Comparison of Parallel Algorithms for Path Expression Query in Object Database Systems

Guoren Wang, Ge Yu
School of Information Science and Engineering, Northeastern University, Shenyang 110006, P.R. China
{wanggr,yuge}@mail.neu.edu.cn

Kunihiko Kaneko, Akifumi Makinouchi
Graduate School of Information Science and Electrical Engineering, Kyushu University, Fukuoka 812, Japan
{kaneko,akifumi}@db.is.kyushu-u.ac.jp

Abstract

In this paper, We proposed a new parallel algorithm for computing path expression, named *Parallel Cascade Semi-join (PCSJ)*. Moreover, a new scheduling strategy called right-deep zigzag tree is designed to further improve the performance of the PCSJ algorithm. The experiments have been implemented in a NOW distributed and parallel environment. The results show that the PCSJ algorithm outperforms the other two parallel algorithms(the parallel version of forward pointer chasing algorithm*(PFPC)* and the index splitting parallel algorithm*(IndexSplit)*) when computing path expressions with restrictive predicates and that the right-deep zigzag tree scheduling strategy has the better performance than the right-deep tree scheduling strategy.

Key Words: Object databases, path expressions, parallel algorithms, scheduling strategies

1 Introduction

On one hand, data managed by database management systems are becoming very large in data-intensive applications such as E-Commerce, Digital Library, DNA Bank, Geographic Information Systems(GIS). Thus efficient parallel algorithms for accessing and manipulating a large volume of data are required to provide high performance for users. The parallel processing is an important approach for realizing high-performance query processing in such data-intensive applications. Up to now, a lot of research work has been done for parallel algorithms in the context of relational database systems. For example, DeWitt et al. proposed a parallel hybrid hash join algorithm in the paper[4] and Kitsuregawa presented a centralized GRACE join algorithm[7] and corresponding parallel algorithm[8]. On the other hand, relational database systems can not effectively and efficiently meet the requirements of emerging advanced database applications due to the limitations of their modeling capability. In these application domains, data types and semantics among data are much richer than conventional applications. One of promising approaches to meet the needs of these domains is object-orientation. One distinguished feature of object database system is path expression and most queries on an object database are based on path expression because it is the most natural and convenient way to access the object database, for example, to navigate the hyper-links in a web-based database. Because of the orthogonality of OQL[1], a path expression can be placed in *SELECT*, *FROM*, and *WHERE* clauses. The following is a typical example of OQL statement which uses path expressions in the *SELECT*, *FROM*, and *WHERE* clauses, respectively. In this paper, we will focus on the parallel execution of the path expressions in *WHERE* clauses, this kind of path expressions is also called as complex predicates. In the ODMG 2.0 standard, the expressing capability of path expressions is limited to some degree[1]. For example, a predicate can be applied only on the last class in a path. Actually, the definition of path expressions can be extended such that each class in a path expression can be qualified with one or more predicates. The general form of a extended path expression is as follows.

$$rv[p_1].NA_1[p_2].NA_2[p_3]...NA_n[p_{n+1}] \qquad (1)$$

where rv is a range variable, $NA_i(1 \leq i \leq n)$ is a nested attribute of class C_i, $p_i(1 \leq i \leq n+1)$ is a predicate applied on class C_i. The extent of class C_i is denoted as C_i itself.

In recent years, more and more researchers are moving their research interests from parallel relational database systems to parallel object database

Proceedings of the Seventh International Conference on Database Systems for Advanced Applications. Hong Kong, April 18-21, 2001.

systems[6][13], and correspondingly some parallel join algorithms have been proposed for object database systems[9]. Parallel pointer-based join techniques have been presented for object-oriented databases in the paper[9]. However, to the best of our knowledge, few of approaches have been presented for computing path expressions in parallel. It is sure that parallel pointer-based join techniques can be employed for computing path expressions and they are easy to be parallelized. However, we think that they can be improved from the following three aspects. First, these join algorithms do not utilize the cascade feature of path expressions. That is, a path expression can be computed in a pipelining way by converting the path expression to a right-deep tree. Secondly, a path expression can be converted to an equivalent semi-join rather than join expression and thereby reducing CPU cost and communication cost. Thirdly, during the execution of a semi-join expression, an object's OID rather than the whole object is necessary to send from a semi-join operation to the next one, thus, the cost of communication for computing the path can be reduced further.

The main contribution of this paper is that it presents a new parallel algorithm for computing path expressions: *parallel cascade semi-join(PCSJ)* which computes a path expression using semi-join operations rather than join and thereby dramatically reducing the cost of computing the path expression compared with the Pointer-based parallel join algorithm. The experimental results show that the PCSJ algorithm greatly outperforms the other two typical parallel algorithms. In order to deal with limited memory size, this paper proposes a new scheduling strategy called right-deep zigzag tree to further improve the performance of the PCSJ algorithm by avoiding extra I/O disk overhead. Our preliminary experimental result shows that the right-deep zigzag tree scheduling strategy has the better performance than the right-deep tree scheduling strategy.

The remainder of this paper is organized as follows. Section 2 presents a parallel cascade semi-join algorithm. Section 3 introduces two other kinds of parallel algorithm for computing path expressions, including the parallel forward pointer chasing algorithm and the index splitting approach. Section 4 gives the performance analysis and comparison of the PCSJ algorithm with the other two parallel path algorithms. Finally, Section 5 concludes the paper.

2 Parallel cascade semi-join algorithm

In this section, we first explain how to use semi-join operation to compute path expressions, and then describe how to schedule the computation of a path expression, finally discuss the parallel execution of a sub-path expression.

2.1 Using semi-join to compute path expressions

If there is an extent for each class in a path expression, then the implicit joins in the path expression can be converted into explicit joins for computing the path expression. For example, the path (see expression (1)) described in Section 1 can be computed by the following join expressions.

$$sel(C_1,p_1) \overset{NA_1}{\bowtie} sel(C_2,p_2) \overset{NA_2}{\bowtie} \cdots \overset{NA_{n-1}}{\bowtie} sel(C_n,p_n) \overset{NA_n}{\bowtie} sel(C_{n+1},p_{n+1}) \quad (2)$$

If each join operation in formula (2) is replaced by one semi-join operation, an equivalent semi-join expression can be obtained as follows.

$$(sel(C_1,p_1) \propto (sel(C_2,p_2) \propto (\cdots (sel(C_n,p_n) \propto sel(C_{n+1},p_{n+1}))\cdots))) \quad (3)$$

Formula (2) can be evaluated and performed in two reversed directions: forward and backward, respectively, while formula (3) can be performed in only one direction, i.e., backward. The forward semi-join expression is not equivalent to the backward semi-join expression because semi-join operator is not commutable. In formula (3), the execution of any semi-join operation, to say $sel(C_i,p_i) \propto sel(C_{i+1},p_{i+1})(1 \leq i \leq n)$ can be roughly described as follows. First, the extent of class C_i is scanned and predicate p_i is applied on the scanned objects, and then the selected objects are used to build a hash table for class C_i by applying a hash function on the nested attribute NA_i. Secondly, the result objects of the previous semi-join operation, i.e. $sel(C_{i+1},p_{i+1}) \propto sel(C_{i+2},p_{i+2})$, are used to probe the hash table of class C_i with the same hash function as that used in the building phase. If an object in the hash table is matched, then OIDs of the matched objects is passed to the next semi-join operation, i.e. $sel(C_{i-1},p_{i-1}) \propto sel(C_i,p_i)$. The semi-join approach to path expressions has the following advantages over the pointer-based join approach.

(1) Due to replacement of join operation with semi-join operation, the execution cost of the path expression is greatly reduced, including CPU and communication cost.

(2) For the pointer-based join approach, a projection operation is needed for projecting objects from the

final results of the join expression, while in the semi-join approach it is obvious that this projection operation is unnecessary because just OIDs rather than objects are sent from semi-join operation to the next one.

(3) A cascade semi-join expression can be implemented as a semi-join right-deep tree. The other feature of a semi-join cascade query is that the position of any semi-join operation in the expression is fixed. Thus optimization of a cascade query is relatively easier to be done than a general join query.

2.2 Right-deep zigzag tree scheduling strategy

For the parallel evaluation of multi-join queries, there are some interesting scheduling strategies: left-linear(left-deep) trees, left-oriented bushy trees, wide bushy trees, right-oriented bushy trees, right-linear(right-deep) trees described in the paper[14]. The paper reports the experiences with the implementation of these strategies for the implementation of multi-join queries on PRISMA/DB. In addition, the paper[2] extended the right-deep tree strategy to the segmented right-deep tree strategy for the execution of pipelined hash joins and gave the simulation experimental comparison of these two strategies. The paper[16] presented a new scheduling strategy called zigzag tree, which is intermediate between left-deep tree and right-deep tree, and the paper[10] analyzed and compared the zigzag strategy with the right-deep tree and bushy tree strategies. It is obvious that these scheduling strategies can not be directly applied to the parallel computation of path expressions except for the right-deep tree strategy.

For a cascade semi-join operations in a path expression, if there is not enough available main memory holding the hash tables for a path expression, then the path expression can be broken into several sub-path expressions such that the hash tables for each sub-path expression can be held in the available main memory. For example, for the path expression as shown in Fig. 1(a). If all hash tables for the path are expected to fit into memory, the path can be executed in parallel with the algorithm described in the next subsection. However, if there is not enough memory, then this path has to be broken into several sub-path expressions. Assume that the path is broken into two sub-path expressions p_1 = {result $\longleftarrow$ $(sel(C_1, p_1) \propto (sel(C_2, p_2) \propto F'$ } and p_2 = { F' $\longleftarrow$ $(sel(C_3, p_3) \propto (sel(C_4, p_4) \propto (sel(C_5, p_5) \propto (sel(C_6, p_6)$ }, as shown in Fig. 1(b). First, the sub-path expression p_2 is scheduled and executed. The result of the sub-path is written into the disk file F'. And then, the sub-path expression p_1 is scheduled and executed. During the execution of sub-path p_1 disk file F' is read to probe corresponding hash tables. This is similar to the idea of right-deep tree scheduling strategy described in papers[3] and [11]. Although the paper[2] has shown that the schema using segmented right-deep trees for pipelined hash joins outperforms the schema using right-deep trees, the segmented right-deep tree schema can not be directly applied to compute path expressions because of the cascade feature of path expression. In order to avoid extra disk I/O overhead, we present a new scheduling strategy called right-deep zigzag tree as shown in Fig. 1(c) and the dynamic bottom-up scheduling strategy is adopted to deal with the memory constraint. Our scheduling strategy is not only a special form of zigzag tree but also a special form of segmented right-deep tree, so this scheduling strategy is referred to as right-deep zigzag tree, which has the following characteristics.

(1) For any sub-path p_i of a right-deep zigzag tree, if i is an odd number then p_i is a right-deep sub-tree. Otherwise, p_i is a left-deep sub-tree and the length of all left-deep sub-trees in a right-deep zigzag tree is 2. The first sub-path is a right-deep sub-tree.

(2) Just all right-deep sub-paths in a right-deep zigzag tree are scheduled to execute in parallel. The result of the ith($i \neq 1$) right-deep sub-path is directly used to build corresponding hash table for the execution of the (i-1)th right-deep sub-path rather than to write back into a disk file and thereby to avoid extra disk overhead occurring in right-deep tree scheduling strategy.

(3) Right-deep zigzag tree is an extreme form of zigzag tree. Right-deep zigzag tree also is an extreme form of right-deep tree. A right-deep zigzag tree can also be viewed as a cascade segmented right-deep tree, i.e. the result of the ith right-deep sub-path is the left input of the last semi-join operation of the (i-1)th right-deep sub-path. So, right-deep zigzag tree can also be named cascade segmented right-deep tree, to reflect the cascade feature of path expressions.

2.3 Parallel execution of sub-path expressions

In this subsection, we will discuss the parallel execution of a right-deep sub-path in a right-deep zigzag tree. It works in two phases: building and probing. In the building phase, extents of the classes except for the last

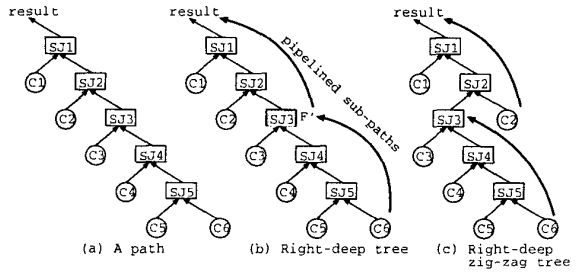

Figure 1: Scheduling strategies

one class in sub-path expression are partitioned using a split table and then the hash tables corresponding to various partitions are built. These n partitioning operations are performed in parallel. The procedure of partitioning the extent of class C_i is shown in Fig. 2. First, the extent of class C_i is scanned and the operation $sel(p_i)$ is applied on the scanned objects. After a partitioning hash function *phf* is imposed on the join attribute NA_i of the selected objects, a split table is used to determine which hash table, to say hash table HT_j, these objects should be inserted into. Then, the hash function hf_j different from *phf* is used to build hash table HT_j.

The number of partitions of each class in the path expression are the same, which is just dependent on the number of sites available, that is times of the number of sites available. We assume that the number of sites available be m, then the number of partitions h is times of m, that is, h = k * m(k $\geq$ 1). These partitions are allocated to m sites in round-robin way with formula *siteno(the ith partition)* = *i modulo m*.

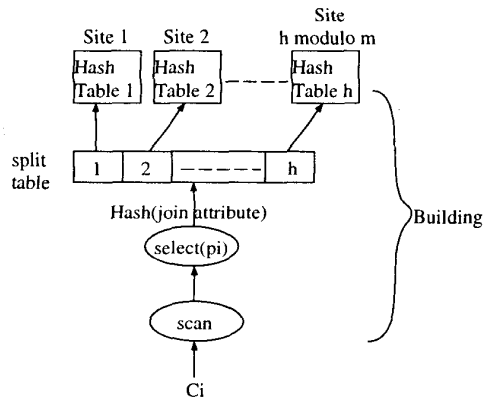

Figure 2: Building phase of the PCSJ algorithm

During the probing phase, the extent of class C_{n+1} is scanned and predicate p_{n+1} is applied on the scanned objects. And then the same hash function *phf* as that used in the building phase is computed on the OID of each selected object, and the hashed value is used to determine which hash table is used to probe through the split table. If hash table HT_i, for example, is selected, then only the OID part of the selected object of class C_{n+1} rather than the whole object is sent to site *i modulo m* to probe hash table HT_i of class C_n. Similarly, if an object, to say o', is matched in hash table HT_i of class C_n, then the OID value of o' is used to probe a hash table of class C_{n+1}, to say HT_j, at site *j modulo m*. Finally, in the hash tables of class C_1, all the matched objects are results of the path expression. The probing phase of the PCSJ algorithm is demonstrated in Fig. 3.

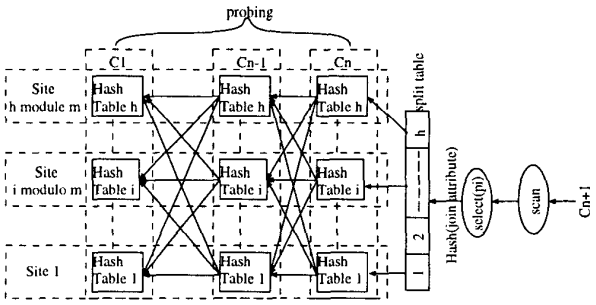

Figure 3: Probing phase of the PCSJ algorithm

3 Other two parallel algorithms for path expressions

In this section, we introduce other two kinds of parallel algorithms, the parallel forward pointer chasing algorithm and the index splitting approach, for computing path expressions so that we can analyze and compare the performance of PCSJ with that of them.

3.1 Parallel forward pointer chasing algorithm

Since relationships among objects are stored in the database, one natural way to execute a path expression is forward pointer chasing along path instances. In this way, all objects in the extent of the first class C_1 in the path are first fetched and then predicate p_1 is imposed on them. For each fetched and matched object o, its next object is obtained by forward chasing the pointer along the path instance and the corresponding predicate is checked. This forward chasing is continuously going if the object is satisfied with the predicate and the last object of the path instance is not encountered. Otherwise, the next path instance is forward chased.

If all predicates in the path expression are satisfied, then the object o is one of the result objects with respect to the path expression and is put to the result buffer. For the sake of performance comparison, we have been designed and implemented a parallel version of forward pointer chasing(short for PFPC). Readers can refer paper the paper[5] for the detailed design and implementation techniques of the PFPC algorithm.

3.2 Index splitting Approach

An index splitting approach for path expression is proposed in [12]. In the scheme, index is split vertically and horizontally into sub-indices, each of which is placed on a separate PE (Process Element). Index retrieval is done in parallel with minimal communication cost. Each sub-index of multi-index is called as a V-partition and each sub-index obtained by splitting a V-partition horizontally is called a HV-partition. Fig. 4 shows index element denoted by $< m, n >$. For example, path expression is $C_1.C_2.C_3.C_4$; c_1, c_2, c_3 and c_4 are corresponding OID values in a path instance, so the path $c_1.c_2.c_3.c_4$ is divided vertically into $< c_1, c_2 >$, $< c_2, c_3 >$, $< c_3, c_4 >$. Before running the algorithm, all the HV-partitions are stored into the PEs. When a simple value or an OID is given as a retrieval request, the retrieval operation is performed as shown in Fig. 4.

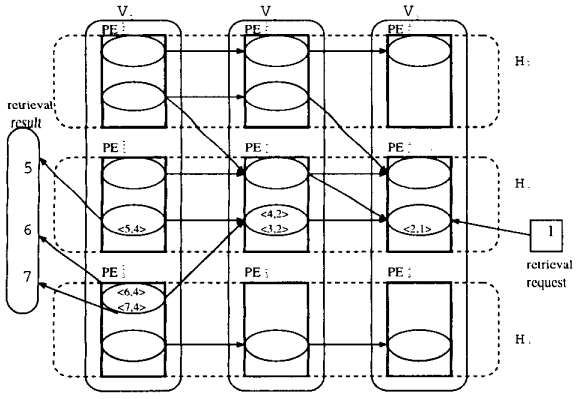

Figure 4: Parallel algorithm of index splitting

4 Performance evaluation

In order to further evaluate the performance of the PCSJ algorithm, we have designed and implemented the PFPC and IndexSplit algorithms, and have done much actual test work. This section first introduces the test environment and the test database, and then analyzes and compares the speedup and scaleup performance of the PCSJ algorithm with the PFPC and IndexSplit algorithms. Finally, we also compare the performance of the right-deep zigzag tree scheduling strategy and the right-deep tree scheduling strategy.

4.1 Test environment and test database

In order to test the performance of these algorithms, we design a special test database. Performance tests include the speedup and scaleup tests. The speedup means the ability to grows with the system size while keeping the problem size as constant. Scaleup measures the ability to grow with both the system size and the problem size. Scaleup is defined as the ability of an N-times larger system to perform an N-times larger job in the same elapsed time as the original system. We define the class in test database as follows.

```
class BasicClass: public d_Object {
  d_Ref<BasicClass> NA;
  int               selAttr;
  char              otherAttr[strLength];
    }
```

The attribute selAttr is used to set restrictive predicates applied on the class. NA is a nested attribute of class BasicClass(the item for the last class in the path is set to null). The attribute otherAttr is used to set the size of objects in the test database. The size of objects is 252 bytes.

In the speedup test, we design test database to compute path expression when the path length is 3, that is $C_1.C_2.C_3.C_4$. So we construct 4 extents for 4 classes on 4 sites, and there is 100,000 objects in each extent. The tested path expression is $C_1[P_1].C_2[P_2].C_3[P_3].C_4[P_4]$. At the same time, we require: (1)The selectivity of p_i is 90% for class C_i; (2)90% objects in two collection can be joined.

According to above requirements, the number of results is about:

$$100,000 * 0.9 * 0.9^3 = 65,610 \quad (4)$$

In the scaleup test, the length of path expression is added as sites increase. So eight extents is built to form the path expression:$C_1[P_1].C_2[P_2]...C_7[P_7].C_8[P_8]$ at maximum. We will test $C_1[P_1].C_2[P_2]...C_i[P_i]$, when i sites are working (1<i<=8).

The hardware configuration for testing is eight PCs connected with high-speed switch. Eight PCs are all the same in configuration: AMD-K6 233Mhz CPU, 64MB memory and 4.3GB hard disk. The operating system is Solaris 2.5 and Shusse-Uo system is 2.0[15]. On every working site, the WAKASHI server is running. Shusse-Uo is a distributed and parallel object

database system that has been under development at Kyushu University of Japan and Northeastern University of China. In the Shusse-Uo system, heaps are provided for building databases. A heap consists of fixed-length pages. It is persistent when it is mapped onto a disk, otherwise volatile. A database may consist of more than one persistent heap. Each heap can be globally shared by any client in the system. In the testing database, objects of each class are stored in different persistent heap, i.e. 8 persistent heaps are used for the performance evaluation.

We have tested the parallel forward pointer chasing algorithm(PFPC), parallel cascade semi-join algorithm(PCSJ) and parallel index splitting algorithm(IndexSplit) based on the test database. Also, we give the performance comparison and analysis of the right-deep zigzag tree scheduling strategy with the right-deep tree scheduling strategy.

4.2 Speedup performance

Fig. 5(a) shows the speedup response time to compute $C_1[P_1].C_2[P_2].C_3[P_3].C_4[P_4]$ from 2 to 8 sites; and Fig. 5(b) shows the corresponding speedup curve for three parallel algorithms. We can see that the response time of PFPC and IndexSplit are much longer than that of PCSJ, especially when there are less working sites. The speedup performance shows PCSJ is faster than PFPC and IndexSplit when computing path expressions with restrictive predivates. For PFPC and IndexSplit, all objects of classes in the path have to be scanned and read into memory to check if the predicates on the path is satisfied, so for every traversing operation from one class to the next class one I/O may occur. Because PFPC and IndexSplit algorithms do not consider the memory size constraint, one object may be read from disk several times and thereby extra I/O overhead occurs. Thus the so-called I/O thrashing problem might occur. Because PCSJ considers the memory size constraint, there is no extra I/O overhead. So, PCSJ is faster than PFPC and IndexSplit. Because IndexSplit has to pay additional cost for dealing with the index, IndexSplit is slower than PFPC. The common trendency for the three parallel algorithms is that response time decreases as the number of sites increases. And the graph shows that response time will continue to drop down when the number of sites exceeds 8.

4.3 Scaleup performance

Fig. 6(a) shows scaleup response time for computing $C_1[P_1].C_2[P_2]...C_i[P_i]$ when i sites are available and

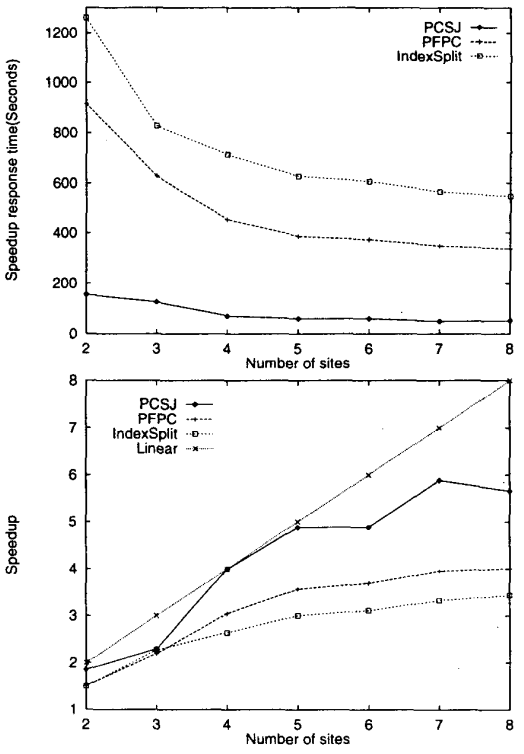

Figure 5: (a)Speedup response time (b)Speedup rate

Fig. 6(b) shows the corresponding scaleup performance curve for three parallel algorithms. Obviously, the scaleup performance of PCSJ is much better than that of PFPC and IndexSplit. In PCSJ, the scaleup curve is always near to linear. There is a little interference between sites in the process of PCSJ, since every hash table locates different storage space within DSVM space and the amount of communication always keeps constant. On the other side, PFPC and IndexSplit have worse scaleup. The main reason is that the PFPC and IndexSplit do not adopt the partitioning technique and thereby the interference among sites increases as the number of sites increases.

4.4 Discussion

From the previous performance analysis, we can see that PCSJ has the best speedup and scaleup performance among the three parallel algorithms when computing path expressions with restrictive predicates. However, what happens for path expressions without restrictive predicates? that is to say, how about the performance of the three parallel algorithms for path expressions occurring in the *SELECT* and/or *FROM*

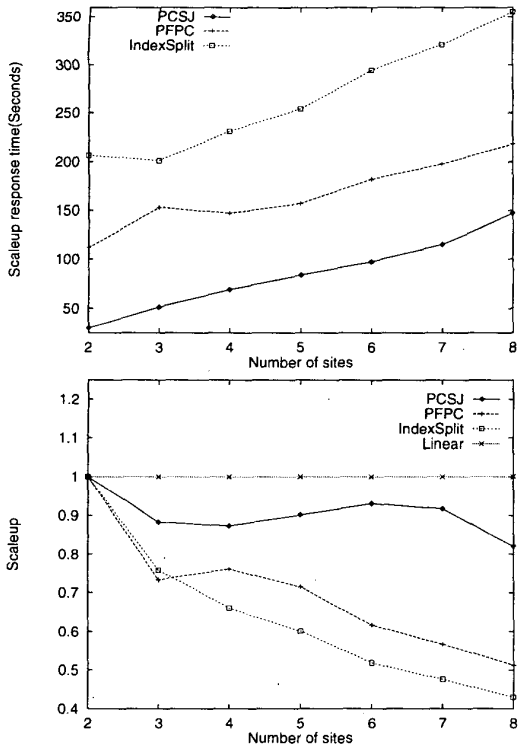

Figure 6: (a)Scaleup response time (b) Scaleup rate

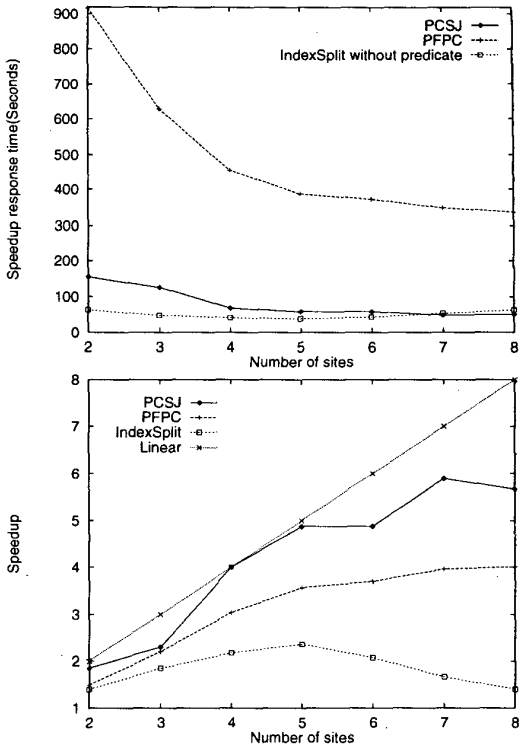

Figure 7: (a) Speedup response time without predicates (b) Speedup rate without predicates

clauses? Fig. 7(a) and Fig. 7(b) show the speedup response time and the speedup curves of the three parallel algorithms in the case of path expressions without predicates, respectively. From the response time, IndexSplit is fastest among the three algorithms within 6 sites. This is mainly because PCSJ and PFPC have to read all objects in the path expression from disk no matter whether or not there exist predicates in the path expression while IndexSplit need not read any object by the help of index. However, the PCSJ algorithm outperforms the IndexSplit algorithm as the number of sites increases beyond 6. From Fig. 7(b), we can see that IndexSplit has the worst speedup in the case of without predicates although IndexSplit has the fastest response time. This is because the overhead of parallelization dominates the whole response time. The fact shows that it is unnecessary to parallelize the execution of path expressions based on the splitting index.

4.5 Right-deep zigzag tree vs right-deep tree

This subsection analyzes the performance of different scheduling strategies. Fig. 8 gives the performance curves of right-deep zigzag tree and right-deep tree scheduling strategies with limited memory, 8 sites. The length of the path expression is set to 8. The y-axis value is the response time and the x-axis value is the percentage of available memory. The percentage of available memory is given in the formula: $percentage = \frac{2^x}{8}$. From Fig. 8, we can see that the response time of two strategies are same when 100% hash tables are expected to fit into memory. However, as the size of the available memory decreases the response time of two schedule strategies increases since the path expressions has to be broken into two several sub-path due to the memory size constraint. The right-deep zigzag tree outperforms the right-deep tree. This is mainly because that when a path expression is broken into several sub-path expressions to compute, the right-deep zigzag tree schedule strategy just spends overhead to start the parallel execution of these sub-path expressions while the right-deep tree scheduling strategy needs extra I/O disk overhead besides the startup time for each sub-path expression.

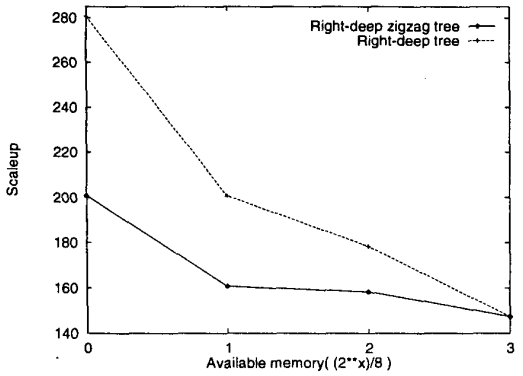

Figure 8: Right-deep zigzag tree vs right-deep tree with limited memory

5 Conclusions

This paper studies one of the open problems in parallel object database systems, namely parallel computation of path expressions. After analyzing the existing approaches to path expressions, this paper has presented a new parallel algorithm PCSJ to compute path expressions in object database systems, and given the performance analysis and comparison of algorithm with the other two typical parallel algorithms. The experimental results show that the PCSJ algorithm outperforms the other two parallel algorithms when computing path expressions with restrictive predicates. At the same time, a new scheduling strategy *right-deep zigzag tree* is proposed to further improve the performance of the PCSJ algorithm by avoiding extra disk overhead. The experimental result shows that the *right-deep zigzag tree* outperforms the right-deep tree in the case of limited memory. The algorithm has been implemented in an object database system Shusse-Uo which is under development at Kyushu University of Japan and Northeastern University of China in the environment of Networks Of Workstations(NOW).

References

[1] R.Cattell, *The object database standard: ODMG 2.0*. Morgan Kaufmann Publishers, Inc. 1997.

[2] M.-S.Chen, M.Lo, P.S.Yu, et al. "Using segmented right-deep trees for the execution of pipelined hash joins." *Proceedings of the 18th VLDB Conference*, Vancouver, Canada, 1992, pp.15-26.

[3] M.-S.Chen, M.Lo, P.S.Yu, et al. "Applying segmented right-deep trees to pipelining multiple hash joins," *IEEE TKDE*, 1995, 7(4): pp.656-668.

[4] D.DeWitt, S.Ghandeharizadeh, D.Schneider, et al. "The Gamma database machine project," *IEEE TKDE*, 1990, 2(1): pp.44-62.

[5] Q.Fang, G.Wang, G.Yu, K.Kaneko, and A.Makinouchi, "Design and Performance Evaluation of Parallel Algorithms for Path Expressions." *1999 International Symposium on Database Applications in Non-Traditional Environments*, Kyoto, Japan, 1999, pp.373-380.

[6] K.-C.Kim, "Parallelism in object-oriented query processing," *Proc. of the 6th ICDE Conference*, Feb. 1990, pp.209-217.

[7] M.Kitsuregawa, H.Tanaka, and T.Moto-oka, "Application of hash to data base machine and its architecture," *New Generation Computing*, 1983, 1(1).

[8] M.Kitsuregawa, S.Tsudaka, and M.Nakano, "parallel GRACE hash join on shared-everything multiprocessor: Implementation and performance evaluation on symmetry s81," *Proceedings of the 8th ICDE Conference*, Temper, Arizona, Feb. 1992, pp.256-264.

[9] D.Lieuwen, D.DeWitt, and M.Mehta, "Parallel pointer-based join techniques for object-oriented databases," *Proceedings of the 2nd PDIS Conference*, January 1993.

[10] R.S.G.Lanzelotee, P.Valduriez, and M.Zait, "On the effectiveness of optimization search strategies for parallel execution spaces." *Proceedings of the 19th VLDB Conference*, Dublin, Ireland, 1993, pp493-504.

[11] D.A.Schneider and D.J.DeWitt, "Tradeoffs in processing complex join queries via hashing in multiprocessor database machines." *Proceedings of the 16th VLDB Conference*, Btishbane, Australia, 1990, pp.469-480.

[12] T.Tsuji and T.Hochin, "Parallel Index Retrieval of Complex Objects", *Advanced Database Systems for Integration of Media and User Environments'98*, Singapore, 1998, pp.179-184.

[13] A.K.Thakore, S.Y.W.Su, and H.X.Lam, "Algorithms for asynchronous parallel processing of object-oriented databases," *IEEE TKDE*, 1995:7(3), pp.487-504.

[14] A.N.Wilschut, J.Folkstra, and P.M.G.Apers, "Parallel evaluation of multi-join queries." *Proceedings of the SIGMOD Conference*, San Jose, CA USA, pp.115-126

[15] G.Yu, K.Kaneko, G.Bai, and A.Makinouchi, "Transaction management for a distributed object storage system WAKASHI-design, implementation and performance," *Proc. of the 12nd ICDE Conference*, New Orleans, 1996, pp.460-468.

[16] M.Ziane, M.Zait, and P.Borla-Salamet, "Parallel Query Processing with Zigzag Trees." *VLDB Journal*, 1993, 2(3): pp.277-301.