STUDIA UNIV. BABEŞ-BOLYAI, MATHEMATICA, XXXIX, 3, 1994

DISTRIBUTED PROCESSING IN EXTENDED B-TREE

Florian Mircea BOIAN* **and Alexandru VANCEA***

Dedicated to Professor Sever Groze on his 65th anniversary

Received: February 10, 1995
AMS subject classification: 68Q22, 65Y05, 65Y10

REZUMAT. - Procesarea distribuită în B-arbori extinşi. În această lucrare se arată că structura de B-arbore este o structură de date foarte indicată pentru procesarea sa într-un mediu distribuit în care comunicarea se face prin transmitere de mesaje. În acest context se propun unele tehnici de procesare distribuită în B-arbori, tehnici inspirate de algoritmii clasici de mapare a taskurilor într-un sistem distribuit. Nu poate fi stabilită o tehnică optimă în cazul general, problema în acest caz fiind o problemă NP-hard.

1. Preliminaries

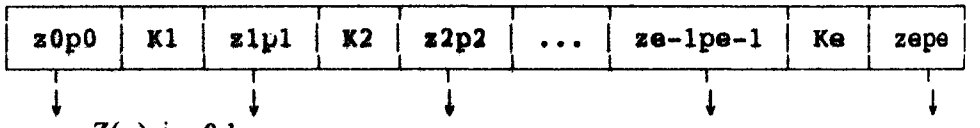
A B-tree was formally defined in [Knuth76]. We denote by **m** the order of the B-tree, and we denote by **e** the number of keys from the current B-tree node. By p, with possible subscripts we denote pointers to B-tree nodes. Finally, by **K**, with possible subscripts, we denote value(s) of key(s) from B-tree. If p is a pointer to a B-tree node, we denote by S(p) the B-subtree having the root in the node pointed by p.

Definition 1. The **possession** of S(p) is defined as the total number of keys from S(p). We denote this number by Z(p).

Let $a = K_{i+1}K_{i+2}...K_{i+r}$ be the word of the r succesive keys from a particular node of B-tree. Let p_i, p_{i+1}, p_{i+2}, ..., p_{i+r} be the neighbour pointers for the keys from a. By S(a) we denote the B-subtree which has in its root only the keys from a and the descendents $S(p_i)$, $S(p_{i+1})$, $S(p_{i+2})$, ..., $S(p_{i+r})$.

* *"Babeş-Bolyai" University, Faculty of Mathematics and Computer Science, 3400 Cluj-Napoca, Romania*

Definition 2. An **Extended B-tree** [Boian89] is a B-tree having in its nodes the following information:



where $z_i = Z(p_i)$, i = 0,1,...,n.

An example. In figure 1, an extended B-tree is presented. In each node, only the values of keys are presented. For leafless nodes there are two arrows near each key: one on the left and the other on the right. On the left of each row, in brackets, the value of possession appears, and on the right, the value of the pointer (here is the number of the node) appears.

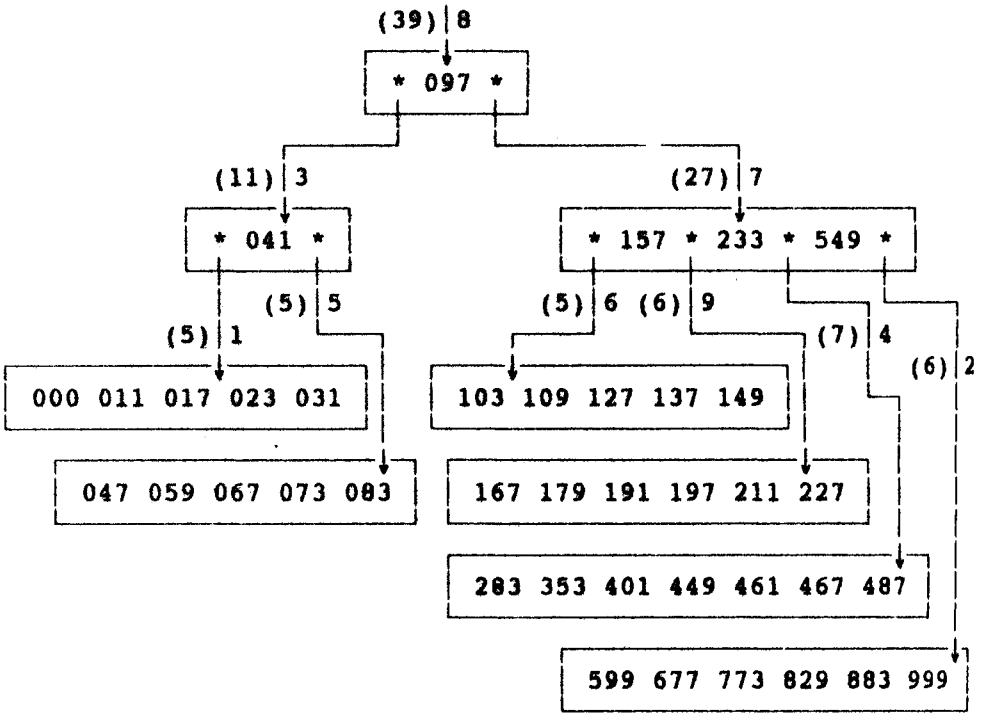


Figure 1. An extended B-tree

For example, S(7) has the nodes 7, 6, 9, 4, 2, and Z(7) = 27. If a = "157 233", then S(a) is S(7) without the key "549" and without the node 2, and Z(a) = 20.

Extended B-tree transformation. The operations with B-tree are presented in [Knuth76]. In [Boian89] and [Boian89a] we have described some ideas to implement an extended B-tree. In figures 2, 3 and 4 three pairs of transformations are presented: rotate left/right, transform a node into two or viceversa and the transformation of two nodes into three or viceversa. In these figures, we denote by lower case (a, b, ..., h, i) the sequences, possibly empty, from consecutive keys (from the same node), and by an uppercase (C, E, G) a key from a node.

When a transformation is applied, the possessions for new nodes must be computed only from the old ones, without considering the other nodes from the B-tree. In the following, for the four usual transformations, the new possessions are:

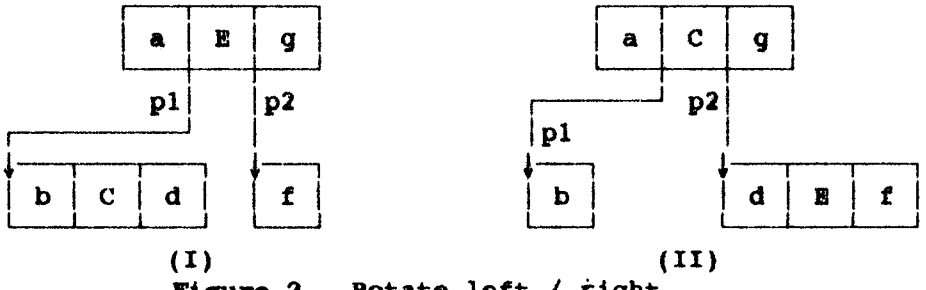


Figure 2. Rotate left / right

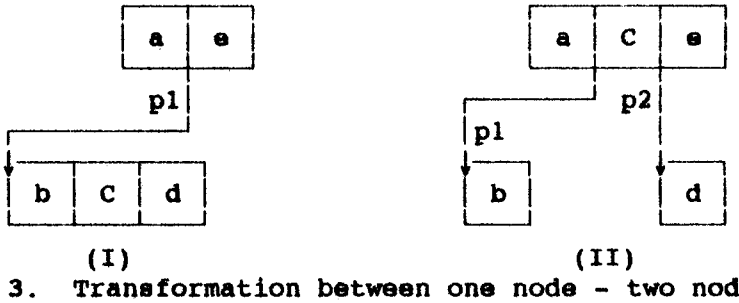


Figure 3. Transformation between one node - two nodes

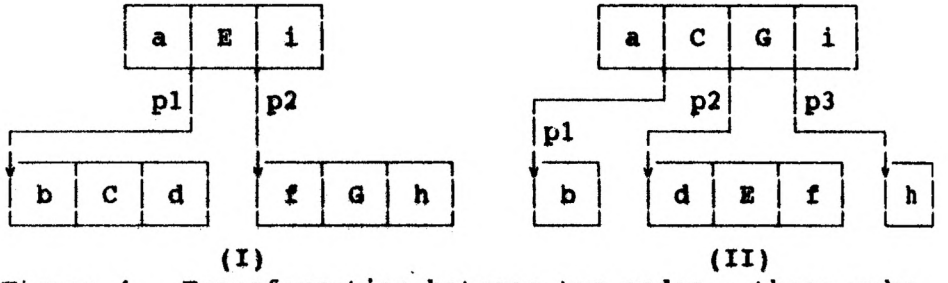


Figure 4. Transformation between two nodes - three nodes

1) From (I) to (II) of the fig.2 (rotate to right):

$Z(p_1) := Z(b)$;

$Z(p_2) := Z(d) + 1 + Z(f)$;

2) From (II) to (I) of the fig.2 (rotate to left):

$Z(p_1) := Z(b) + 1 + Z(d)$;

$Z(p_2) := Z(f)$;

3) From (II) to (I) of the fig.3 (fusion of two nodes into one):

$Z(p_1) := Z(b) + 1 + Z(d)$;

4) From (I) to (II) of the fig.4 (transformation of two nodes into three):

$Z(p_1) := Z(b)$;

$Z(p_2) := Z(d) + 1 + Z(f)$;

$Z(p_3) := Z(h)$.

We have used these four transformations in [Boian89] for implementation. If only these are used, at most two nodes are necessary for operations with the B-tree.

2. Distributed processing techniques for extended B-trees

One of the most important aspects in obtaining good execution efficiency in parallel and distributed computing is *load balancing*. Before executing a program on a distributed

computing system the work to be done must be partitioned among the available processors and for this to be an efficient action this work needs to be *balanced*.

Therefore, having at our disposal a distributed computing system, we consider that an adequate B-tree nodes distribution onto the available network configuration is of an extremely importance.

There are many ways for assigning the B-tree nodes to the available processing elements. We present below some of them, called *source initiated schemes*, which we consider well suited solutions for our problem of obtaining good distributed execution efficiency. In the following, we will identify the necessary activity at each B-tree node with a process, having then the correspondence "a node" - "one process" for simplifying our discussion.

The nature of the transformations that are taking place in a B-tree suggests a *message passing* distributed modelling. For example, the necessary messages for the cases described in figure 2, 3 and 4 are given below. Except create and remove, all the other remaining actions can be executed in parallel:

i). rotate left to right (fig.2, I to II):

p_0 to p_2 sends "E";

p_1 to p_2 sends "d";

p_1 to p_0 sends "C";

ii). rotate right to left (fig.2, II to I):

p_2 to p_0 sends "E";

p_2 to p_1 sends "d";

p_0 to p_1 sends "C";

iii). divide a node from two (fig.3, I to II):

Create a new node p_2;

p_1 to p_0 sends "C";

p_1 to p_2 sends "d";

iv). join the nodes p_1 and p_2 into p_1 (fig.3, II to I):

p_0 to p_1 sends "C";

p_2 to p_1 sends "d";

Remove the node p_2;

v). divide the nodes p_1 and p_2 from three (fig.4, I to II):

Create a new node p_3;

p_0 to p_2 sends "E";

p_1 to p_0 sends "C";

p_1 to p_2 sends "d";

p_2 to p_0 sends "G";

p_2 to p_3 sends "h";

vi). join the nodes p_1, p_2 and p_3 into p_1 and p_2 (fig.4 II to I):

p_3 to p_2 sends "h";

p_0 to p_2 sends "G";

p_2 to p_1 sends "d";

p_0 to p_1 sends "C";

p_2 to p_0 sends "E";

Source initiated schemes are characterized by the fact that the work splitting is performed only when an idle processor (called the *source* in this context) requests some work to do. Hence, the schemes presented here are all demand driven schemes. In all such schemes when a processor runs out of work it generates a request for work. What differentiates all the following different load balancing schemes is the way in which is made the selection of the target for this work request. This selection should be such as to minimize the total number of work requests and to balance the load among processors with fewest possible work transfers. The basic load balancing algorithm is the following [Gra91]:

```
while (not terminated)
        while (work not available)
                determine target;
                send request for work to target;
                receive message if any;
                if (message is work request) send a reject;
                if (message is a reject) reset flag to indicate
                   that a fresh target has to be determined and
                   another request for work be generated;
                service work requests and termination messages;
        end-while
        do work until exhausted and at the same time service
        work requests;
end-while;
```

In the *Asynchronous Round Robin* (ARR) scheme each processor maintains an independent variable *count*. Whenever a processor runs out of work it reads the value of *count* and sends a work request to that particular processor. The value of *count* is incremented (modulo P) each time its value is read and a work request sent. Initially, the value of *count* is set to ((p+1) modulo P) where p is the processor identification number. Since each processor has a counter of its own, work requests can be generated by each processor independent of the other processors.

Global Round Robin (GRR) considers the variable *count* stored in the processor 0 of a hypercube. When a processor needs work it requests and gets the value of this variable and the processor 0 increments the value by 1 modulo P before responding to another request. The processor needing work is sending now a request to the processor whose number was supplied by processor 0. This algorithm ensures that the work requests are uniformly distributed over all processors. A potential drawback of this scheme is the possible competition for reading *count*.

In the *Nearest Neighbour* scheme, a processor running out of work sends a work request to its nearest neighbours in a round robin fashion (for example, on a hypercube a processor will request its logN neighbours). Thus we have locality of communication for both work requests and actual work transfers. For networks in which the distance between any two processors is the same this scheme is the same as the Global Round Robin. In a way, this

scheme can be considered an adaptation of ARR for networks that are not completely connected. A potential drawback of this scheme is that localized concentration of work takes a longer time to globally balance the load among all processors.

To avoid competition for reading *count* in GRR, in *GRR with message combining* all the requests to read the value of *count* at processor 0 are combined at some intermediate processors. Thus, the total number of requests that have to be solved by processor 0 and its neighbours is greatly reduced. This technique is basically a software implementation of the *fetch&add* operation [Ski91].

In the *scheduler based load balancing* scheme a processor is designated as a scheduler. This processor maintains a list of all possible processors which can donate work. Initially this list contains just one processor which has all the work. Whenever a processor goes idle it sends a request to the scheduler. The scheduler then enquires the processors in the list of active processors in a round robin fashion till it gets work from one of the processors. This processor is then placed at the tail of the list and the work received by the scheduler is forwarded to the requesting processor.

Let's notice that the performance of this last scheme can be degraded significantly by the fact that all messages (including messages containing the actual work) are routed to the scheduler. This poses an additional bottleneck for the work transfer. We can improve this scheme so that the poll be still generated by the scheduler but the work be transferred directly to the requesting processor instead of being routed through the scheduler.

3. Conclusions and further reasearch

We presented in this paper some proposals on efficiently distribute the nodes of a B-tree on the processing elements of a distributed computing system. It's difficult to say in the general case which is the best strategy to use, taking into account the fact that the distribution of processes among processors in the general case in known to be NP-hard [Tao92].

As further research we hope to have access at some parallel architectures and make experimental evaluations of these load balancing techniques for a large number of B-tree applications and compare the performances with those theoretically estimated. Also, we want to determine more exactly the most efficient contents of the notion of process in this context,

this paper making the simplifying assumption "one node" - "one process", not necessarily the best choice.

REFERENCES

[Akl90] S.G.Akl - *The Design and Analysis of Parallel Algorithms*, Prentice Hall, 1989.

[Boian89] Boian F. M. - Sistem de fişiere bazat pe B-arbori, în *Lucrarile celui de-al VII-lea colocviu naţional de informatică*, INFO-IASI, 1989, pp. 33-40.

[Boian89a] Boian F. M. - Căutare rapidă în B-arbori, în *Lucrările simpozionului "Informatica şi aplicaţiile sale"*, Zilele academice Clujene, Cluj-Napoca, 1989.

[Gra91] A.Y.Grama, V.Kumar, V.N.Rao - Experimental Evaluation of Load Balancing Techniques for the Hypercube, in *Parallel Computing '91* (D.J.Evans et al.editors), Elsevier Science Publishers, pp.497-512.

[Knuth76] Knuth D.E. - *Tratat de programarea calculatoarelor*, vol III, Sortare şi căutare. Ed. Tehnică, Bucureşti, 1976.

[Kri89] Krishnamurthy E.V. - *Parallel Processing. Principles and Practice*, Addison-Wesley, 1989.

[Ski91] D.B.Skillicorn - Models for Practical Parallel Computation, *International Journal of Parallel Programming*, vol.20, no.2, pp.133-158, 1991.

[Tan92] A.S.Tanenbaum - *Modern Operating Systems*, Prentice Hall, 1992.

[Tao92] Tao L., Zhao Y.C., Narahari B. - *Efficient Heuristics for Task Assignment in Distributed Systems*, in Proceedings of 1992 International Conference on Parallel and Distributed Systems, December 16-18, 1992, Hsinchu, Taiwan, pp.134-141.

[Tou93] Tout W.R., Pramanik S. - *A Distributed Load Balancing Scheme for Data Parallel Applications*, Proceedings of the 1993 Int. Conference on Parallel Processing, pp.II-213 - II-216.