DISTRIBUTED LINEAR HASHING AND PARALLEL PROJECTION IN MAIN MEMORY DATABASES

C. Severance, S. Pramanik & P. Wolberg

Computer Science Department, Michigan State University
East Lansing, Michigan 48824

ABSTRACT

This paper extends the concepts of the distributed linear hashed main memory file system with the objective of supporting higher level parallel database operations. The basic distributed linear hashing technique provides a high speed hash based dynamic file system on a NUMA architecture multi-processor system. Distributed linear hashing has been extended to include the ability to perform high speed parallel scans of the hashed file. The fast scan feature provides load balancing to compensate for uneven distributions of records and uneven processing speed among different processors. These extensions are used to implement a parallel projection capability. The performance of distributed linear hashing and parallel projection is investigated.

1. INTRODUCTION

The availability of multi-processor computers with large main memories has made main memory database applications feasible. With a large number of processing nodes, these systems can have a large amount of memory at relatively low cost. The aggregate data transfer rate between the memories and the processing nodes is also very large for these systems.

While the aggregate performance and memory size of these systems is very high, the central control structures used in traditional database systems will prevent the system from achieving high levels of performance. Distributed linear hashing provides a technique for implementing parallel main memory database systems which minimize the adverse effect of these architectural constraints, while exploiting the NUMA architecture to enhance performance of key based access to individual records stored in a hash based file system.

To provide high speed access for operations which must access all records in a database, fast scan exploits the locality of data by using primarily local memory references.

Relational projection can be viewed as composed of two sub-tasks, scanning the input relation and creating the result relation. In the process of creating the output relation any duplicate records created as a result of the projection must be removed. The duplicate elimination phase of projection is usually the time consuming part of the projection operation.

Distributed linear hashing is implemented on the BBN Butterfly. Fast scan has been added to support database wide operations. We have implemented parallel projection with duplicate removal as an example application using this file system. The performance of the hash file system and parallel projection is shown.

1.1. Previous Work

Several hashing methods [Ghos 86, Wied87] for dynamic files have been proposed since the mid 1970's, including extendible hashing[FNPS79], linear hashing [Lars80], and dynamic hashing [Lars78]. Complete reorganization of the data file is avoided in these techniques by allowing the directories to adjust to the records of the overflowing buckets. These hashing methods reduce the search time by minimizing the number of disk accesses.

Linear hashing as a search structure for databases was developed by Litwin [Litw 80]. A solution for concurrent linear hashing was proposed by C. S. Ellis [Elli 87]. Concurrent linear hashing adds a locking protocol and extends the data structures to enhance concurrency of

Proceedings of the 16th VLDB Conference
Brisbane, Australia 1990

the entire linear hashed file. Like Litwin's design, concurrent linear hashing of [Elli 87] was intended for a disk environment.

Hash based accesses to main memory databases have also been analyzed in [GLV 84]. Parallel processing in main memory databases has also been investigated in [PrKi 88a, PrKi88b]. Parallel main memory data bases are also explored in [SePr 90] and [Sev 90].

Previous efforts on projection in parallel or main memory environments include work in parallel algorithms for relational operations [BBJW 83] and main memory databases [LeCa 86] and parallel projection of main memory databases [RoJa 87]. Parallel projection in main memory databases was also explored in [Wolb 89].

1.2. Linear Hashing

Before we describe the motivation for our approach to design parallel main memory databases, we define two important shared variables used in linear hashing called P and M. These variables have significant impact on the performance of concurrent accesses. The variable M is the number of buckets and the variable P is the next bucket to be split or merged. M and P are used in computing the proper bucket as follows:

```
bucket_number := key mod M;
if bucket_number < P then
    bucket_number := key mod (M*2)
```

These two variables are also used in database reorganization. When a bucket in the database overflows or the database exceeds a preset load factor, the bucket pointed to by P is split and P is advanced by one. Once P reaches M-1, instead of advancing P, it is set to zero and M is doubled. As records are deleted and buckets underflow, P is decremented by one and the Pth. bucket is merged with the P+Mth. bucket. Once P reaches zero as a result of repeated merge operations, M is halved and P is set to one less than the new value of M. The database is organized in a linear fashion by only splitting or merging the Pth. bucket.

1.3. Motivation

The work by Ellis and Litwin is suitable for a disk based system but there are a number of problems with this approach when it is used for searching main memory databases on a multi-processor NUMA system. The first problem is the cost of accessing central variables such as M and P, and the central locks associated with these variables. In a multi-processor NUMA environment, access to a central data structure may cause a hot spot problem resulting in a serious performance degradation with increasing degree of parallelism. We present a distributed linear hashing scheme where access to costly centralized variables and locks are reduced by using distributed variables and locks. The cost of maintaining consistent copies is significantly reduced by using retry logic which is based on the fact that these variables are updated very seldom.

Database reorganization is single threaded in previous implementations of linear hashing. This is a problem when a large number of parallel nodes are able to insert records at a much faster rate than the database can be reorganized. Experimental results have shown that even if a processor continuously splits buckets sequentially, the database can never be re-organized fast enough to keep the load factor within a reasonable bound. We present multi-threaded reorganization technique to solve these problems. Here, multiple splits and merges are performed concurrently by the processing nodes.

Distributed linear hashing with retry logic and multi-threaded reorganization provides a high performance hash based main memory database system with a very high rate of continuous inserts and deletes. This type of database system is useful for implementing temporary files for main memory databases and relational operators such as projection and join for main memory databases. In this paper we give details of distributed linear hashing scheme and its performance analysis.

The rest of the paper is organized as follows. Section 2 describes the important features of distributed linear hashing scheme. Section 3 describes an implementation of distributed linear hashing, on the BBN Butterfly, called the KDL_RAMFILE system. Section 4 presents performance analysis of the KDL system. Concluding remarks are given in section 5.

2. DISTRIBUTED LINEAR HASHING

In linear hashing the records are distributed into buckets which are normally stored on disk. In distributed linear hashing, the buckets are stored in main memory. Fast accesses to a record within a bucket is accomplished through a hash directory. Two computations are used to locate a record. Distributed linear hashing is used to locate the bucket and an additional computation is performed to find record chain within the bucket. One simple approach to this two dimensional address mapping is

to partition the hash address bits into two. One part locates the directory and the other part finds the record chain. A bucket consists of a hash directory and the records pointed by the hash directory. Note that the records within a bucket can be placed in any memory module by linking them through pointers. An index is used to point to the bucket directories. The index expands or contracts as needed by linear hashing. Figure 2.1 shows the access structure of a main memory distributed linear hashed file. The index is cached in each processor. Each entry in the directory points to the head of a record chain. Collisions in a particular entry in the directory are resolved by adding the record to the chain and tracking collisions to determine the average non-empty chain lengths. A directory is said to be overflowing or underflowing when the average chain length for that directory exceeds or falls below some predetermined value. Directories are split or merged in linear order as soon as some directory overflows or underflows. Splitting is done by rehashing the records using hash function (key mod 2*M) as in standard linear hashing.

In linear hashing bucket address computation requires M and P for every access to a record. Accessing these central variables for every record access will cause a hot spot. To avoid this problem, bucket address computation in distributed linear hashing does not use central copies of M and P. Instead local copies of M and P, called Local_M and Local_P, are used in each processor. The techniques for maintaining distributed copies of these variables at a significantly lower cost than the cost of accessing central copies of these variables are described in the following sections. Central copies of M and P, called Global_M and Global_P, are only used for database reorganization. The Local_M and Local_P used in the hash computations may at times be out of date causing incorrect bucket address computation. To solve the problem of incorrect bucket address computation, retry logic is used. The details of retry logic is described in the next section.

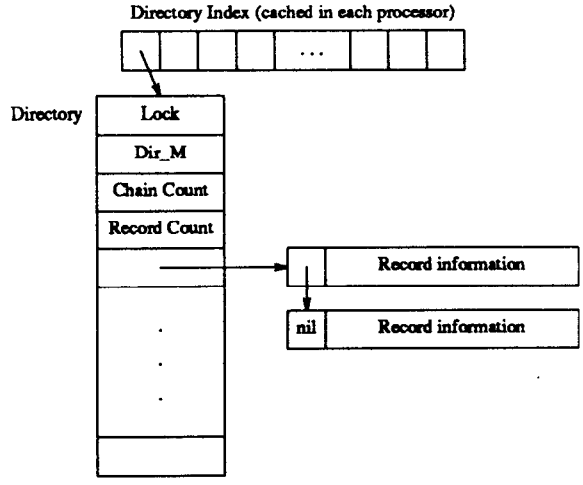

Figure 2.1: Access Structure for Distributed Linear Hashing

2.1. Retry Logic Using Local_M and Local_P

Besides maintaining Local_M and Local_P in each node, a directory contains a value for M. This M, called Dir_M, is the value which was used in the hash computation to place the records into the directory during the most recent split or merge. The value of Dir_M is used to determine if a particular record belongs to this directory regardless of the value of Local_M.

To find a record, the directory number is computed first using the record key value and Local_M stored at that node. The directory indicated by the computation is locked. The directory number is re-computed using the record key value and the value for Dir_M stored in the directory. If this computes to the same directory as the one currently being locked, the proper directory has been found, otherwise the process is repeated by locking the new directory.

The performance enhancement by retry logic over that using central variables is described in section 4. Figure 2.2 shows the algorithm to find the correct directory using retry logic.

```
dir_number := hash(key, Local_M, Local_P);
dir_pointer := dir_lock(dir_number);
new_dir_number := key mod dir_pointer.Dir_M

while (new_dir_number <> dir_number) do begin
  dir_unlock(dir_number);
  dir_number = new_dir_number;
  dir_pointer := dir_lock(dir_number);
  new_dir_number := key mod dir_pointer.Dir_M
```

```
end
```

Figure 2.2: Retry Logic for Locating the Bucket for a Given Key

The following section describes techniques for maintaining the local copies of P and M in each processor.

2.2. Maintaining Copies of Local_P and Local_M

Local_M in each processor is maintained using a central array of pointers to each copy. To double or halve Global_M, a processor must have a lock on Global_P. When the Global_M is updated, the processor updating Global_M must also update all of the copies of Local_M in all of the processors. This technique for updating the values for Local_M uses a very small amount of memory bandwidth compared to the bandwidth requirements for a shared value for M. The values are only updated when the value for M is doubled or halved as a result of database reorganization. In our experiment of a typical run of inserting 500,000 records, Global_M is modified in one out of 50,000 database insertions. During the period a processor updates each copy of Local_M incorrect hash computations may occur because of incorrect values of Local_M. The retry logic described in the previous section will recover from these incorrect computations.

Updating all Local_Ps every time Global_P changes is not efficient because Global_P changes quite frequently. So each local processor updates local_P based on the values for Dir_M in each directory which the processor has to access during every normal database operations. This technique keeps the Local_P values close to the value for Global_P. Every time a key is mapped into a directory address through Local_M and Local_P, the Dir_M is compared with the value of Local_M. If the directory number is higher than the Local_P of the processor and the Dir_M is twice Local_M, Local_P of the processor is too low and is set to the directory number. If the directory number is lower than the Local_P of the processor and the Dir_M is the same as Local_M, the Local_P is too high and is set to the directory number.

2.3. Multi-threaded Reorganization

In concurrent linear hashing of Ellis[Elli 87] and sequential linear hashing of Litwin [Litw 80], P always determines the next bucket to be split or the next bucket to be merged. For example, if a split operation was in progress, P always points to the directory being split. This forces all aspects of database reorganization to be single threaded. With retry logic it is not necessary to complete a split or merge operation before another split or merge operation can start. In distributed linear hashing with multithreaded split Global_P is moved before the directory is merged or split. The updates to Global_P are serialized but the actual database reorganization is performed in parallel. The algorithm to implement this is shown in Figure 2.3.

```
procedure multithreaded_split()
begin
 if ( lock_no_wait(Global_P) = busy ) return;
 Tmp_P := Global_P;
 Global_P := Global_P + 1;
 if ( Global_P == Global_M ) do begin
   Global_P := 0;
   double_M();
 end
 unlock(Split_P);
 Perform_Split(Tmp_P);
end
```

Figure 2.3 Multithreaded split operations

In multithreaded split there are no lock waits while a directory lock is held. When a busy lock on a central variable is encountered the split is not performed. The critical aspect of multithreaded splitting is that Global_P is unlocked before the directory is actually split. The only aspect of database reorganization which is single threaded is the update to Global_P. After Global_P has been unlocked another processor can lock Global_P and begin a split operation on the following directory. Hash computation does not use the value for Global_P to compute the proper directory for a given key so inaccuracies in Global_P because of multithreaded split do not affect the hash computations. Retry logic locates the proper directory regardless of the order of split completions because the values for Dir_M in each bucket are maintained properly when the split directories are unlocked.

Multithreaded split allows database reorganization to keep up with continuous insert operations regardless of the number of processors. As more processors insert records causing directories to overflow more processors begin to split the directories to keep up with the incoming records. Merge operations are multithreaded as well using the same technique as split. Global_P is decremented and M is halved if necessary before the actual merge is

performed.

3. THE KDL SYSTEM

We have developed a KDL_RAMFILE (Key access, Distributed Lock, RAMFILE) system which is based on distributed linear hashing, as described in the previous section. The KDL_RAMFILE system is currently operational on BBN's Chrysalis [BBN 86], GP-1000 [BBN 88] and TC2000 systems.

3.1. Distributed Linear Hashing

The distributed linear hashing index and data blocks are distributed randomly among the processors to avoid any hot spot contention. The data structures were extended to keep track of all the index and data blocks which are stored on each processor. Using these data structures a complete file scan is implemented using only local memory accesses.

Fast scan consists of a loop that sequentially scans each data block in a processor's local memory. Load balancing is a direct extension of fast scan. When a processor has no more data blocks in its local memory, it extends the scan to examine data blocks in other processors. Thus the KDL data block is the unit of work for load balancing.

Because of load balancing, other processors may be accessing a particular processor's data blocks, in addition to the processor that owns them. To provide exclusive access to individual data blocks, each processor maintains a variable, under exclusive access via a lock, that specifies the address of the next unprocessed block in that processor's list of local data blocks. When a processor has no more unprocessed blocks, this variable is cleared.

The KDL_RAMFILE system can be used for many applications requiring a fast file system. One application which has been implemented is parallel projection. Parallel projection employs all of the important features of the KDL_RAMFILE system.

3.2. Parallel Projection

We have implemented parallel projection using fast scan and distributed linear hashing.

Fast scan is used to scan the input relation quickly using a minimum of global memory resources. As each record is accessed by fast scan, it is projected and inserted into the output relation. As the records are inserted into the output relation duplicates are detected by the KDL system and discarded. The KDL system compares a record to be inserted into the relation only when the hashed key matches the key of a record already in the database. This reduces the actual number of full record comparisons relative to parallel sorting methods. The duplicates are removed without the need for a separate hash table or separate pass over the data for duplicate removal. Because the KDL files are dynamic there is no need to estimate the size of the output relation.

4. PERFORMANCE ANALYSIS

4.1. Experimental Details

The performance of the KDL_RAMFILE and the performance of parallel projection using the KDL_RAMFILE as an example application, are presented. The goal of the KDL_RAMFILE performance evaluation was to measure the maximum continuous throughput of the database system. This implies that system overhead such as creation of the application processes, operating system overhead to initialize and terminate tasks on each processor, are ignored. For parallel projection the time to complete a projection was measured. Since projection requires creation of an output file, the time to create this file is included in the total time for parallel projection.

The total number of database operations per second were measured while varying the number of processors. Under ideal conditions, the total number of operations per second should increase linearly as the number of processors are increased. Total number of operations per second was chosen over the number of effective processors because it shows the speedup while allowing quantitative comparison between different types of runs on the same graph. For the KDL_RAMFILE performance, the operations are record read, writes, and deletes. For parallel projection the operations are record projections. The KDL_RAMFILE performance tests were conducted on a GP1000 running the Chrysalis operating system [BBN 86]. The parallel projection performance tests were conducted on a TC2000 running the nX operating system.

4.2. Performance impact of Local_P and Local_M

This section examines the performance impact of distributing the global variables as described in section 2. It should be noted that Local_P and Local_M are used only for hash address computation. Global_P and Global_M are used for database reorganization. We will

compare the performance of two different models of distributed linear hashing with standard linear hashing. These schemes are defined in terms of the type of variables used in computing the hash addresses:

Scheme 1: Local_P, Local_M, Dir_M
Scheme 2: Global_P, Local_M, Dir_M
Scheme 3: Global_P, Global_M

Note that schemes 1 and 2 use retry logic while scheme 3 is the standard linear hashing scheme accessing global variables. The performance of the three schemes for continuous read operations is shown in Figure 4.1. The figure shows that the overall performance improves for scheme 3 using global data structures, protected by central locks, until about 10 nodes. Then the overhead of the hot spots caused by these global data structures begins to dominate the computation, and performance stops improving. Similar performance degradation has been observed for BBN's implementation of parallel main memory file system, called RF_RAMFILE system, where central lock structure has been used [BBN 86]. The performance of RF_RAMFILE system levels off after 15 nodes doing continuous activity. Figure 4.1 shows that Scheme 2 maintains linear performance for up to 40 nodes. We have observed that avoiding the use of central locks and Global_ M for hash address computation reduces the amount of central memory accesses significantly (by about 75% in our experiment). In the implementation using scheme 3 each database operation accesses the lock twice and the values for Global_P and Global_M. In the implementation using scheme 2 there is only a single global access per operation. Thus the performance is impacted by the remaining central accesses at a high level of parallelism. The implementation using scheme 1 has the best performance at high levels of parallelism because it never accesses global variables. So performance of scheme 1 is only limited by the performance of the memory sharing network under random memory access patterns. Figure 4.1 shows that performance is still improving for scheme 3 at 80 processors.

Insert operations require database reorganization and updating global information.

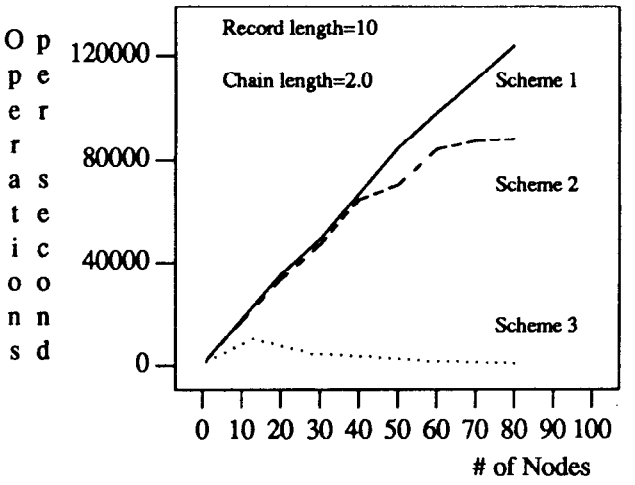

Figure 4.1 Read performance of different implementations

Figure 4.2 shows the performance comparisons of the three schemes when performing continuous insert operations. Scheme 1 maintains performance improvement until 15 processors where the memory conflicts limit any additional performance improvement.

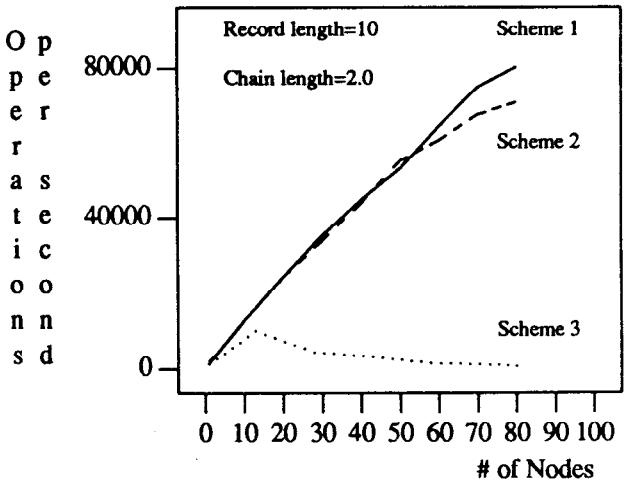

Figure 4.2 Insert performance

Scheme 1 and 2 continue to show performance improvement at higher level of parallelization. However, the performance of these two schemes is much closer for inserts than for reads. At 50 nodes and below, the implementation using scheme 2 performs as well and at times performs better than scheme 1. Above 50 nodes performance continues to improve for scheme 1 while for scheme 2 it begins to level off.

There are several reasons for this performance characteristics. (1) Insert operations are inherently slower because they include database reorganization. (2) The database reorganization makes use of the global variables. This means that when performing inserts even scheme 1 will have to access global data structures causing some memory contention. (3) As the database is reorganized, the value for Global_P is changing causing the independent copies of Local_P to become inaccurate. This causes incorrect directory to be locked requiring additional directory locks due to retry logic.

The percentage of retries required for scheme 2 is very small. At 80 processors 3 out of 1000 bucket accesses will have to be retried because the value for P is inaccurate. For scheme 1, retry percentage ranges from 4 to 7.

Unlike the overhead of accessing global data structures, the overhead of retry logic does not result from hot spot accesses. Retry logic uses only the bucket locks and bucket locks are randomly distributed among the processors. However, retry logic causes some additional *random* memory references but the bandwidth of random memory references for Butterfly type architecture increases with the number of processors. Thus the overhead of retry logic remains steady with increasing number of processors. This allows the overall system performance to improve as additional processors are added.

4.3. Performance of Multi-Threaded Database Reorganization

All of the performance figures given above include cost of database reorganization for continuous insert and delete operations. Under continuous insert or delete loads, the database is being reorganized continuously to maintain the desired maximum allowed chain length. Distributed linear hashing allows this database reorganization to be performed in a parallel fashion. This allows the reorganization to keep up with the insert or delete operations. Figure 4.3 compares the average chain lengths for single and multi-threaded reorganization under continuous inserts. Both implementations are trying to maintain a maximum chain length of 2.0 records.

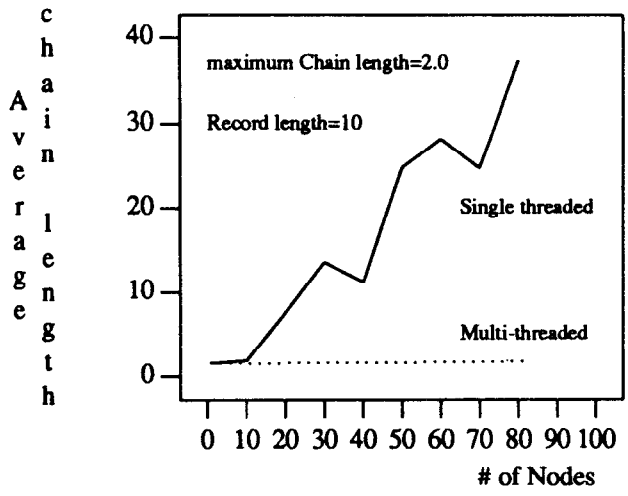

Figure 4.3 Chain length control using multi-threaded reorganization

The multi-threaded database reorganization is able to maintain the desired chain length regardless of the number of processors. The single threaded implementation cannot keep up with the required reorganization above 10 processors.

4.4. Performance of Parallel Projection

Parallel projection takes advantage of the fast scan feature of KDL for reading input records and the distributed hashing structure for duplicate removal. The effectiveness of fast scan's load balancing was tested by comparing runs with and without load balancing. We compared performance with an even distribution of records among processors, and an uneven distribution of records among processors. For the uneven distribution of records, the number of records processed by each processor was a uniform random number between zero and the number of records processed in the even distribution case.

For an even distribution of records, performance improved slightly at higher numbers of nodes because load balancing offsets the speed differential among processors due to variations in memory conflicts among the processors [Figure 4.4]. These memory conflicts increase as the number of nodes used increases, so the performance improvement in the even distribution case should be even greater as more nodes are used.

As expected, load balancing provided a dramatic performance improvement with an uneven distribution of records because there will be some processors with relatively few records that will assist those processors with

more records [Figure 4.5].

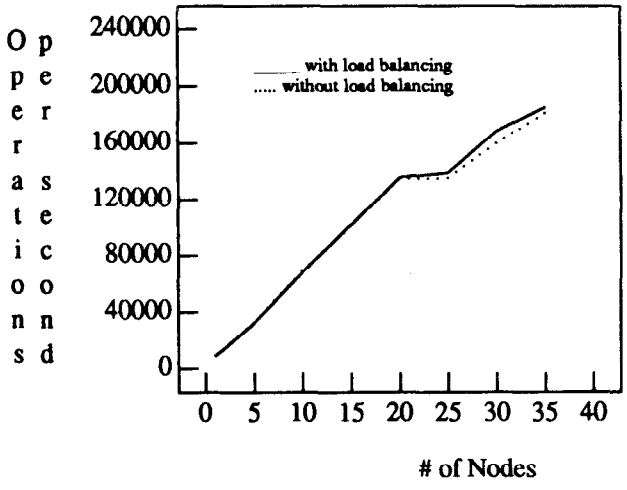

Figure 4.4 Performance with an even distribution of records

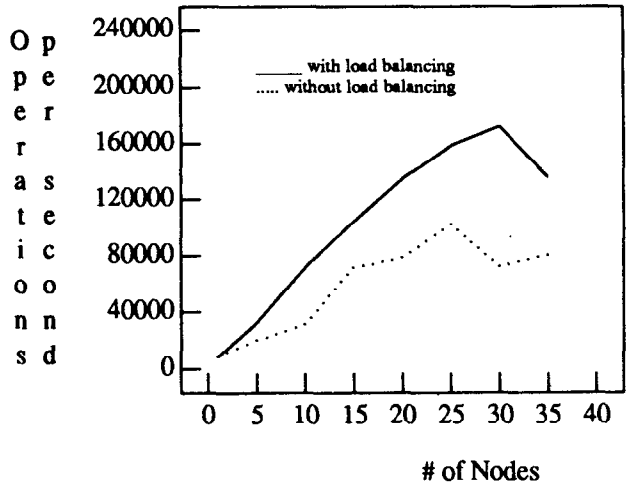

Figure 4.5 Performance with an uneven distribution of records

To test processing of duplicate records, each processor created duplicates by copying existing records from other processors. Runs with 0%, 20%, 50% duplicates were made [Figure 4.6]. The performance improves with increasing number of duplicates, even though duplicate removal requires global memory references to compare records when two records have keys that match. This cost is more than offset by the reduction in cost of creating the output file, which is smaller because of discarded duplicate records.

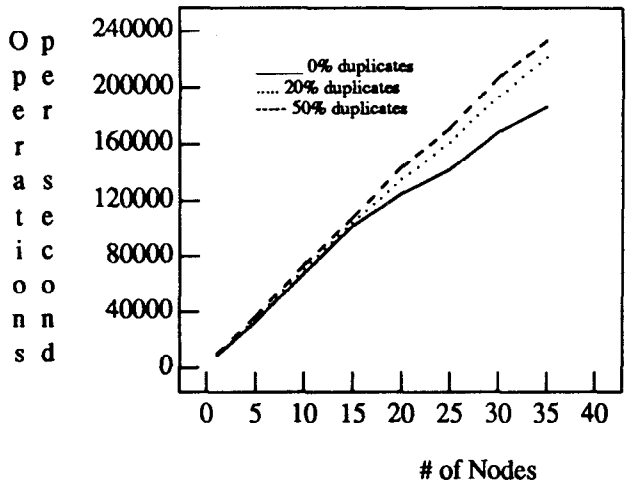

Figure 4.6 Performance with different percentages of duplicates

5. CONCLUSION

Distributed linear hashing is shown to have excellent performance as a main memory database on a NUMA architecture system. Distributed techniques have been employed to minimize the use of central variables and locks. The additional fixed overhead of the retry logic for these distributed techniques is small compared to the cost of accessing central data structures at high levels of parallelism.

Distributed linear hashing is an effective platform for implementation of parallel projection of main memory data bases on a general purpose NUMA architecture system. Fast scan is an important extension to distributed linear hashing to support parallel projection.

Performance has been analyzed by building a working main memory file system on BBN's Butterfly parallel processor. The file system has been used to implement high performance parallel projection with duplicate removal.

6. BIBLIOGRAPHY

[BBJW 83] Britton, D., Boral, H., DeWitt, D., and Wilkinson, W. "Parallel algorithms for the execution of relational database operations". *ACM Trans. Database Syst.* 8, 3 (Sept. 1983), 324-353.

[BBN 88] GP-1000 Tutorial, BBN Advanced Computers, Inc., 1987

[BBN 86] Chrysalis Programmers Guide, BBN Advanced Computers, Inc., 1986.

[Elli 87] Ellis, C., "Concurrency in Linear Hashing", *ACM Trans. on Database Systems*, June 1987

[FNPS 79] Fagin, R., Nievergelt, J., Pippenger, N., Strong, H.R., "Extendible Hashing - A Fast Access Method for Dynamic Files", *ACM TODS*, 1979, Vol. 4(3), pp. 315-344.

[Ghos 86] Ghosh, S. P., "Database Organization for Data Management"', *Second Edition, Academic Press*, 1986

[GLV 84] Garcia-Molina H, Lipton R, Valdes J, "A massive memory machine"', *IEEE Transaction on Computers*, Vol. c-33, No. 5, May 1984

[Lars 78] Larson, P.A., "Dynamic Hashing", *BIT*, 1978, Vol. 18(2), pp. 184-201.

[Lars 80] Larson, P.A., "Linear Hashing with Partial Expansions", *Proc. 6th VLDB Conference*, 1980, pp. 224-232.

[LeCa 86] Lehman, T.J., Carey, M., "Query Processing in Main Memory DBMS"', *Proc. 1986 SIGMOD*, pp. 239-250.

[Litw 80] Litwin, W., "Linear Hashing: A New Tool for File and Table Addressing", *Proc. 6th VLDB Conference*, 1980, pp. 212-223.

[PrKi 88a] Pramanik S, Kim M, "Generalized Parallel Processing models for Database Systems", *Int. Conference on Parallel Processing*, 1988.

[PrKi 88b] Pramanik S, Kim M, "Optimal File Distribution For Partial Match Retrieval", *Proc. ACM SIGMOD Conf.*, 1988.

[PrKi 89] Pramanik S, Kim M, "Parallel Processing of Large Node B_trees", *To appear in IEEE Transaction on Computers.*

[RoJa 87] Rosenau, T., Jajodia, S. "Parallel relational database operations on the Butterfly parallel processor: projection results", Naval Research Laboratory report (July 1987).

[SePr 90] Severance C, Pramanik S, Rosenau T, "A High Speed KDL-RAM File System for Parallel Computers", *Proc. PARBASE-90, IEEE Computer Society Press*, 1990, pp 195-203.

[Sev 90] Severance C, *A Linear Hashed Main Memory Database in a Non-uniform Multi-processor System*,, M.S. Thesis, Michigan State University, February 1990, Available as *Technical Report MSU-CPS-DB-290.*

[Wied 87] Wiederhold G, *File Organization for Database Design*, McGraw-Hill, New York, 1987.

[Wolb 89] Wolberg P, "An Application of Distributed Linear Hashing to Relational Projection in a Multi-Processor Main Memory Database", *Technical Report*, Computer Science Department, Michigan State University, Dec. 17, 1989.