Regular Paper

A Parallel Navigation Algorithm with Dynamic Load Balancing for OODBMSs

LAWRENCE MUTENDA,† TAKANOBU BABA,†† TSUTOMU YOSHINAGA †† and KANEMITSU OOTSU ††

High-performance navigational access to objects is crucial if the demands of object-oriented database management system (OODBMS) target application areas are to be met. We propose an algorithm to navigate, in parallel, objects in a parallel OODBMS for a shared-nothing environment. The algorithm exploits, among others, qualified path expressions and the set-valued nature of OODBMS attributes to generate independent paths, that can be traversed in parallel if the objects referenced therein are declustered across multiple nodes. We also propose a decentralised dynamic load balancing algorithm based on the premise that the cost of arbitrary user-defined functions (UDF) in the predicates of qualified path expressions can be estimated accurately by an average calculated over time and used in determining the load at each node in a shared-nothing system. Analytical and simulation results are presented which show that our navigation and load balancing strategies can improve query response time significantly. Simulation shows that if objects are uniformly partitioned across system nodes, our algorithm can achieve a speedup of up to 13.1 from 2 to 32 nodes. If the data is skewed, dynamic load balancing decreases the response time for navigating a seven level path expression by up to 34 %.

1. Introduction

Object-oriented database management systems (OODBMS) are considered viable alternatives to relational databases for new application areas such as CAD, CAM and GIS, because they provide rich facilities for modeling and processing of structural as well as behavioural properties of the complex data structures found in these new areas[5] and many commercial systems like ObjectStore and Objectivity[12] have been developed.

Parallelism has been highly effective in improving the performance of RDBMSs, with focus mainly on the join operation.[15)7)21)]. Due to the inherent generality of OODBMS and the performance requirements of corresponding applications, high performance implementation of system operations is required. Some work has been done on parallelism in OODBMSs which has demonstrated that parallel processing can be effectively applied in such environments[5)6)9)17)]. Recognizing that high-performance navigation is crucial to efficient OODBMS queries, this paper proposes an algorithm to navigate, in parallel, objects in a shared-nothing (SN) OODBMS. The algorithm exploits, among others, qualified path expressions and the set-valued nature of OODBMS attributes to generate independent paths, that can be traversed in parallel if the objects referenced therein are declustered across multiple nodes.

The OODBMS reference model allows the user to include user-defined functions (UDF) in query predicates. Examples include scaling geometric objects in CAD applications or the *veg* function in the Sequinoa benchmark for GIS systems[11]. Such UDFs are computationally or I/O expensive. This coupled with generality in access pattern for different queries, very likely introduces load imbalance in the course of executing queries including such UDFs, in an SN OODBMS. We propose a decentralised dynamic load balancing algorithm based on the premise that the cost of such arbitrary UDFs can be estimated over time and used in determining the load at each node in an SN system. An analytical model is presented as well as simulation results which indicate that our navigation and load balancing strategies can improve query response time significantly.

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 discusses sources of parallelism in OODBMS. Section 4 describes the proposed

† Institute of Industrial Science, University of Tokyo
†† Department of Information Science, Utsunomiya University

parallel navigation algorithm together with the load balancing algorithm. An analytical model is described in section 5 together with analytical evaluation results. Simulation experiments are described in section 6 and conclusions and further work are discussed in section 7.

2. Related Work

One of the earliest descriptions of parallelism in OODBMSs is done by Kim[14] who identifies path parallelism, node parallelism and class-hierarchy parallelism as three types of parallelism possible in an OODBMS, all based on the notion of object query graph. Path parallelism is closely related to the work presented in this paper, but algorithm details are different

Lieuwen et al[17] describes a number of parallel pointer-based joins for OODBMSs, among them hash-loops, hybrid-hash/page-pointer and others. We use a strategy similar to the hybrid-hash/page-pointer scheme, but we apply it to a path-expression of arbitrary length instead of the 2 classes they consider. Unlike them we include UDFs in query predicates and analyze of how they affect response time in the presence of object placement skew and propose a dynamic load balancing algorithm for such situations. To the best of our knowledge this is the first time that load balancing is applied to the processing of a path expression of arbitrary length in the presence of UDFs.

Dewitt et. al.[6] uses par-sets to parallelize traversals of objects in a parallel OODBMS. A par-set is simply a set of database objects, declustered on different SN nodes and on whom specified methods are applied in parallel. We share goals with this work, i.e. speeding up the navigation of objects but the concepts and algorithms used are different. Chen et. al.[5] introduce elimination and identification based techniques, that use the concept of multi-wavefronts. This work exploits the graph nature of OODBMS schemas. The strategies used in this paper are different to ours, but we note that the type of queries they address can also be solved by our proposed algorithm.

There is a large body of work discussing dynamic load balancing for SN RDBMS[13)18)21)19)], but virtually no corresponding work for parallel OODBMSs, to our knowledge. However some work was reported in Cario et. al.[3] on load-balancing in processing UDFs for object relational databases . We believe as well that executing UDFs constitutes a significant portion of processing in OODBMS unlike in relational systems where simple access and compare operations are dominant. We therefore apply the same technique with Cario et. al.[3], of estimating the method execution time by an average which is valid when time variation about the mean is low.

3. Parallelism In OODBMS Query Processing

3.1 An Parallel OODBMS Reference Model

In an OODBMS a class is defined by specifying its name, its attributes, its methods and the names of its superclass(es). A *simple attribute* has a primitive domain like integer and a *complex attribute* has a class domain. An attribute can be single-valued or set-valued. An objects is an instance of a class. Consider an SN processing system hosting a parallel OODBMS. Class objects are declustered across system nodes using some partitioning scheme, like round-robin, hashing range, or a user-defined scheme. Each object of a class has a unique physical 4-tuple *object-identifier (OID)* (c, k, p_k, u), that specifies the class c to which it belongs, the node k at which the object is stored, the disk page, p_k at node k in which it is stored i.e. the *page-identifier (PID)* of that *OID*, and u a unique system assigned value.

A path expression is called qualified if it includes predicates. A qualified path expression[8] is defined as $\mathcal{P} = C_1(P_1).A_1.(P_2)A_2 \ldots (P_n)A_n$ for $(n \geq 1)$. C_1 is the *root class* of the qualified path expression; A_i is an attribute of a class C_i such that C_i is the domain of the complex attribute A_{i-1}, of class C_{i-1}, $(1 < i \leq n)$; subscript i refers to the position of the class or attribute in the path expression; and P_i is a predicate executed on a class C_i object. Moreover $len(\mathcal{P})$ denotes the length of $\mathcal{P}$, i.e. the number of classes in the path expression. To denote the fact that instances of class C_i are declustered across nodes we use C_{ik}, for $1 \leq k \leq N$, N the number of nodes. A_n may or may not be a complex attribute. A single-valued complex attribute has one OID as a value, and a set-valued complex attribute has a set of OIDs as a value. The predicate P_i may be a UDF or it may be simple comparison.

If we define an object-tuple OT_i as the combination of all the projected attributes of the object O_i of class C_i and one OID, called the OT_iOID from the object's attribute participat-

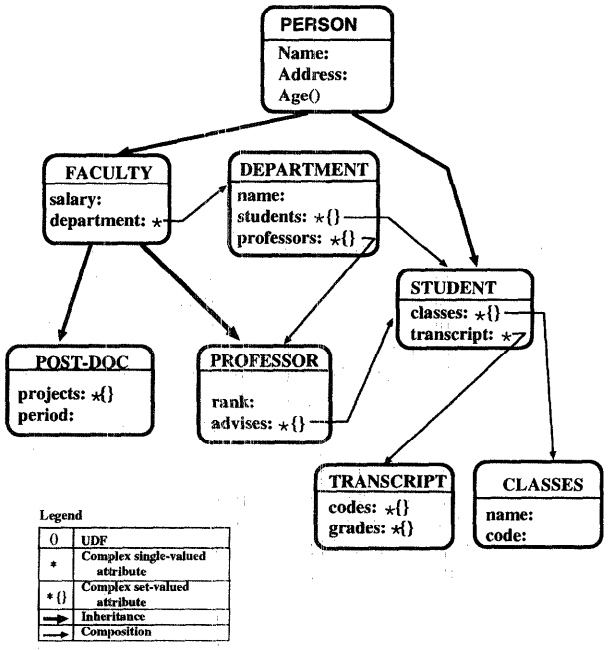


Fig. 1 An OODBMS schema graph

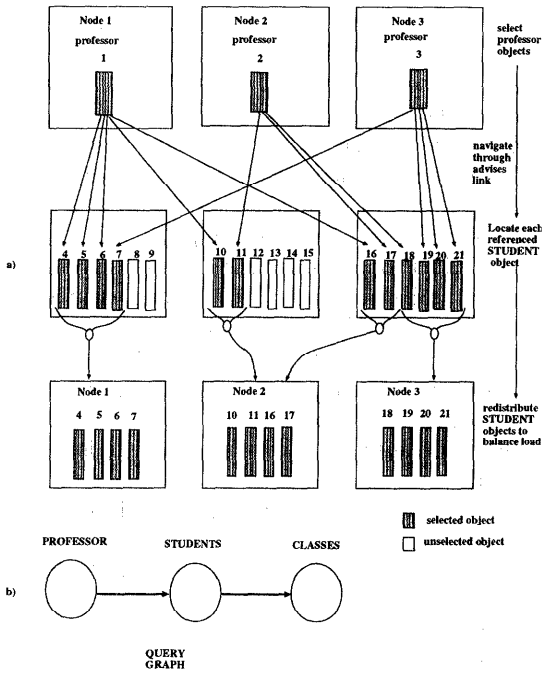


Fig. 2 Navigating in Parallel with Load-Balancing

ing in the corresponding path expression, then a *path instance* (*PI*) of $\mathcal{P} = C_1.A_1.A_2.\ldots.A_n$, is defined as a sequence of p object-tuples denoted $OT_1.OT_2.\ldots.OT_p$, where $p = n$ if A_n is a complex attribute or $p = n - 1$ if A_n is a simple attribute. If $q < p$, then $OT_1.OT_2.\ldots.OT_q$ is a *partial path instance* (PPI_q) of $\mathcal{P}$ where q is the number of classes traversed. Attributes projected depend on the query requirements.

Figure 1 illustrates a university OODBMS schema based on the reference model just described. The following query retrieves , from this database, the names of the courses taken by students over 35 years old, who are advised by professors whose rank is department chairman.

Q1:

```
SELECT R.Name , S.Name , C.name
FROM   R IN PROFESSOR, S IN STUDENT,
       C IN CLASSES
WHERE R(rank = 'dept chair').
       advises.(age() > 35 )classes.()name
```

A example of qualified path expression here is $\mathcal{P} = PROFESSOR(rank = 'deptchair').advises.(age() > 35)classes.()name$ with $len(\mathcal{P}) = 3$. The *STUDENT* class has a predicate with UDF *age*() inherited from class *PERSON*. The predicate for the *PROFESSOR* class is '*age*() > 35' and the predicate for *CLASSES* is empty.

3.2 Set/Path Parallelism

In this section we describe a form of parallelism we call *set/path parallelism* closely related to the path parallelism in Kim[14].

Consider the situation depicted in **Fig. 2**.a, where class objects are declustered onto disks in a 3-node SN environment. Consider a query **Q1** to instantiate the path expression *P(rank = 'dept chair')advises.(age() > 35)classes.()name*. The corresponding query graph is shown in Fig. 2.b. Each node, in parallel, scans its portion of the PROFESSOR class and applies the predicate *rank = 'dept chair'*. Noting that the OIDs used here are symbolic, assume that three objects OID 1, 2 and 3 satisfy the predicate. Node 1 examines the *classes* attribute of PROFESSOR object 1. This turns out 5 OIDs (4,5,6,10,16), three local (4,5,6) and 2 external (10,16). The required data for the *professor 1* object is projected and combined with each OID into OT_1, essentially a PPI_1.

Each such PPI_1 is either kept locally or sent to its respective node. PPI_1 whose Object-

tuple has OT_iOID 10 is sent to node 2 and that with OID 16 to node 3. Each node then dereferences STUDENT class objects according to the OT_iOIDs of the OT of its PPI_1s. If the execution of UDF $age() > 35$ is computationally expensive then load imbalance will occur between the 3 nodes since the number of target STUDENT objects are unequal. We transfer STUDENT objects 16 and 17 from node 3 to node 2 to balance the UDF execution load and then initiate UDF execution. If an object is selected, each PPI_1 referencing it is extended with an OT from that object to create PPI_2s, instantiating the partial path *P.advises*. Execution continues until the whole path is instantiated.

We term this parallelism exhibited here *set/path parallelism*. The difference between *path parallelism* in Kim[14] and the *set/path parallelism* here is that in the former different paths in the query graph, at the class level, are followed in parallel, whilst in the later the same path in a query graph at the class level, generates multiple paths at the object level, because of object declustering and the existence of set-valued attributes. Note that even with single-valued attributes multiple paths are still available from declustering alone.

4. A Parallel Navigation Algorithm

4.1 Algorithm Description

In this subsection we present the *parallel navigation with dynamic load balancing (PNDLB)* algorithm, that exploits set/path parallelism for instantiating path expressions. A description of the algorithm is given in **Fig. 3**. This is a more detailed description of the scheme in Fig. 2. The input to the algorithm is a path expression $\mathcal{P} = C_1(P_1).A_1.(P_2)A_2 \ldots (P_n)A_n$ with length $len(\mathcal{P})$.

The algorithm uses a breadth first fetch (BFF) traversal of a link (C_i, C_{i+1}) of a path expression $\mathcal{P}$ at each node in the SN system. Referring to Fig. 3, the algorithm begins in parallel at each at step 1, where i is a counter to indicate the class currently being processed. We assume that there is no object placement skew so that the number of objects in C_{1k}, the portion of C_1 stored at node k, is equal for all nodes. Hence there is no need for load balancing in processing the root class C_i.

Transmitting PPI_is is done in pages as much as possible to save on communication overhead.

For N nodes in an SN system, execute the following at each *node* k, $1 \leq k \leq N$:
INPUT:$\mathcal{P} = C_1(P_1).A_1.(P_2)A_2.\ldots.(P_n)A_n$
(1) **BEGIN:** $i = 1$:
(2) Scan C_{1k}. Execute P_1 on each object in C_{1k} including any received from load balancing step. Create OT_1s, i.e. PPI_1s from each selected object.
(3) If $len(\mathcal{P}) = 1$ then **END** algorithm.
(4) Send PPI_is to node pointed to by $OT_i - OID$.Send partial path instances in pages as much as possible. As each PPI_i arrives insert into a hash table on the PID of OT_i-OID. Group PPI_is pointing to same object.
(5) **<Barrier Synchronize>**:At end of synchronization, each node k broadcasts number of objects to load from next class. $i = i + 1$.
(6) **If** P_i is not a UDF skip this step. **Otherwise generate load balancing plan** at each node. Distribute data according to plan.
(7) Probe remaining hash table, loading each page of C_{ik} in turn. Store referenced objects with referencing group of PPI_is.
  (a) Execute P_i on each C_{ik} object referenced or received during load balancing.
(8) if $i = len(\mathcal{P})$ then **END** algorithm. **Otherwise** extend each PPI_{i-1} to PPI_i from each selected C_i object and its group of referencing PPI_{i-1}s and **GOTO STEP 4.**

Fig. 3 Parallel Navigation with Dynamic Load Balancing Algorithm

Locally originating PPI_is and received PPI_is are put into the hash table on the PID of $OT_i - OID$ such that PPI_is pointing to the same page are in the same hash link and those referencing the same C_{i+1} object are grouped together. If there is overflow in building the hash table, overflow PPI_is are written into disk. Note that in analytical and experimental evaluation, sufficient memory to avoid overflow is assumed and justified.

Barrier synchronization is used to allow gathering of statistics about the number of objects referenced by each node from the hash table, to determine need for load balancing, which we explain in the next subsection. Suffice it to say that load balancing is initiated only for cases when P_i is a UDF, an expensive operation.

Load balancing may transfer some PPI_is and their referenced objects into or out of nodes. The remaining hash table is probed, and P_i is executed on each referenced object loaded from disk or received during load balancing. If all classes in the path expression have been processed the algorithm terminates else each selected object of class C_i is use to extend PPI_{i-1} to PPI_i and the algorithm loops to step 4.

```
LoadBalance()
(1)INPUT: H[k], k := 1 to N ;
    /*H[k] is number of Objects to load at node k */
(2)BEGIN at each node in parallel;
(3)  Avg := SUM(H[k] : k := 1 to N) ÷ N;
(4)  HvyLim := AvgLoad * (1 + ε);
(5)  LgtLim := AvgLoad * (1 − ε);
    /*Balanced range defined */
    /* ε is the load balancing factor */
(6)  Create HL = {List of Nodes k : H[k] > HvyLim};
(7)     sort HL in desc. order of load size;
(8)  Create LL = {List of Nodes k : H[k] < LgtLim};
(9)     sort LL in asc. order of loadsize;
(10) If HL or LL is empty END algorithm.
    /* HL or LL empty: no load imbalance */
(11) Set Transfer Load: node HL[1] → node LL[1];
    /*Transfer as many objects as possible from
      HL[1] to LL[1] - bring both nodes to bring
      both nodes near or into balanced range */
(12)    if any of HL[1], LL[1] enter balanced range
(13)       remove node from list ;
(14)       move next node to list top ;
(15) GOTO 11 until either HL or LL is empty;
(16) Nodes not originally in HL END algorithm;
(17) Nodes originally in HL transmit data to nodes
     designated in plan: units of
     (C_i object + group of PPI_i s) in pages ;
(18)END algorithm.
```

Fig. 4 Decentralised Dynamic Load Balancing Algorithm

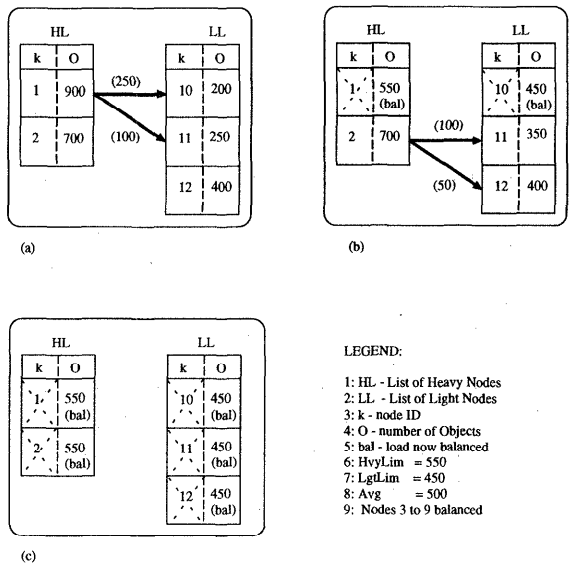


Fig. 5 Data Re-distribution Plan in Load Balancing

4.2 Load Balancing

In this section we describe the load balancing part of the *PNDLB* algorithm i.e. line (6) in Fig. 3. A description of the algorithm is given in **Figure 4**. A typical dynamic load-balancing technique addresses the following issues: load imbalance detection; granularity of work transfer; size of work transfer, which processor to transfer to which processor and actual load transfer. Our load balancing algorithm addresses these issues as follows.

4.2.1 Imbalance Detection and Transfer Granularity

Our proposed load balancing algorithm is based on the premise that the execution of *UDF*s in a qualified path expression is an expensive operation. Therefore if some nodes in the SN system have to access a larger number of objects than others during hash table probing, load imbalance is said to exist if a *UDF* is to be executed for that class of objects. The implicit assumption here is that the execution time of the UDF can be estimated. Estimating the execution time of abitrary code is difficult but we believe that statistics gathered over time can be used to estimate UDF execution time as used in Cario et. al for ORDBMS[3]). We distinguish between low-variant UDFs and high-variant UDFs. Low variant UDFs have execution times that vary within a limited range about an average and our load balancing algorithm can be applied. High-variant UDFs have a wide range of variations and average estimates are inaccurate.

During barrier synchronization each node within the broadcasts the number of $C_{i+1,k}$ objects referenced by the PPI_is in its hash table. In parallel, each node will use this information to detect load imbalance, which is said to exist only if both list HL and list LL defined in Fig. 4 are non-empty. The value of the load balancing factor ϵ is determined experimentally. The *granularity of work transfer* is one UDF execution, i.e., one C_i object and all $PPI_{(i-1)}$s referencing that object.

4.2.2 Transfer Plan Generation

Transfer plan generation answers both the size of work transfer question and the question which processor to transfer to which processor. After detecting load imbalance, each node will, in parallel, generate a work transfer plan. Our algorithm creates two lists, HL sorted in descending order, and LL in ascending order as shown in **Fig. 5**. The transfer plan results in each node in the system having within $(AvgLoad \pm \epsilon)$ C_i objects.

Transfer plan generation is detailed in Fig. 5. In this example $\epsilon = 0.1$ and $AvgLoad = 500$

giving a balanced range of 450-550 objects ($AvgLoad \pm \epsilon$). This is a best-fit decreasing strategy. For example in Fig. 5.a, node 1 sets to transfer 200 objects to node 10 and the latter becomes balanced so it is removed from list LL in Fig. 5b. Node 1 is removed from its list only after it is set to transfer a further 100 objects to node 11 since it becomes balanced then. Node 2 is set to transfers 100 objects to node 11 and 50 objects to node 12. The algorithm stops when the one of the lists is empty.

Note that planning is done in parallel at every node and exactly the same plan is generated. This makes the algorithm decentralised and in a large SN can help to avoid the bottleneck of waiting for one controller node to generate a plan.

4.2.3 Actual Load Transfer

The transfer plan of Fig. 5 is generated in parallel at each node. Nodes not included in the HL list in the plan terminate the algorithm and continue to next step. Nodes in list HL begin actual tranfer as in line 17, Fig. 4 . For example to transfer 50 C_i objects in from node 2 to node 12 in Fig. 5, node 2 will load from disk 50 C_i objects referenced from its hash table and transfer these together with all referencing $PPI_{(i-1)}$s in units of pages. Receiving nodes process data as in the next step on the $PNDLB$ algorithm.

4.3 Target Queries

For a navigation algorithm to be effective in an OODBMS it must be able to navigate abitrarily complex query graphs. We show below that our algorithm can be used to navigate such kinds of query graphs. **Figure 6** shows the type of query graphs that can be solved using the PNDLB algorithm. Shown are 3 types of graphs, namely single path expression graphs, multiple path-expression graphs and cyclic path expression graphs.

We already described our algorithm as processing single path graphs. However to process multiple path-expression graphs, we divide the graph into a number of constituent paths and then process these paths individually. For example for Fig. 6b we would process path $C_1.A_1.A_2.A_3$, and store the results. To process $C_1.A_1.A_2.A_4.A_5.A_6$, we would utilize the data obtained from $C_1.A_1.A_2.A_3$ so that we only use those objects C_1 that are selected here. This reduces the amount of data if the first path is highly selective. The same is done for last

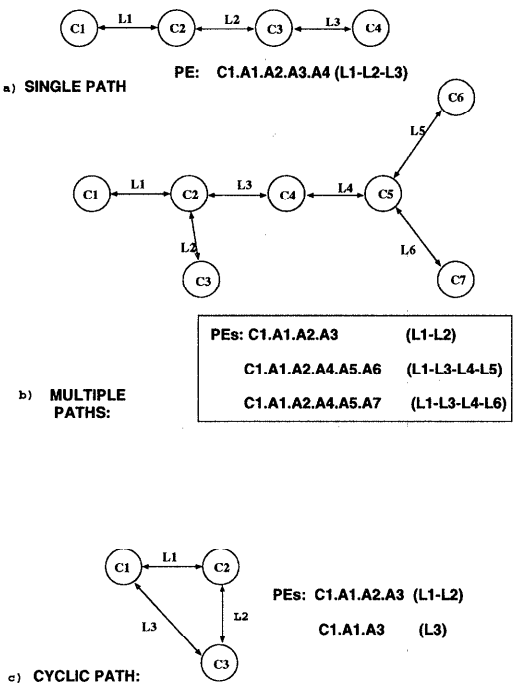


Fig. 6 Types of Query Graphs for Parallel Navigation Algorithm

path $C_1.A_1.A_2.A_4.A_5.A_7$. For the cyclic query in Fig. 6c we follow much the same way.

Note that to process multiple path expression and cyclic path expression graphs, we would need to provide a new entry point to the PNDLB algorithm other than the scan operation but doing so is straightforward. In this way we can locate all the data that meets the requirements of the query graph. Note also that to process this kind of graph efficiently the query optimizer must divide the graph into the most optimum way so that the paths with least selectivity are processed first.

5. Analytical Evaluation

In this section we discuss an analytical evaluation of the navigation algorithm. The model is validated by comparing it to results obtained from simulation in the next section.

5.1 Analysis Parameters

Table 1 lists the parameters that used in the analysis. We focus on four important aspects of PNDLB algorithm namely, disk page I/O; communication time for inter-node traversals; CPU time for hash table maintenance and method execution time for UDFs. We define $fan\_out(C_i, C_j)$ and $fan\_in(C_i, C_j)$ to model the extent of the connection between two classes

Table 1 Parameter Definition List

Name	Description
Pg	size of a disk page
N	Total number of nodes
s	size of an object
C_{ik}	The portion of class C_i stored at node k
$\|\|C_{ik}\|\|$	Total number of class C_{ik} objects at node k
$\|C_{ik}\|$	Total number of class C_{ik} pages at node k
sel_{P_i}	The selectivity of predicate P_i
$locality_{C_{ijk}}$	The percentage of pointers from class C_{ik} objects that point to C_{jk} objects at node k
$fan\_out(C_i, C_j)$	The average number of pointers from a class C_i object to a class C_j objects.
$fan\_in(C_i, C_j)$	The average number of pointer from class C_i objects into one class C_j object.
y_{ijkl}	The average fraction of the total number of external pointers that point from node k to node l for class C_i to class C_j
ϵ	The load balancing factor

participating in a path expression. These parameters are particularly important in a parallel OODBMS since they indicate the degree to which parallelism can be applied (cf: set-valued attribute) and also the level of data sharing during access. The parameter y_{ijkl} is an indicator of the amount of message passing between 2 nodes l and k. Closely related to this parameter is the locality of reference which indicates $locality_{C_{ijk}}$ the fraction of pointers from a class at node k that point to the same node.

We make the assumption that the time to execute a UDF is on average t_{UDF} for each class. As mentioned ealier, this assumption is valid for low variant UDFs but needs modification for high variant UDFs. However curve fitting techniques are used to make estimates for t_{UDF} based on accumulated statistics in some researches[11]. In this paper we use the simpler method of averaging.

The analysis here and indeed the algorithm assumes that each node has enough memory to maintain any data its had read in main memory together with all hash tables. We therefore assume that there is no page I/O resulting from memory overflow. This assumption is not simplistic since recent typical shared-nothing machines have 64MB of main memory per node. An example is the PC cluster in Kitsuregawa[16], where each PC in the cluster has 64MB of main memory. It is also shown in Dewitt et. al.[6] that an 8 node machine has sufficient memory to hold all OODBMS data.

5.2 Cost Equations

Let us consider the instantiation of the following path expression:

$$\mathcal{P} = C_1(P_1).A_1.(P_2)A_2.\ldots.(P_n)A_n$$

.

5.2.1 Scanning Root Class C_1

Node k scans class C_{1k} and the number of disk pages read is given as:

$$IO_{C_{1k}} = \left\lceil \frac{||C_{1k}|| \cdot s}{Pg} \right\rceil \quad (1)$$

Each object read has predicate P_1 (i.e. a UDF) executed on it and the number of such operations is:

$$UDF_{C_{1k}} = ||C_{1k}|| \quad (2)$$

The number of PPI_1s generated at node k is:

$$PPI_{ik} = UDF_{C_{1k}} \cdot sel_{P_i} \cdot fan\_out(C_i, C_{i+1}) \quad \text{where} \quad i = 1. \quad (3)$$

5.2.2 Send and Receive Operations

The number of PPI_is sent out from node k is calculated as follows:

$$PPI\_Snd_{C_{ik}} = PPI_{ik} \cdot (1 - locality_{C_{i,i+1,k}}) \quad (4)$$

The algorithm arranges these in pages and sends them a page at time. The number of send operations is therefore:

$$Snd_{C_{ik}} = \left\lceil \frac{PPI\_Snd_{C_{ik}} \cdot sizeof(PPI_i)}{Pg} \right\rceil \quad (5)$$

where the $sizeof()$ operator returns the size of PPI_i. Note that here the number of received PPI_is is:

$$PPI\_Recv_{C_{ik}} = \sum_{l=1}^{N, l \neq k} PPI_{ik} \cdot (1 - locality_{C_{i,i+1,l}}) \cdot y_{ijlk} \quad (6)$$

Since these are received in pages the number of receive operations is:

$$Recv_{C_{ik}} = \left\lceil \frac{PPI\_Recv_{C_{ik}} \cdot sizeof(PPI_i)}{Pg} \right\rceil \quad (7)$$

5.2.3 Hashing Object-Tuples

PPI_is generated from class C_{ik} and those received from external nodes are hashed on their PID of OT_i-OID. The total number of such hash operations is:

$$Hash_{C_{ik}} = PPI_{ik} \cdot locality_{C_{i,i+1,k}} + PPI\_Recv_{C_{ik}} \quad (8)$$

5.3 Probing the Hash Table

To find the time spent in probing the hash ta-

ble we have to estimate the number of disk I/O operations executed. This is done by the Yao function[22]. Given n objects evenly distributed among m pages, Yao estimates the number of pages to be read to access x randomly selected objects as follows:

$$Yao(n,m,x) = \left\lceil m * (1 - \prod_{i=1}^{x} \frac{n - \frac{n}{m} - i + 1}{n - i + 1}) \right\rceil$$

If X_{ik} is the number of objects to access at node k for class C_i then the number of I/O operations is:

$$Probe\_IO_{C_{ik}} = Yao(||C_{ik}||, |C_{ik}|, X_{ik}) \quad (9)$$

where:

$$X_{ik} = \left\lceil \frac{Hash_{C_{i-1,k}}}{fan\_in(C_{i-1}, C_i)} \right\rceil \quad (10)$$

The number of probe operations is given by

$$Probe\_CPU_{C_{ik}} = Hash_{C_{i-1,k}} \quad (11)$$

Equation 11 means that the number of hash operations for PPIs from class C_{i-1} determine the number of probe operations for class C_i.

5.3.1 Load Balancing

To create an analytic model for the load balancing algorithm we used the approach suggested in Walton et. al.[21]. Here data skew for a parallel relational system is modeled as a scalar quantity Q. This represents the case where one node in the SN system has Q times more tuples than any of the other nodes, which each have a number of tuples equal to some value X. In our case Q is calculated as: $\frac{X_{ik}}{X_{il}}$. where node k is the heavily loaded node and node l is any other node.

The time taken to do load balancing is determined by the time taken to transfer the extra load from node k. If we assume that node 1 is heavily loaded then if X_{i2} is the number of objects to access at a lightly loaded node 2 (equal to all other light nodes) then the average load in the system is given as:

$$Avg = \frac{Q \cdot X_{i2} + (N-1) \cdot X_{i2}}{N} \quad (12)$$

The heavily loaded node must transfer load such that the number of objects remaining in its hash table is within the threshold as defined in Fig. 4. In this case the $HvyLim$ is given as follows:

$$HvyLim = (1 + \epsilon) \cdot Avg\_Load$$

where ϵ is the load balancing factor. The total load transferred is calculated as:

$$T\_Load = (Q \cdot X_{i12} - (1 + \epsilon) \cdot Avg) \quad (13)$$

The unit size of data transferred, T_data is given as follows:

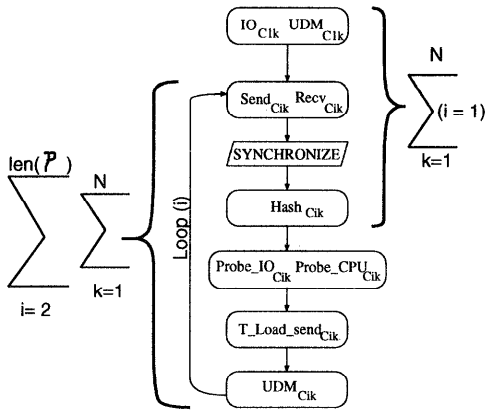


Fig. 7 Calculating Response Time

$$T\_data = sizeof(PPI_{i-1}) \cdot fan\_in(C_{i-1}, C_i) + s \quad (14)$$

The number of send operations in pages is given as follows:

$$T\_Load\_snd = \left\lceil \frac{T\_Load \cdot T\_data}{Pg} \right\rceil \quad (15)$$

5.3.2 UDF Execution

After load balancing, each node then executes the UDF for objects from class C_i. The number of UDFs executed here are calculated as follows:

$$UDF_{C_{ik}} = \begin{cases} X_{ik} & \text{no load balancing} \\ Q \cdot X_{i2} - T\_Load & \text{load balancing} \end{cases} \quad (16)$$

In the load balancing case the assumption is that the heavily loaded node, node 1 takes the most time here and it has Q times more objects than all other nodes for example node 2. We therefore only calculate the time for the heavily loaded node 1. Therefore in equation 17 the value of k is 1 when load balancing is used.

The total number of PPI_is generated at this point is:

$$PPI_{ik} = UDF_{C_{ik}} \cdot fan\_in(C_{i-1}, C_i) \cdot sel_{P_i} \cdot fan\_out(C_i, C_{i+1}) \quad (17)$$

For each object there are $fan\_in(C_{i-1}, C_i)$ PPI_{i-1}s pointing to it, and each object has $fan\_out(C_i, C_{i+1})$ pointers from which to create new PPIs given in equation 18.

5.3.3 Total Response Time

We calculate the time to reach the barrier synchronization point for each state of the path expression. The time for each of the stages of the algorithm, is determined by the slowest node. **Figure 7** show a diagram on how

the total response time is calculated. The response time calculated in the equation 18 takes the time at the slowest node.

To calculate the total time taken to traverse the whole path expression input into the algorithm, we need to calculate the maximum time taken to reach the synchronization point:

(1) From the beginning of the algorithm until the first time the synchronization point is reached;

(2) From the time the synchonisation point is crossed until the next time its crossed looping until the whole path expression is traversed.

$$
\begin{aligned}
&TotalTime = \qquad (18)\\
&MAX\{(IO_{C_{1k}} + UDF_{C_{1k}} + Snd_{C_{1k}}\\
&+Recv_{C_{1k}} + Hash_{C_{1k}}) : \{k = 1, N\}\}\\
&+ \sum_{i=2}^{i=len(\mathcal{P})} MAX\{(Probe\_IO_{C_{ik}}\\
&+Probe\_CPU_{C_{ik}} + T\_Load\_snd\\
&\quad +UDF_{C_{ik}} + Snd_{C_{ik}} + Recv_{C_{ik}}\\
&+Hash_{C_{ik}}) : k = 1, N\}\\
&\text{where:}\\
&Snd_{C_{ik}} = Recv_{C_{ik}} = Hash_{C_{ik}} = 0\\
&\quad \text{if i} = len(\mathcal{P}),\\
&T\_Load\_send = 0 \quad \text{if no load balancing}
\end{aligned}
$$

It is clear then that the maximum time for each of these 2 phases is determine by the slowest node. So to find the response time we calculate the time for the slowest node. For point 1, the time is calculated by the first term in equation 18. The summation term gives the time for point 2. The sum is given from 2 to $len(\mathcal{P})$ to indicate that the algorithm loops this section that many times. When there is no load balancing we do not need to include the load balancing message overhead T_Load_send note also that load balancing reduces the number of UDFs according to equation 17 at the heavily loaded node, which is still the slowest node because it still has the most UDFs to execute.

The total time for execution is therefore a summation of the all the above terms as shown in equation 18

5.4 Analytical Evaluation Results

We collected some results from the analytical model described above. **Table 2** shows the constants used in this evaluation. The time is given in time units. A size of 200,000 objects is chosen as it represents a reasonable size for the database and is derived from the OO7 benchmark[2]. A fan out of 3 and a fan in of 1 are derived from the OO1 benchmark[4]. For the

Table 2 Time Constants List

Name	Value	Name	Value
DBsize(objects)	200,000	t_{disk}	200
Pg	8kbytes	t_{send}	70
object size s	256bytes	t_{recv}	100
N	2-32	t_{UDF}	40-70
$fan\_out(C_i, C_j)$	3	t_{hash}	3
$fan\_in(C_i, C_j)$	1	t_{probe}	3
$locality_{C_{ik}}$	0.5	ϵ	0.1
y_{ijkl}	$\frac{1-locality_{C_{ik}}}{N}$	$len(\mathcal{P})$	7

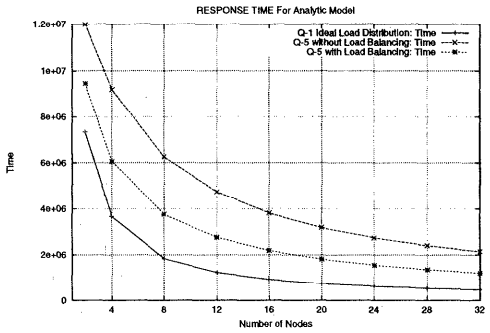


Fig. 8 Response Time generated from Analytical Model

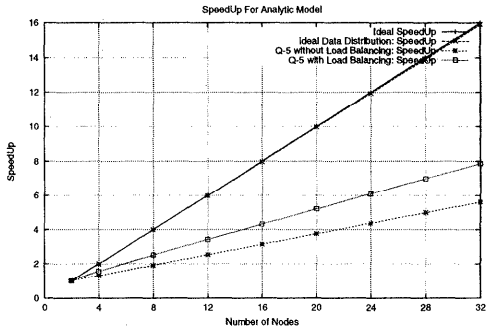


Fig. 9 Speedup from Analytical Model

UDF time we chose a range of 40-70 for the low variance case and this is equal for all classes. The value of y_{ijkl} is adjusted when load imbalance exists such that each lightly loaded node has Q times more pointers to the heavily loaded node than to any other node.

Figure 8 shows the response time for the analytic model described in the previous subsection. We include a plot for the case where data is uniformly distributed at all levels of a path expression. In the graph it can be seen that ideal distribution achieves almost linear speedup. When data is skewed with a Q value of 5 speedup decreases to a value of about 5.7

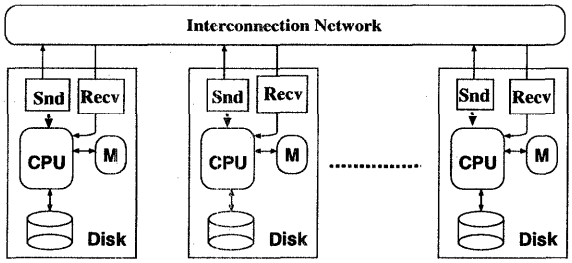


Fig. 10 The Simulated Shared-Nothing model for Navigation Algorithm

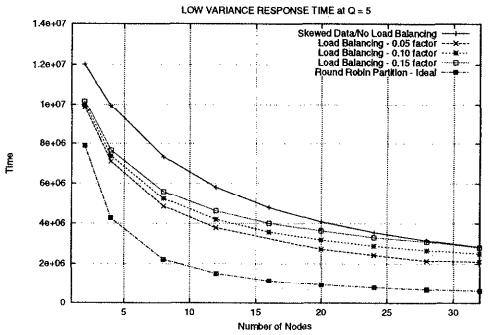


Fig. 11 Response Time for Low Variance at Q=5

from 2 to 32 processors. This shows that this algorithm is sensitive to load-imbalance. Applying the load balancing algorithm improves the speedup to 7.9 over 32 nodes as shown in **Fig. 9**. If the response time at 2 nodes is RT_2 and the response time for N nodes is RT_N then speedup here is defined as: $\frac{RT_2}{RT_N}$. However note the improvement in response time at 8 nodes is over 33% if we compare the graphs when load balancing is not used and the graph when load balancing is used.

The analytic model indicates that instantiating a path expression in a parallel OODBMS can benefit from parallel processing but good data distribution is necessary to obtain maximum parallel processing benefit. In the next section we will discuss these analytical results further in comparison simulation results.

6. Simulation Evaluation

6.1 Simulation Model

We developed a simulation model to validate our PNDLB algorithm. **Figure 10** depicts the simulation model. Each node in the system communicates with other nodes via a high speed network. Each node has enough send and receive buffers to hold whatever messages going to and from the network. In the case of receive, all messages sent to a node are held in the buffer until the CPU processes them. We also assume a node memory large enough that any all objects and tables at a particular node are held in memory without the need to resort to overflow processing. The parameters in the simulation model are similar to those in Table 2. The simulation program was written in C and experiments performed on a Sparc Center Server with 2GB of memory.

6.2 Low Variance Results

The graphs in **Fig. 11** show the response time when the load balancing factor is varied from 0.5-0.15. This graph list experiments for the case for low variance UDFs using a range 40-70 time units selected at random with average 55. We note that as predicted in the analytical model the algorithm is sensitive to load imbalance. Note also that the results depicted in this graph agree very well with the data in Fig. 8. When data is uniformly distributed across all nodes using round-robin partition the speedup is about 13.1 from 2-32 nodes. However a data skew of Q-factor 5 results in a speedup to 4 on 2-32 nodes as shown in **Fig. 14**. However note that the worst increase is in the response time. For 8 nodes response time increases about 3.5 times. At 28 nodes the increase is about 2.5 times. This emphasizes the need for load balancing in the navigation of links in a parallel OODBMS since the ideal partition realised by round robin partition may only be practical at object creation time but arbitrary access pattern may result in load imbalance. Having mentioned that we would like to note that even without load balancing Fig. 11 shows that parallel processing can decrease the response time for navigation in OODBMS.

We note that for the low variance case load balancing at ϵ 0.05, 0.1 and 0.15 improves response time significantly. A maximum decrease of 34% in response time is achieved at 8 nodes for $\epsilon = 0.15$. However if we look at the case for $t_{UDF} = 5$ shown in **Fig. 12**, we see that load balancing actually increases the response time since the t_{UDF} and load balancing communication time introduces overhead that is not compensated for by the executing low time UDFs. We can therefore say that load balancing bene-

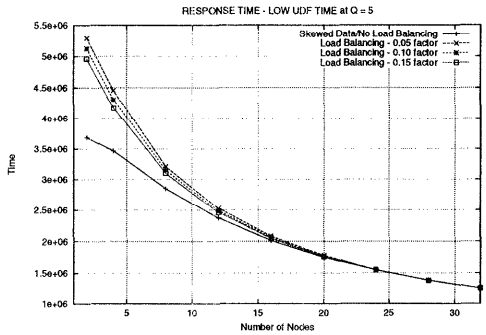


Fig. 12 Response time at $t_{UDF} = 5$

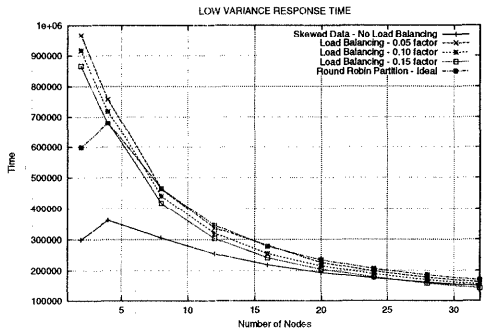


Fig. 13 Communication Time for Low Variance

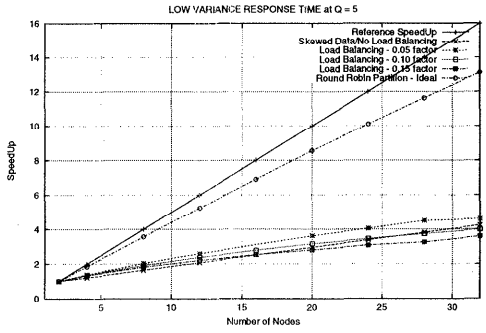


Fig. 14 Speedup For Low Variance

fits the case when t_{UDF} is high.

Figure 13 shows the communication time for the PNDLB algorithm in the low variance case. In general we can infer that communication time varies from 10% - 4 % of response time for 2 and 32 nodes respectively. The dominant time is obviously UDF execution and disk access. Load balancing increases communication time but this is an increase in a factor whose influence is limited.

One of the purposes of the experiment depicted in Fig. 11 - Fig. 14 was to determine the best value for the load balancing factor ϵ. We varied ϵ from 0.5-0.15 an noted the response time. It shows from these three figures that the bigger the load transferred the better the improvement in response time. However if we compare the speedup for 0.05 and the speed up for 0.15 we note that the latter's is lower than that without load balancing at a higher number of nodes. However 0.05 results in a response time that is converges faster towards the case without load balancing as the number of processors increases. These two factors suggest that a load balancing factor of 0.10 is a good trade-off between an improvement in response time and speedup.

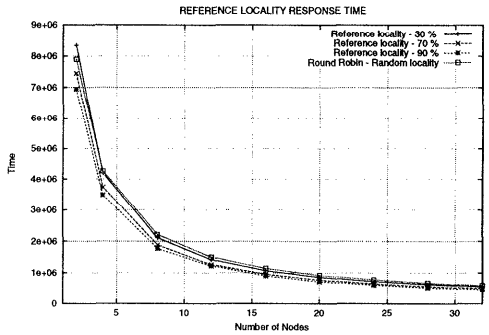


Fig. 15 Response Time for Locality of Reference

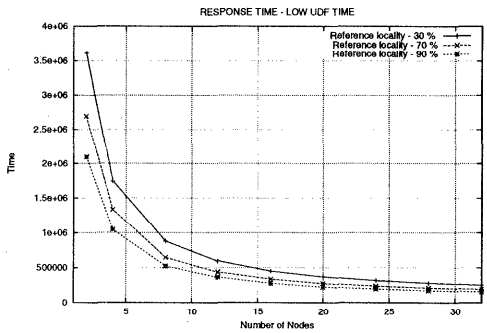


Fig. 16 Response Time for $t_{UDF} = 5$ with Locality Variation

6.3 Variation of Locality of Reference

One of the aims of research in parallel OODBMS is to reduce the number of inter-node references. We experimented with varying levels of locality of reference, i.e. the percentage of OIDs pointing to external nodes for a class C_{ik} at node k. Locality of reference was varied from 30% to 90%. **Figure 15** shows that

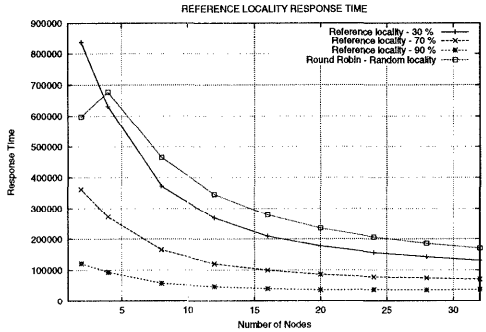


Fig. 17 Communication Time for Locality Of Reference

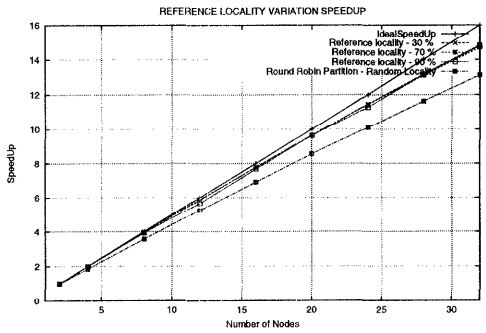


Fig. 18 Speed Up For Locality of Reference

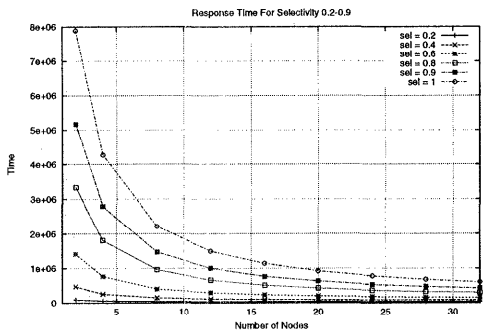


Fig. 19 Response Time for selectivity 0.2-1

the response time does not vary very much with changes in locality of reference in the low variance UDF case. This is not the case for the case with $t_{UDF} = 5$ as shown in **Fig. 16**. Here response time increases more significantly as locality of reference decreases. Communication time increases as locality of reference decreases as shown in **Fig. 17**. This time is more influential when t_{UDF} is low. **Figure 18** shows that locality of reference does not affect speedup.

6.4 Effect of Selectivity

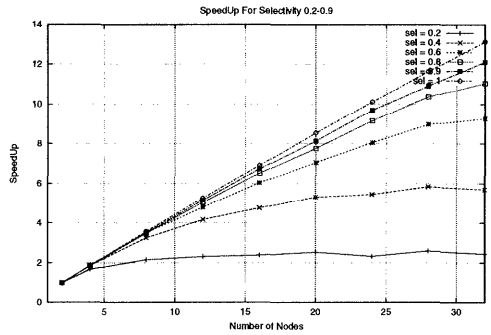


Fig. 20 SpeedUp for selectivity 0.2-1

Experiments with the selectivity of the UDF show that a decrease in selectivity results in a decrease in the response time. A surprising result is that the lower the selectivity the lower the speedup. We may infer from this result that a low selectivity UDF will reduce the benefit of parallel processing since the number of methods executed in subsequent levels of the path expression is correspondingly reduced. Parallel processing will be of greatest benefit to methods with high selectivity.

7. Conclusion and Further Work

We have presented a parallel navigation algorithm, PNDLB. The algorithm exploits the existence of set-valued attributes and declustering to navigate path expressions in parallel and includes a dynamic load balancing algorithm. We argue that UDF execution is important in parallel OODBMS and that load balancing is needed to ensure that UDF load is balanced across in an SN system. We estimate UDF time using the average value of the execution time for previous executions of the corresponding UDF assuming that the average is a reliable estimate of future execution times. This assumption holds for low variance UDFs.

We have presented analytical evaluation results and simulation evaluation results to validate the efficiency of the PNDLB algorithm. Simulation also shows that dynamic load balancing is beneficial for situations when the UDF execution time is high, but is not when UDF time is low. A speedup of 13.1 obtained when there is no load imbalance for 2-32 nodes. When there is load imbalance a decrease of up to 34% is obtained for response time when load balancing is used. Simulation also shows that the best value of the load balancing factor is 0.1.

We intend to do an actual implementation of the PNDLB algorithm on a real machine. We would also like to compare the performance of our algorithm with the algorithm in Chen et. al.[5].

Acknowledgments This research is supported in part by the Japanese Ministry of Education, Science, Sports and Culture under Grant-in-Aid for Scientific Research (B) and (C).

References

1) Bancilhon F. et. al.: *Building an Object-Oriented Database System: The Story of 0-2*, Morgan Kauffman, (1992).
2) Carey M.J., DeWitt D.J. and Naughton J.F.: The 007 Benchmark, *Univ. Wisconsin Tech. Report*, (1994).
3) Cario F.,and OConell W.,:Plan-Per-Tuple Optimization - Parallel Execution of Expensive User-Defined Functions, *Proc. of the VLDB Conf.*, pp. 690-695 (1998)
4) Cattell R.G.G., Skeen J.: Object Operations Benchmark,*ACM TODS*, Vol 17, No. 1, (1992).
5) Chen Y.H. and Su S.: Identification and Elimination-based Parallel Query Processing Techniques for Object-Oriented Databases, *Journal of Parallel and Distributed Computing*, Vol. 18, No. 2, (1995).
6) DeWitt D.J., Naughton J.F., Shafer J.C. and Venkataraman S.: ParSets for Parallelizing OODBMS Traversals: Implementation and Performance, *Univ. Wisconsin Tech Report* (1995).
7) DeWitt D.J., et. al: Practical Skew Handling Algorithms for Parallel Joins, *Proc Conf. on Very Large Databases*, (1992).
8) Gardarin G., Gruser, J.R. and Zhao-Hui, T.: Cost-Based Selection of Path Expression Processing Algorithms in Object-Oriented Databases, *Proc. of the 22nd VLDB Conference*, (1996).
9) Ghandeharizadeh S. Choi V., Ker, C. and Lin, K. : Design and Implementation of the Omega object-based system, *Proc. of the Fourth Australian Database Conference*, (1993).
10) Ghandeharizadeh S., Wihite D., Lin K. and Zhao X.: Object Placement in Parallel Object Oriented Database Systems, *Proc 10th International Conference on Data Engineering*, pp 253-262 (1994).
11) Hellerstein J.M.: Optimization Techniques for Queries with Expensive Methods, *ACM Trans. on Database Systems* (to appear).
12) Kempter A. and Moerkotte G.: *Oject Oriented Database Management*, Prentice Hall, pp609-620, (1994).
13) Kien H.A. and Su J.X.W: Dynamic Load Balancing in Very Large Shared-Nothing Hypercube Database Computers, *IEEE Trans. Computers*, Vol. 42, No. 12, (1993).
14) Kim K.C Parallelism in Object-Oriented Query Processing, *Proc. 6th Int'l Conf on Data Engineering*,pp. 209-217, (1990)
15) Kitsuregawa M., Tanaka H. and Motooka T.: Application of Hash to Data Base and its Architecture *New Generation Computing No 1*, pp62-74 (1983).
16) Kitsuregawa M., Tamura T. and Oguchi M.: Parallel Database Processing/Data Mining on Large Scale Connected PC Clusters, *Proc. of Parallel and Distributed Systems Euro-PDS' 97*, pp313-320, (1997).
17) Lieuwen, D.F., DeWitt, D.J., Mehta M. Parallel Pointer-based Join Techniques for Object-oriented Databases, *Univ. Wisconsin Tech Report*, (1993)
18) Rahm E. and Marek R.: Analysis of Dynamic Load Balancing Strategies for Parallel Shared Nothing Database Systems,*Proc of the 19th VLDB Conf.*,pp 182-193 (1993).
19) Shekhar S., et. al.: Declustering and Load-Balancing Methods for Parallelizing Geographic Information Systems, *IEEE Trans. on Knowlwdge and Data Engineering*, Vol. 10, No. 4, (1998).
20) Tout W.R. and Pramanik S.: Distributed Load balancing for Parallel Main Memory Hash Join, *IEEE Trans. on Parallel and Distributed Systems*, Vol. 6, No. 8, (1995)
21) Walton C.B., Dale A.G. and Jenevein R.M.: A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins, *Proc. of the Int. Conf. on VLDB*, pp537-548 (1991).
22) Yao S.B.: Approximating Block Accesses in Database Organisations, *Comm. ACM*, Vol. 20, pp260-261, (1977).

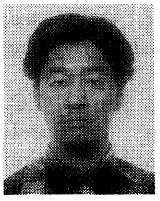
(Received December 20, 1998)
(Accepted February 1, 1999)

(Editor in Charge: *Hiroshi Ishikawa*)

Lawrence Mutenda was born in 1966. He received his BSc(Electrical Engineering) degree from the University of Zimbabwe in 1989. He worked as a communications engineer at the Zimbabwe Telecommucations Corporation from 1990 to 1991. In 1992 he enrolled at University of the Witwatersrand in South Africa where he received the Graduate Diploma in Engineering (Computer Engineering). In 1993 he enrolled at Utsunomiya University and received the M.E. degree in 1995. He studied for the PhD degree in Production and Information Systems at Utsunomiya University from 1995. He left Utsunomiya University in March 1998, to work as a foreign researcher at the Institute of Industrial Science, University of Tokyo, where he is presently. His research interests are parallel OODBMSs and parallel spatial databases.

Takanobu Baba was born in 1947. He received his B.E., M.Eng., and Dr.Eng. degrees from Kyoto University in 1970, 1972, and 1978, respectively. From 1975 to 1979 he was a Research Associate and then a Lecturer at the University of Electro-Communications. Since 1979, he has been with the Department of Information Science, Utsunomiya University, where he is now a Professor. In 1982 he spent one year leave as a Visiting Professor at University of Maryland. His interests include computer architecture, and parallel processing. He received IPSJ Best Author award in 1992. He is author of the books, "Microprogramming"(Shoukoudoh, 1985), "Microprogrammable Parallel Computer"(The MIT Press, 1987), "Computer Architecture"(Ohmsha, 1994). Dr. Baba is a member of IPSJ, IEICE and IEEE Computer Society.

Tsutomu Yoshinaga was born in 1963. He received his M.E. and D.E. degrees from Utsunomiya University in 1988 and 1997 respectively. Since 1988, he has been in Department of Information Science, Utsunomiya University. From 1997 to 1998, he was a visiting researcher at Electrotechnical Laboratory. His current research interests are architecture of parallel computers and reconfigurable computing systems. He is a member of IPSJ and IEICE.



Kanemitsu Ootsu was born in 1969. he graduated from the department of Information Science, University of Tokyo in 1993. He obtained a Master of Engineering degree from the same university in 1995. He was a PhD student at the University of Tokyo from 1995, leaving in 1997 to join the University of Utsunomiya as a research associate. His main interests are designing efficient computer systems, especially microprocessor architecture.