,200}). In our implementation, we assume the OID is simply an integer data type. This query also used in our implementation. We referred to the *Journal* class as class A and *Proceedings* class as class B in our subsequence section of our explanation.

2.2 Double Sort-Merge Join

This algorithm can be divided into two main steps. The first is the data-partitioning step for both *class A* and *B*, which produces disjoint partitions, and the second step is joining. In the partitioning step, the object instance for a class is partitioned based on their first elements (for lists/arrays), or their minimum value OID elements (for bags/sets). In the joining step, the collection elements for all object instance of *class A* and *B* need to be sorted in ascending order (for sets/bags only). These two classes are then sorted based on the first element of the collection type attribute. Subsequently, these two classes are merged in class and *collection* level. The merging in class level involves finding two object instance of *class A* and *B* that has the same first element of its collection attribute. On the other hand, the collection level merge involves merging that two matched first element object instances start from the second element of their collection attribute values.

2.3 Sort Hash Join

The first step of this algorithm is the same as the sort-merge algorithm. In the second step, the collection attribute element of *class A* and *B* needs to be sorted (for set/bag only). But for sorting object class, only *class A* needs to be sorted. The chained hash table is then built by the hash object instance of *class A* based on the first element of its collection attribute. In the joining step, the hash table is examined for a match object instance for *class B*. To do so, *class B* is hashed by using the same hash function of *class A* to find the correct hash table bucket. The chained objects of *class A* are then checked and found matches at a collection level.

3 Implementation Environment

3.1 Parallel Architecture

The parallel implementation environment is based on a shared nothing architecture. In this architecture each process has its own memory, CPU and disk [6]. This architecture is chosen because it scales well and doesn't have the bottleneck that had in other architecture [7]. Interconnected LINUX workstations were used for the simulation.

3.2 Programming Language

The algorithm was developed using C and Message Passing Interface library (MPI). MPI libraries consist of message-passing function and can be used for distributed and parallel algorithms. Message passing function is simply a function that explicitly transmits data from one process to another. By using MPI library each process that is running on an individual workstation is assigned a unique number called process rank. The root process will coordinate most of the operation for the parallel data joining implementation.

The equi-join employed the data parallelism paradigm. Data is partitioned to different process for local processing. The main MPI library functions used in our implementation are Send, Receive, Broadcast, Gather, and All-to-All communication [6].

4 Parallel Data Partitioning

Parallel collection equi-join can be divided into two main phases. The first phase is data partitioning based on disjoint partition, and the second phase is performing the join algorithm mentioned earlier (i.e. sort-merge or hash join). The general algorithms is as follows,

1. Using a *Send* or *Scatter* function, the root process sends equal (nearly) portions of raw object instances to all other processes.

2. Each processor partitions its local objects based on a uniform *range partitioning vector* into p (number of processor) partitions and using the *all-to-all* MPI function to send each partition to its destination processor.
3. Each processor performs a local collection equi-join algorithm.

Figure 1 shows the definition of an object class. The OID for an object instance is stored as oid; and *collection_id* is the collection attribute that will store the *Person object* OIDs. In our implementation we predefined the maximum size of (that is *COL_MAX*) collection attribute because dynamic memory allocation will make the message passing between processes more complex. We do not include other attribute in the object class definition to simplify the implementation, but it can be easily included.

```
typedef struct Class_ {
   int  oid;                   //      OID for this object instance.
   int collection_id[COL_MAX];//      collection attributes
   int col_size;               //      collection size
   short int col_type;         //      type of collection
}OBJ_CLASS
```

Fig. 1. Object class

4.1 Raw Data Transfer

This step divides the raw object instance data of *class A or B* into equal size and transfers them to p processes. This is necessary because the local file is not stored in the each workstation disk. Figure 2 shows the code for this step.

The data is divided evenly between the numbers of processors p. The root process uses *MPI_Send* to send an equal size of *class A* object instances to all other processes. If the whole database is too large to load into root process' memory, the root process can transfer part of the objects that has been read, reclaim the free memory and read the remaining. At this stage all processes will have it local distinct set of *class A* objects. We perform the same data transfer to *class B* not until we obtain the range partitioning vector for object *class B* explained in the next section.

4.2 Data Partitioning

The main drawback in data parallelism algorithm is partition skew. It is hard to derive a fixed range partition vector without properly keep tracks the distribution of partition attribute. A sample based partitioning algorithm [8, 9] is used to partition the local *class A* objects in each processor. The partition is based on the smallest OID element if the collection type is bag or set, whereas for list/array, the first element (that is *collection_id* [0]) will be used.

```
if (my_rank == ROOT_PROC){
   ClassA_Size = read_object(&ClassA,fin);
    //read class A object instances.
   num_obj = ClassA_Size / processor_size;
}
MPI_Bcast(&num_obj,1,MPI_INT,0,io_comm);
//allocate memory for receive buffer
if (my_rank == ROOT_PROC){
  //send each processor from root it raw data portion.
  for (i=0; i<  processor_size-1;i++)
     MPI_Send((ClassA + (i *   num_obj)),num_obj,
              obj_cls_typ,i+1,i+1,io_comm);
} else
   MPI_Recv(ClassA,num_obj,obj_cls_typ,ROOT_PROC,my_rank,
            io_comm,&status);
```

Fig. 2. Raw object transfer

In the sample based partitioning, each processor selects p OID from its local objects. After the sample OID's are collected using *Gather* in the root processor, it is sorted locally. The root processor then picks up $p - 1$ distinct OIDs to form a range-partitioning vector. The range vector is then *Broadcast* to all other processors.

Each process then sorts its local instances of class A based on the first element of collection attribute (for set/bag it is the smallest element). The sorted objects are divided into p partitions based on a uniform range partition vector. Using *All-total communication*, the first partition is sent to processor rank 0, second partition to processor 1 and so on. At this stage, each processor has disjoint portion objects of *class A*. We also partition object of *class B* based on first element of collection attribute using the range vector obtained earlier.

5 Parallel Local Join

In OODB, collection equi-join is based on *'total'* equality. That is, for list/array each collection type element (OID) must be equal in the same order. Whereas for set/bag two objects are equal if all collection type element object of *class A* exist in object of *class B* in any order. Before the joining can take place, the local objects for *class A and B* needs to be sorted. We used *quick sort* with algorithm in Figure 3 as the comparison function.

The algorithm performs element-by-element comparison between two collection attributes until one of the OIDs is not equal or it reaches the end of one of the collection attributes. For this comparison to perform correctly, collection attributes of type set or bag must be pre-sorted.

5.1 Double-Sort-Merge

Parallel double-sort-merge algorithm is based on *nested loop* at the *collection level*. It first finds the *lower bound* index for sorted object *class A* and *class B*

```
int coll_merge(const int *col1,int size1, const int *col2, int
size2){
   int i=0,j=0;
   while((i< size1) && (j < size2) && (col1[i] == col2[j])){
        i++; j++;
   } //return 1 if col1>col2,-1 if col1<col2,or 0 col1== col2
}
```

Fig. 3. Total Comparison Function

that has same first elements of its collection attribute. It then finds the *upper bound* index of these two classes that has the same first element as the lower bound index objects.

As an illustration consider an example in Figure 5. The lower bound for both class A and B is 1. Their upper bounds are 5 for class A and 4 for class B. In this example the first elements of their collection attribute that equal is OID 4. After this boundary is found, a nested loop is performed at the collection level using a comparison function in Figure 3 to find a total equality of two collection attributes. The code snapshot for this step is shown in Figure 4.

```
nested_loop_matched(OBJ_CLASS *obj_clsA,int obj_sizeA,OBJ_CLASS
*obj_clsB, int obj_sizeB)
{
   long i=0;j=0;m=0;n=0;
   //object counter for class A and class B respectively
   while((i<obj_sizeA)&&(j<obj_sizeB))
   {
   //find the lower bound for Class A and Class B.
     if (coll_merge (obj_clsA, obj_clsB) == 1)  j++;
     else if  (coll_merge (obj_clsA, obj_clsB) == -1) i++;
     else
     {
       m = i + 1;
       //find the upper bound for object Class A.
       while((coll_merge(obj_clsA,obj_clsB)==0)&&(m<obj_sizeA))
         m++;  n = j + 1;
       //find the upper bound for object Class B
       while((coll_merge(obj_clsA, obj_clsB)==0)&&(n<obj_sizeB))
         n++;
       //class A and class B collection merging.
       while(i<m){
         k = j;
         while (k < n){
           print_pairwise_obj(&obj_clsA[i],&obj_clsB[k]);
           k++;
         } //while
         i++;
       }
       i = m; j = n;            //start next nested loop
   } } }
```

Fig. 4. Collection Merging

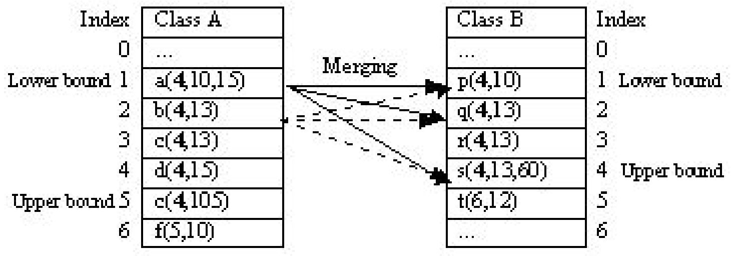

Fig. 5. Finding lower and upper boundary

5.2 Hash-Join

In this algorithm, the sorting step is not necessary. The only item that needs to be sorted is the collection attribute of type set or bag. A single chained hash table is used to store the object instances for *Class A*. The structure for the hash table is illustrated in Figure 6.

Initially, the object *class A* is placed into the hash table by hashing the first elements of collection attributes. By doing so, all the object instances with the same first element value will go to the same bucket. These object instances are chained together using a *list* data structure.

To perform the collection equi-join, each object instances of *class B* is hashed based on the first element of its collection attribute. This will directly find the object of *class A* that has the same first element of collection attribute. The object instances of *class B* are then compared to all chained object of *class A* by using the comparison algorithm in Figure 3.

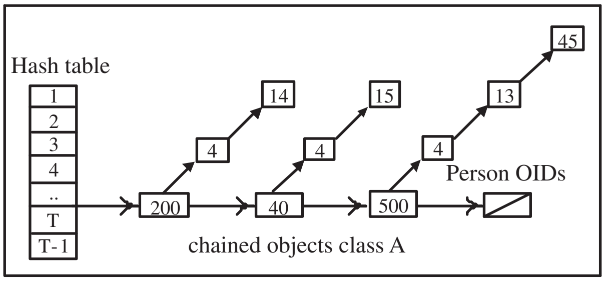

Fig. 6. Hash table organization

6 Performance

The experiment for performance analysis of the two algorithms was carried out. Sample OIDs were generated for object of *class A, B* and *Person*. The experiment setup is based on the architecture mentioned in section 3.1 using 8 Linux workstations. Although the number of workstations seems to be small, it is expected the same trend will follow if more workstations are used.

The result in Figure 7 shows that the performance of sort-merge is better than the sort-hash algorithms. This variation is significant if less workstations or CPUs are used. Comparison of the overhead for sorting between the *set/bag* and the *list/array* collection type is also shown in the same figure. The results indicate that this overhead is not significant and it is expected to incur some overhead if the collection size is large.

There are two reasons why sort-merge outperforms sort-hash. Firstly, there are overheads to create the hash table by chaining the "whole" object instance of *class A* together. In the algorithm, each object instance is ported from an array structure into a hash table structure. Whereas for the sort-merge, the array of object instance class A is ready to merge with *class B*. Secondly, the sort-merge filter object instances of *class A and B* that could not be joined. This was achieved by specifying the lower bound and the upper bound index of the two object classes to merge. These make the actual comparison in sort-merge using collmerge less. On the other hand, for hash join, each object instance of *class B* needs to be probed to find the matching object instances of *class A*. In this case, it may result in many redundant comparison (i.e. not filtered), which is not desired.

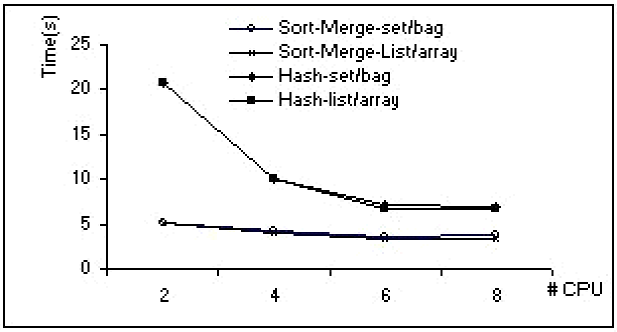

Fig. 7. Execution time

Figure 8 shows the execution time by varying the selectivity of *class B*. The time taken for each join algorithm is highly dependent on the distribution of the data partition. We found out, a sampled-based method will result in small partition skew if more workstation is being used. Better improvement can be achieved by also collecting sample OID's from *class B* objects.

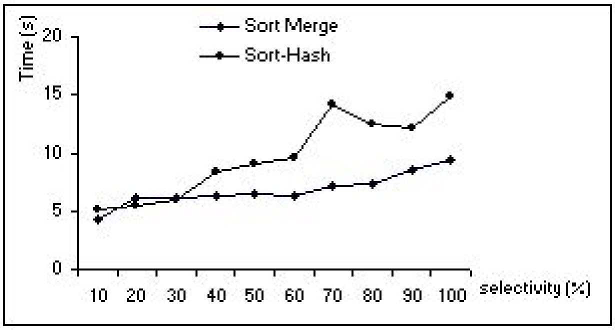

Fig. 8. Class B selectivity

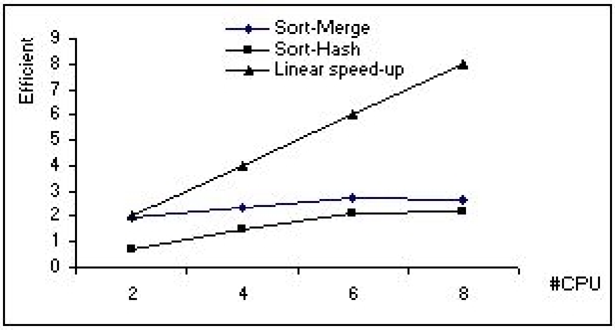

Fig. 9. Speed-Up

Figure 9 shows the speed-up result of both algorithms. The result shows that the sort-merge achieved higher speed-up as compared to the sort-hash. The overall speed-up for the two algorithms is low. The main reason for low speed-up is the high overhead of communication cost in the shared nothing architecture. These communications cost incurred on finding the range partitioning vector and data redistribution using *MPI_Alltoall* primitive. In our experiment, we did not consider disk I/O as one of the overhead.

7 Conclusion

The two algorithms implementation shows a great potential for performing collection equi-join. In overall sort-merge algorithm is more efficient than the sort-hash due to less numbers of comparison during merging.

There are many improvements that can be made on these two algorithms, especially in terms of memory management and the representation of OID. For the sort-hash method, an improvement on the representation of chained list can be made. For example it may be possible to represent each object instances as single valued entity rather than the whole object itself. Another possible improvement is how to filter out the unnecessary comparison during the probe phase.

They are few limitation of the implementation that can also be improved. In our implementation we did not deal with NULL collection values. One possible solution is to filter out objects that have NULL collection type during the reading because these objects are not useful for the collection equi-join. We assume a fixed size of collection attribute type in our implementation. More efficient data structures to deal with dynamic collection attribute size are desired to save memory space.

References

1. Cattel, R.G.G. (ed.), The Object Database Standard: ODMG-93, Release 1.1, Morgan Kaufmann, 1994.
2. Taniar, D., and Rahayu, W., "Object-Oriented Collection Join Queries", Proceedings of TOOLS Pacific'96 International Conference, Melbourne. pp. 115-125, 1996.
3. Taniar, D. and Rahayu, J.W., "Chapter 5: A Taxonomy for Object-Oriented Queries", Current Trends in Data Management Technology, A. Dogac, M.T.Ozsu, and O.Ulusoy (eds.), ISBN: 1-878289-51-9, Idea Group Publishing, pp. 69-96, 1999.
4. Taniar, D. and Rahayu, J.W., "Parallel Collection-Equi Join Algorithms for Object-Oriented Databases", Proceedings of International Database Engineering and Applications Symposium IDEAS'98, IEEE Computer Society Press, pp. 159-168, 1998.
5. 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.
6. Pacheco P.S., Parallel Programming with MPI, Morgan Kaufmann, 1997.
7. Ramakrishnan R. and Gehrke J. Management Systems 2nd Edition, McGraw-Hill 2000.
8. Goil S. and Choudhary A., "High Performance OLAP and Data Mining on Parallel Computers", Technical Report 1997, Department of Electrical & Computer Engineering, Northwestern University.
9. Wang X., Luk W.S., "Parallel Join Algorithm on a Network of Workstations", Proceedings of the first international symposium on Databases in Parallel and Distributed Systems, 1988.