数理解析研究所講究録
511巻 1984年 1-25

A Fast Parallel Merging Algorithm for 2-3 trees

東工大・情報科学 柴山悦哉
(Etsuya SHIBAYAMA)

Introduction

A parallel algorithm is presented which merges two sorted lists represented as 2-3 trees of lengths m and n ($m \leq n$), respectively, with at most 2m processors within $O(\log n)$ time. The consideration for the time complexity includes comparisons, allocation of processors, and construction of an output 2-3 tree. The algorithm is performed without read conflicts.

Since a 2-3 tree is a dynamic data structure, the algorithm inevitably becomes dynamic and thus it is hard to be expressed on a static parallel computation model such as a SIMD (single instruction stream, multiple data stream) model. In section 2 we introduce a new parallel computation model consisting of processors which are connected via communication lines to form a 2-3 tree. The most significant feature of the model is that its communication lines are allowed to be changed dynamically during execution time. The parallel merging algorithm introduced in section 3 is designed and analyzed on the model.

The algorithm presented in this paper is competing with those which merge sorted lists represented as arrays (e.g. [8]). Generally speaking, however, it takes at least $O((m+n)/p)$ time to construct an array of length m+n using p processors. Therefore our merging algorithm is more effective than those whose outputs are arrays especially when n is by far greater than m and p.

Section 1 2-3 trees

Definition 1.1

A 2-3 tree [1] is a tree which satisfies that:

(1) each non-leaf node has 2 or 3 sons (i.e. descendant nodes).

(2) each path from the root to a leaf is of the same length.

□

In the sequel, if a non-leaf node has 3 sons, they are called the first son, the second son, and the third son, respectively, from left to right. On the other hand, if a node has 2 sons, they are called the first son and the second son, respectively, from left to right.

Remark that a tree which consists of the single node is also a 2-3 tree.

Definition 1.2

A 2-3 tree represents a sorted list in the following manner [1].

(1) Every element of the sorted list is assigned to a leaf of the 2-3 tree in increasing order from left to right.

(2) Every non-leaf node N which has 3 sons contains 4 values min(N), 1stmax(N), 2ndmax(N), and max(N) defined below, whereas every non-leaf node N which has 2 sons contains 3 values min(N), 1stmax(N), and max(N).

min(N) = " the least element assigned to the leaves of the subtree whose root is N "

max(N) = " the largest element assigned to the leaves of the subtree whose root is N "

1stmax(N) = " the largest element assigned to the leaves of the subtree whose root is the first son of N "

2ndmax(N) = " the largest element assigned to the leaves of the subtree whose root is the second son of N "

For each leaf node N, we also define min(N) = max(N) = "the element assigned to N".

□

The 2-3 tree of Fig. 1.1 represents the sorted list {2, 3, 5, 7, 8, 9, 11, 15, 16}. Here N_1, N_2, $\cdots$, and N_7 are the names of the non-leaf nodes. In each non-leaf node N_i ($0 \leq i \leq 7$), min(N_i), 1stmax(N_i), 2ndmax(N_i) (if N_i has 3 sons), and max(N_i) are written from left to right.

In order to execute dictionary operations (e.g. search, insert, and delete operations) efficiently on a 2-3 tree, 1stmax(N) and 2ndmax(N) provide sufficient information [1]. However, min(N) and max(N) are necessary for the algorithm presented in section 3.

Example 1

We show a way to delete the leaf in Fig. 1.1 to which the element 9 is assigned.

STEP 1 : Traversing each node N which satisfies $\min(N) \leq 9 \leq \max(N)$ from the root to a leaf. That is, N_1, N_3, N_6, and the leaf to which the element 9 is assigned are traversed.

STEP 2 : Detaching the leaf. (See Fig. 1.2.)

STEP 3 : Fusing N_6 and N_7. (See Fig. 1.3.)

STEP 4 : Fusing N_2 and N_3. (See Fig. 1.4.)

STEP 5 : Deleting N_1. (See Fig. 1.5.)

Definition 1.4

(1) The height of a node in a 2-3 tree is the length of a path from the node to a leaf. The height of a leaf node is 0.

(2) The height of a 2-3 tree is that of its root. If T is a 2-3 tree, high(T) stands for the height of T.

(3) The depth of a node in a 2-3 tree is the length of the path from the root to the node. The depth of the root is 0. □

For instance, the height of the 2-3 tree in Fig. 1.1 is 3.

Section 2 Parallel Computation Model

In this section we introduce a parallel computation model consisting of processors which form a 2-3 tree via communication lines. Here, for convenience, we regard a tree as a 2-3 tree even if some of its nodes has only one son. Remember that in Example 1 the delete operation is performed cleverly by allowing some node to have only one son temporarily.

Definition 2.1

A 2-3 parallel computation model consists of either the

single processor or the ones which are formed via bi-directional communication lines so that:

(1) the root processor has 2 or 3 sons.

(2) each processor which is neither the root nor a leaf has 1, 2, or 3 sons. However, each one which has no brothers must have 2 or 3 sons.

(3) each leaf is of the same depth.

(See Fig. 2.1.)

□

In order to change communication lines of a 2-3 parallel computation model dynamically, we introduce the following operations, which are a generalization of the delete operation in Example 1. Prune Operations detach useless leaves and Re-balance Operations attempt to let each non-leaf processor have two or three sons. Remark that the height of a non-root processor never changes by the operations until it is detached.

Prune Operation :

A processor whose sons are leaves detaches some but not all of them. (See Fig. 2.2.) Detached processors are disabled.

□

Re-balance Operation :

(1) When the root processor P_r has only 2 or 3 grandsons, it detaches the sons and adopts its grandsons as its own sons. (See Fig. 2.3)

(2) When the root processor P_r has more than 3 grandsons and some of them has just one son, processors are re-formed as follows: first, P_r adopts some grandsons as sons of their uncles in order that every son of P_r has 0, 2, or 3 sons; next, P_r detaches each son which has no sons. (See Fig. 2.4.)

(3) If a non-root processor P_x has 2 or 3 sons and some of them has just one son, processors are re-formed as follows: first, P_x adopts some grandsons as sons of their uncles in order that each son of P_x has 0, 2, or 3 sons; next, P_x detaches each son which has no sons. (See Fig. 2.5.)

□

When a processor P_x executes Re-balance Operation, communication lines between P_x and its sons as well as those between its sons and its grandsons are changed. Therefore, Re-balance Operation of P_x and that of its parent conflict with each other if they are performed simultaneously. On the other hand, since Re-balance Operation of P_x and that of its grandparent do not change the same communication lines, they do not conflict with each other even if they are performed parallelly. Supposing that the parent of P_x in Fig. 2.5 has just one son, the re-formation violates the condition (2) of Definition 2.1. In order to avoid this problem, P_x and its grandparent must perform Re-balance Operations in parallel.

Definition 2.2

A 2-3 dynamic computation model satisfies:

(1) the conditions as a 2-3 parallel computation model (i.e. Definition 2.1).

(2) each non-leaf processor is able to perform Prune Operation or Re-balance Operation in parallel with arbitrary processors except its parent and sons.

□

With Prune and Re-balance Operations, an effective dynamic scheduling algorithm for a sequence of associative operations can be implemented.

Example 2 : Computing 3.1+5+6+18+7+2+11+4.

In Fig. 2.6, each node is a processor, each edge is a communication line, each number enclosed in a node is stored on the local memory of the corresponding processor.

Attaching the symbol '+' to each non-leaf node in Fig 2.6, the processor system is regarded as a parse tree of ((3.1+5)+(6+18))+((7+2)+(11+4)), which suggests a natural way to solve the problem in parallel. However, if we assume that it takes ten time units to add a decimal fraction and an integer number, whereas it takes only one time unit to add two integer numbers, the parsing does not give an efficient answer. In this case the problem is solved more cleverly by Prune and Re-balance

Operations as follows.

Step 1: Each processor of a height 1 receives a value from the second son, sends it to the first son, and detaches the second son. Here Prune Operations are used. (See Fig. 2.7.)
Step 2: Each leaf processor adds the numbers allocated. In parallel, each processor of a height 2 performs Re-balance Operation. (See Fig. 2.8.)
Step 3: The leftmost leaf processor continues to compute 3.1+5. The rightmost processor of a height 1 performs the similar operation to that in Step 1. In parallel, the root processor performs Re-balance Operation. (See Fig. 2.9.)

After Step 3, the processors are re-formed as Fig. 2.10. Here, the leaf processors add numbers and the non-leaf ones devote to allocation of their respective sons. This problem is solved as if the parsing (3.1+5)+((6+18)+((7+2)+(11+4))) was used. Neglecting the cost of communication and that of Re-balance Operations, this method is more efficient than that given by the parsing ((3.1+5)+(6+18))+((7+2)+(11+4)).

In the following, we show a parallel algorithm which concatenates several 2-3 trees on a 2-3 dynamic computation model. This algorithm will be used as a subroutine in the parallel merging algorithm in section 3.

Definition 2.3

(1) Suppose that T and T' are 2-3 trees whose roots are N and N', respectively, T<T' stands for max(N)<min(N'). In other word, T<T' means that each element in the sorted list represented by T is less than each element in the sorted list represented by T'

(2) If 2-3 trees T and T' (T<T') represent the sorted lists $\{u_1,u_2,\cdots,u_k\}$ $(u_1<u_2<\cdots<u_k)$ and $\{v_1,v_2,\cdots,v_l\}$ $(v_1<v_2<\cdots<v_l)$, respectively, a 2-3 tree which represents $\{u_1,u_2,\cdots,u_k, v_1,v_2,\cdots,v_l\}$ is called a concatenation of T and T'. □

A concatenation of T and T' (T<T') is obtained by applying

the following operation, repeatedly. For simplicity, we omit the detail of updating information on the non-leaf nodes.

Concatenate Operation :

CASE 1 high(T)=high(T')

Creating a new node and letting the roots of T and T' be its first son and the second son, respectively. (See Fig. 2.11.)

CASE 2 high(T)<high(T')

The leftmost node of T' whose height is high(T)+1 is called N. If N has 2 sons, T and T' is concatenated as Fig. 2.12. Otherwise, they are splited and fused as Fig 2.13.

CASE 3 high(T)>high(T')

Similar to CASE 2.

□

In case of Fig. 2.13, a concatenation of T and T' is obtained by applying Concatenate Operation, repeatedly. In any case, the operation produces one or two 2-3 trees which are higher at least by one than the lower one of T and T'.

[Algorithm - Concatenating several 2-3 trees-]

For simplicity, first, we consider the case of Fig. 2.14, where each node is a processor, each edge is a communication line, each triangle is a 2-3 tree, and each arrow represents a pointer to the corresponding 2-3 tree. We assume that the 2-3 trees are on a shared memory and that $T_1 < T_2 < \cdots < T_8$ is satisfied.

Step 1: Each processor of a height 1 performs Re-balance Operation as if each 2-3 tree pointed by an arrow from its son were its grandson. (See Fig. 2.15.)

Step 2: Each processor of a height 2 performs Re-balance Operation. In parallel, each leaf processor applies Concatenate Operation. (See Fig. 2.16.)

Step 3: Each processor of an odd height performs Re-balance Operation. (See Fig. 2.17.)

□

8

After completing Step 3, Re-balance and Concatenate Operations are repeatedly performed until only one 2-3 tree remains. (Fig. 2.18)

Generally, the following program computes a concatenation of several 2-3 trees. Here the processors are synchronized such that each processor which finishes Step 1 (or 2) waits until the others complete the Step.

Program 1 :

Comment: Assume that each non-leaf processor has 2 or 3 sons and that each leaf processor has just one 2-3 tree;

While there remain more than one 2-3 trees Do

Step 1: Each processor of an odd height performs a Re-balance Operation;

Step 2: Each leaf processor performs Concatenate Operation. In parallel, each non-leaf processor of an even height performs Re-balance Operation;

EndWhile;

□

By the following lemma, the 2-3 dynamic computation model does not violate the condition of Definition 2.1 in execution time.

Lemma 2.4

If a 2-3 dynamic computation model satisfies the following condition (1), Re-balance Operations of the non-leaf processors whose heights are even re-form the model so that the condition (2) is satisfied. On the other hand, if the model satisfies the condition (2), Re-balance Operations of the non-leaf processors whose heights are odd re-form the model so that the condition (1) is satisfied.

(1) Each non-leaf processor of an even height has 2 or 3 sons and each non-leaf processor of an odd height has 1, 2, or 3 sons.

(2) Each non-leaf processor of an odd height has 2 or 3 sons and each non-leaf processor of an even height has 1, 2 or 3 sons.

Proof: Obvious. □

Theorem 2.5

On a 2-3 dynamic computation model with at most 2m processors, m 2-3 trees which contain n leaves can be concatenated within O(log n) time by Program 1.

Proof:

By Lemma 2.4, before entering Step 2 in Program 1, each leaf processor has 2 or 3 2-3 trees. If it has 2 2-3 trees, it can apply Concatenate Operation to them so that the height of the lowest 2-3 trees is increased at least by 1. Otherwise, by applying the operation at most twice, the height of the lowest 2-3 trees is increased at least by 1. Since there exist only n leaves, Step 2 is not executed more than $\log_2 n$ time. □

Section 3 A Parallel Merging Algorithm

In this section, we design and analyze a parallel algorithm which merges two sorted lists $A = \{a_1, a_2, \cdots, a_m\}$ $(a_1 < a_2 < \cdots < a_m)$ and $B = \{b_1, b_2, \cdots, b_n\}$ $(b_1 < b_2 < \cdots < b_n)$. For simplicity, we assume that:

1) A and B are disjoint from each other; 2) $m \leq n$; 3) $b_n < a_m$. The last assumption seems strange but the algorithm can be easily modified so that it merges two sorted lists which do not satisfy the assumption. In the sequel, $\bar{A}$ and $\bar{B}$ are 2-3 trees which represent A and B, respectively.

In order to use pipelining, we had better consider that a list of subtrees in $\bar{B}$ represents a subset of B. To speak more precisely, when $\bar{B}_1, \bar{B}_2, \ldots, \bar{B}_k$ $(\bar{B}_1 < \bar{B}_2 < \cdots < \bar{B}_k)$ are disjoint subtrees of $\bar{B}$ and represent the subsets $B_1, B_2, \ldots,$ and B_k, respectively, the list of them represents the union of $B_1, B_2, \ldots,$ and B_k.

In this section, for convenience, the notations for the sets (e.g. $\in$ and $\cup$) will be used as if the sorted lists and the lists of 2-3 trees were sets. For instance, if S is a sorted

list, $v \in S$ means that v belongs to S.

Definition 3.1

The list of 2-3 trees which represents the sorted list $\{b_j \in \bar{B} \mid u < b_j < v\}$ with the fewest number of disjoint subtrees in $\bar{B}$ is denoted as $\bar{B}[u,v]$.

□

The sorted list $\bar{B}[u,v]$ is determined uniquely by Theorem 3.2 which are to be stated.

In the sequel, we assume that the sorted list A is represented by a 2-3 dynamic computation model, whereas B is represented by an ordinal 2-3 tree. For instance, Fig. 3.1 illustrates the 2-3 dynamic computation model which represents the sorted list {6, 10, 13, 21, 23}. Here each node is a processor, each edge is a communication line, P_1, P_2, $\cdots$, P_8 are the names of the processors, each value enclosed with a node is stored on the local memory of the corresponding processor.

The merging algorithm is divided into the following 3 phases.

Phase 1: Dividing $\bar{B}$ into m lists of subtrees $\bar{B}[-\infty, a_1]$, $\bar{B}[a_1, a_2]$, $\cdots$, and $\bar{B}[a_{m-1}, a_m]$. Here, we assume that $-\infty$ is less than every element of $A \cup B$.

Phase 2: Constructing m 2-3 trees which represent $\{b_j \mid b_j < a_1\} \cup \{a_1\}$, $\{b_j \mid a_1 < b_j < a_2\} \cup \{a_2\}$, $\cdots$, and $\{b_j \mid a_{m-1} < b_j < a_m\} \cup \{a_m\}$, respectively.

Phase 3: Concatenating the 2-3 trees constructed in Phase 2 using the algorithm introduced in section 2.

□

By Phase 1,2, and 3, $\bar{A}$ and $\bar{B}$ can be obviously merged. For example, when $\bar{A}$ and $\bar{B}$ are the 2-3 trees in Fig. 3.1 and 3.2, respectively, each Phase is executed as follows. For simplicity, in Fig. 3.2, the information stored on the non-leaf nodes is omitted. In the sequel, "P_i sends a 2-3 tree to P_j" means that P_i sends the address of the root in the 2-3 tree (i.e. a pointer to it) to P_j. Before execution the processors and $\bar{B}$ are formed as Fig. 3.3.

Phase 1:

Step 1: P_1 divides $\bar{B}$ into $\bar{B}[-\infty,10]$ and $\bar{B}[10,23]$, and sends the 2-3 trees belonging to them to P_2 and P_3, respectively. Here the value 10 is the 1stmax of the root in $\bar{A}$.

Step 2: P_2 receives the 2-3 trees in $\bar{B}[-\infty,10]$, divides it into $\bar{B}[-\infty,6]$ and $\bar{B}[6,10]$, and sends the 2-3 trees belonging to them to P_4 and P_5, respectively. In parallel, P_3 receives the 2-3 trees in $\bar{B}[10,23]$, divides it into $\bar{B}[10,13]$, $\bar{B}[13,21]$, and $\bar{B}[21,23]$, and sends the 2-3 trees belonging to them to P_6, P_7, and P_8, respectively.

□

Step 1 and 2 mentioned above is improved by pipelining. For instance, just after P_2 receives the leftmost tree in Fig. 3.4, the processor can determine that the tree belongs to $\bar{B}[-\infty,6]$ and send it to P_4. For this, P_2 need not wait until it receives the other 2-3 trees. In this respect it is advantageous to represent a sorted list by a list of 2-3 trees instead of a 2-3 tree itself.

Phase 2:

Each leaf processor concatenates the 2-3 trees received in Phase 1 and one which represents the sorted list consisting of the element assigned to the processor. For instance, P_4 concatenates the 2-3 trees in $\bar{B}[-\infty,6]$ and one which represents $\{6\}$.

Phase 3:

By the algorithm introduced in section 2, the 2-3 trees developed in Phase 2 are concatenated.

□

By Theorem 2.5, Phase 3 is completed in $O(\log n)$ time. In Phase 2, each leaf processor performs several Concatenate Operations.

The most essential part of Phase 1 is to divide some $\bar{B}[u,v]$ into $\bar{B}[u,w]$ and $\bar{B}[w,v]$ $(u<w<v)$. That is, in Fig. 3.3 - 3.5, P_2 and P_3 obviously divide $\bar{B}[-\infty,10]$ and $\bar{B}[10,23]$, respectively, and if we consider that $\bar{B}$ and $\bar{B}[-\infty,23]$ consisting of $\bar{B}$ alone are

identical, P_1 divides $\bar{B}[-\infty,23]$. For the time being, we show some properties of $\bar{B}[u,v]$ and an efficient way to compute $\bar{B}[u,w]$ and $\bar{B}[w,v]$ from $\bar{B}[u,v]$ $(u<w<v)$. By the following theorem, we can determine which 2-3 trees belong to $\bar{B}[u,v]$.

Theorem 3.2

Assuming that N is a node in $\bar{B}$, the following (1) and (2) are equivalent.

(1) the 2-3 tree whose root is N belongs to $\bar{B}[u,v]$.

(2) $u<\min(N)\wedge\max(N)<v$ is satisfied but if N has its parent N_p in $\bar{B}$, $u<\min(N_p)\wedge\max(N_p)<v$ is not satisfied.

Proof :

Assume that T and T_p are the 2-3 trees whose roots are N and N_p, respectively.

(1) => (2)

The 2-3 tree T represents a subset of $\{b_j | u<b_j<v\}$ since it belongs to $\bar{B}[u,v]$. Therefore $u<\min(N)\wedge\max(N)<v$ is satisfied. On the other hand, if $u<\min(N_p)\wedge\max(N_p)<v$ were satisfied, by removing each subtree of T_p from $\bar{B}[u,v]$ and adding T_p to it, we would get a list of 2-3 trees which represents $\{b_j | u<b_j<v\}$ but which has fewer elements than $\bar{B}[u,v]$. Since this would contradict the definition of $\bar{B}[u,v]$, $u<\min(N_p)\wedge\max(N_p)<v$ is not satisfied.

(1) <= (2)

By $u<\min(N)\wedge\max(N)<v$, the 2-3 tree T, a subtree of T, or a supertree of T must belong to $\bar{B}[u,v]$. If a subtree of T belonged to $\bar{B}[u,v]$, by removing each subtree of T in $\bar{B}[u,v]$ and adding T to it, we would get a list of 2-3 trees which represents $\{b_j | u<b_j<v\}$ but which has fewer elements than $\bar{B}[u,v]$. If a supertree of T belonged to $\bar{B}[u,v]$, $\bar{B}[u,v]$ could not represent $\{b_j | u<b_j<v\}$ since $u<\min(N_p)\wedge\max(N_p)<v$ is not satisfied. Therefore T belongs to $\bar{B}[u,v]$.

□

Corollary 3.3

Assuming that N is a node in $\bar{B}$, the following (1) and (2) are equivalent.

(1) N is a node in a 2-3 tree which belongs to $\bar{B}[u,v]$.

(2) $u<\min(N) \wedge \max(N)<v$ are satisfied.

Proof :

Obvious from Theorem 3.2.

□

The next theorem suggests how to compute $\bar{B}[u,w]$ from $\bar{B}[u,v]$ $(u<w<v)$.

Theorem 3.4

Assume that T whose root is N is a subtree of $\bar{B}$. When $u<w<v$ is satisfied, a necessary and sufficient condition for T's belonging to $\bar{B}[u,w]$ is that either (1) or (2) is satisfied.

(1) T is an element of $\bar{B}[u,v]$ such that $\max(N)<w$ is satisfied.

(2) T is a subtree of some element in $\bar{B}[u,v]$ such that $\max(N)<w$ is satisfied but when N_p is the parent of N, $\max(N_p)<w$ is not satisfied.

Proof :

=> (1) or (2)

By Theorem 3.2, $\max(N)<w$ is obvious. By Corollary 3.3, N is a node in some 2-3 tree which belongs to $\bar{B}[u,v]$. Therefore, T is either a 2-3 tree in $\bar{B}[u,v]$ or one which is a subtree of some 2-3 tree in $\bar{B}[u,v]$. In the former case, (1) is satisfied. In the latter case, $u<\min(N_p) \wedge \max(N_p)<w$ is not satisfied. In this case, however, since N_p is also in some 2-3 tree which belongs to $\bar{B}[u,v]$, $u<\min(N_p)$ is satisfied. Therefore, $\max(N_p)<w$ is not satisfied.

<= (1)

By Theorem 3.2, $\min(N)<u$ is satisfied. Therefore if N does not have its parent in $\bar{B}$, T belongs to $\bar{B}[u,v]$. Otherwise supposing that N_p is the parent of N, $u<\min(N_p) \wedge \max(N_p)<v$ is not satisfied by Theorem 3.2. Therefore $u<\min(N_p) \wedge \max(N_p)<w$

is not satisfied and so T belongs to B[u,w].

<= (2)

Obvious.

□

Similarly supposing that T whose root is N is a subtree of $\bar{B}$ and that u<w<v is satisfied, a necessary and sufficient condition for T's belonging to $\bar{B}$[w,v] is that either (1') or (2') is satisfied.

(1') T is an element of $\bar{B}$[u,v] such that w<min(N) is satisfied.

(2') T is a subtree of some element in $\bar{B}$[u,v] such that w<min(N).

When $\bar{B}$ is the 2-3 tree in Fig. 3.2, $\bar{B}[-\infty,21]$ consists of the 2-3 trees $\bar{B}_1$, $\bar{B}_2$, $\bar{B}_3$, and $\bar{B}_4$ in Fig. 3.8. As an example, we consider how to determine $\bar{B}[-\infty,6]$ and $\bar{B}[6,21]$ from them. First, by Theorem 3.2, $\bar{B}_2$, $\bar{B}_3$, and $\bar{B}_4$ belong to $\bar{B}[6,21]$. On the other hand, $\bar{B}_1$ which does not belong to neither $\bar{B}[-\infty,6]$ nor $\bar{B}[6,21]$ has to be divided. By the theorem, if N is a node in $\bar{B}_1$ and is the root of some 2-3 tree in either $\bar{B}[-\infty,6]$ or $\bar{B}[6,21]$, min(N)<6<max(N) must be satisfied. Therefore, if only we examine the sons of N_1, N_3, and N_4 in Fig. 3.8, $\bar{B}[-\infty,6]$ and $\bar{B}[6,21]$ are determined. By the following lemma, every node N which satisfies min(N)<6<max(N) are searched by traversing them from the root of $\bar{B}_1$ to a leaf.

Lemma 3.5

Assume that N and N'(N≠N') are nodes in a 2-3 tree. For some value w, if min(N)<w<max(N) and min(N')<w<max(N') are satisfied, N is either an ancestor or a descendant of N'.

Proof :

If N is neither an ancestor or a descendant of N', the 2-3 trees whose roots are N and N', respectively, would be disjoint from each other. This would contradict min(N)<w<max(N) and min(N')<w<max(N').

□

Now we show that Phase 1 can be performed in O(log n) time with at most 2m processors. The following program, which is a generalization of the example of Fig. 3.3-3.5, expresses how each processor works in Phase 1. Here, min(P), 1stmax(P), 2ndson(P), and max(P) are defined analogously to Definition 1.2. $pred(a_i)=a_{i-1}$ (if $i>1$), and $pred(a_1)=-\infty$. The processors are synchronized by message passing.

```
[Program 2 - dividing B̄ -]
 For each processor P:
   For h From high(B̄) To 0 Step -1 Do
     If P is not the root Then
        P receives each 2-3 tree of a height h which belongs to
        B̄[pred(min(P)),max(P)] from the parent
     EndIf;
     If P has 2 sons
       Then
          P sends the 2-3 trees of a height h which belong to
          B̄[pred(min(P)),1stmax(P)] or B̄[1stmax(P),max(P)] to
          the respective sons
       Else If P has 3 sons Then
          P sends the 2-3 trees of a height h which belong to
          B̄[pred(min(P)),1stmax(P)], B̄[1stmax(P),2ndmax(P)], or
          B̄[2ndmax(P),max(P)] to the respective sons
     EndIf EndIf;
   EndFor
```

□

The following lemmas and theorem guarantee that Phase 1 is completed in O(log n) time.

Lemma 3.6

Suppose that P_a is the root processor of $\bar{A}$ and it has just 2 sons. By executing Program 2, P_a sends each 2-3 tree of a height h ($0\le h\le high(\bar{B})$) which belongs to either $\bar{B}[-\infty,1stmax(P_a)]$ or $\bar{B}[1stmax(P_a),max(P_a)]$ to the appropriate son within $O(high(\bar{B})-h+1)$ time.

Proof :

If h=high($\bar{B}$), the lemma is obviously satisfied. When $0 \le h < high(\bar{B})$, P_a searches the node N_b in $\bar{B}$ of a height h+1 which satisfies that $min(N_b) < 1stmax(P_a) < max(N_b)$ within $O(high(\bar{B})-h)$ time if exist. Therefore the 2-3 trees which belong to $\bar{B}[-\infty, 1stmax(P_a)]$ or $\bar{B}[1stmax(P_a), max(P_a)]$ are determined in $O(high(\bar{B})-h+1)$ time.

□

Similarly, if the root processor P_a has 3 sons, it is proved that P_a sends the 2-3 trees of a height h which belong to $\bar{B}[-\infty, 1stmax(P_a)]$, $\bar{B}[1stmax(P_a), 2ndmax(P_a)]$, or $\bar{B}[2ndmax(P_a), max(P_a)]$ within $O(high(\bar{B})-h+1)$ time to the respective sons.

Lemma 3.7

Assume that a processor P_a which is neither the root nor a leaf in $\bar{A}$ is in a depth d and has just 2 sons. If P_a receives the 2-3 trees of a height h ($0 \le h \le high(\bar{B})$) which belong to $\bar{B}[pred(min(P_a)), max(P_a)]$ within $O(high(\bar{B})-h+d)$ time, the processor sends the 2-3 trees of a height h which belong to $\bar{B}[pred(min(P)), 1stmax(P)]$ or $\bar{B}[1stmax(P), max(P)]$ to the respective sons within $O(high(\bar{B})-h+d+1)$ time.

Proof :

Suppose that T is a 2-3 tree of a height h which belongs to $\bar{B}[pred(min(P)), 1stamx(P)]$ or $\bar{B}[1stmax(P), max(P)]$. If T also belongs to $\bar{B}[pred(min(P)), max(P)]$, P_a can obviously send it to the appropriate son within $O(high(\bar{B})+d-h+1)$ time. Otherwise the lemma is proved similarly to Lemma 3.6.

□

Similarly, if P_a receives the 2-3 trees of a height h ($0 \le h \le high(\bar{B})$) which belong to $\bar{B}[pred(min(P_a)), max(P_a)]$ within $O(high(\bar{B})-h+d)$ time, the processor sends the 2-3 trees of a height h which belong to $\bar{B}[pred(min(P_d)), 1stmax(P_d)]$, $\bar{B}[1stmax(P_d), 2ndmax(P_d)]$ or $\bar{B}[2ndmax(P_d), max(P_d)]$ to the respective sons within $O(high(\bar{B})-h+d+1)$ time.

Theorem 3.8

$\bar{B}[-\infty,a_1]$, $\bar{B}[a_1,a_2]$, $\cdots$, and $\bar{B}[a_{m-1},a_m]$ are obtained within $O(\log n)$ time by Program 2.

Proof :

By Lemma 3.6 and 3.7, the following (*) is proved by induction on h.

(*) If P_a is a non-root processor of $\bar{A}$ whose depth is d, the processor receives the 2-3 trees of a height h ($0 \le h \le high(\bar{B})$) which belong to $\bar{B}[pred(min(P)),max(P)]$ within $O(high(\bar{B})+d-h)$ time.

Therefore, the i-th leaf processor in the left to right order has received the 2-3 trees in $\bar{B}[pred(a_i),a_i]$ within $O(high(\bar{B})+high(\bar{A}))=O(\log n)$ time. □

By Theorem 3.8, Phase 1 is completed in $O(\log n)$ time.

Lemma 3.9

Supposing that $\bar{B}[u,v]$ is the list of 2-3 trees $\{T_1,T_2,\cdots,T_k\}$ ($T_1<T_2<\cdots<T_k$), there exists k' ($1 \le k' \le k$) such that the following conditions (1),(2), and (3) are satisfied.

(1) $high(T_1) \le high(T_2) \le \cdots \le high(T_{k'}) \ge \cdots \ge high(T_k)$

(2) In $\{T_1,T_2,\cdots,T_{k'}\}$, there are at most two 2-3 trees of the same height.

(3) In $\{T_{k'+1},T_{k'+2},\cdots,T_k\}$, there are at most two 2-3 trees of the same height.

Proof :

By dividing a list of 2-3 trees which satisfies the condition (1), (2), and (3) in the way we have described, two lists of 2-3 trees which also satisfy the conditions are obtained.

□

If a list of 2-3 trees which represents a sorted list of a length n' satisfies the conditions (1),(2), and (3) in Lemma 3.5, it is known that the 2-3 trees can be concatenated within $O(\log n')$ time [1]. Therefore Phase 2 is performed within $O(\log n)$ time. Since Phase 3 is also completed within $O(\log n)$ time

by Theorem 2.5, the parallel merging algorithm is performed within O(log n) time.

At last we show that the parallel merging algorithm is executed without read conflicts (i.e. simultaneous reading of the same memory word). Obviously, Phase 2 and 3 are executed without read conflicts. On the other hand, in Phase 1, the following are satisfied.

(1) Each non-root processor reads a node only if it is in a 2-3 tree which has been received from the parent.

(2) Each non-leaf processor does not read a node in a 2-3 tree which has been sent to a son.

(3) Each non-leaf processor sends 2-3 trees which are disjoint from one another.

By (1) and (2), if a processor is either a descendant or an ancestor of another, read conflicts between them never happen. By (3), if a processor is neither a descendant nor an ancestor of another, read conflicts between them never happen.

Acknowledgement

The author would like to express his deep gratitude to Professor Reiji Nakajima for his kind and appropriate advice. The author also thanks Tatsuya Hagino, Tatsuyuki Akiyama, and Takashi Sakuragawa who read earlier drafts and gave him worth-while suggestions.

Reference

1. AHO, A.V., HOPCROFT, J.E., and ULLMAN, J.D. The Desingn and Analysis of Computer Algorithms. Addison-Wesley, (1974).

2. BROWN, M.R. and TARJAN, R.E. A Fast Merging Algorithm. JACM, Vol. 26(1979), No. 2, pp. 211-226.

3. DEKEL, D. and SAHNI, S. Binary Trees and Parallel Scheduling Algorithms. CONPAR 81, Lecture Notes on Computer Science Springer-Verlag, Vol. 111, pp. 480-492.

4. GAVRIL, F. Merging with Parallel Processors. CACM, Vol. 18, No. 10(1975), pp. 588-591.

5. HIRSCHBERG, D. S. Fast Parallel Sorting Algorithms. CACM, Vol. 21, No. 8(1978), pp. 657-661

6. HWANG, F.K. and LIN, S.A. A Simple Algorithm for Merging Two Disjoint Linearly Ordered Sets. SIAM J. on Computing, Vol. 1, No. 1(1972), pp. 31-39.

7. PREPARATA, F. P. New Parallel Sorting Schemes. IEEE Trans. on Computers, C-27 No. 7(1978) pp. 669-673.

8. SHILOACH, Y. and VISHIKIN, U. Finding the Maximum, Merging, and Sorting in a Parallel Computation Model. Journal of Algorithms, Vol. 1, No. 2(1981), pp. 81-102.

9. VALIANT, L.G. Parallelism in Comparison Problem, SIAM J. on Computing, Vol. 4, No. 3(1975), pp. 348-355.

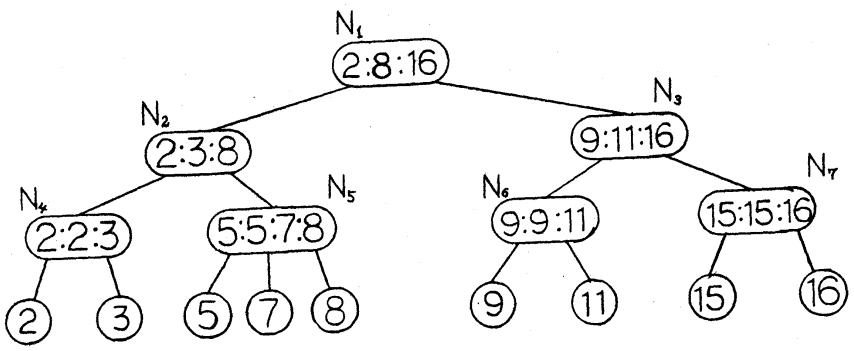

Fig. 1.1

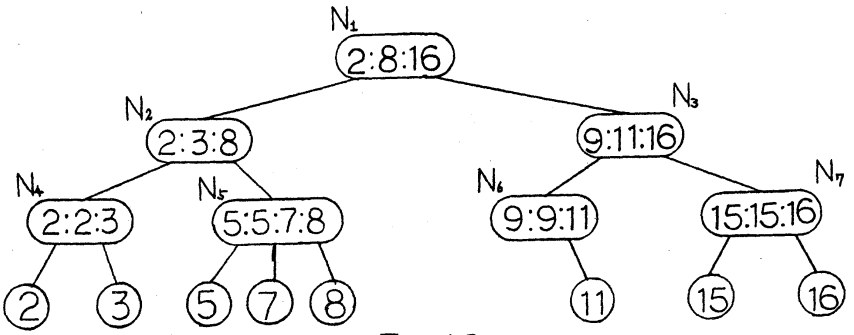

Fig. 1.2

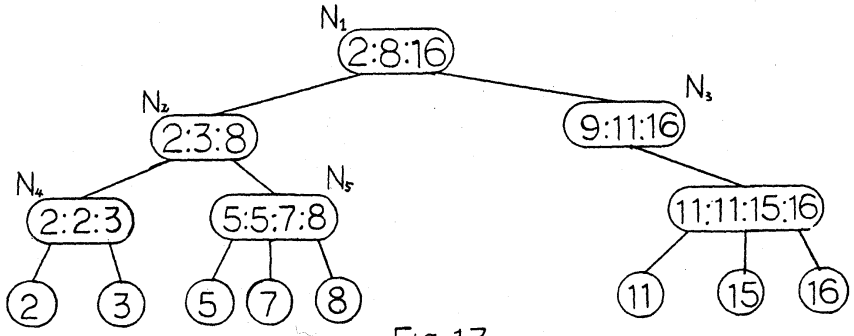

Fig. 1.3

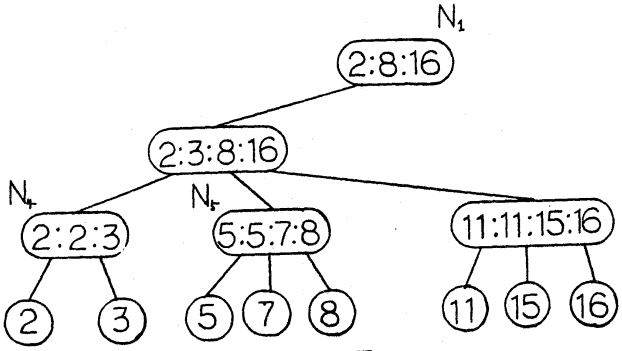

Fig. 1.4

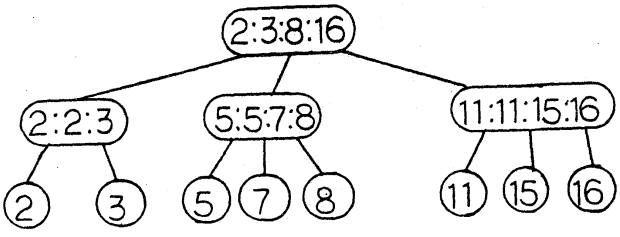

Fig. 1.5

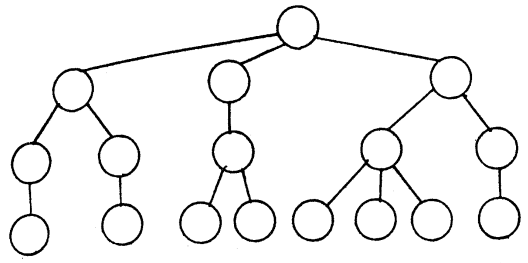

Fig. 2.1 — A 2-3 computation model —

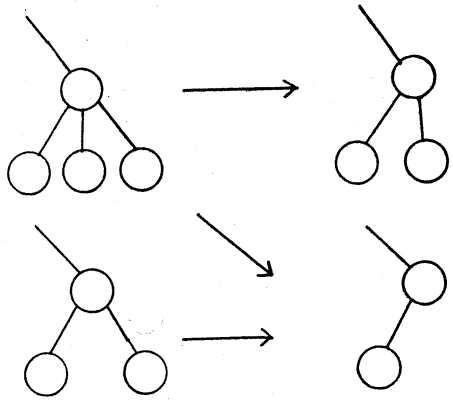

Fig. 2.2

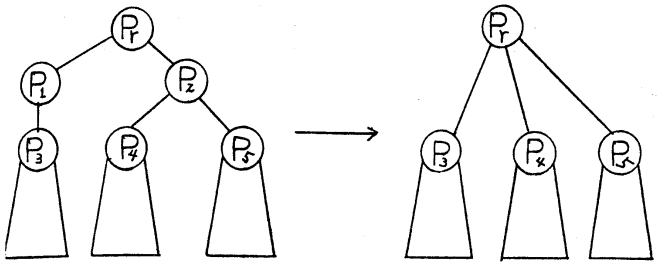

Fig. 2.3

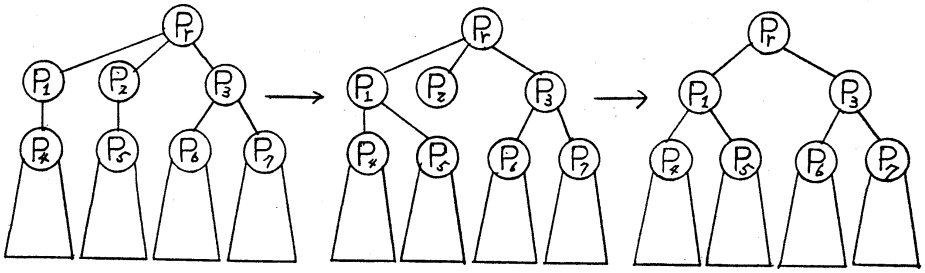

Fig. 2.4

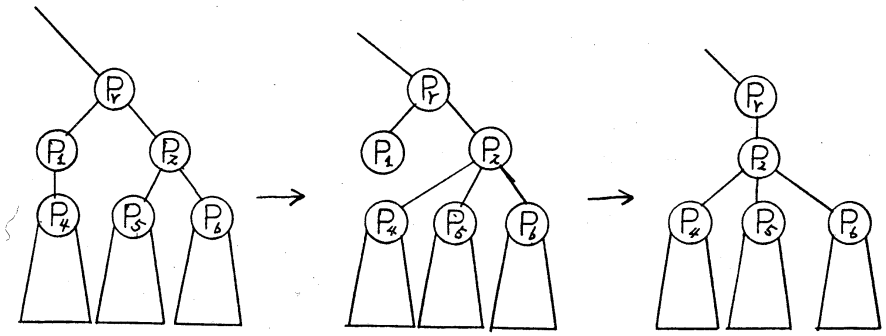

Fig. 2.5

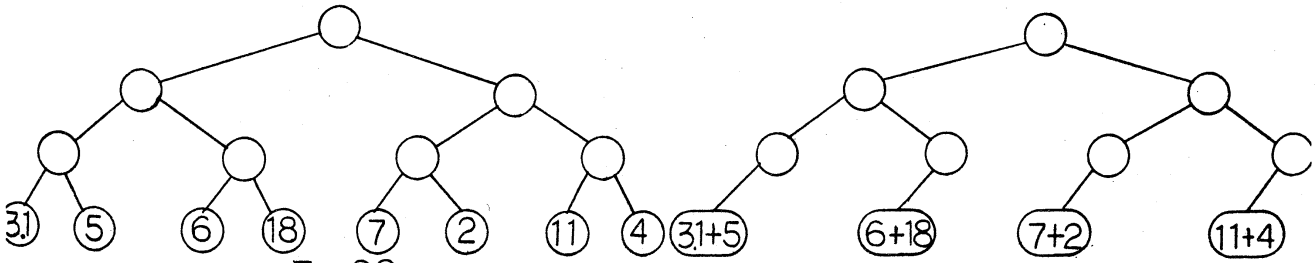

Fig. 2.6

Fig. 2.7

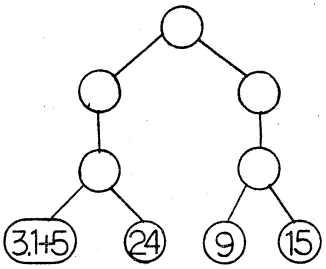

Fig. 2.8

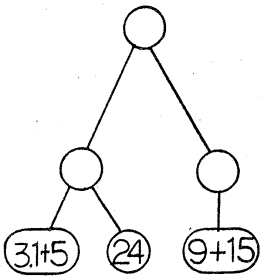

Fig. 2.9

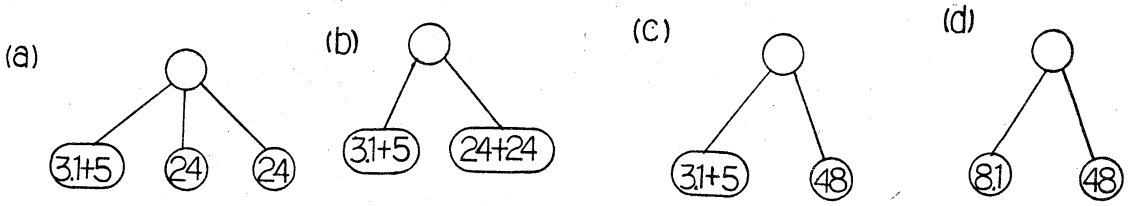

Fig. 2.10

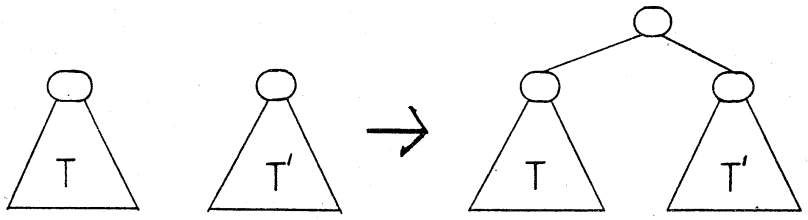

Fig. 2.11

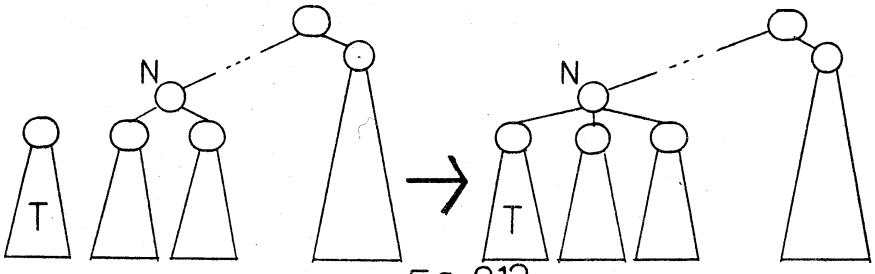

Fig. 2.12

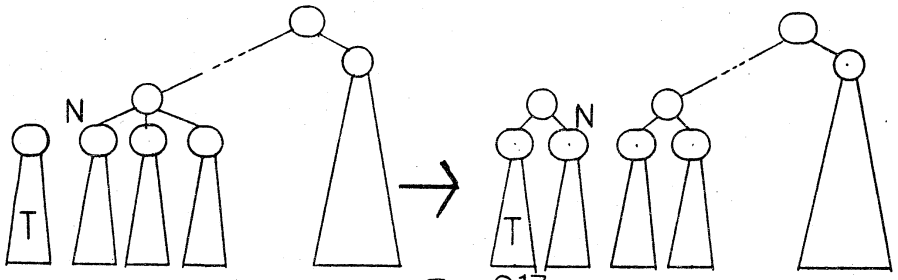

Fig. 2.13

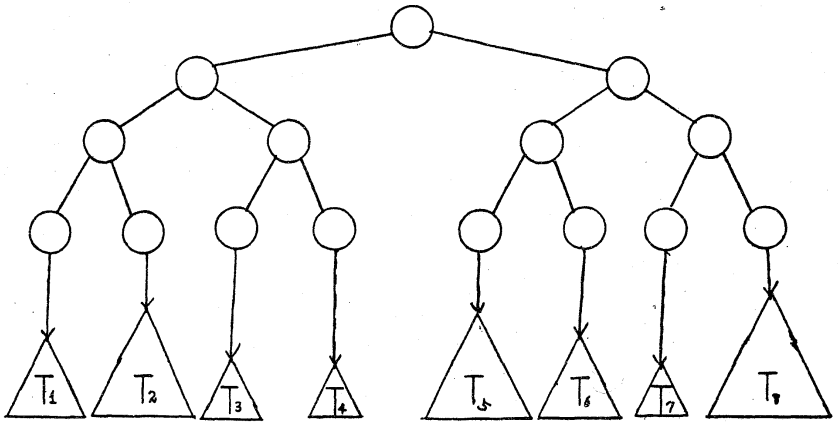

Fig. 2.14

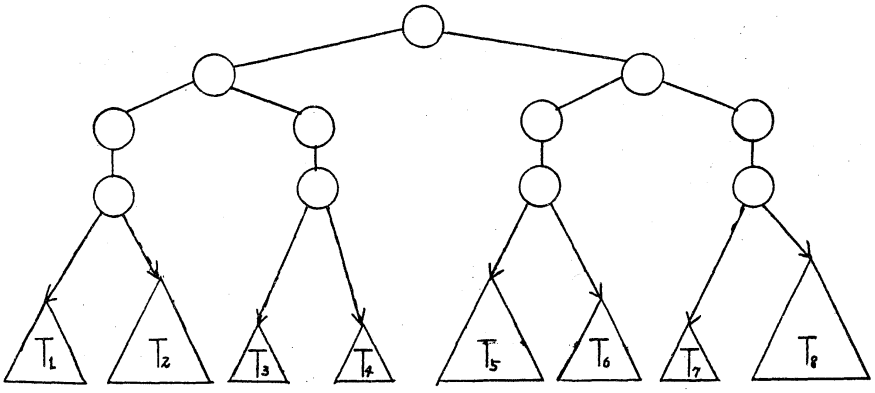

Fig. 2.15

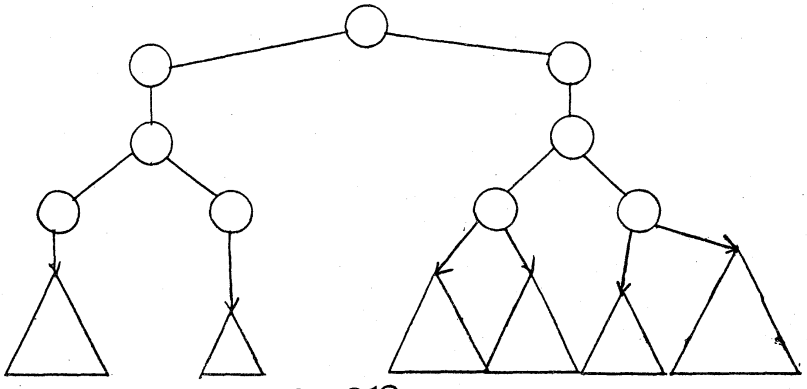

Fig. 2.16

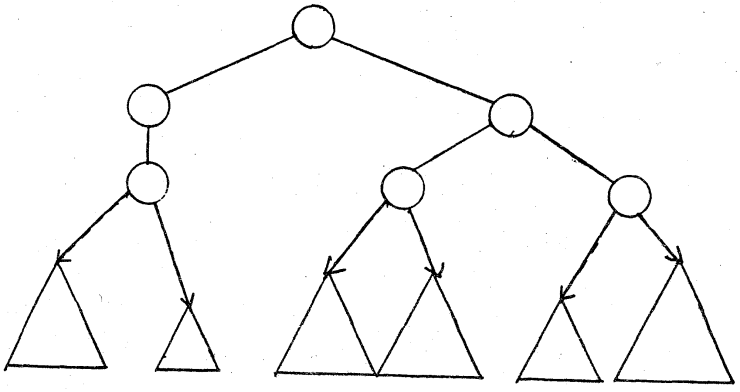

Fig. 2.17

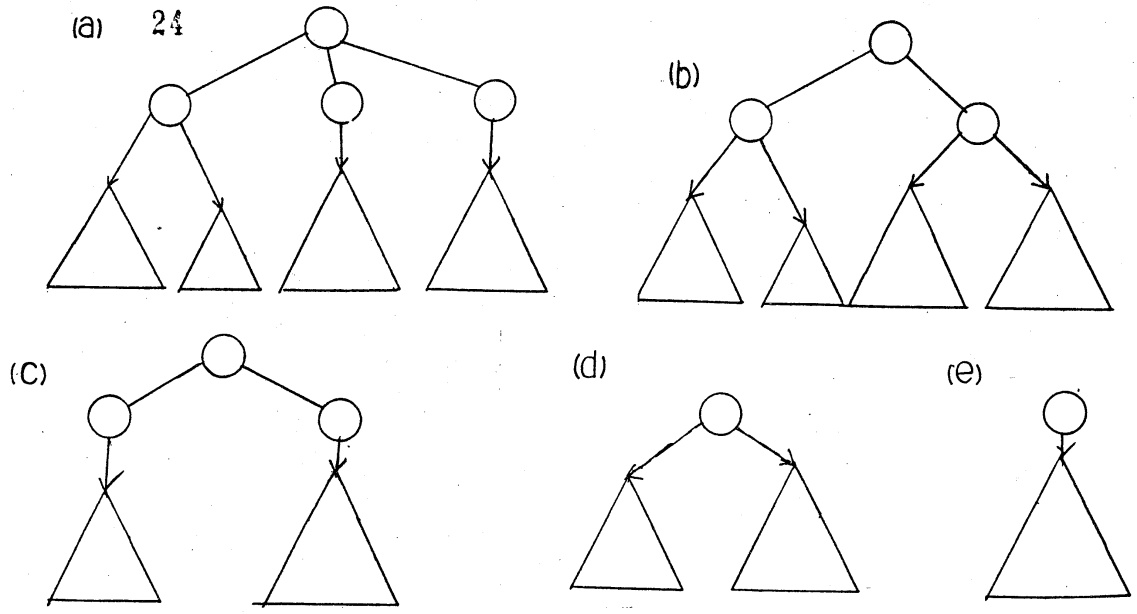

Fig. 2.18

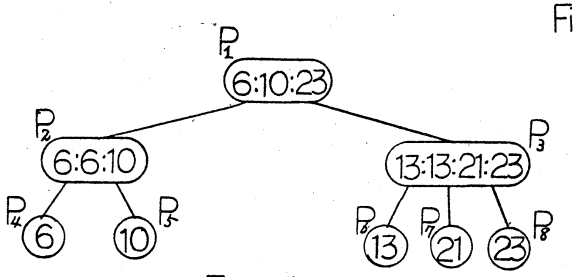

Fig. 3.1

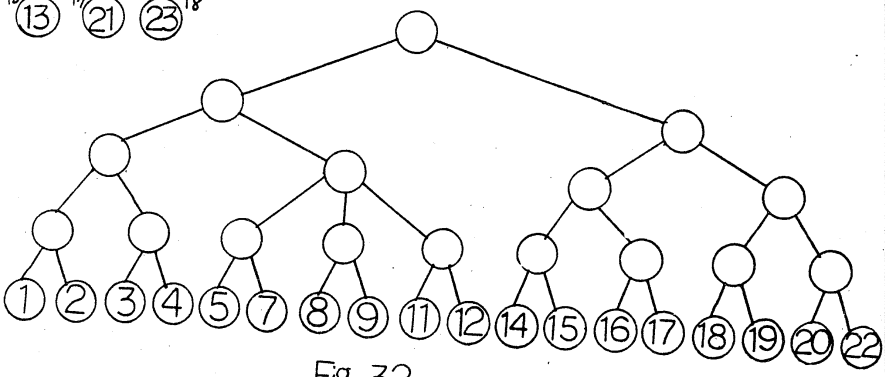

Fig. 3.2

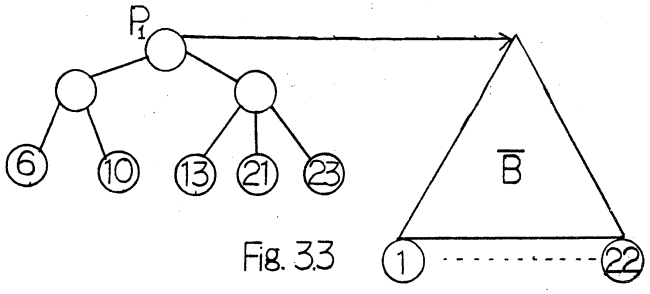

Fig. 3.3

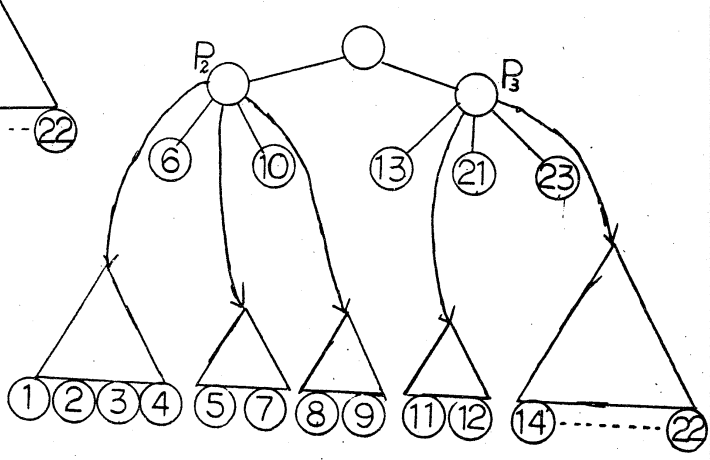

Fig. 3.4

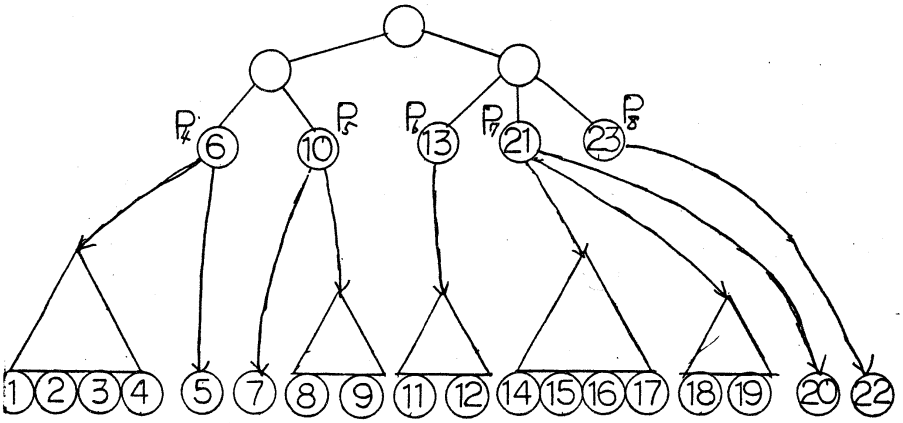

Fig. 3.5

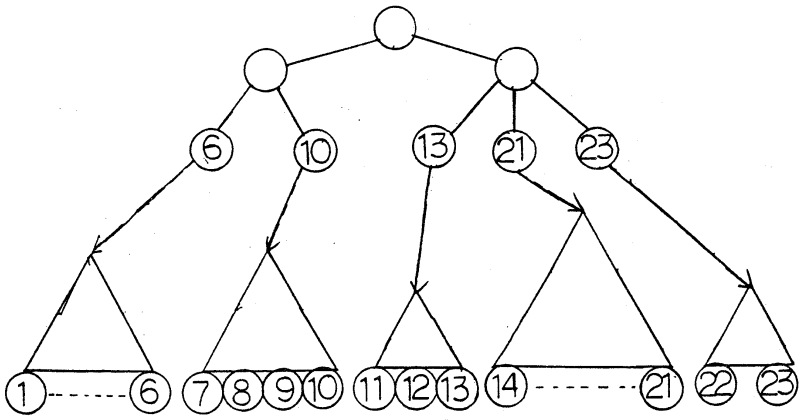

Fig. 3.6

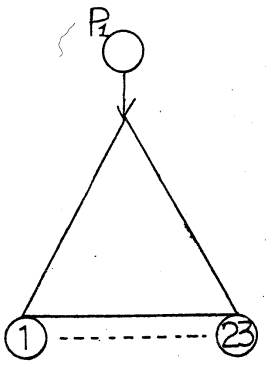

Fig. 3.7

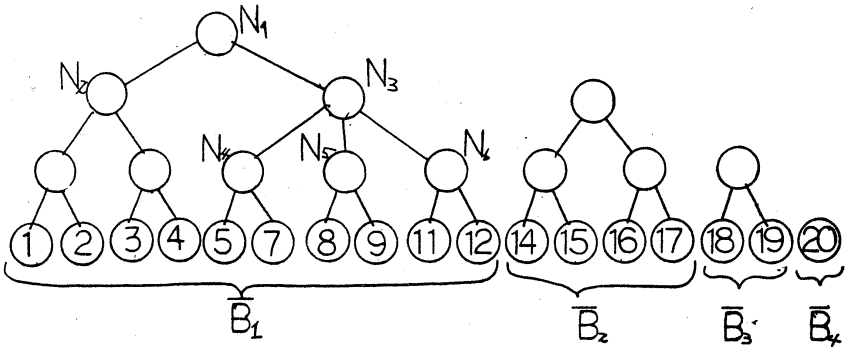

Fig. 3.8