RAIRO. INFORMATIQUE THÉORIQUE

W. PAUL
U. VISHKIN
H. WAGENER

Parallel computation on 2-3-trees

RAIRO. Informatique théorique, tome 17, nº 4 (1983), p. 397-404
<http://www.numdam.org/item?id=ITA_1983__17_4_397_0>

© AFCET, 1983, tous droits réservés.

L'accès aux archives de la revue « RAIRO. Informatique théorique » implique l'accord avec les conditions générales d'utilisation (http://www.numdam.org/conditions). Toute utilisation commerciale ou impression systématique est constitutive d'une infraction pénale. Toute copie ou impression de ce fichier doit contenir la présente mention de copyright.

NUMDAM
Article numérisé dans le cadre du programme
Numérisation de documents anciens mathématiques
http://www.numdam.org/

R.A.I.R.O. Informatique théorique/Theoretical Informatics
(vol. 17, n° 4, 1983, p. 397 à 404)

PARALLEL COMPUTATION ON 2-3-TREES (*)

by W. PAUL [1], U. VISHKIN [2], H. WAGENER [3]

Communicated by J. BERSTEL

Abstract. — *Our model of computation is a parallel computer with k synchronized processors* $P_1, \ldots, P_k$ *sharing a common random access storage, where simultaneous access to the same storage location by two or more processors is not allowed. Suppose a 2-3 tree T with n leaves is implemented in the storage, suppose* $a_1, \ldots, a_k$ *are data that may or may not be stored in the leaves, suppose* $a_1 \leqq \ldots \leqq a_k$ *and for all i processor* P_i *knows* a_i. *We show how to search for* $a_1, \ldots, a_k$ *in the tree T, how to insert these data into the tree and how to delete them from the tree in* O *(log n + log k) steps.*

Résumé. — *Notre modèle de calcul est un ordinateur parallèle avec k processeurs synchronisés* $P_1, \ldots, P_k$ *partageant une mémoire commune à accès aléatoire où un accès simultanément à la même adresse mémoire par deux processeurs ou plus est interdit. On suppose qu'un arbre 2-3 T à n feuilles est implanté dans la mémoire, que* $a_1, \ldots, a_k$ *sont des données qui peuvent être stockées dans les feuilles, on suppose que* $a_1 \leqq \ldots \leqq a_k$ *et que pour tout i, le processeur* P_i *connaît* a_i. *On montre comment chercher* $a_i, \ldots, a_k$ *dans l'arbre T, comment insérer ces données dans l'arbre et comment les supprimer de l'arbre en* O *(log n + log k) pas.*

1. INTRODUCTION

Technology will make it possible to build computers with a large number of cooperating processors in the near future. However, building such computers will only be worthwhile if the increased computing power can be used to reduce considerably the execution time of sufficiently many basic computational problems. In particular, one would like to have datastructures, where k processors can solve many problems about k times faster than a single processor. One such datastructure are 2-3-trees as will be demonstrated here.

(*) Received in April 1982, revised in February 1983. Part of this research was done while the first author was visiting the Institut de Programmation of the Université Paris-VI.

[1] IBM Research Laboratory, San Jose, California 95193.
[2] Courant Institute, New York University, 251 Mercer Street, New York, New York 10012.
[3] Technische Universität Berlin, Institut fuer Software und Theoretische Informatik, Franklinstr. 28/29, D-1000 Berlin 10, West Germany.

R.A.I.R.O. Informatique théorique/Theoretical Informatics, 0399-0540/1983/397/$ 5.00
© AFCET-Bordas-Dunod

Protocols that avoid read or write conflicts, if several processors are working simultaneously on the same balanced tree, have been studied previously [2, 3], but apparently no attempt was made to design fast algorithms and to analyse their running time.

In the sequel, we say very little about how to avoid read or write conflicts. In the situations where they are possible, there are easy ways to avoid them. We will, however, have to say some words about storage allocation.

2. 2-3-TREES

A 2-3-tree T is a tree in which all leaves have the same depth and each interior node v has two or three sons: the left son $l(v)$, the right son $r(v)$ and in case there are three sons the middle son $m(v)$. Data from a totally ordered domain are stored in the leaves with smaller data to the left of larger ones. For each node v, the value $L(v)$ of the largest element stored in the subtree of T with root $l(v)$ is stored in v. If v has three sons, then the value $M(v)$ of the largest element stored in the subtree of T with root $m(v)$ is also stored in v. The depth of a node in T is its distance from the root, the height of v is its distance from the leaves. We assume the reader to be familiar with the usual search, insertion and deletion routines as described say in [1].

3. SEARCH

A chain is a subsequence $a_f, a_{f+1}, \ldots, a_l$ of the input sequence $a_1, \ldots, a_k$. Such a chain corresponds in a natural way to a chain of processors P_f, $P_{f+1}, \ldots, P_l$. The search algorithm starts with the chain $a_1, \ldots, a_k$ at the root of the 2-3-tree T. This chain is subsequently split into many subchains which are wandering down the tree. Among the processors of a chain $a_f, \ldots, a_l$ only the first one, i. e., P_f, is active. P_f knows l and of course f. If at some time the chain is split into $a_f, \ldots, a_{m-1}$ and $a_m, \ldots, a_l$, then processor P_f will inform processor P_m and transmit the value l to P_m.

The search algorithm proceeds in stages. During each stage s, the active processor of each chain C will access the data in some node v of the 2-3-tree T. We say that C is in node v at stage s. The chain $a_1, \ldots, a_k$ is in the root at stage 1. During each stage, each active processor processes its chain once. We describe how this is done.

Suppose a chain $C = a_f, \ldots, a_l$ is in node v at stage s, the node v has two or three sons and the labels $L(v)$ and possibly $M(v)$ are stored in v. We say that C hits a label X, if $a_f \leqq X < a_l$.

Chains C that hit no label are sent to the appropriate son of v, more precisely: C is at stage $s+1$ in node:

$l(v)$ if $a_l \leqq L(v)$;

$m(v)$ if $L(v) < a_f$ and $a_l \leqq M(v)$ and v has 3 sons;

$r(v)$ if $M(v) < a_f$ and v has 3 sons;
or: $L(v) < a_f$ and v has 2 sons.

For chains $C = a_f, \ldots, a_l$ let $C_1 = a_f, \ldots, a_{m-1}$ and $C_2 = a_m, \ldots, a_l$ with $m = [(f+l)/2]$. If C hits a label, then it is split into C_1 and C_2. If C_i, $i = 1, 2$, hits no label, then it is sent to the appropriate son, else it remains in v, i. e., C_i is in node v at stage $s+1$.

Clearly, a chain can be processed in $O(1)$ steps. For all s, the chains which are in some node at stage s form—with a slight abuse of language—a partition of $a_1, \ldots, a_k$, and the partition in stage $s+1$ is a refinement of the partition in stage s. Hence, for each node v, at most two of the chains that are sent to v hit its labels. Part of these chains remain in v for some stages, their length is halved in each stage. All other chains that are sent to v are not split in v.

Let $A = a_f \ldots a_l$ and $A' = a_e \ldots a_k$ be chains. Define $A \leqq A'$ iff $a_l \leqq a_e$. By induction on the depth of u one verifies for all nodes u:

If v is a son of u and s is the number of any stage, then at most two chains are sent from u to v in stage s. If chain A is sent from u to v in stage s, then either $A \leqq A'$ for all chains A' sent from u to v during stages $s' < s$ or $A' \leqq A$ for all such chains A'. If chains A_1, A_2 are sent from u to v in stage s and $A_1 \leqq A_2$, then $A_1 \leqq A' \leqq A_2$ for all chains A' sent from u to v during stages $s' < s$.

This implies, that for each s and v at most four chains are in v at stage s. Thus, each stage lasts $O(1)$ steps. Moreover, for all s and d, every chain that is at stage s in a node of depth d has at most length $k \cdot 2^{-(s-d-1)}$. Once a chain $a_f, \ldots, a_l$ has arrived in a leaf b, the processors $P_{f+1}, \ldots, P_l$ have to be informed of the value of b. This is done recursively. If processor P_f wants to inform $P_{f+1}, \ldots, P_l$, it informs P_m of l and b, where $m = [(f+l)/2]$. Then in parallel P_f, informs $P_{f+1}, \ldots, P_{m-1}$ and P_m informs $P_{m+1}, \ldots, P_l$.

4. INSERTIONS

If $a_1, \ldots, a_k$ are to be inserted into T, we first run the search algorithm. This results in splitting the input into chains $C_f = a_f, \ldots, a_l$ that arrive in leaves say b_f for certain $f \in \{1, \ldots, k\}$. First, we describe a simple algorithm

for the special case where all chains consist of single elements and no chain is to the left of the smallest leaf in T. This algorithm works in stages. In stage 1 for all i processor P_i makes a_i a son of a father of b_i and then stands by on a_i. Now the algorithm works such that for all s after stage s the following holds:

All leaves in the tree have the same depth, all interior nodes of height $\neq s$ have two or three sons.

With the help of the processors that are standing by on nodes of height $s-1$ and modifying the standard insertion procedure for 2-3-trees in an obvious way, it is easy to organize stage s such that it works in O(1) steps and that only nodes of height $s-1$, s and $s+1$ are accessed. Care has to be taken in order to avoid several processors accessing the same node simultaneously and how to choose among the processors that are standing by those that do the work (the others become inactive). This is easy. Also, in each stage, several new nodes of the tree may be created simultaneously. We will say later how to do this without occupying too much storage space. If a_1 is smaller than the smallest leaf in T, we insert it first and then proceed as above.

The problem of inserting a long chain $a_f, \ldots, a_l$ at a leaf b_f is reduced to the problem of inserting shorter chains by first inserting the middle element $a_m (m=[(f+l)/2])$ at leaf b_f and then inserting $a_f, \ldots, a_{m-1}$ at a_m and $a_m, \ldots, a_l$ at b_f. This is done for all chains in parallel and the middle elements are inserted by the simple algorithm described above. After the chains have been split $\log k$ times, they are reduced to length one. Thus, running the simple algorithm $\log k$ times would do the job in $O(\log n \log k)$ steps. For $i \leqq \log k$, let T_i be the tree obtained by running the simple algorithm i times. Now for all i running the simple algorithm the i'th time results in a wave of processors running up T_{i-1} at a speed of one level per stage, and below this wave, the tree already looks like T_i. Thus, before starting the $(i+1)-$st run of the simple algorithm and with it the $(i+1)-$st wave of processors, one has not to wait until the i'th wave has reached the root, but only long enough to ensure that the two waves will not overlap. Three stages will certainly suffice.

5. DELETIONS

Like in the case of insertions, we will parallelize and pipeline a suitable sequential deletion algorithm, i. e., an algorithm which makes a single processor p delete a single element a from a 2-3-tree T. Consider the following algorithm:

1. Using the standard search algorithm, find the path $\pi(a)=(p_1, \ldots, p_t)$ from the root of T to a. Delete the whole path $\pi(a)$ and all edges adjacent to

it from T. One is left with a forest of subtrees of T some of which were to the left of the path $\pi(a)$ and the others were to the right of $\pi(a)$. Let us call these subtrees of T the *left*, resp. *right*, *side trees* of path $\pi(a)$.

If a was not stored in the tree, then a was eventually compared to a leaf b of T with the result $a<b$ or $a>b$. In the first [second] case treat b and possibly its right [left] brothers as right [left] side trees of $\pi(a)$.

2. Join the left [right] side trees of $\pi(a)$ into a 2-3-tree L [R].

3. Join L and R into a 2-3-tree.

If the tree T has n leaves, all this can be done in $O(\log n)$ steps [1].

We now parallelize and pipeline this algorithm in order to delete simultaneously elements $a_1, \ldots, a_k$ from a 2-3-tree T:

1. Run the search algorithm for $a_1, \ldots, a_k$ and mark the paths $\pi(a_1), \ldots, \pi(a_k)$. As we do not require the elements a_i to be stored in the tree T, these paths are not necessarily distinct. These paths cut T into $k+1$ pieces. The notion of left and right forests defined below suggests a further refinement of these $k+1$ pieces into 2 k pieces.

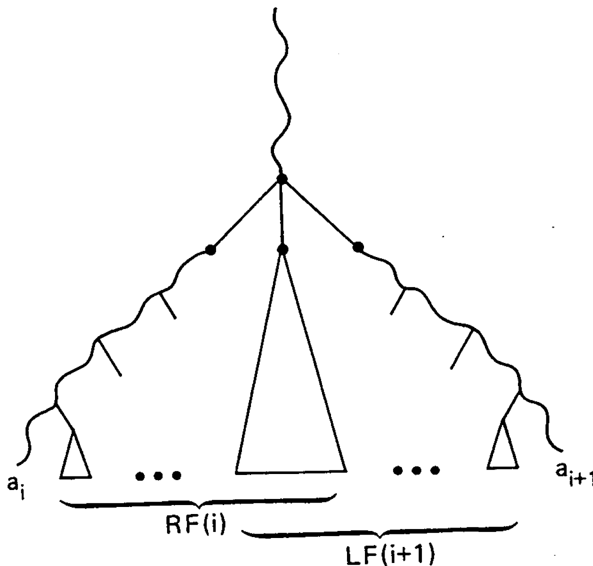

Figure 1

For all i, we define the *left* [*right*] *forest* $LF(i)$ [$RF(i)$] of path $\pi(a_i)$ as the set of left [right] side trees of $\pi(a_i)$ whose root is not marked and that are not left [right] side trees of $\pi(a_{i-1})$ [$\pi(a_{i+1})$]. The example of figure 1 shows that $RF(i)$ and $LF(i+1)$ may have a tree in common.

2. Rerun the search algorithm, but for all chains $a_f, \ldots, a_l$ that are created have the processors p_f and p_l both active. Processor $P_f\,[P_l]$ keeps track which trees are in $LF(f)\,[RF(l)]$. Also delete in this run the paths $\pi a_1), \ldots, \pi(a_k)$ and the adjacent edges. For all chains $a_f \ldots a_l$ that have reached a leaf only processor P_f remains active. P_f remembers the index $f'=l+1$ of the next active processor. The following commands are only executed by active processors P_i:

3. For all i processor P_i joins the left forest of path $\pi(a_i)$ into a 2-3--tree L_i. [For all i such that $RF(i) \cap LF(i') \neq \emptyset$ consider $L_{i'}$ now as a tree in $RF(i)$.]

4. For all i processor P_i joins the right forest of path $\pi(a_i)$ into a 2-3-tree R_i. [For all i such that $RF(i) \cap LF(i') \neq \emptyset$ we have now $R_i = L_{i'}$.]

5. For any tree T, let $\lambda(T)\,[\rho(T)]$ denote the leftmost [rightmost] leaf of T. For all i processor P_i determines $\lambda(L_i)$ and $\rho(L_i)$.

6. For all i processor P_i determines $\lambda(R_i)$ and $\rho(R_i)$.

7. For all i such that $\rho(R_i) \neq \rho(L_{i'})$ processor P_i joins L_i and R_i into a 2-3-tree T_i. For all i such that $\rho(R_i) = \rho(L_{i'})$ let $T_i = L_i$. For each i such that P_i was inactive, let T_i be an empty tree. Reactivate all processors.

8. We are left with the problem of joining the trees $T_1, \ldots, T_{k+1}$. This will be done in phases $j=0,1, \ldots$

The following is true for $j=0$ and will remain true: At the beginning of phase j we are left with $[k/2^j]+1$ trees $T_1^{(j)}, T_2^{(j)}, \ldots$, where for each $\nu \in \{1, \ldots, [k/2^j]+1\}$ the tree $T_\nu^{(j)}$ is a 2-3-tree obtained by joining $T_{(\nu-1)2^j+1}, \ldots, T_{\nu 2^j}$. For each ν we have not yet used processor $P_{\nu 2^j}$ and processor $P_{\nu 2^j}$ knows $\rho(T_\nu^{(j)})$ and $\lambda(T_{\nu+1}^{(j)})$.

The following is done in phase j: For each odd ν processor $P_{\nu 2^j}$ runs up the right branch of $T_\nu^{(j)}$ and the left branch of $T_{\nu+1}^{(j)}$ and joins the two trees into $T_{(\nu+1)/2}^{(j+1)}$. Notice:.

$$\lambda(T_{(\nu+1)/2}^{(j+1)}) = \begin{cases} \lambda T_\nu^{(j)}) & \text{if } T_\nu^{(j)} \neq 0; \\ \lambda(T_{(\nu+1)}^{(j)} & \text{otherwise.} \end{cases}$$

In the second case $P_{\nu 2^j}$ informs $P_{(\nu-1)2^j}$.

$$\rho(T_{(\nu+1)/2}^{(j+1)}) = \begin{cases} \rho(T_{\nu+1}^{(j)}) & \text{if } T_{(\nu+1)}^{(j)} \neq 0; \\ \rho(T_\nu^{(j)}) & \text{otherwise.} \end{cases}$$

In the second case $P_{\nu 2^j}$ informs $P_{(\nu+1)2^j}$.

Finally, observe that the phases can be pipelined, i. e., for all j, phase $j+1$ can be started a constant number of steps after phase j.

6. STORAGE ALLOCATION

Nodes of a 2-3-tree with n leaves are stored in the first say N rows of some two-dimensional array A. During the insertion algorithm, each processor P_i may create $n_i \leqq \log n$ new nodes. Therefore, for each processor P_i $\log n$ consecutive rows of some other array B are reserved, where the new nodes are created. For each i let $N_i = \sum_{j<i} n_j$. After the insertion algorithm, the numbers N_i are computed in parallel and for all i processor P_i copies the nodes that it created into rows $N+N_{i+1}, \ldots, N+N_i+n_i$ of A.

During part 2 of the deletion algorithm, each processor P_i may cancel $m_i \leqq \log n$ nodes, i. e., rows in A. Each processor P_i stores the numbers of these rows in its private memory in the array B. After part 2 of the deletion algorithm $M = \sum_{i=1}^{k} m_i$ is computed. Now the rows with numbers $>N-M$ that were not cancelled have to be copied into the rows with numbers $\leqq N-M$, that have been cancelled: Rows $N-M+1, \ldots, N$ of A are partitioned into blocks $B_1, \ldots, B_k$, each consisting of at most $\log n$ consecutive rows of A. Each processor P_i determines the number d_i of rows in B_i that were not cancelled. The numbers $D_i = \sum_{j<i} d_j$ are computed.

Next, each processor P_i determines the set ρ_i of indices $\leqq N-M$ that were cancelled by processor P_i and its cardinality r_i. The numbers $R_i = \sum_{j<i} r_j$ are computed. Each processor P_i writes the indices in ρ_i in places $R_i+1, \ldots, R_i+r_i$ of some array C. Once all processors are done with this, P_i copies the rows of block B_i that were not cancelled in those rows of A whose indices are in places $D_i+1, \ldots, D_i+d_i$ of array C.

Later in the deletion algorithm, every processor may create O ($\log n$) new nodes. Storage allocation is handled as in the case of insertions.

ACKNOWLEDGEMENTS

The authors thank Professor J. Berstel for inspiring discussions.

REFERENCES

1. A. AHO, J. HOPCROT and J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Massachusetts, 1976.
2. R. BAYER and M. SCHKOLNICK, *Concurrency of Operations on B-Trees*, Acta Informatica, Vol. 9, 1977, pp. 1-21.
3. Ellis C. SCHLATTER, *Concurrent Search and Insertion in* 2-3-*Trees*, Acta Informatica, Vol. 14, 1980, pp. 63-86.
4. H. WAGENER, *Parallele Bearbeitung von* 2-3-*Bäumen*, Diplomarbeit, Fakultät für Mathematik, Universität Bielefeld, 1982.