Parallel Computing 16 (1990) 75–79
North-Holland

Short communication

A note on the load balancing problem for coarse grained hypercube dictionary machines *

Frank DEHNE † and Michel GASTALDO ‡

† *Center for Parallel and Distributed Computing, School of Computer Science, Carleton University, Ottawa, Canada K1S 5B6*
‡ *Laboratoire de l'Informatique du Parallelisme - IMAG, Ecole Normale Superieure de Lyon, 69364 Lyon cedex 07, France*

Received March 1990

Abstract. The main problem for the design of dictionary machines on coarse grained hypercube multiprocessors, in comparison to the widely studied dictionary problem for fine grained hypercube multiprocessors, is that due to unequal distribution of the inserted and deleted records, the sizes of the sets stored at the individual processors may vary considerably. This problem, which is usually referred to as the *load balancing problem*, may lead to considerable degradation of the dictionary machine's performance. In this note we show that the load balancing problem for coarse grained hypercube dictionary machines can be solved with provable bounds on the sizes of the data sets, and with only little computational overhead.

Keywords. Dictionary machine, Load balancing, Hypercube multiprocessors, 2-3 trees, Performance.

1. Introduction

We present a solution to the load balancing problem arising in the design of dictionary machines for coarse grained hypercube multiprocessors.

The use of parallel architectures for implementing dictionary machines has been widely studied in the literature. Some authors design specialized architectures (e.g. [1,2,3,6,11,12]) while others use existing and widely available processor networks (e.g. [4,5,10,13]. Particular attention has been given to the problem of designing dictionary machines for hypercube multiprocessors [5,10,13]. A common feature of the presented solutions is that the records currently stored in the dictionary are stored in sorted order according to a pre-defined sequence of processors (called e.g. "snake" and "sorted chain" in [5] and [10], respectively). This sorted order is used to detect and avoid redundant deletions and duplicate insertions, respectively. The presented methods assume fine grained hypercube multiprocessors; i.e. every processor stores

* The first author's research is partially supported by the Natural Sciences and Engineering Research Council of Canada (Grant A9173). The second author's research is partially supported by the Direction de la Recherche et des Etudes Techniques and the Programme de Recherche Coordonnees C3 (France).

at most one record. In practice, one would often like to use coarse grained hypercube multiprocessors; that is, one would like to store a set of records at each processor. For the above methods, the snake (or sorted chain) of records would be replaced by a sorted sequence of sets of records (each of them usually represented as a balanced search tree). The main additional problem arising here, which is commonly referred to as the *load balancing problem*, is that due to unequal distribution of the inserted and deleted records, the sizes of the sets may vary considerably. This may lead to considerable degradation of the dictionary machine's performance.

In [10] it is noted that one way of solving the problem is a periodic "shut down" of the dictionary machine followed by a global rebalancing phase (which is obviously not very desirable) and that "local balancing on the fly is considerable more difficult". They sketch a heuristic for shifting records between processors with unbalanced data sets, but do not give any performance guaranties.

In this note we show that the load balancing problem for coarse grained hypercube dictionary machines can be solved with provable bounds on the sizes of the data sets and with only little computational overhead. We present our result for the dictionary machine described in [5]; it can however be easily adapted to the solutions shown in [10] and [13].

2. Balancing sequences of 2-3 trees for coarse grained hypercube dictionary machines

The hypercube dictionary machine presented in [5] consists of two structures, a snake and a broadcast net which are both embedded into a d-dimensional hypercube (with $p = 2^d$ processors) and operate on it simultaneously. The operations *insert, delete, find-min*, and *extract-min* are handled by the snake, while *search* operations are handled by the broadcast net (see [4,5] for more details). The snake, which is basically a systolic priority queue [8], is embedded into the hypercube as shown in Fig. 1a and 1b (note that, in order to detect/avoid redundant deletions/duplicate insertions, the data elements are stored in sorted order with respect to the snake).

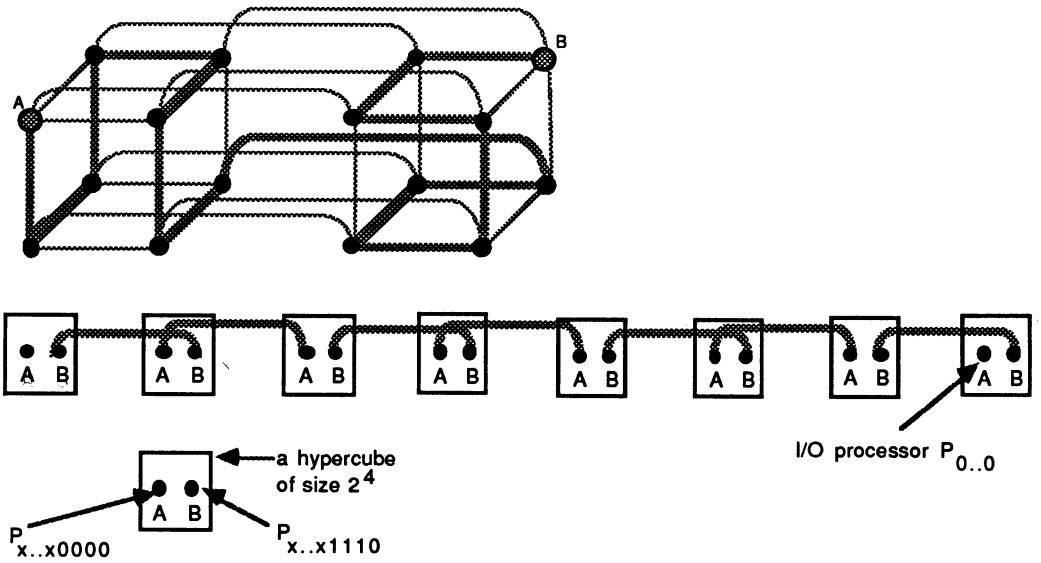

Fig. 1. a. Local structure of the snake in each 4-dimensional subcube. b. Global structure of the snake in a 7-dimensional hypercube.

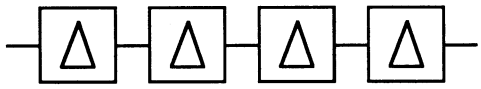

Fig. 2. A snake of processors each storing a 2-3 tree.

Let us now consider the coarse grained case, $n \gg O(p)$, where n is the number of data items stored in the dictionary. For the remainder we will assume that $n \geqslant 2\ p\ d$. As indicated above, each processor will now store a set of data items as a balanced search tree; here, we will use 2-3 trees. Let us assume that at the beginning the global data structure is balanced, i.e. each processor stores the same number of entries. Note that these sets are sorted with respect to the snake ordering (i.e. all items in processor P_i are smaller than all items in P_j, if P_i is a predecessor of P_j).

The load balancing problem now consists of keeping the sizes of the different search trees balanced while insertions and deletions into/from the data sets occur. In addition to having to shift around data items, a major problem is for the processors to determine whether they are unbalanced and to compute their correct size. Note that, since we allow duplicate insertions and redundant deletions, and because of the communication delay, it is impossible for each processor even to know, at any point in time, the total number of records currently stored in the dictionary.

Our approach to solving this problem is to compute at every processor approximate values of the current total number of records as well as the number of records to be shifted to the neighbors. We will ensure, however, that this information is precise enough to guarantee that the sizes of two arbitrary trees differ by at most $O(\log p + \log m)$, where m is the size of the largest tree in the dictionary machine. This, in turn, guarantees that the depths of two arbitrary trees differ by at most O(1).

The general structure of our approach is as follows: Assuming that the sizes of two trees differ by at most $O(\log p + \log m)$, we present an $O(\log p + \log m)$ time hypercube algorithm to rebalance the trees to equal size (if the trees where not changed during this period). If this algorithm is concurrently executed with the general dictionary machine algorithm, then after $O(\log p + \log m)$ steps the dictionary machine algorithm can only create $O(\log p + \log m)$ imbalance between two arbitrary trees. Thus, if our rebalancing algorithm is always restarted immediately after its termination, the sizes of two trees will never differ by more than $O(\log p + \log m)$.

In the remainder, we will outline our $O(\log p + \log m)$ time rebalancing algorithm. It consists of two phases: *approximate counting* and *shifting subtrees*. In our model, one exchange of an O(1) length message between two neighboring processors takes time O(1), and we account for communication as well as for local processing time.

3. Approximate counting

We assume that during the entire dictionary machine operation, every processor keeps a counter of the current size of its local tree. At the beginning of the first phase, every processor creates a copy (σ) of that counter, and then for every processor p the following three numbers are computed:

n: the sum of the sizes (σ) of all trees

n_p the sum of the sizes (σ) of all preceding trees (with respect to the snake), and

n_s: the sum of the sizes (σ) of all succeeding trees (with respect to the snake).

We observe that, except for the 4-dimensional subcubes, the processor numbers of the processors in the snake sequence increase monotonically (see Fig. 1b). Therefore, the above

operation is essentially equivalent to a partial sum operation (see [7] for more details) and can hence be executed in time $O(\log p)$.

4. Shifting subtrees

With the above information, every processor determines the number $s \leqslant O(\log p + \log m)$ of records it has to shift to the left or right, and from our initial assumption it follows that its tree contains at least s records. What remains to be shown is how to shift these records in time $O(\log p + \log m)$ while maintaining the 2-3 tree structures at all processors.

Clearly, if we would shift every record individually and insert it into the neighboring tree, the required time would be $O((\log p + \log m) \log m)$; i.e. it would be a factor of $O(\log m)$ too slow. We will therefore proceed as follows:

Assume that a processor P_i has to send $s \leqslant O(\log p + \log m)$ records to its left neighbor and receive $r \leqslant O(\log p + \log m)$ records from its right neighbor (in the snake):

(1) Processor P_i cuts the s smallest records from its 2-3 tree T, rebuilds these s records into a 2-3 tree T', and rebuilds the remainder into a valid 2-3 tree T''. It then sends the 2-3 tree T' to its left neighbor.

(2) Processor P_i receives a 2-3 tree T' of size r from its right neighbor and merges it with its tree T'' into a new 2-3 tree.

An example for the cutting and rebuilding operation referred to in Step 1 is given in Fig. 3. We assume that at every node of the 2-3 tree we store, in addition to the minimum values of the middle and right subtree, also the minimum of the left subtree, the maximum of the right subtree, as well as the total number of leaves in all (at most three) subtrees. First, we navigate from the root down the tree to the $s + 1$st leaf (leaf number 10 in Fig. 3). Then we rebuild both trees, T' and T'', in a bottom-up fashion similar to the standard insertion and deletion procedure for 2-3 trees. Note that at each level, only a constant number of links and nodes need to be shifted across the border of the two trees (or copies of nodes need to be created).

Summarizing we obtain that, assuming that the sizes of the subtrees differ by at most $O(\log p + \log m)$, the sequence of subtrees can be rebalanced in time $O(\log p + \log m)$. Note that, since the cutting and rebuilding operations (without transmission of the subtrees) can be executed in the same time as insertions or deletions into/from the 2-3 trees, problems with concurrent insertions and deletions are avoided.

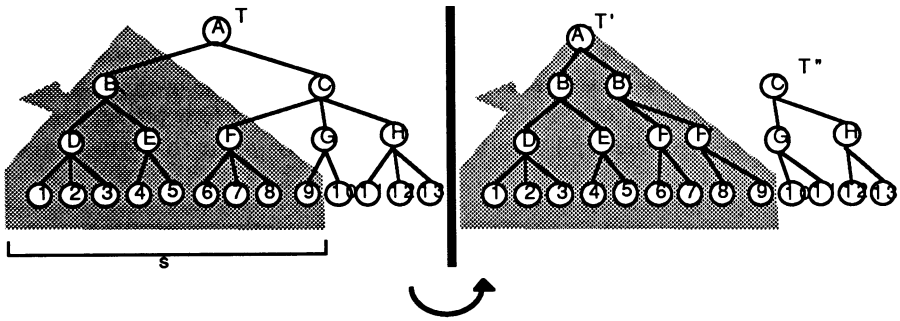

Fig. 3. An example for cutting and rebuilding 2-3 trees.

5. Conclusion

In this note we have shown that the load balancing problem for coarse grained hypercube dictionary machines can be solved with provable bounds on the sizes of the data sets and with only little computational overhead. Assuming that the sizes of two data sets differ by at most $O(\log p + \log m)$, we presented an $O(\log p + \log m)$ time hypercube algorithm for rebalancing the sets to equal size (if they where not changed during this period). If this algorithm is concurrently executed with the general dictionary machine algorithm, then after $O(\log p + \log m)$ steps the dictionary machine algorithm can only create $O(\log p + \log m)$ imbalance between two arbitrary data sets. Thus, if our rebalancing algorithm is always restarted immediately after its termination, the sizes of two data sets will never differ by more than $O(\log p + \log m)$.

References

[1] M.J. Atallah and S.R. Kosaraju, A generalized dictionary machine for VLSI, *IEEE Trans. Comput.* 34 (2) (1985) 151–155.
[2] J.L. Bentley and H.T. Kung, A tree machine for searching problems, *Proc. 1979 Internat. Conf. Parallel Processing* 1979 (May 1981) 257–266.
[3] J.H. Chang, O.H. Ibarra, M.J. Chung and K.K. Rao, Systolic tree implementation of data structures, *IEEE Trans. Comput.* 37 (6) (1988) 727–735.
[4] F. Dehne and N. Santoro, Optimal VLSI dictionary machines on meshes, *Proc. Internat. Conf. Parallel Processing* (1987) 832–840.
[5] F. Dehne and N. Santoro, An optimal VLSI dictionary machine for hypercube architectures, in: *Parallel and Distributed Algorithms*, M. Cosnard, ed. (North-Holland, Amsterdam, 1989) 137–144.
[6] A.L. Fisher, Dictionary machines with small number of processors, *Proc. Internat. Symp. Computer Architecture* (June 1984) 151–156.
[7] C.P. Kruskal, L. Rudolph and M. Snir, The power of parallel profix, *Proc. Internat. Conf. Parallel Processing* (1985) 180–184.
[8] C.E. Leiserson, Systolic priority queues, Report CMU-CS-79-115, Carnegie-Mellon University, April 1979.
[9] D. Nassimi and S. Sahni, Brodcasting data in SIMD Computers, *IEEE Trans. Comput.* 30 (2) (1981) 101–107.
[10] A.R Omondi and J.D. Brock, Implementing a dictionary on hypercube machines *Proc. Internat. Conf. Parallel Processing* (1987) 707–709.
[11] T.A. Ottman, A.L. Rosenberg and L.J. Stockmeyer, A dictionary machine for VLSI, *IEEE Trans. Comput.* 31 (9) (1982) 892–897.
[12] A.K. Somani and V.K. Agarwal, An efficient unsorted VLSI dictionary machine, *IEEE Trans. Comput.* 34 (9) (1985) 841–852.
[13] A.M. Schwartz and M.C. Loui, Dictionary machines on cube-class networks, *IEEE Trans. Comput.* 36 (1) (1987) 100–105.