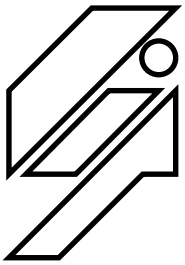
Towards a dynamic parallel database machine: data balancing techniques and pipeline

Thibault Duboux, Afonso Ferreira



Laboratoire de l'Informatique du Parallélisme

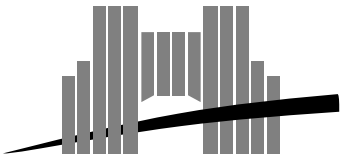
Ecole Normale Supérieure de Lyon
Unité de recherche associée au CNRS n°1398

Towards a dynamic parallel database machine: data balancing techniques and pipeline

Thibault Duboux
Afonso Ferreira

December 1994

Research Report N^{O} 94-47



Ecole Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : (+33) 72.72.80.00 Télécopieur : (+33) 72.72.80.80
Adresse électronique : lip@lip.ens−lyon.fr

Towards a dynamic parallel database machine: data balancing techniques and pipeline

Thibault Duboux
Afonso Ferreira

December 1994

Abstract

The fast development over the last years of high performance multicomputers makes them attractive candidates as the base technology for scalable and performance oriented database applications. In this paper, we address the problem of how to process utility commands while the system remains operational and the data remain available for concurrent access. In particular, we focus on the on-line reorganization of a *dictionary*, a database reduced to its simplest instance, showing its implementation on a multicomputer. As is the case with implementations of dynamic structures on distributed memory architectures, a crucial load balancing problem has to be solved. We propose an elegant solution and prove that it solves this problem. Experimental results are shown and analyzed.

Keywords: Dictionary Machine, Parallel Data Structures, Parallel Databases, Load Balancing

Résumé

Le développement des ordinateurs massivement parallèles rendent ces machines intéressantes pour des applications de bases de données qui soient extensibles et performantes. Dans ce rapport, nous abordons le problème de la mise à jour de ces bases (insertions - suppressions de données) tout en les laissant disponibles et opérationnelles. En particulier, nous nous penchons sur la redistribution en temps réel d'un *dictionnaire*, la plus simple des bases de données, sur une machine parallèle. Comme pour toute implantation de structures dynamiques sur des architectures à mémoire distribuée, il est crucial de résoudre le problème de l'équilibrage de la charge. Nous proposons une solution pour traiter ce problème, nous prouvons son efficacité et nous analysons les résultats expérimentaux obtenus.

Mots-clés: Machine dictionnaire, Structures de données parallèles, Bases de données parallèles, Équilibrage de charge.

Towards a dynamic parallel database machine: data balancing techniques and pipeline*

Thibault Duboux Afonso Ferreira†

Laboratoire de l'Informatique du Parallélisme
URA 1398 du CNRS
Ecole Normale Supérieure de Lyon
69364 Lyon Cedex 07, FRANCE
e-mail: duboux@lip.ens-lyon.fr

Abstract

The fast development over the last years of high performance multicomputers makes them attractive candidates as the base technology for scalable and performance oriented database applications. In this paper, we address the problem of how to process utility commands while the system remains operational and the data remain available for concurrent access. In particular, we focus on the on-line reorganization of a *dictionary*, a database reduced to its simplest instance, showing its implementation on a multicomputer. As is the case with implementations of dynamic structures on distributed memory architectures, a crucial load balancing problem has to be solved. We propose an elegant solution and prove that it solves this problem. Experimental results are shown and analyzed.

Keywords: Dictionary Machine, Parallel Data Structures, Parallel Databases, Load Balancing.

1 Introduction

The fast development over the last years of high performance multiprocessor machines makes them attractive candidates as the base technology for scalable and performance oriented applications. In order to handle the increasing amount of data and the query complexity, parallelism appears as one of the most promising research axes for future database applications [DG92].

A consensus architecture on parallel and distributed database systems has emerged. Such architecture is based on a *shared-nothing* hardware design in which processors communicate with one another only by sending messages via an interconnection network. The data is partitioned across processors. Data partitioning is therefore the first step in parallel query optimization, i.e.,

*This work was partially supported by Stratagème project of the French CNRS and the DRET.

†CNRS.

the parallelization of an input query to be executed on parallel machines [Val93]. Such architectures were pioneered by Teradata in the late seventies. Different approaches yielded the prototypes XPRS by Stonebraker [SPKO88], GAMMA by DeWitt [DeW90], and DBS3 by Valduriez [VCB91]. It should be noted that none of these prototypes have implemented a dynamic data partitioning.

As explained in [DG92], loading or reorganizing a terabyte database takes over several days. In the SQL world, typical utilities create indices, add or drop attributes, and physically reorganize the data, changing its clustering. Clearly, parallelism is needed if such utilities are to complete within a reasonable time. Even then, it will be essential that the data be available while the utilities are operating.

One unexplored and difficult problem is how to process database utility commands while the system remains operational and the data remain available for concurrent reads and writes. The fundamental properties of such algorithms is that they must be *online* (operate without making data unavailable), *incremental* (operate on parts of a large database), and *parallel*.

In this paper, we focus on this on-line reorganization applied to a *dictionary*, a database reduced to its simplest instance. The dictionary is an important data structure used in applications such as sorting and searching, symbol-table and index-table implementations [Knu72]. It is a basic data type which provides update and retrieval operations on a set of records. There is a unique search key K from a totally ordered set associated with each record. The standard dictionary operations are INSERT, DELETE and SEARCH. In addition, the EXTRACT-MIN *priority queue* operation may also be provided [AHU83]. Note that some of these operations require a response to be produced.

The dictionary task can be loosely defined as the problem of maintaining a set of key-record pairs (K,R). For simplicity, the record whose associated key is K will be denoted record K. The dictionaries we consider support at least the following set of operations on its entries.

- INSERT(K,R): inserts key-record pair (K,R) in the dictionary;
- DELETE(K): deletes record K from the dictionary;
- SEARCH(K): retrieves record K if currently stored, does nothing otherwise;
- EXTRACT-MIN: returns the current minimum record and deletes it.

Insert and Delete can be *redundant*. An insertion is redundant when the key being inserted already exists in the dictionary; a deletion is redundant when the key being deleted does not exist.

Because of its general and fundamental capabilities, one important problem consists in designing special-purpose multiprocessor systems, called *dictionary machines*, implementing dictionaries of more or less restricted types. In a dictionary machine, a sequence of instructions (i.e., requests to perform dictionary or priority queue operations) is received through an I/O port. The machine executes the corresponding operations in a pipelined fashion, and reports the responses, if any, via the I/O port. Thus, performance of such a machine can be measured in terms of the following parameters.

Response time: The elapsed time between initiation and completion of an instruction.

Pipeline interval: The minimum elapsed time between the initiation of two distinct instructions.

Capacity: The maximum number of records that may be stored in the machine.

Several specially designed parallel architectures have been proposed in the literature for the implementation of dictionary machines on VLSI chips [Nar86]. Almost all proposed designs are based on the complete binary tree structure [Lei79, ORS82, AK85, SA85, LP90, FC91], while few papers report on other topologies, like systolic meshes [SS85], Cube Connected Cycles (CCC) [SL87] or hypercubes [DS89]. The problem of scalability of these VLSI designs has been studied [DFG94].

Since those parallel dictionary machines are primarily intended for implementation in VLSI, one of their main characteristics is complete processor utilization, i.e., there is one data item per processor in the system. Only a few papers proposed dictionary machines where the number of processors is several orders of magnitude smaller than the number of records [Fis84, OB87, DG90]. Some of them have led to implementations in existing parallel systems as on the Maspar MP1, a SIMD architecture, and on the Volvox i860 [Gas93, DFG92].

In this paper, we consider the implementation of a dictionary machine on a real parallel computer. Our goals are to show that good performance can be achieved with general purpose parallel machines, and to study the influence of the MIMD approach on the algorithms and the results. We also develop a general technique to dynamically solve the load balancing problem arising in many on-line applications on distributed memory machines.

We present the main features of our design in the following section. We propose an elegant solution to the load balancing problem, and show interesting properties of the chosen strategy. The corresponding implementation is then described and analyzed after a presentation of the target architecture and its characteristics, necessary to understand the choices made during the implementation. Some results of our experiments are shown, and figures comparing different executions are analyzed. We end the paper with some concluding remarks.

2 Design issues

In this section, we present all the points and ideas independent of the target architecture. The ideas of partitions, partial and total sum calculation for balancing were first introduced in [DG90].

2.1 Partitioning the space of the keys

The idea to get a distributed data structure is to split S, the space of the keys. We sort and make a partition of this space. Each node i will handle a *working domain* D_i, with $\bigcup D_i = S$ and $\bigcap D_i = \emptyset$, such that if $k_i \in D_i$ and $k_j \in D_j$ then $k_i < k_j$ if and only if $i < j$. One can notice that assuming a balanced structure, with the same amount of data on each node, means that the distribution law of the keys is known. Indeed, with this law, it is easy to split S in domains that have the same

probability concerning instructions. Saying that processor p_i handles D_i does not mean that p_i can store all the keys belonging to D_i, as the space can be very large and the distribution of the keys very sparse. Therefore, a bad or moving distribution of the keys may cause a memory overflow on a node, even though others are empty.

2.2 Broadcasting a query and getting the response

Let us consider an instruction to be performed on the (global) dictionary machine. It has to be broadcast to the processors, to be executed by one of them. After this broadcast, the response has to be collected on the host. It is important to note here that we take into account the time necessary for input/output, which is not the case in many other suggested machines.

2.3 The local data structure: balanced trees

We have now to define the local data-structure to be stored and maintained on each processor. In the case of sequential algorithms, balanced trees are the most powerful. Several data structures yield such balancing, but just a few can support exact balancing along with logarithmic time operations needed for distributed load balancing, SPLIT and CONCAT (see Section 2.4). We have chosen to work with 2-3-4 trees because of these parameters, and used binary colored trees (BCT) where data are stored in the leaves, to implement them ([Meh84]). Refer to Figure 1 for an example of a 2-3-4 tree and its BCT counterpart.

A SEARCH is implemented exactly as in a binary search tree. INSERT, DELETE, EXTRACT-MIN instructions all have the same behavior: first, locate the required leaf, then modify this leaf (remove it or duplicate it) and perform rotations (when necessary) in the path from this leaf to the root. Each of these instructions are executed in $O(\log N)$ time, N being the number of elements in the tree.

CONCAT instruction is used to merge two trees. Let H be the height of the first tree and h the height of the second tree. We suppose, w.l.o.g., that $H \geq h$. CONCAT can be described as follows.

1. Selection of the largest node $\mathcal{N}$ belonging to level $H - h - 1$ in the tree of height H.
2. Creation of a new node, whose left child is the right child of $\mathcal{N}$ and its right child is the root of the other tree.
3. Insertion of this new node as right child of $\mathcal{N}$ (with the same rotations in the path as for a standard insertion).

In the same way, SPLIT instruction is used to split a given number of the largest elements from a tree. To perform it in optimal time, we append to the content of each inner node the number of elements in the subtree rooted at that node. This value is updated at each modification in the tree but does not change the order of complexity. SPLIT is described as follows.

1. Determine the set of vertices to be split.

2. Remove the inner nodes at the boundary of the two trees (with a depth-first traversal).

3. CONCAT the disconnected branches to their respective trees.

Both of these instructions are executed in $O(\log N)$ time.

2.4 Load balancing the dictionary

So far we have assumed that the distributed data was balanced, i.e., that the distribution law of the keys was known beforehand. However, this is not the case in general, and this assumption does not make sense if the law changes in time. In general, under conditions like real time constraints, the law is not known. It means that no hypothesis can be made *a priori* about the key distribution among the processors. Moreover, the load of a processor (in memory space as well as in work) directly depends on the size of its data structure. In fact, the larger this structure, the larger the time for executing one instruction.

To solve this problem, we propose a simple strategy, based on local data exchanges. The balancing algorithm can be decomposed in two phases. During the first one, each processor p_i computes TS, the size of the whole data structure, and PS_i, the size of the dictionary handled by processors with a number smaller than i (this for $i = 0...P-1$ where P is the total number of processors). Formally, with n_j the size of the local data structure on processor j :

$$TS = \sum_{j=0}^{j<P} n_j \qquad\qquad PS_i = \sum_{j=0}^{j<i} n_j$$

From this, each processor can compute where there is an imbalance and decide if data have to be sent to its left, to its right, to both sides or none of them.

The second phase consists of exchanging data with neighbors, according to the previous calculation. It follows the updates of the dictionary, the size, and the bounds of the working domain on each processor. After that, the distributed data structure is balanced.

Theoretically, this strategy was proven to be efficient in [DG90]. In this paper, we shall show how to take into account the parameters of real machines in order to design a balancing strategy that really works.

Let `Split&Send(`$dict_i$`,size,dest)` be the function that splits the `size` smallest (or largest) data from structure $dict_i$ (on the current processor i) and sends them to processor `dest`. Note that reception is implicit. Let `Concat&Update(`$dict_i$`)` restore a coherent data structure $dict_i$. Let further MIN and MAX be constants, and DR_i and DL_i be the amount of data to be sent to the right and left respectively. If n_i is the size of the local data structure $dict_i$ on processor p_i, the algorithm for balancing can be written as follows:

```
BALANCE ALGORITHM:
/* Sum Calculation */
   for all i do in parallel
```

$TS = \sum_{j=0}^{j<P} n_j$;
$PS_i = \sum_{j=0}^{j<i} n_j$;
$DL_i = \lceil \frac{TS}{P} \times i - PS_i \rceil$;
$DR_i = \lfloor \frac{TS}{P} \times (P - i - 1) - (TS - PS_i - n_i) \rfloor$;

/* Data Balancing */
for all i do in parallel
If $DL_i > MIN$ Then
Split&Send($dict_i$, Min(MAX, DL_i), p_{i-1});
If $DR_i > MIN$ Then
Split&Send($dict_i$, Min(MAX, DR_i), p_{i+1});
Concat&Update($dict_i$);
$n_i = n_i +$ Size(Received) $-$ Size(Sent);
Update($bounds_i$);

The constant MIN corresponds to the minimum number of elements in excess to justify balancing. MAX is the maximum number of elements sent during a balancing.

One can remark that data are exchanged only between neighbors in the ring. Furthermore, if $DR_i < 0$ (or DL_i), processor i only receives data.

Lemma 1 *The following statements are always true:*

1. $DL_0 = 0$
2. $DR_{P-1} = 0$
3. $\forall i \in [0, P-2], DR_i = -DL_{i+1}$

Proof: Immediate for statements 1 and 2. And, for statement 3, we have:

$$\begin{aligned} DR_i &= \left\lfloor \frac{TS}{P} \times (P - i - 1) - (TS - PS_i - n_i) \right\rfloor \\ &= \left\lfloor \frac{TS}{P} \times P - TS - (\frac{TS}{P} \times (i+1) - (PS_i + n_i)) \right\rfloor \\ &= -\left\lceil (\frac{TS}{P} \times (i+1) - PS_{i+1}) \right\rceil \\ &= -DL_{i+1} \end{aligned}$$

□

Clearly, a good strategy to use BALANCE efficiently depends upon the programming mode of the target architecture. In the following section we shall show how we take it into account in our implementation.

3 Solution in the target architecture

Our target architecture, the Volvox machine, is a coarse grained, distributed memory parallel computer. The implementation we describe below is based on a ring of Dictionary Machines (DM's), as shown in Figure 2. Instructions to be executed are pipelined in this ring. Each DM processes the subset of the instructions belonging to its working domain, and forwards the other instructions to the next DM. Input (respectively, output) is supposed to come from (respectively, go to) users outside the machine, being supported by the host.

Instructions sent by the host are analyzed by the processors in the ring. Once a processor has selected a particular instruction to treat, it will access the local data structure. Algorithms corresponding to dictionary instructions are described in Section 2.3. Just recall that SEARCH, INSERT, DELETE and EXTRACT-MIN algorithms are locally performed in $O(\log n_i)$ time, where n_i corresponds to the size of a local data structure.

3.1 The Volvox IS860: a distributed memory architecture

The test-bed for the implementation of our distributed dictionary machine was a Volvox IS860 from Archipel. The Volvox Supercomputing server (see Figure 3(a)) is a distributed memory architecture implementing the CSP message passing model. It has 48 available nodes, accessed via 4 independent communication boards lying in a Sun workstation (the host). Each communication board is a *Transputer T800* with 1MB of memory. In order to allow the user to define any variable topology, each physical node (see Figure 3(b)) is composed of a Transputer T800 with 4 reconfigurable communication links, as a Communication Processor, and an Intel 860 as an Application Processor. The total memory of a node is formed by 4MBytes of Transputer's private memory plus 16MBytes RAM independently accessed by the two processors via a double port mechanism. The application described in this paper takes into account the extensibility of the machine.

An application is described by a set of communicating tasks. Tasks are mapped onto processors as defined by the user without any limitations except that each i860 can support only one task. Tasks communicate by primitives from Volvox library. Communication can be synchronous (blocking) or asynchronous. Message routing is automatic and implicit.

Communication costs can be modeled by the well-known linear model. Thus, for a message of length L, the time for a communication is: $t = \beta + L\tau$ where β is the startup time and τ the time for a byte to be transmitted. Table 1 summarizes values of β and τ for various cases of communication: between two neighboring Transputers, two neighboring i860 and a T800-i860 communication inside a node. We verify experimentally that β increases with the number of user tasks in the application.

For our application, communications between a T800 and the i860 inside the same node with system primitives is too slow to be used. Indeed, we need to communicate once for each instruction processed, and communication takes at least 1040μs while an elementary instruction manipulating trees on i860 takes at most 130μs (for instance an insertion, see Section 2.3). To avoid this problem, we must implement a low level technique via the double-port shared memory.

Figure 1: A 2-3-4 tree and its binary colored implementation.

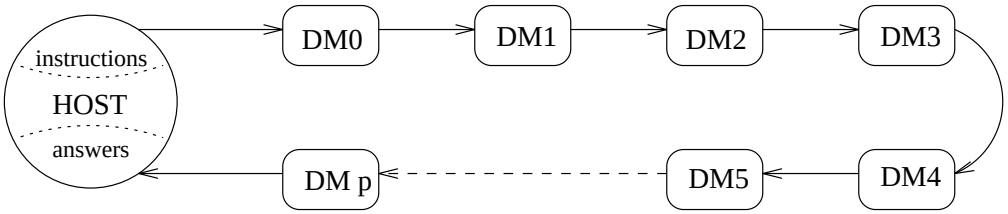


Figure 2: Architecture of the Dictionary Machine.

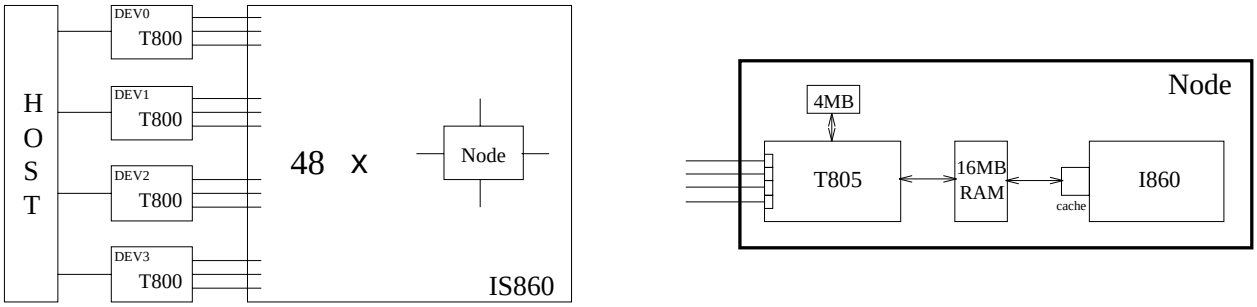


Figure 3: (a) The Volvox IS860 architecture, (b) a node of the machine.

Communication	β	τ
T800-T800 (neighbors)	766.5	1.1
i860-i860 (neighbors)	1305	1.23
T800-i860 (same node)	1040	0.1

Table 1: Costs of communications in μs.

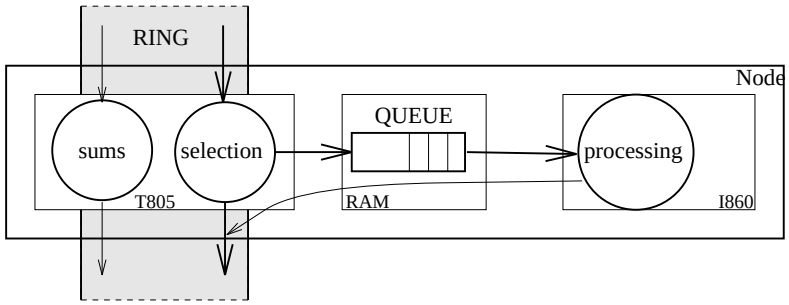


Figure 4: Organization of the processes.

It is also interesting to compare T800-T800 communication costs (at least 766μs), with our computing cost (at most 130μs). We will analyze the impact of such values in Section 3.5.

3.2 The parallel dictionary machine

We describe here the global behavior of the proposed dictionary machine. As seen before, each physical node is a dictionary machine by itself, and they are connected in a ring architecture. The instructions circulate on the ring, starting at the host. Because of the internal organization of the nodes of the Volvox parallel computer, we implemented a client/server protocol inside each node (see Figure 4). The Transputer selects instructions within its working domain. It also forwards output messages that arrive from its predecessor in the ring.

The binary colored tree storing the elements of the dictionary is located in the i860 memory, which is used as the actual processing element of the application. The i860 executes the current instruction, modifying the data structure. Eventual output messages are sent by the i860 directly to the next node on the ring.

Communications between the two processors of a single node are performed via the shared memory. It stores a circular queue containing the instructions selected by the *selection* task on the Transputer and waiting to be processed by the i860. The head of the queue is controlled by the *processing* task on the i860, i.e., every time an instruction is to be executed, it is extracted from the queue. Similarly, the *selection* task on the Transputer controls the tail of the same circular queue. Every time an instruction is selected from the instruction flow in the ring, it is inserted into the queue.

The working domain of a node is given by a lower and an upper bound on the key values. Any instruction containing a key in that interval should be treated by that node. As described

in Section 2.4, a dynamic updating of the working domains is essential to ensure load balancing. For this reason, we add a task on each Transputer, called *summation* task, to compute total and partial sums. These values are useful for our distributed load balancing technique. First, these sums are globally computed, concurrently with instruction processing, and then, instruction processing is suspended during data balancing. After that, the normal behavior is restored till the next balancing phase.

3.3 Details on dynamic load balancing

During the instruction processing phase, *summation* tasks keep computing total sum and partial sums, as described in Section 2.4. The algorithm for computing these sums uses a simple strategy corresponding to a token circulating two rounds on the ring of nodes. Other strategies were tested (e.g. on a hypercube structure as proposed in [DG90]) but the simplest one is also the most powerful for synchronization reasons on our machine.

When partial and total sums are available, each *summation* task can evaluate the difference between the size of its local data structure and the average size of the other structures. It can also evaluate which sides show a deficit or an excess (through the values DL_i and DR_i). If the difference is large enough, we enter the data balancing phase.

For a node, this balancing phase consists of using *split* on a part of the local structure to be sent to the side showing a deficit or using *concat* on a set of elements received from the side showing an excess. *Split* and *concat* are done in $O(\log n_i)$ by the corresponding sequential algorithms described in Section 2.3.

3.4 Correctness and Performance Analysis

In this section, we use the following notations:

- L denotes the size in bytes of an element (key, record), and l the size of a key.
- t_i denotes the maximal time for an instruction to be processed locally in a node (tree manipulations).
- t_c corresponds to the time for a neighbor-to-neighbor communication of a message containing an instruction. We assume that $t_c = \beta + L\tau$ where β and τ are machine-dependent values (linear communication model).
- t_s is the time of total and partial sum calculation (during processing phase).
- t_b is the time of a balancing phase.
- Int is the minimum pipeline interval between two instructions sent by the host.
- MIN and MAX are machine dependent constants corresponding to numbers of elements.

Global consistency of this dictionary machine is ensured by its working mode. Here are some details of its behavior during balancing phases. Each time two nodes are involved in a balancing phase, they freeze the instruction flow upstream on the ring, then they process instructions waiting in their queues. Only after that they perform the data balancing between them. In this way, we ensure that every instruction is processed by the node corresponding to the appropriate working domain.

Furthermore, because the only synchronization mechanism used in the machine is this pairwise synchronization, our application is deadlock free. This is true if the *logical capacity* of the nodes is not exceeded.

Definition 1 *The* logical capacity *of a node is* $\frac{N}{P} - MAX$ *elements (*N *is the global physical capacity of the machine).*

We will see that this logical limit is necessary during balancing phases. Now, we want to prove that our machine maintains a balanced data structure.

Definition 2 *The data structure is said to be* balanced *if and only if* $DR_i \leq MIN$ *for all* i *(and* $DL_i \leq MIN$ *for all* i*).*

Lemma 2 *The execution of* $\mathcal{I}$ *instructions can increase the imbalance by at most* $\mathcal{I}$*.*

Proof: The proof is based on the fact that the execution of a single instruction increases the imbalance by at most one.

As seen previously, the imbalance is defined by $DL_j = \lceil \frac{TS \times j}{P} - PS_j \rceil$ (the case of DR_j is handled analogously). Let us consider DL_j before a given instruction, $Instr$, and after the execution of this instruction. Suppose that $Instr$ is a *delete*. Clearly, TS decreases by one, and PS_j stays constant or decreases by one (depending on j). Hence, DL_j does not change or decreases by one. The case of the instruction *insert* is symmetric, so an insertion increases DL_j by 0 or 1.

Using this result, we can say that $\mathcal{I}$ instructions can modify DL_j by at most $\mathcal{I}$, for all j. □

In the remainder of this section, we suppose that each node handles at least MAX elements at the beginning of balancing phases. This restriction is not severe as it just means that the data structure is not completely empty. However, even if this condition is not true, the balancing strategy can be applied. It can be shown that the resulting structure is not balanced immediately after one balancing phase, but becomes more and more balanced in time. This phenomenon is illustrated in the experimental results by Figure 9. A similar analysis for empty structures can be found in the case of a SIMD implementation [Gas93].

Proposition 1 *If the instruction flow is globally frozen during the balancing phase, then, for fixed values of* MIN *and* MAX*, we have the following relation:*

$$If, \forall i \in [0..P-1], max_i(DR_i) \leq MAX \textit{ just before balancing, then}$$

$$max_i(DR_i) \leq MIN, \forall i \in [0..P-1], \textit{ just after the balancing phase.}$$

Proof: During a data balancing phase, as described in subroutine `balance`:
"`If` $DR_i > MIN$ `Then Split&Send`$(dict_i, \texttt{Min}(MAX, DR_i), p_{i+1})$;"

- Thus, if $DR_i > MIN$, each node can send up to MAX data, which is enough to balance every DR_i.
- If $DR_i \leq MIN$, `Send` is not performed, DR_i is unchanged, but the proposition still holds.

Since, according to lemma 1,$DL_i = -DR_i$, we only need to study DR_i. □

Definition 3 *The value $MAX - MIN$ represents the* capacity of balancing *for a node, i.e., the maximum number of elements balanced during one balancing phase.*

Lemma 3 *At most $\frac{t_s}{Int}$ instructions are processed between two successive balancing phases.*

Proof: Our application alternates processing phases (with sum calculations) and balancing phases. The partial and total sum calculations are done simultaneously with the processing phase. The balancing phase starts at the end of this calculation. Therefore, between two successive balancing phases, instructions arrive and are processed, with a pipeline interval Int, during time t_s. □

Hence, there is an imbalance of at most $\frac{t_s}{Int}$ between two successive balancing phases (by Lemma 2). So, our data structure remains balanced whenever:

$$MAX - MIN \geq \frac{t_s}{Int}.$$

Now, we want to relax the hypothesis of Proposition 1 so that dictionary instructions may be processed on some nodes while other nodes are balancing. This is very important in order to use the asynchronous capabilities of the host parallel computer at their best.

To ensure efficiency of our balancing strategy, the *capacity of balancing* has to be greater than the number of instructions potentially processed during a complete cycle (processing phase plus balancing phase):

$$MAX - MIN \geq \frac{t_b + t_s}{Int}.$$

Lemma 4 *The cost of one balancing phase is bounded by:*

$$t_b = 2t_i + \beta + MAX \times (L + l + 10)\tau.$$

Proof: During a balancing phase, each node may execute one *split* (cost: t_i), followed by a send of size up to MAX elements, followed by one *concat* (cost: t_i). Since we need to maintain information from the split tree to ensure a CONCAT in $O(\log n)$ time in the destination node, each element (of size L) is encapsulated in a structure of size $(L + l + 10)$ bytes. Thus, the cost of the send is bounded by $\beta + MAX \times (L + l + 10)\tau$. □

Theorem 1 *Our balancing strategy is correct if we respect the following constraint:*

$$MAX \geq \frac{(\beta + L\tau) \times MIN + 2t_i + \beta + t_s}{\beta - (l+10)\tau}.$$

Proof: As dictionary instructions are sent sequentially by the host, the pipeline interval for instruction processing is at least the time to send an instruction: $Int \geq t_c$. So,

$$\begin{aligned}
& MAX - MIN \geq \frac{t_b + t_s}{Int} \\
\Leftarrow \quad & MAX - MIN \geq \frac{t_b + t_s}{t_c} \\
\Leftrightarrow \quad & MAX - MIN \geq \frac{2t_i + \beta + MAX \times (L + l + 10)\tau + t_s}{\beta + L\tau} \\
\Leftrightarrow \quad & MAX \times (\beta - (l+10)\tau) \geq 2t_i + \beta + t_s + MIN \times (\beta + L\tau).
\end{aligned}$$

If $\beta > (l+10)\tau$, we obtain the desired constraint.

In a computer with a very small start up time for communication ($\beta \leq (l+10)\tau$), our strategy is not efficient in the worst case but it can be used in an average case (when the imbalance does not reach the capacity of balancing every time). □

Our strategy for partial and total sum calculations needs $2P-1$ communications to complete ($t_s = (2P-1)(\beta + s\tau)$, where s is the size of an integer). As $L \geq s$, at most $2P$ instructions are processed during this time ($\frac{t_s}{Int} \leq 2P$). The constraint to be satisfied is now:

$$MAX \geq \frac{(\beta + L\tau) \times (MIN + 2P) + 2t_i + \beta}{\beta - (l+10)\tau}.$$

For instance, typical values for a variety of existing machines are: a ring of 32 nodes ($P = 32$), communication rate τ equals to $1\mu s$, the size of an element $L = 40$ with a key of size $l = 4$, the time of one local processing of an instruction $t_i = 130\mu s$, and the constant MIN fixed to 10.

With a communication startup β equals to $767\mu s$ (the actual value on the Volvox), we obtain $MAX > 80$, which is feasible. Even if β was small ($\beta = 50\mu s$), we would obtain $MAX > 193$ which still corresponds to a realistic value.

When P becomes large, it is possible (and even necessary) to implement partial and total sum calculations so that $\frac{t_s}{Int} \leq O(\log P)$, see for example [DG90].

Now, if available local memory is large enough to allow us a larger MAX, we can introduce a delay between the end of a balancing phase and the beginning of the next sum calculation. Let WAIT be the number of instructions processed during this delay. We have to maintain the new capacity of balancing:

$$MAX - MIN \geq WAIT + \frac{t_b + t_s}{Int},$$

which leads to the following constraint:

$$WAIT \leq MAX - MIN - \frac{t_b + t_s}{Int}.$$

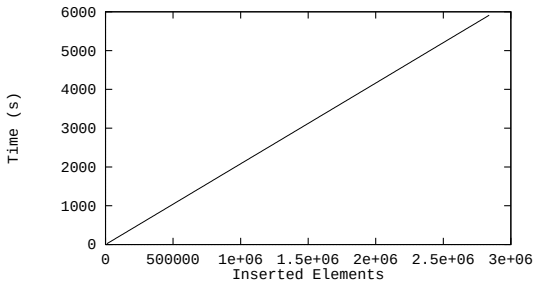


Figure 5: Filling time on 8 nodes.

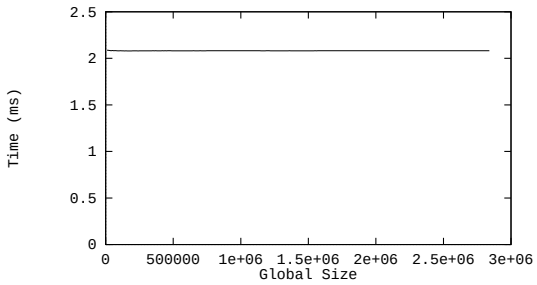


Figure 6: Int versus current population.

In our machine, with a reasonable value of $MAX = 1000$, we can process 920 instructions during this delay (0.7 seconds). Clearly, we can increase this delay by increasing MAX.

We remark that a finer analysis of the behavior of the queues storing instructions between the two processors inside each node is not necessary. In fact, with an appropriate value of Int, these queues are always empty with the exception of the duration of the balancing phases. Queues are made empty at the beginning of the next processing phase. In our target machine, we fix the value of Int to t_c since $t_c > t_i$. With a better computation/communication ratio, Int has to be greater than $\max(t_c, \frac{t_i}{P})$.

Further, we can easily calculate a lower bound for the response time of the dictionary machine: $(P+1)t_c + t_i$. This value is close to the average response time experimentally measured.

3.5 Experimental results

Our balancing mechanism has a small cost. To ensure an imbalance of up to 1% ($MAX = 3600$ for a local capacity of 360000), our machine spends 3% of its time in balancing operations.

The following experimental results were obtained with the values: $MIN = 0$, $MAX = 5000$, $L = 4$, $P = 8$, and waiting time $WAIT \times Int = 3.9s$. As the memory of a node is 16 MBytes, the capacity of our dictionary is 360000 elements per node.

We can see in Figure 5 the time to "fill up" the machine. It corresponds to the time spent for the machine to insert a given amount of elements. It is drawn by a line, up to the capacity of the dictionary (2.88 million elements in this configuration).

In Figure 6, obviously, the minimum possible pipeline interval is constant whatever the current population is in the dictionary. The throughput is independent of the amount of data already inside the machine. This is explained by t_c being much greater than t_i in this machine, thus $Int = t_c$ that is a constant.

Figure 7 shows that the communication corresponds to the model proposed: the minimum possible value for the pipeline interval increases linearly with the size of the processed elements.

In Figure 8, we can see that the pipeline interval increases slightly with the number of processors involved in the dictionary. We could hope that the number of processors would not change the pipeline interval since, indeed, processors are along the pipeline. But this phenomenon is justified by

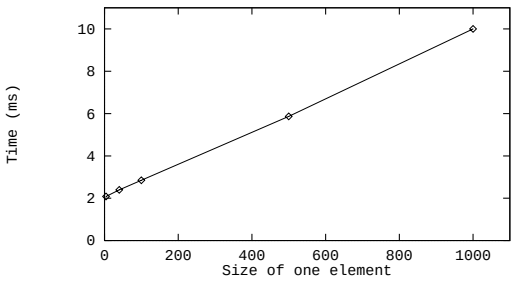


Figure 7: Int vs L.

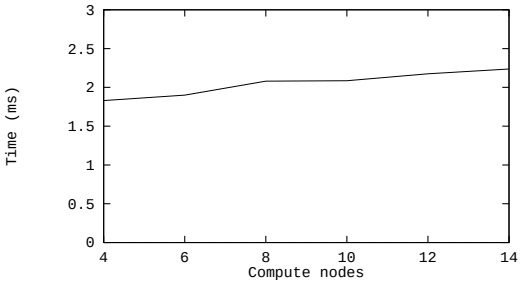


Figure 8: Int vs P.

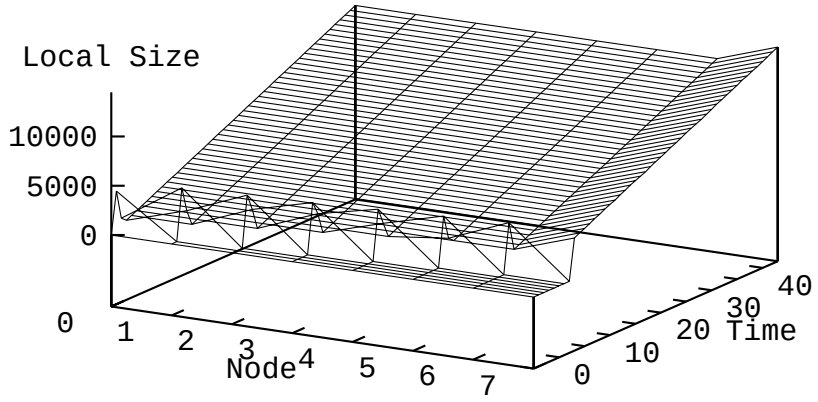


Figure 9: Behavior for key-increasing insertions.

the machine-dependent remark that β increases abnormally with the number of user tasks running on the Volvox.

Figures 9 and 10 represent the behavior of the machine quantified by the evolution of the current amount of elements in each node during series of insertions processed.

The first one (Figure 9) corresponds to series of insertions of elements with keys that increase continuously. We can verify that, even in that extreme case, after a first period where not enough data were inserted (as explained in the analysis), our dictionary machine becomes and remains balanced.

Finally, the other situation visualized in Figure 10 corresponds to series of insertions performed in a random order for a while (only a small part of these insertions are visualized in the plot), and suddenly changed to an increasing order like in the previous figure. We can see that even when the distribution of the keys inserted changes, the structure stays balanced.

4 Conclusion

Multicomputers represent the cutting edge technology for database applications. Unfortunately, implementing dynamic data partitioning seems to be a very challenging problem. In this paper we gave a first step towards its solution, by presenting an implementation of a dictionary machine – a

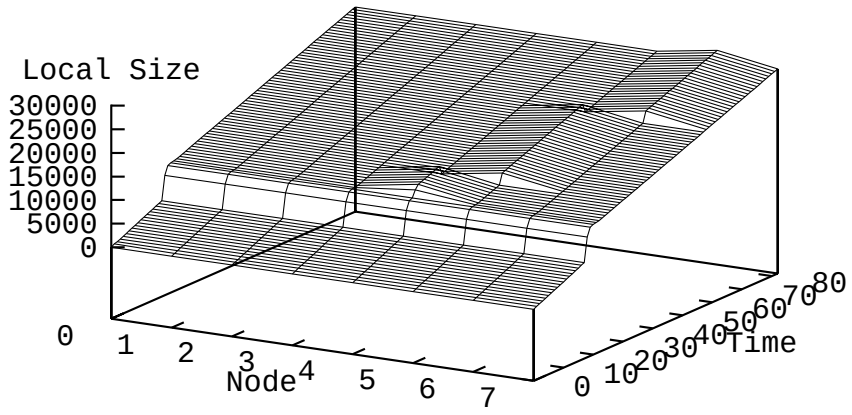


Figure 10: Behavior for random then key-increasing insertions.

database reduced to its simplest instance –, on a distributed memory architecture. Our goal was to achieve good performance, which required a dynamic balancing scheme. Using this scheme and exploiting the capacities of the target architecture, we have proved that choosing an adapted local data structure, good performance can be attainable for information maintenance and retrieval on a general purpose parallel architecture.

More generally, we have proposed a powerful balancing strategy, and we have proved that it solves the load balancing problem. This strategy induces a very reasonable overhead and can be used with success in many other situations, where a bad dynamic data distribution would lead to weak performance.

A possible extension to our load balancing technique should be to adapt the number of processors (P) to the current population of the dictionary machine. By this way, the efficiency of the machine would be increased while currently storing only a small number of elements.

In the near future, we hope to implement the techniques shown in this paper into a parallel database application. Interesting applications are evolutionary ones, where the insertions and deletions of data are significant compared to the amount of information retrievals (queries).

Acknowledgments

The authors would like to thank Nashib Qadri whose comments have greatly improved the quality of presentation of the manuscript.

References

[AHU83] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *Data structure and algorithms*. Addison-Wesley, 1983.

[AK85] M.J. Atallah and S.R. Kosaraju. A generalized dictionary machine for VLSI. *IEEE Trans. on Computers*, 34:151–155, 1985.

[DeW90] D.J. DeWitt et al. The GAMMA database machine project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), June 1990.

[DFG92] T. Duboux, A. Ferreira, and M. Gastaldo. MIMD dictionary machines : from theory to practice. In Bouge et al., editor, *Parallel Processing: CONPAR 92 - VAPP V*, number 634 in LNCS, pages 545–550. Springer-Verlag, 1992.

[DFG94] T. Duboux, A. Ferreira, and M. Gastaldo. A scalable design for dictionary machines. In *3rd International Workshop on Algorithms and Parallel VLSI Architectures*, Leuven, Belgium, August 1994.

[DG90] F. Dehne and M. Gastaldo. A note on the load balancing problem for coarse grained hypercube dictionary machines. *Parallel Computing*, 16:75–79, 1990.

[DG92] D.J. DeWitt and J. Gray. Parallel database systems : the future of high performance database systems. *Communications of the ACM*, 35(6), June 1992.

[DS89] F. Dehne and N. Santoro. An optimal VLSI dictionary machine for hypercube architectures. In M. Cosnard, editor, *Parallel and Distributed Algorithms*, pages 137–144. North Holland, 1989.

[FC91] Z. Fan and K.-H. Cheng. A generalized simultaneous access dictionary machine. *IEEE Trans. on Parallel and Distributed Systems*, 2(2):149–158, 1991.

[Fis84] A. Fisher. Dictionary machines with small number of processors. In *Int. Symp. on Computer Architecture*, pages 151–156, June 1984.

[Gas93] M. Gastaldo. Dictionary Machine on SIMD Architectures. Technical Report RR 93-19, LIP ENS-Lyon, July 1993. Submitted to Publication.

[Knu72] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley Publishing Co., 1972.

[Lei79] C. E. Leiserson. Systolic priority queues. Rep. CMU-CS-79-115, Dept. Comp. Sci., Carnegie-Mellon University, 1979. also in Proc. Caltech VLSI Conf., Jan. 1979.

[LP90] H.F. Li and D.K. Probst. Optimal VLSI dictionary machines without compress instructions. *IEEE Trans. on Computers*, 39:332–340, 1990.

[Meh84] K. Mehlhorn. *Data structure and Algorithms I : Sorting and searching*. Springer Verlag, 1984.

[Nar86] T.S. Narayanan. A survey of dictionary machines. Technical Report CSD-86-2, Concordia University, 1986.

[OB87] A.R. Omondi and J. D. Brock. Implementing a dictionary on hypercube machines. In *Int. Conf. on Parallel Processing*, pages 707–709, 1987.

[ORS82] T. Ottmann, A. Rosenberg, and L. Stockmeyer. A dictionary machine (for VLSI). *IEEE Trans. on Computers*, c-31(9):892–897, sep 1982.

[SA85] A. Somani and V. Agarwal. An efficient unsorted VLSI dictionary machine. *IEEE Trans. on Computers*, C-34(9):841–851, September 1985.

[SL87] A.M. Schwartz and M.C. Loui. Dictionary machines on cube-class networks. *IEEE Trans. on Computers*, 36:100–105, 1987.

[SPKO88] M. Stonebraker, D. Patterson, R. Katz, and J. Ousterhout. The design of XPRS. In *International Conference of Very Large Databases*, Los Angeles, California, August 1988.

[SS85] H. Schmeck and H. Schroder. Dictionnary machine for different models of VLSI. *IEEE Trans. on Computers*, C-34(5):472–475, May 1985.

[Val93] Patrick Valduriez. Parallel database systems : Open problems and New issues. *Distributed and Parallel Databases*, 1(2), April 1993. Kluwer Academic Publishers.

[VCB91] P. Valduriez, M. Couprie, and B.Bergstein. Prototyping DBS3, shared-memory parallel database system. In *International Conference on Parallel and Distributed Information System*, Miami Beach, Florida, 1991.